

Programación de Sistemas y Concurrencia

Tema 5: Interacción entre Procesos

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Índice

- Tipos de procesos
- El problema de los jardines (Exclusión Mutua)
- El problema del productor consumidor (Condiciones de sincronización)
- Solución al problema del productor consumidor con espera activa
- Solución al problema de la exclusión mutua con espera activa para dos procesos
- El problema de la panadería
- Corrección de un programa concurrente
- Justicia
- El problema de los lectores/escriptores
- El problema de los filósofos

Procesos vs Recursos

- En un programa concurrente intervienen tres tipos de entidades:
 - **Entidades activas**: modeladas como procesos (hebras, tareas,...)
 - **Entidades pasivas**: son recursos que necesitan los procesos para realizar su trabajo
 - **Con control de acceso**: para acceder a algunos recursos es necesario que se satisfagan ciertas condiciones de seguridad, como por ejemplo la exclusión mutua.
 - **Sin control de acceso**: recursos a los que se puede acceder en cualquier momento.

El problema de los jardines

- Supón que hay un jardín al que un número arbitrario de personas puede visitar
- Al jardín se puede acceder a través de dos puertas distintas
- El problema consiste en conocer cuantas personas hay en el jardín en cada momento



El problema de los jardines

- Cada puerta es simulada por un proceso, que se ejecuta concurrentemente
 - Puerta1 || Puerta2
- Una variable global entera representará en cada momento el número total de personas que ha entrado por cada una de las puertas

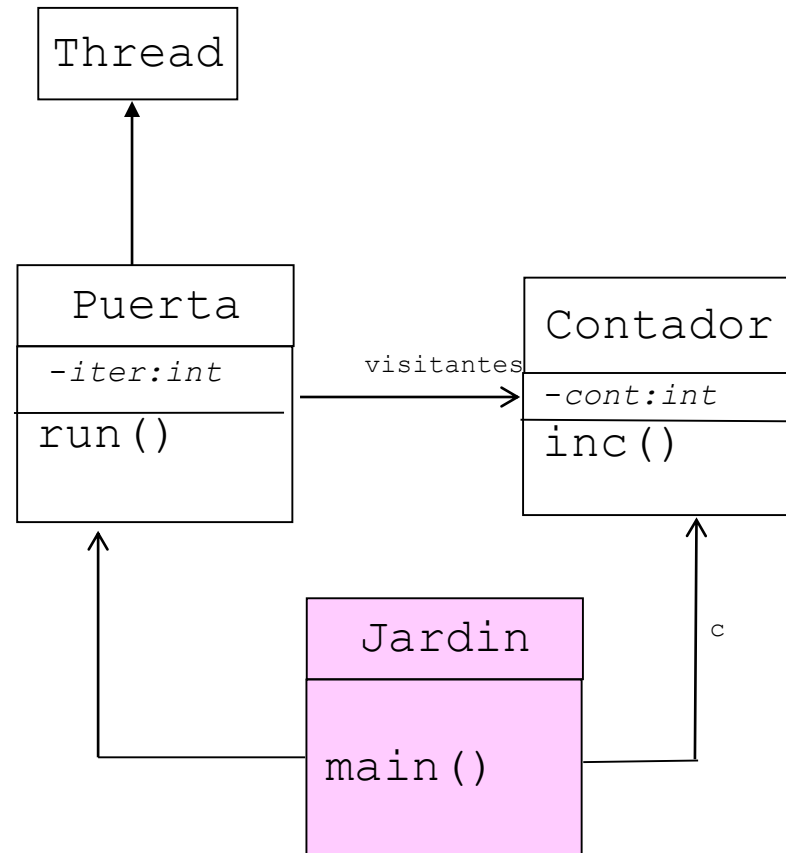


El problema de los jardines

Una implementación en Java

- Cada hebra **Puerta** simula la llegada periódica de un visitante cada cierto tiempo llamando al método **inc()** de un objeto **Contador** que lleva la cuenta del número total de visitantes

- Diagrama de clases



El Problema de los jardines

- El método **main** crea las hebras **p1**, **p2** y el objeto **Contador**

```
public class Jardines {  
    public static void main(String[] args){  
        Contador c = new Contador();  
        Puerta p1 = new Puerta(c,10000000);  
        Puerta p2 = new Puerta(c,10000000);  
  
        p1.start();  
        p2.start();  
  
        try{  
            p1.join();  
            p2.join();  
        } catch (InterruptedException e){  
            System.out.println("La hebra ha sido interrumpida");  
        }  
        System.out.println(c.valor());  
    }  
}
```

El Problema de los jardines

```
public class Puerta extends Thread{  
    private Contador visitantes;  
    private int iter;  
    public Puerta(Contador c,int iter){  
        visitantes = c;  
        this.iter = iter;  
    }  
  
    public void run(){  
        for (int i = 0; i< iter; i++){  
            visitantes.inc();  
        }  
    }  
}
```

```
public class Contador {  
    private int cont = 0;  
  
    public void inc(){  
        cont++;  
    }  
  
    public int valor(){  
        return cont;  
    }  
}
```

El método **run** termina (y también la hebra) cuando han entrado **iter** visitantes

El Problema de los jardines

P1	P2	Suma	Suma Correcta	Diferencia
10000	10000	17439	20000	2561



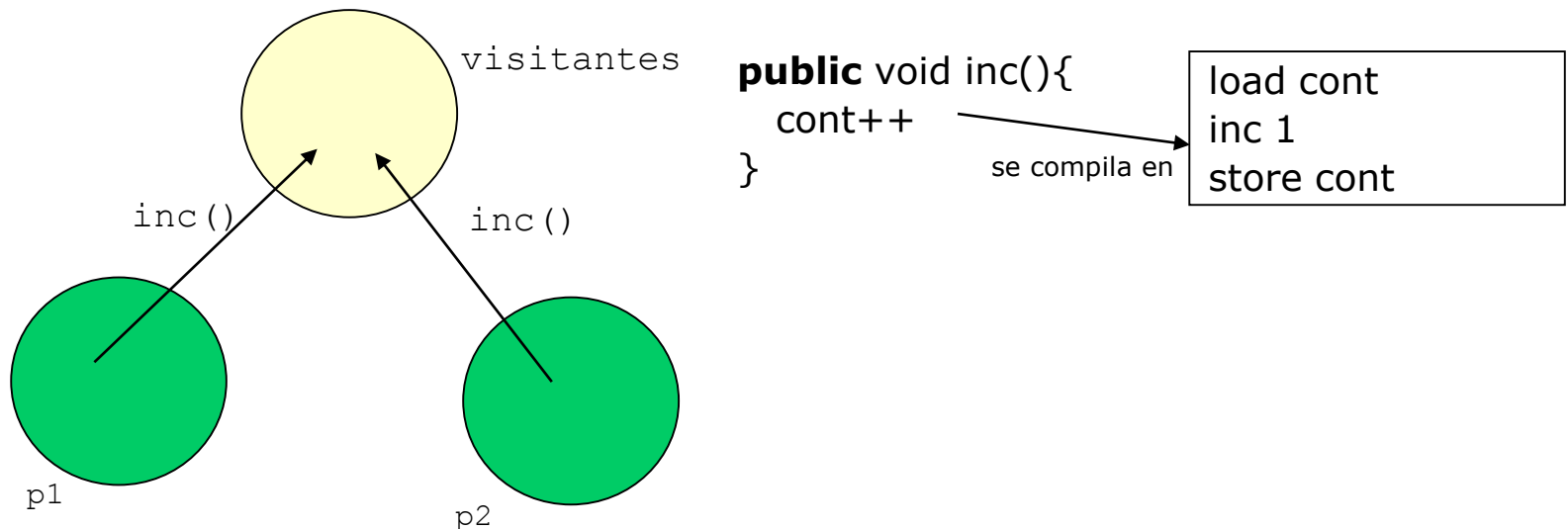
Ejemplo de ejecución

Entran 10000 visitantes por cada puerta, y la suma es 17439 en lugar de 20000

¿Por qué?

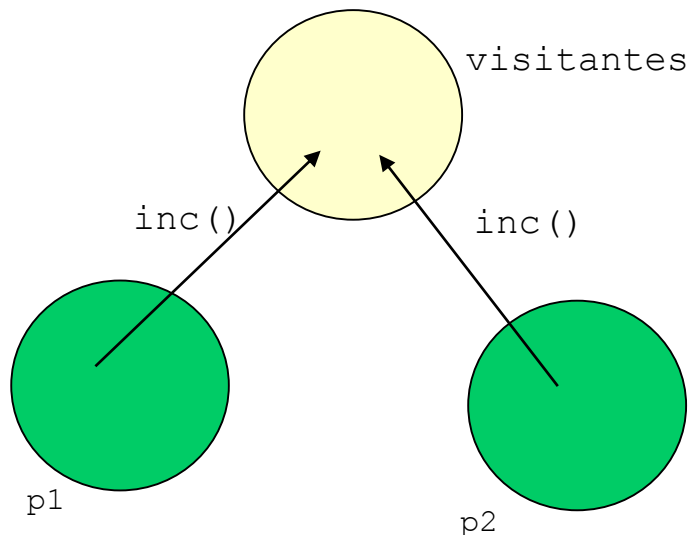
El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución
p1: load cont

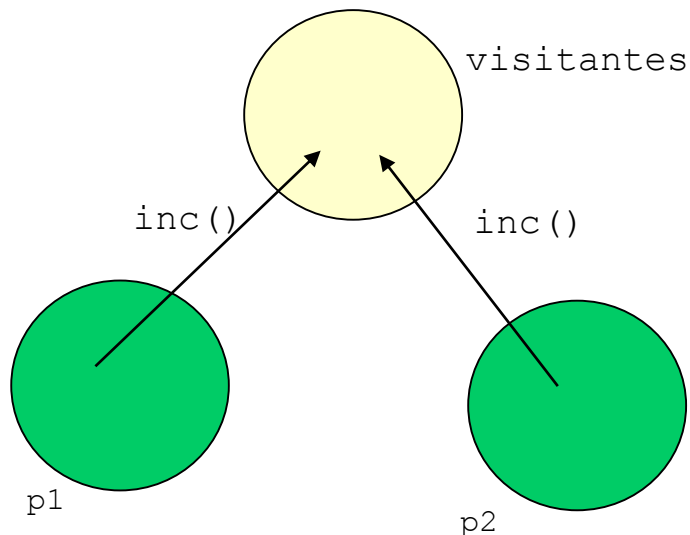
```
cont = 0
```

CPU

?

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución
p1: load cont

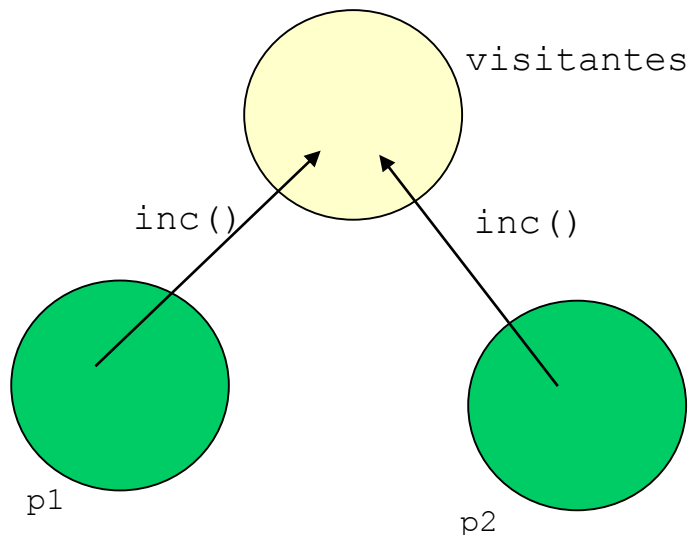
```
cont = 0
```

CPU

0

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución

p1: load cont

p2: load cont

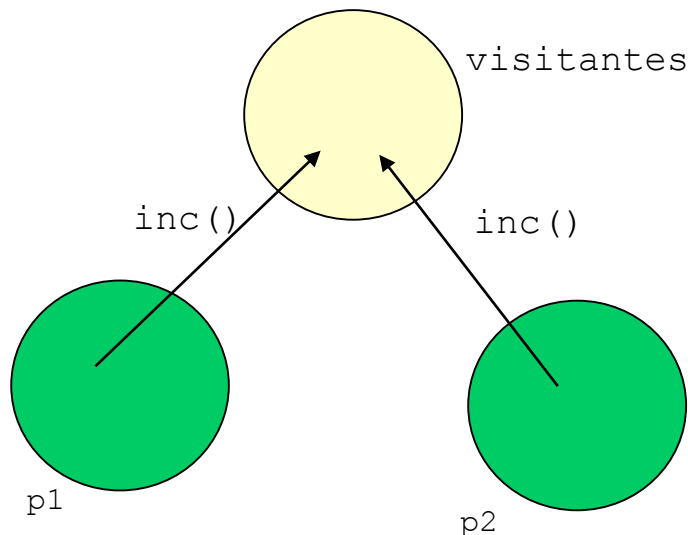
```
cont = 0
```

CPU

0

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de **visitantes** por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución

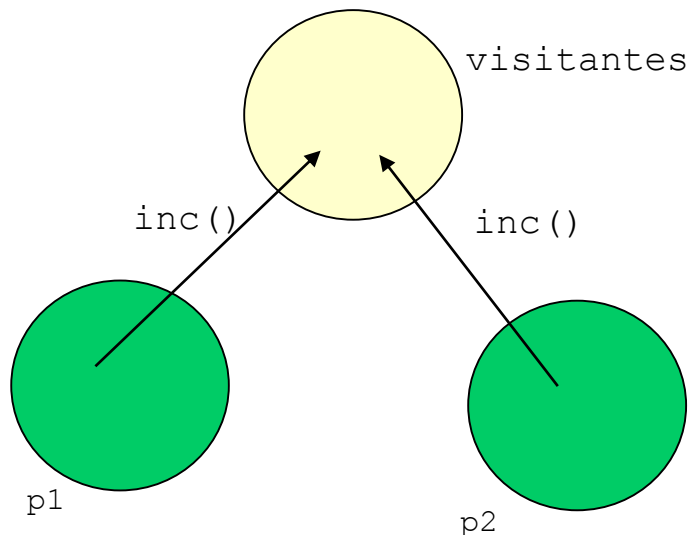
```
p1: load cont  
p2: load cont  
p1: inc 1
```

```
cont = 0
```

CPU
1

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución

```
p1: load cont  
p2: load cont  
p1: inc 1  
p2: inc 1
```

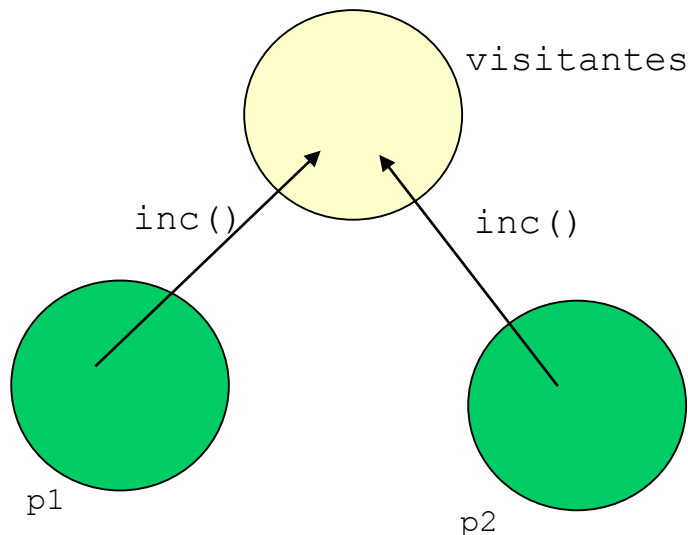
```
cont = 0
```

CPU

1

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución

```
p1: load cont  
p2: load cont  
p1: inc 1  
p2: inc 1  
p1: store cont
```

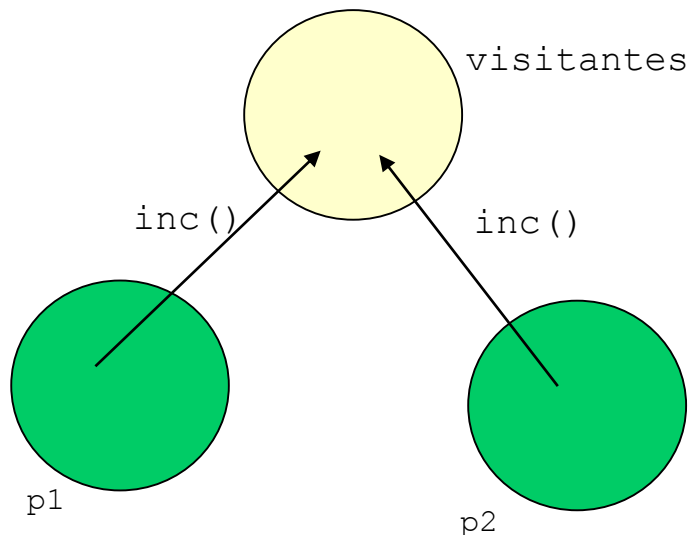
```
cont = 1
```

CPU

1

El problema de los jardines

- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



```
public void inc(){  
    cont++  
}
```

se compila en

```
load cont  
inc 1  
store cont
```

Ejemplo de traza de ejecución

```
p1: load cont  
p2: load cont  
p1: inc 1  
p2: inc 1  
p1: store cont  
p2: store cont
```

```
cont = 1
```

CPU

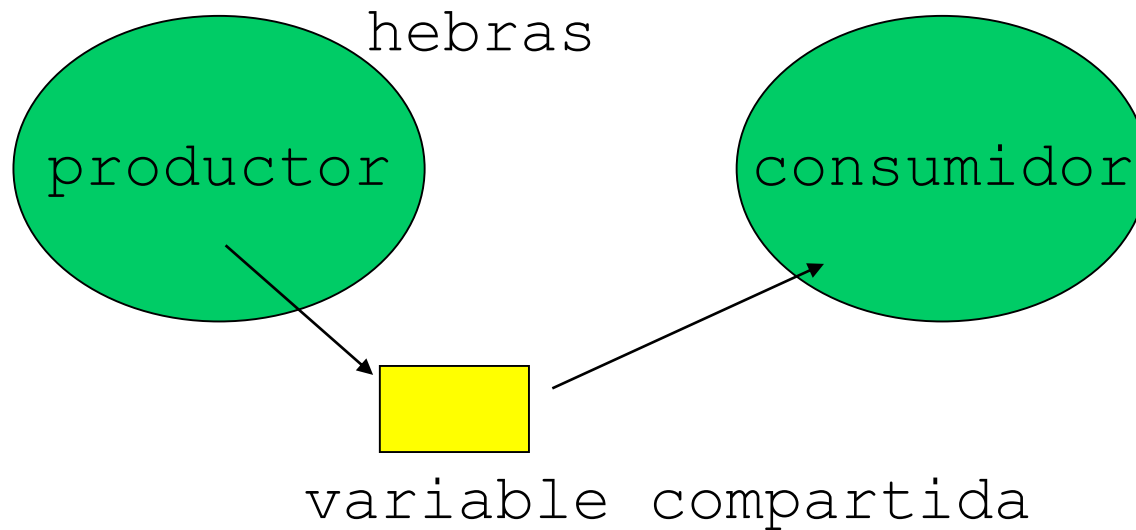
1

El problema de los jardines

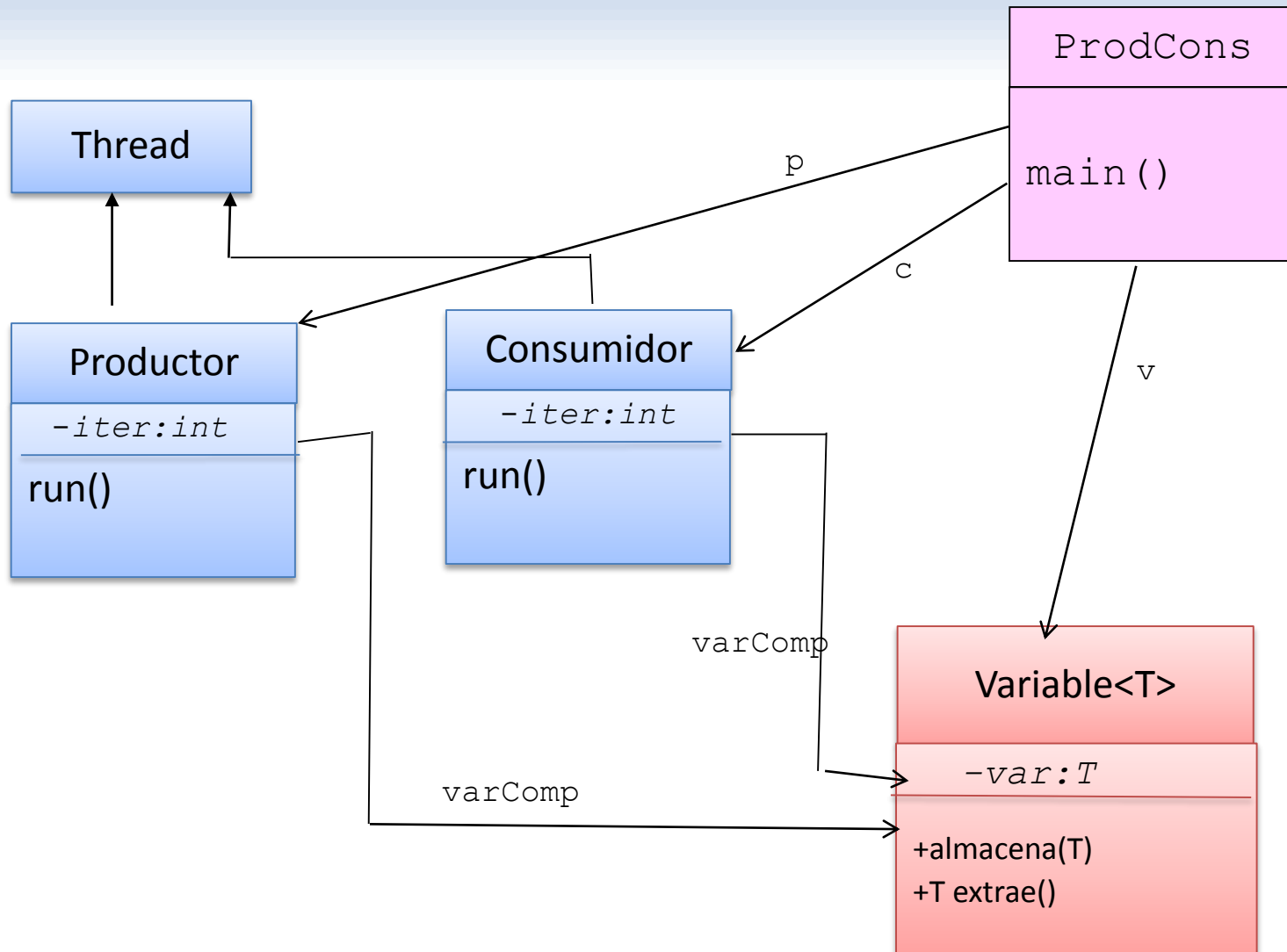
- Este ejemplo muestra uno de los grandes problemas de la programación concurrente
 - Detectar cuándo un recurso (**cont**) compartido necesita algún **control de acceso** para asegurar su integridad
 - Debemos impedir que varios procesos utilicen **simultáneamente** el recurso para que no se produzcan **interferencias**.
 - El código de acceso al recurso en cada una de las hebras se denomina **sección crítica**
 - La no interferencia entre secciones críticas significa que su ejecución no debe solaparse en el tiempo
 - Las secciones críticas deben aparecer como **códigos atómicos** para el resto de los procesos
 - Cuando dos secciones críticas no se interfieren se dice que se ejecutan en **exclusión mutua**.
 - Garantizar la exclusión mutua requiere **sincronizar** a los procesos involucrados
 - Si P1 quiere ejecutar su sección crítica cuando P2 la está ejecutando, **P1 debe esperar** a que P2 termine y viceversa

El problema del productor/consumidor

- Un proceso **productor** produce de forma ininterrumpida datos que deben ser consumidos por otro proceso **consumidor**
- En su versión más simple el productor deja el dato producido en una variable compartida, a la que accede el consumidor para extraer dicho dato



Productor/Consumidor: Diagrama de Clases



Productor/consumidor: Clase principal

```
public class ProductorConsumidor {  
    public static void main(String[] args) {  
        Variable<Integer> v = new Variable<Integer>();  
        Productor p = new Productor(10,v);  
        Consumidor c = new Consumidor(10,v);  
  
        p.start();  
        c.start();  
    }  
}
```

- Se crean el objeto **variable compartida** y las hebras **productor** y **consumidor**

Productor/consumidor: Clases

Variable<T>, Productor y Consumidor

```
public class Variable<T> {  
  
    private T var;  
  
    public void almacena(T dato){  
        var = dato  
    }  
    public T extrae(){  
        return var;  
    }  
}
```

```
public class Productor extends Thread{  
    private static Random r = new Random();  
    private int numIter;  
    private Variable<Integer> var;  
    public Productor(int numIter, Variable<Integer> var){  
        this.numIter = numIter;  
        this.var = var;  
    }  
    public void run(){  
        int nDato = 0;  
        for (int i = 0; i<numIter;i++){  
            nDato = r.nextInt(100);  
            System.out.println("Productor "+nDato);  
            var.almacena(nDato);  
        }  
    }  
}
```

```
public class Consumidor extends Thread{  
    private int numIter;  
    private Variable<Integer> var;  
    public Consumidor(int numIter, Variable<Integer> var){  
        this.numIter = numIter;  
        this.var = var;  
    }  
  
    public void run(){  
        int nDato = 0;  
        for (int i = 0; i<numIter;i++){  
            nDato = var.extrae();  
            System.out.println("Consumidor "+nDato)  
        }  
    }  
}
```

Productor/consumidor: ejemplo de una ejecución

Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Consumidor 32
Productor 32
Productor 71
Productor 53
Productor 98
Productor 9
Productor 87
Productor 46
Productor 90
Productor 67
Productor 87

El consumidor ha consumido el mismo dato muchas veces

El productor ha producido datos que el consumidor no ha leído

- En este caso **no tenemos un problema de exclusión mutua** porque el productor **escribe** sobre la variable y el consumidor **lee**
- Hay que imponer dos **condiciones**
 1. El consumidor no puede extraer un dato hasta que no se ha producido uno nuevo
 2. El productor no puede almacenar un nuevo dato hasta que no se haya leído el anterior
- Estas propiedades imprescindibles para que la solución al problema sea correcta se denominan **condiciones de sincronización**
- Por el momento, modelamos estas condiciones utilizando bucles de **espera activa**



clase Variable<T> revisada

```
public class Variable<T> {  
  
    private T var;  
    private boolean hayDato = false;  
  
    public void almacena(T dato){  
        while (hayDato) Thread.yield();  
        var = dato;  
        System.out.println("se ha almacenado el dato "+dato);  
        hayDato = true;  
    }  
  
    public T extrae(){  
        while (!hayDato) Thread.yield();  
        System.out.println("se ha consumido el dato "+var);  
        int v = var;  
        hayDato = false;  
        return v;  
    }  
}
```

Añadimos una variable Booleana para saber en cada momento si hay un nuevo dato

Bucle de espera activa, el productor espera a que no haya datos para almacenar el siguiente (condición de sincronización 2)

Bucle de espera activa, el consumidor espera a que haya un dato para extraerlo (condición de sincronización 1)

Productor Consumidor revisado (salida)

Productor 58
Consumidor 58
Productor 49
Consumidor 49
Productor 48
Consumidor 48
Productor 90
Consumidor 90
Productor 14
Consumidor 14
Productor 93
Consumidor 93
Productor 35
Consumidor 35
Productor 16
Consumidor 16
Productor 24
Consumidor 24
Productor 58
Consumidor 58

- Cada dato producido por el productor es consumido por el consumidor
- El consumidor no consume dos veces ningún dato
- Esta solución no es del todo satisfactoria puesto que la ejecución de los procesos está **muy acoplada** (realizan cada iteración de forma sincronizada)
- La solución general utiliza un **buffer intermedio** para desacoplar a los procesos

Exclusión mutua con espera activa

- Tipo de solución que buscamos

Hebra h0

```
run(){  
    while (true){  
        preProtocolo0  
        SC0  
        postProtocolo0  
        SNC0  
    }  
}
```

Hebra h1

```
run(){  
    while (true){  
        preProtocolo1  
        SC1  
        postProtocolo1  
        SNC1  
    }  
}
```

Requisito 1:

En cada momento, hay, a lo sumo, una hebra ejecutando su sección crítica

Exclusión mutua con espera activa (estructura de la solución)

```
public class Hebra0{
    private Sinc s;
    public Hebra0(Sinc s){
        this.s = s;
    }
    public void run(){
        while (true){
            s.preProt0();
            //SC0
            s.postProt0();
            //SNC0
        }
    }
}
```

```
public class Sinc{
    ...
    public void preProt0(){
        ....
    }
    public void postProt0(){
        ....
    }
    public void preProt1(){
        ....
    }
    public void postProt1(){
        ....
    }
}
```

```
public class Hebra1{
    private Sinc s;
    public Hebra1(Sinc s){
        this.s = s;
    }
    public void run(){
        while (true){
            s.preProt1();
            //SC1
            s.postProt1();
            //SNC1
        }
    }
}
```

```
public static void main(String[] args){
    Sinc s = new Sinc();
    Hebra0 h0 = new Hebra0(s); Hebra1 h1 = new Hebra1(s); h0.start(); h1.start()
}
```

Exclusión mutua con espera activa: primer intento

```
public class Sinc_PI{

    private volatile boolean f0 = false;
    private volatile boolean f1 = false;

    public void preProt0(){
        f0 = true;
        while (f1) Thread.yield();
    }

    public void postProt0(){
        f0 = false;
    }
    ....
}
```

```
....
public void preProt1(){
    f1 = true;
    while (f0) Thread.yield();
}

public void postProt1(){
    f1 = false;
}
}
```

- Cada hebra utiliza una variable booleana f0 y f1
- fi es true sii la hebra i quiere entrar en su SCi (i = 0,1)
- Antes de entrar en la SCi, la hebra i pone fi a true y, a continuación, mira a ver si la otra hebra quiere entrar y, si es así, se espera
- Cuando sale de la SCi, la hebra i pone de nuevo su variable booleana a false

Exclusión mutua con espera activa: primer intento

Requisito 1:

En cada momento, hay, a lo sumo, una hebra ejecutando su sección crítica

Esta solución satisface el Requisito 1

Para probarlo basta observar que

"el proceso h_i está ejecutando SC_i sii f_i es true"

Por inducción

1. Cuando h_0 entra en SC_0 , f_1 es false y, por lo tanto, h_1 no está en SC_1
2. Mientras que h_0 está en su SC_0 , f_0 es true, y, por lo tanto, h_1 no puede entrar en SC_1

Exclusión mutua con espera activa: primer intento

Requisito 1:

En cada momento, hay, a lo sumo, una hebra ejecutando su sección crítica

Sin embargo, esta solución no es válida porque los procesos pueden quedarse bloqueados en sus respectivas instrucciones de espera activa

Ejemplo:

```
h0: f0 = true
h1: f1 = true
h0: ¿f1 == true?
h1: ¿f0 == true?
.....
```

Esta situación se denomina **livelock**

Nuestra solución debe, por lo tanto, satisfacer algún requisito adicional

Requisito 2:

Ausencia de livelock. Si las dos hebras quieren entrar en sus SC simultáneamente, en algún momento, alguna de ellas, debería poder hacerlo

Exclusión mutua con espera activa: Segundo intento

Cambiamos el orden de las instrucciones en el preprotocolo para evitar el livelock

Esta solución no es correcta porque se viola la exclusión mutua

Ejemplo:

h0: ¿f1 == true? No
h1: ¿f0 == true? No
h0: f0 = true
h1: h1 = true
h0 está en SC0
h1 está en SC1
.....

ER

```
public class Sinc_SI{  
  
    private volatile boolean f0 = false;  
    private volatile boolean f1 = false;  
  
    public void preProt0(){  
        while (f1) Thread.yield();  
        f0 = true;  
    }  
  
    public void postProt0(){  
        f0 = false;  
    }  
    ....  
}
```

```
....  
public void preProt1(){  
    while (f0) Thread.yield();  
    f1 = true;  
}  
  
public void postProt1(){  
    f1 = false;  
}  
}
```

Exclusión mutua con espera activa: Tercer Intento

```
public class Sinc_TI{  
  
    private volatile int turno = 0;  
  
    public void preProt0(){  
        while (turno==1) Thread.yield();  
    }  
  
    public void postProt0(){  
        turno=1;  
    }  
    ....  
}
```

```
....  
public void preProt1(){  
    while (turno==0) Thread.yield();  
}  
  
public void postProt1(){  
    turno = 0;  
}  
}
```

- Para evitar el livelock, usamos una variable turno, que toma los valores 0 o 1
- Antes de entrar en su SC, cada hebra mira a ver si le toca, si no espera
- Cuando sale de la sección crítica le pasa el turno a la otra hebra
- Esta solución no es válida porque las hebras deberían poder entrar en cualquier momento en su SC si ésta no está ocupada

Requisito 3:

Si sólo una de las hebras quiere entrar en su SC,
en algún momento debería poder hacerlo

Exclusión mutua con espera activa: Solución de Peterson

```
public class Peterson{
```

```
    private volatile int turno = 0;  
    private volatile boolean f0 = false;  
    private volatile boolean f1 = false;
```

```
    public void preProt0(){
```

```
        f0 = true;  
        turno = 1;  
        while (f1 && turno==1 )  
            Thread.yield();  
    }
```

```
    public void postProt0(){  
        f0 = false;  
    }  
    ....
```

Es una combinación de los intentos 1 y 3. Cuando h0 quiere entrar en SC0 lo indica poniendo f0 a true y le pasa el turno a h1. h1 hace lo mismo.

```
    ....  
    public void preProt1(){
```

```
        f1 = true;  
        turno = 0;  
        while (f0 && turno==0 )  
            Thread.yield();  
    }
```

```
    public void postProt1(){  
        f1 = false;  
    }  
}
```

Exclusión mutua con espera activa: Solución de Peterson

```
public class Peterson{  
  
    private volatile int turno = 0;  
    private volatile boolean f0 = false;  
    private volatile boolean f1 = false;  
  
    public void preProt0(){  
        f0 = true;  
        turno = 1;  
        while (f1 && turno==1 )  
            Thread.yield();  
    }  
  
    public void postProt0(){  
        f0 = false;  
    }  
    ....  
}
```

Antes de entrar en SC0, h0 espera si h1 quiere entrar en SC1 y es su turno.
Cuando sale de SC0, h0 pone f0 a false.
h1 hace lo mismo.

```
....  
public void preProt1(){  
    f1 = true;  
    turno = 0;  
    while (f0 && turno==0 )  
        Thread.yield();  
}  
  
public void postProt1(){  
    f1 = false;  
}  
}
```

Exclusión mutua con espera activa: Solución de Peterson

```
public class Hebra0{  
  private Peterson s;  
  public Hebra0(Peterson s){  
    this.s = s;  
  }  
  public void run(){  
    while (true){  
      s.preProt0();  
      //SC0  
  
      s.posProt0();  
      //SNC0  
    }  
  }  
}
```

```
public class Peterson{  
  private volatile int turno = 0;  
  private volatile boolean f0 = false;  
  private volatile boolean f1 = false;  
  ...  
}
```

```
public void preProt0(){  
  f0 = true;  
  turno = 1;  
  while (f1 && turno==1 )  
    Thread.yield();  
}
```

```
public void postProt0(){  
  f0 = false;  
}
```

```
public void preProt1(){  
  f1 = true;  
  turno = 0;  
  while (f0 && turno==0 )  
    Thread.yield();  
}
```

```
public void postProt1(){  
  f1 = false;  
}
```

```
public class Hebra1{  
  private Peterson s;  
  public Hebra1(Peterson s){  
    this.s = s;  
  }  
  public void run(){  
    while (true){  
      s.preProt1();  
      //SC1  
  
      s.postProt1();  
      //SNC1  
    }  
  }  
}
```

Exclusión mutua con espera activa: Solución de Peterson

```
public class Hebra0{  
    private Peterson s;  
    public Hebra0(Peterson s){  
        this.s = s;  
    }  
    public void run(){  
        while (true){  
            s.preProt0();  
            //SC0  
  
            s.posProt0();  
            //SNC0  
        }  
    }  
}
```

```
public class Peterson{  
    private volatile int turno = 0;  
    private volatile boolean f0 = false;  
    private volatile boolean f1 = false;  
    ...  
}
```

```
public void preProt0(){  
    f0 = true;  
    turno = 1;  
    while (f1 && turno==1 )  
        Thread.yield();  
}
```

```
public void preProt1(){  
    f1 = true;  
    turno = 0;  
    while (f0 && turno==0 )  
        Thread.yield();  
}
```

```
public void postProt0(){  
    f0 = false;  
}
```

```
public void postProt1(){  
    f1 = false;  
}
```

```
public class Hebra1{  
    private Peterson s;  
    public Hebra1(Peterson s){  
        this.s = s;  
    }  
    public void run(){  
        while (true){  
            s.preProt1();  
            //SC1  
  
            s.postProt1();  
            //SNC1  
        }  
    }  
}
```

R1: En cada momento, hay, a lo sumo, un proceso ejecutando su sección crítica

Esta solución satisface R1. Se prueba como en el Primer Intento.

Exclusión mutua con espera activa: Solución de Peterson

```
public void run(){  
    while (true){  
        s.preProt0();  
        //SC0  
  
        s.posProt0();  
        //SNC0  
    }  
}
```

```
public void preProt0(){  
    f0 = true;  
    turno = 1;  
    while (f1 && turno==1 )  
        Thread.yield();  
}
```

```
public void postProt0(){  
    f0 = false;  
}
```

```
public void preProt1(){  
    f1 = true;  
    turno = 0;  
    while (f0 && turno==0 )  
        Thread.yield();  
}
```

```
public void postProt1(){  
    f1 = false;  
}
```

```
public void run(){  
    while (true){  
        s.preProt1();  
        //SC1  
  
        s.postProt1();  
        //SNC1  
    }  
}
```

R1: En cada momento, hay, a lo sumo, un proceso ejecutando su sección crítica

R2: Ausencia de livelock. Si las dos hebras quieren entrar en sus SC simultáneamente en algún momento, alguna de ellas, debería poder hacerlo

Esta solución satisface R2. Si `f0` y `f1` son `true`, la variable `turno` decide a quién le toca entrar.

Exclusión mutua con espera activa: Solución de Peterson

```
public void run(){  
    while (true){  
        s.preProt0();  
  
        //SC0
```

```
public void preProt0(){  
    f0 = true;  
    turno = 1;  
    while (f1 && turno==1 )  
        Thread.yield();  
}
```

```
public void preProt1(){  
    f1 = true;  
    turno = 0;  
    while (f0 && turno==0 )  
        Thread.yield();  
}
```

```
public void run(){  
    while (true){  
        s.preProt1();  
  
        //SC1
```

```
s.posProt0();  
  
//SNC0  
}  
}
```

```
public void postProt0(){  
    f0 = false;  
}
```

```
public void postProt1(){  
    f1 = false;  
}
```

```
s.postProt1();  
  
//SNC1  
}  
}
```

R3: Si sólo uno de los procesos quiere entrar en su sección crítica, en algún momento debería poder hacerlo.

Esta solución satisface R3.

Supongamos que h0 quiere entrar en SC0, y que h1 no quiere entrar en SC1. En este caso, como h1 no quiere entrar en SC1, f1 es falso, lo que significa que la expresión `f1 && (turno == 1)` es falsa, y, por lo tanto, h0 puede entrar en SC0

Exclusión mutua con espera activa: Solución de Peterson

```
public void run(){  
    while (true){  
        s.preProt0();  
  
        //SC0  
  
        s.posProt0();  
  
        //SNC0  
    }  
}
```

```
public void preProt0(){  
    f0 = true;  
    turno = 1;  
    while (f1 && turno==1 )  
        Thread.yield();  
}
```

```
public void postProt0(){  
    f0 = false;  
}
```

```
public void preProt1(){  
    f1 = true;  
    turno = 0;  
    while (f0 && turno==0 )  
        Thread.yield();  
}
```

```
public void postProt1(){  
    f1 = false;  
}
```

```
public void run(){  
    while (true){  
        s.preProt1();  
  
        //SC1  
  
        s.postProt1();  
  
        //SNC1  
    }  
}
```

La solución de Peterson satisface una propiedad adicional

R4: Justicia. Si ambos procesos quieren entrar simultáneamente, primero lo hace uno (h0, por ejemplo) y luego lo hace el otro (h1, en este caso).

Supongamos que h0 y h1 quieren entrar en SC0 y SC1 simultáneamente, entonces f0 y f1 son ambos true. Si, por ejemplo, turno == 0, entonces h0 entra en SC0, y h1 se queda esperando en el bucle de espera activa de su código. Supongamos que h0 sale de SC0 (f0 es false), y quiere volver a entrar, entonces al ejecutar su preprotocolo pone f0 a true, y turno a 1, por lo que él mismo se cierra el paso a SC0, siendo h1 el que puede continuar ahora ejecutando SC1.

Sincronización y el modelo de memoria de java (JMM)

- En el JMM, cada hebra tiene acceso a dos zonas de memoria:
 - La memoria de trabajo, local a la hebra
 - La memoria principal, compartida por todas las hebras.
- La memoria de trabajo se utiliza para optimizar el acceso a los datos y puede contener copias de los datos almacenados en la memoria principal, así como de los registros del procesador.
- La máquina virtual JVM transfiere datos entre la memoria principal y la memoria de trabajo en los siguientes casos:
 - La memoria de trabajo se invalida cuando la hebra accede a un método sincronizado explícitamente, (cuando adquiere el lock de un objeto);
 - La memoria de trabajo se vuelca sobre la memoria principal cuando la hebra libera el lock, es decir, antes de que el método o bloque sincronizado termine, las variables escritas durante la ejecución del método se transfieren a la memoria principal.

Sincronización y el modelo de memoria de java (JMM)

Si no declaramos las variables como volátiles...

```
public class Hebra0{
    private Peterson s;
    public Hebra0(Peterson s){
        this.s = s;
    }
    public void run(){
        while (true){
            s.preProt0();
            //SC0

            s.posProt0();
            //SNC0
        }
    }
}
```

```
public class Peterson{
    private volatile int turno = 0;
    private volatile boolean f0 = false;
    private volatile boolean f1 = false;
    ...
}
```

```
public void preProt0(){
    f0 = true;
    turno = 1;
    while (f1 && turno==1 )
        Thread.yield();
}
```

```
public void preProt1(){
    f1 = true;
    turno = 0;
    while (f0 && turno==0 )
        Thread.yield();
}
```

```
public void postProt0(){
    f0 = false;
}
```

```
public void postProt1(){
    f1 = false;
}
```

```
public class Hebra1{
    private Peterson s;
    public Hebra1(Peterson s){
        this.s = s;
    }
    public void run(){
        while (true){
            s.preProt1();
            //SC1

            s.postProt1();
            //SNC1
        }
    }
}
```

...no hay nada que nos garantice que h1 detecte que f0 es true, porque cada hebra puede tener distintas copias de la variable f0.

Por lo tanto, la exclusión mutua podría violarse en alguna implementación de Java

Variables volátiles

- Java permite definir atributos volátiles (**volatile**)
 - La especificación del lenguaje java obliga a que
 - Los campos volátiles no se almacenen nunca en la memoria local
 - Todas las lecturas y escrituras se realizan sobre la memoria principal
 - Las operaciones sobre los campos volátiles deben hacerse exactamente en el orden en que la hebra los tiene definidos

```
private volatile int turno = 0;  
private volatile boolean f0 = false;  
private volatile boolean f1 = false;
```

Exclusión mutua con espera activa: Solución de Dekker

Utiliza los mismos recursos que la solución de Peterson

```
public class Dekker{

    private volatile int turno = 0;
    private volatile boolean f0 = false;
    private volatile boolean f1 = false;

    public void preProt0(){
        f0 = true;
        while (f1){
            if (turno == 1){
                f0 = false;
                while (turno == 1) Thread.yield();
                f0 = true;
            }
        }
    }

    public void postProt0(){
        turno = 1;
        f0 = false;
    }
    ....
}
```

```
....
public void preProt1(){
    f1 = true;
    while (f0){
        if (turno == 0){
            f1 = false;
            while (turno == 0) Thread.yield();
            f1 = true;
        }
    }
}

public void postProt1(){
    turno = 0;
    f1 = false;
}
}
```

Exclusión mutua con espera activa: Solución de Dekker

```
public class Dekker{

    private volatile int turno = 0;
    private volatile boolean f0 = false;
    private volatile boolean f1 = false;

    public void preProt0(){
        f0 = true;
        while (f1){
            if (turno == 1){
                f0 = false;
                while (turno == 1) Thread.yield();
                f0 = true;
            }
        }
    }

    public void postProt0(){
        turno = 1;
        f0 = false;
    }
    ....
}
```

```
....
public void preProt1(){
    f1 = true;
    while (f0){
        if (turno == 0){
            f1 = false;
            while (turno == 0) Thread.yield();
            f1 = true;
        }
    }
}

public void postProt1(){
    f1 = false;
}
}
```

Satisface R1, R2, y R3, pero no R4

R4:-Justicia. Si ambos procesos quisieran entrar simultáneamente, primero lo hace uno (h0, por ejemplo) y luego el otro (h1, en este caso).

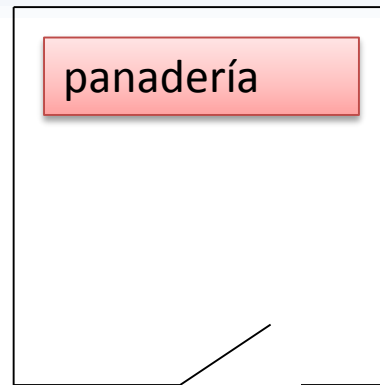


El algoritmo de la panadería (Lamport)

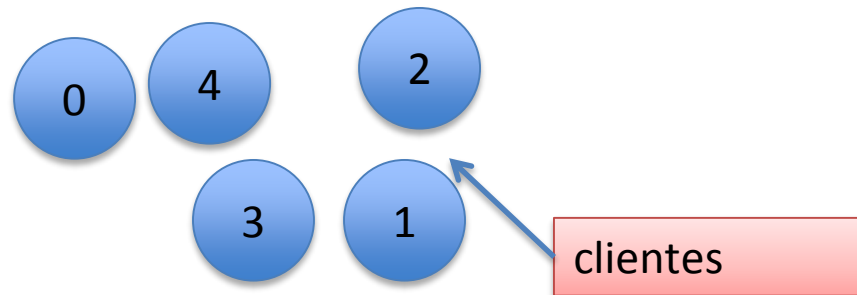
- El algoritmo de la panadería resuelve el problema de la exclusión mutua para $N \geq 2$ procesos, utilizando **espera activa** como mecanismo de sincronización
- No es un algoritmo justo porque no trata a todos los procesos del mismo modo
- Enunciado:
 - Se supone que tenemos N clientes que desean ser atendidos en una panadería. El dependiente representa el **recurso compartido** que debe ser utilizado en **exclusión mutua** por todos los clientes (no se puede atender a dos o más clientes simultáneamente)

El algoritmo de la panadería (Lamport)

Ilustración



En la panadería se atiende a los clientes en exclusión mutua. En la ilustración, la exclusión mutua se representa con un local pequeño en el que no cabe más de un cliente



Estructura del Código: Procesos

```
public class Panaderia {  
    public Panaderia(int N){  
        ....  
    }  
    ....  
}
```

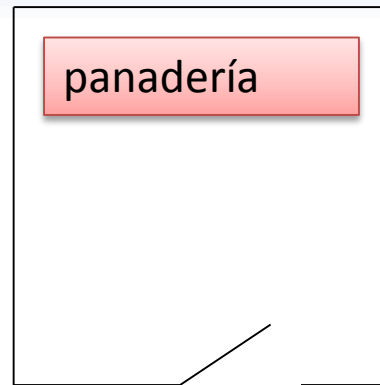
- N (=15) clientes llegan a la panadería y son atendidos en exclusión mutua.
- La panadería es el recurso compartido, es decir, una entidad pasiva, que no hace falta implementar como hebra.

```
class Cliente extends Thread{  
    private int id;  
    private Panaderia pan;  
    public Cliente(int id,Panaderia pan){  
        this.id = id; this.pan = pan;  
    }  
    public void run(){  
        /*Preprotocolo*/  
        // el cliente id es atendido por el dependiente  
        /*Posprotocolo*/  
        // el cliente id sale de la panadería  
    }  
}
```

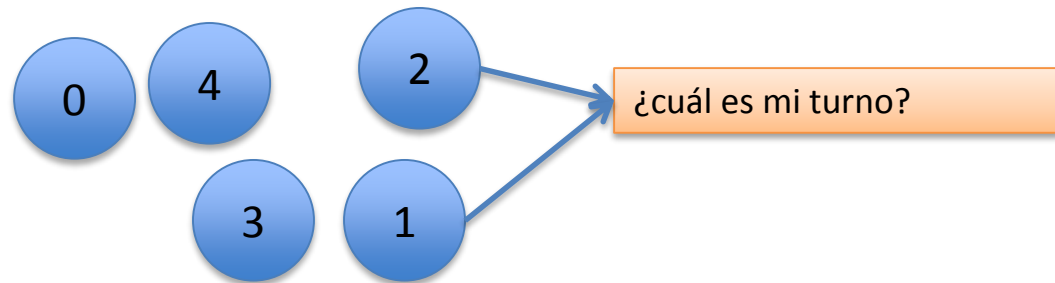
```
public static void main(String[] args){  
    int N= 15;  
    Panaderia pan = new Panaderia(N);  
    Cliente[] c = new Cliente[N];  
    for (int i = 0; i<N; i++){  
        c[i] = new Cliente(i,pan);  
    }  
    for (int i = 0; i<N; i++){  
        c[i].start();  
    }  
}
```

El algoritmo de la panadería (Lamport)

Ilustración




Para modelar la exclusión mutua los clientes van pidiendo su turno cuando llegan a la panadería, para entrar de forma ordenada



Estructura del Código: turno

```
public class Panaderia {  
    private int[] turno;  
    public Panaderia(int N){  
        turno = new int[N];  
    }  
    .....  
}
```



- Para modelar el turno, usamos un array con N componentes (una por hebra)
- Para cada hebra *id*, *turno[id] == 0* sii no quiere acceder a su sección crítica. Por eso, inicialmente todos los turnos están a 0, que es como decir que todos los clientes están fuera de la panadería

```
class Cliente extends Thread{  
    private int id;  
    private Panaderia pan;  
    public Cliente(int id,Panaderia pan){  
        this.id = id; this.pan = pan;  
    }  
    public void run(){  
        /*Preprotocolo*/  
        // el cliente id es atendido por el dependiente  
        /*Posprotocolo*/  
        // el cliente id sale de la panadería  
    }  
}
```

Estructura del Código: Siguiente

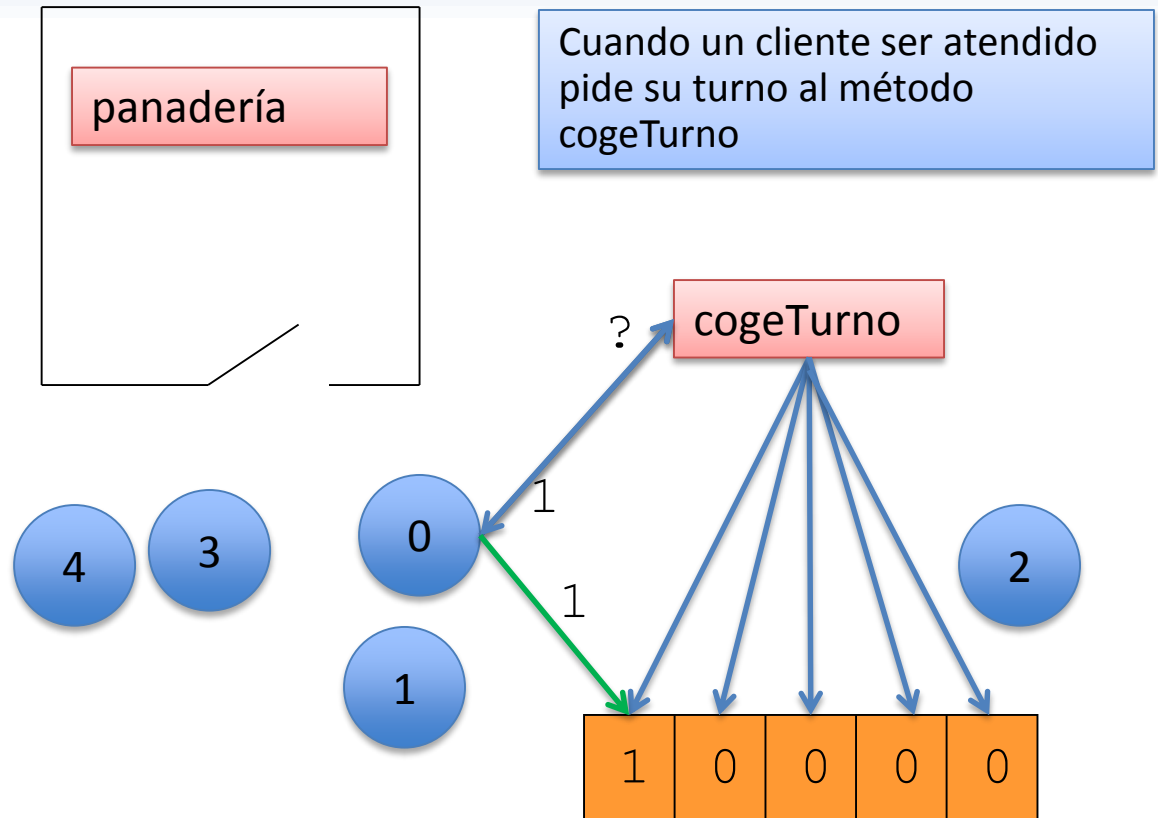
```
public class Panaderia {  
    private int[] turno;  
    public Panaderia(int N){  
        turno = new int[N];  
    }  
  
    public void cogeTurno(int id){  
        int max = 0;  
        for (int i = 0; i < turno.length; i++)  
            if (max < turno[i]) max = turno[i];  
        turno[id] = max + 1;  
    }  
}
```

- Cada cliente pide su turno, utilizando el método *cogeTurno*, que itera por el array *turno*, y le asigna el mayor valor encontrado más 1.

```
class Cliente extends Thread{  
    private int id;  
    private Panaderia pan;  
    public Cliente(int id, Panaderia pan){  
        this.id = id; this.pan = pan;  
    }  
    public void run(){  
        pan.cogeTurno(id);  
        // el cliente id es atendido por el dependiente  
        /*Posprotocolo*/  
        // el cliente id sale de la panadería  
    }  
}
```

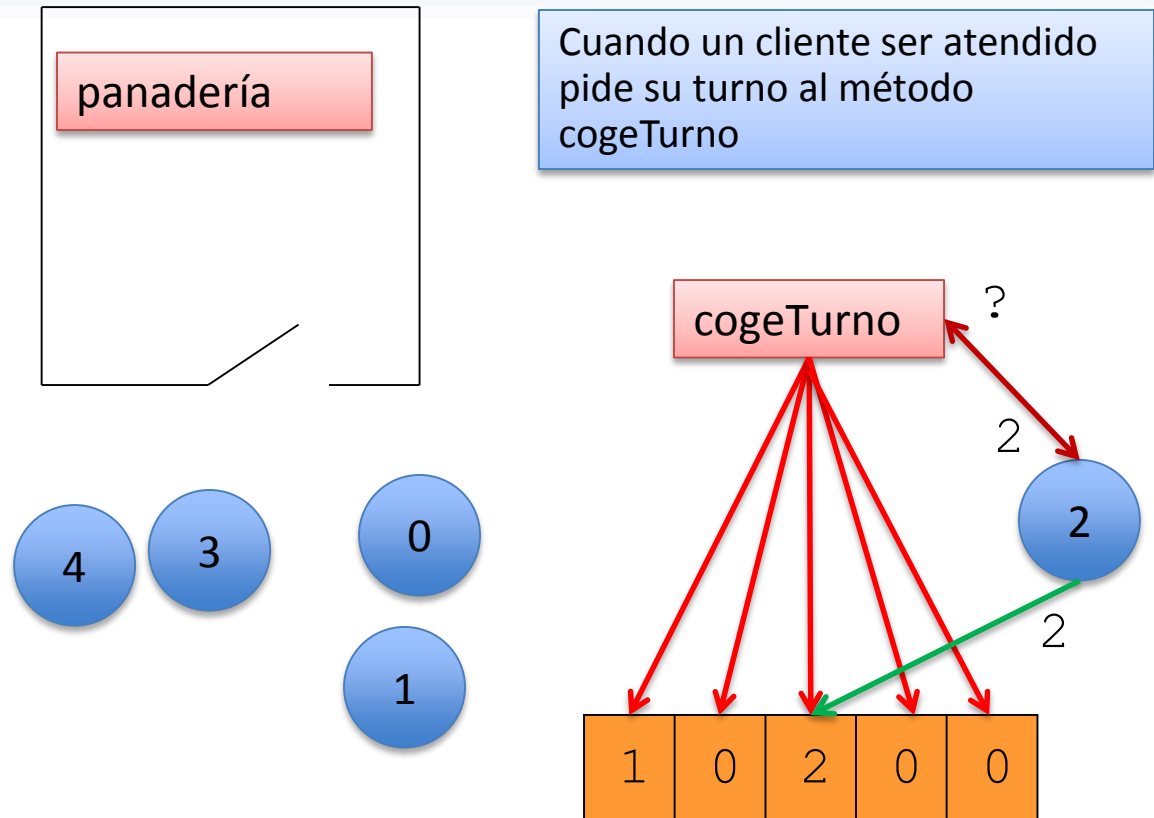
El algoritmo de la panadería (Lamport)

Ilustración



El algoritmo de la panadería (Lamport)

Ilustración



Estructura del Código: esperoTurno

```
public class Panaderia {  
    private int[] turno;  
    public Panaderia(int N){  
        turno = new int[N];  
    }  
    public void cogeTurno(int id){  
        ....  
    }  
  
    public void esperoTurno(int id){  
  
    }  
  
}
```

Una vez que tiene su turno, cada cliente espera hasta que le toca..

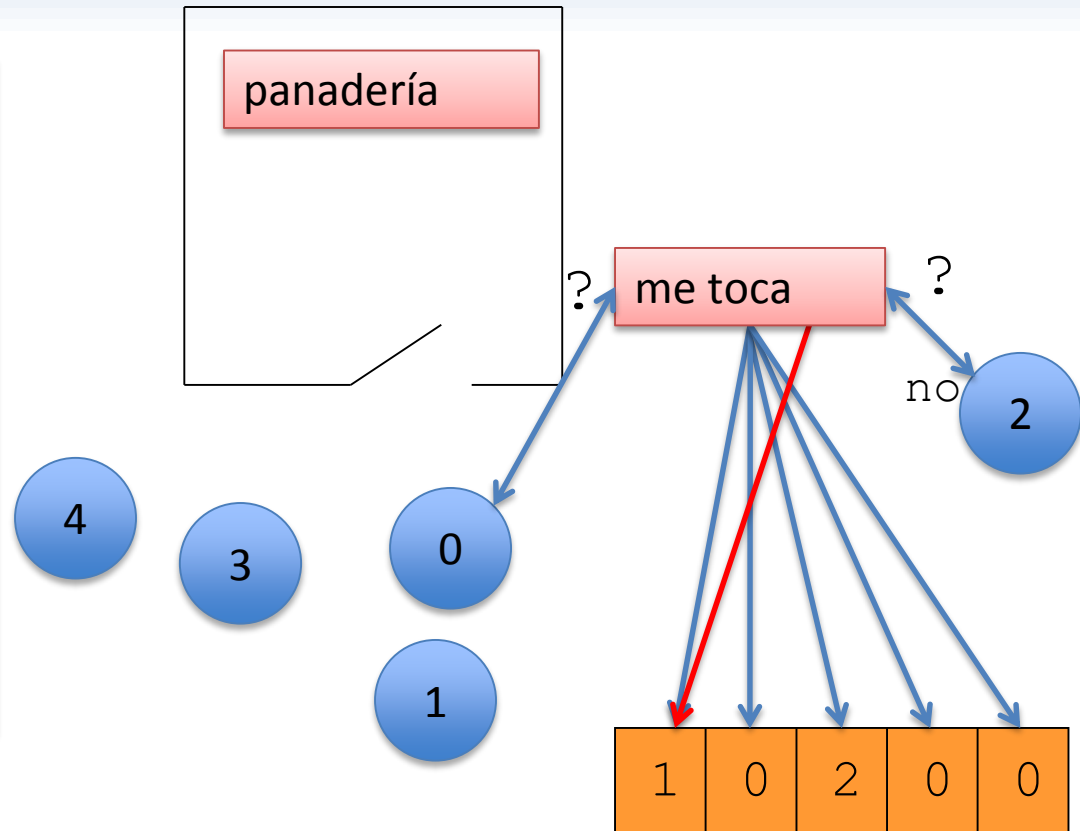
```
class Cliente extends Thread{  
    private int id;  
    private Panaderia pan;  
    public Cliente(int id,Panaderia pan){  
        this.id = id; this.pan = pan;  
    }  
    public void run(){  
        pan.cogeTurno(id);  
        pan.esperoTurno(id);  
        // el cliente id es atendido por el dependiente  
        /*Posprotocolo*/  
        // el cliente id sale de la panadería  
    }  
}
```

El algoritmo de la panadería (Lamport)

Ilustración

Para ver si es su turno, cada cliente comprueba si va antes o después que el resto de los clientes, utilizando el método **meToca**.

Dados dos clientes id y i , **meToca**(id,i) comprueba si el turno de id es anterior al de i . Si una hebra ve que a otra le toca antes que a ella espera



Estructura del Código: meToca

```
public class Panaderia {  
    public Panaderia(int N){... }  
    public void cogeTurno(int id){ ..... }  
  
    private boolean meToca(int id,int i){  
        // devuelve true si el turno de id es anterior al de i  
        if (turno[i] > 0 && turno[i] < turno[id])  
            return false;  
        else if (turno[i] == turno[id] && i < id)  
            return false;  
        else  
            return true;  
    }  
  
    public void esperoTurno(int id){  
        for (int i = 0; i < turno.length; i++)  
            while (!meToca(id,i)) Thread.yield();  
    }  
}
```

- La función *meToca* comprueba qué proceso va antes.
- El bucle de espera activa hace que un cliente espere si hay otro cliente que debe ejecutar su sección crítica antes.
- Cuando el cliente id va delante del resto, puede ejecutar su sección crítica

Estructura del Código: salePanadería

```
public class Panaderia {
    private int[] turno;
    public Panaderia(int N){... }
    public void cogeTurno(int id){ ..... }

    private boolean meToca(int id,int i){
        // devuelve true si el turno de id es anterior al de i
        if (turno[i] > 0 && turno[i] < turno[id])
            return false;
        else if (turno[i] == turno[id] && i < id)
            return false;
        else
            return true;
    }

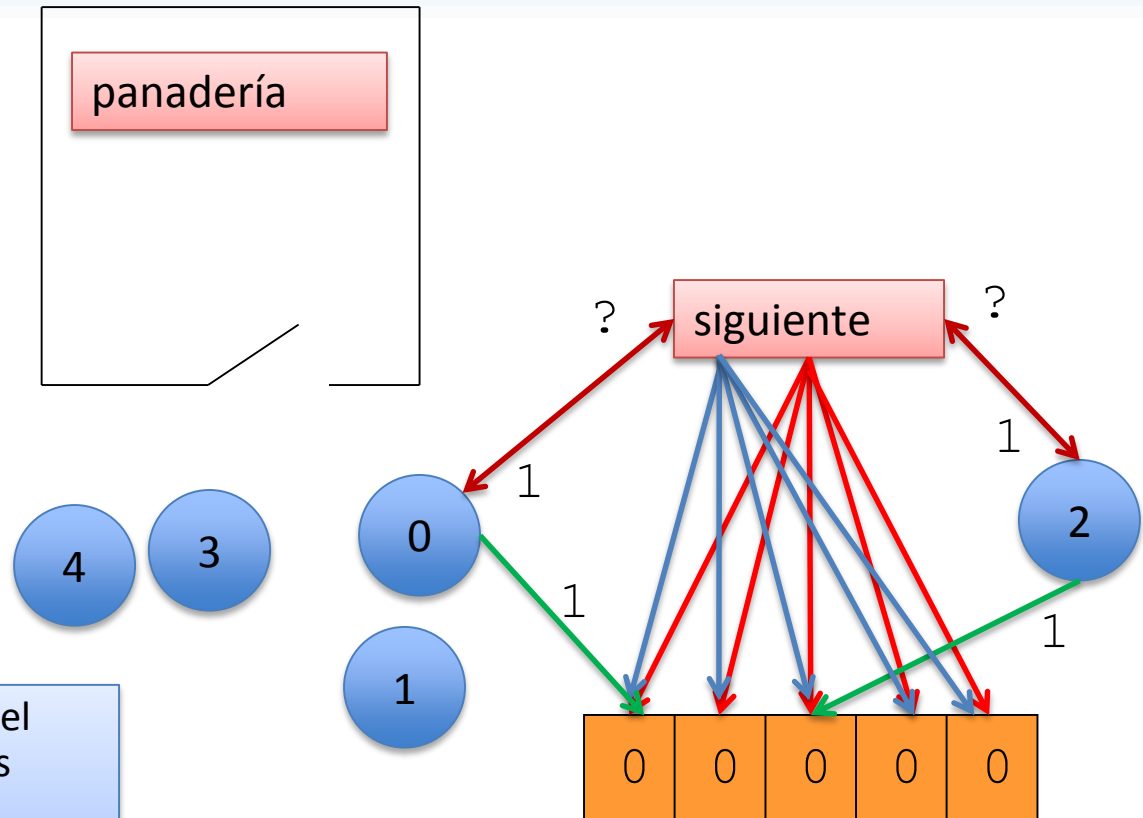
    public void esperoTurno(int id){
        for (int i = 0; i<turno.length; i++)
            while (!meToca(id,i)) Thread.yield();
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

- Cuando termina la sección crítica pone su turno a 0.

```
class Cliente extends Thread{
    private int id;
    private Panaderia pan;
    public Cliente(int id,Panaderia pan){
        this.id = id; this.pan = pan;
    }
    public void run(){
        pan.cogeTurno(id);
        pan.esperaTurno(id);
        // el cliente id es atendido por el dependiente
        pan.salePanaderia(id);
        // el cliente id sale de la panadería
    }
}
```


El algoritmo de la panadería (Lamport)

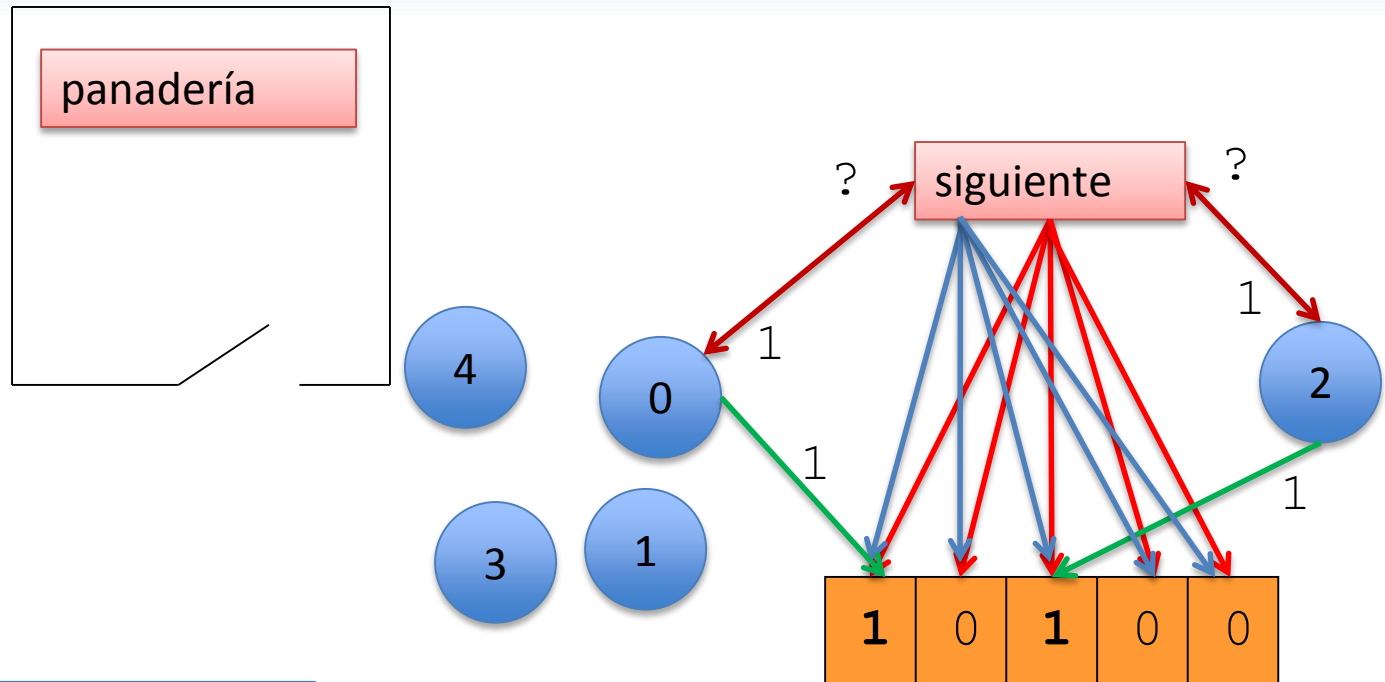
Ilustración



Es posible que siguiente dé el mismo turno a dos procesos distintos, puesto que **no** se ejecuta en **EXCLUSIÓN MUTUA**

El algoritmo de la panadería (Lamport)

Ilustración



Es posible que siguiente dé el mismo turno a dos procesos distintos, puesto que **no** se ejecuta en **EXCLUSIÓN MUTUA**

Las dos iteraciones del método siguiente se intercalan

Estructura del Código: meToca

```
public class Panaderia {
    private int[] turno;
    public Panaderia(int N){... }
    public void cogeTurno(int id){ ..... }

    private boolean meToca(int id,int i){
        // devuelve true si el turno de id es anterior al de i
        if (turno[i] > 0 && turno[i] < turno[id])
            return false;
        else if (turno[i] == turno[id] && i < id)
            return false;
        else
            return true;
    }

    public void esperoTurno(int id){
        for (int i = 0; i < turno.length; i++)
            while (!meToca(id,i)) Thread.yield();
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

- La función *meToca* le da prioridad a la hebra 0 porque tiene menor identificador,
- pero aún no hemos terminado...

Estructura del Código: Traza de error

```
public class Panaderia {

    private int[] turno;
    public Panaderia(int numC){... }
    public void cogeTurno(int id){
        int max = 0;
        for (int i = 0; i < turno.length; i++)
            if (max < turno[i]) max = turno[i];
        turno[id] = max + 1;
    }
    private boolean meToca(int id, int i){
        // devuelve true si el turno de id es anterior al de i
        if (turno[i] > 0 && turno[i] < turno[id])
            return false;
        else if (turno[i] == turno[id] && i < id)
            return false;
        else return true;
    }
    public void esperoTurno(int id){
        for (int i = 0; i < turno.length; i++)
            while (!meToca(id, i)) Thread.yield();
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

```
class Cliente extends Thread{
    private int id;
    private Panaderia pan;
    public Cliente(int id, Panaderia pan){
        this.id = id; this.pan = pan;
    }
    public void run(){
        pan.cogeTurno(id);
        pan.esperaTurno(id);
        // el cliente id es atendido por el dependiente
        pan.salePanaderia(id);
        // el cliente id sale de la panadería
    }
}
```

Instrucción	turno					Acción
Inicialmente	0	0	0	0	0	
c[0] llama a cogeTurno y ejecuta hasta *	0	0	0	0	0	
c[2] llama a cogeTurno y ejecuta hasta **	0	0	1	0	0	
c[2] llama a meToca(2,0), ... meToca(2,4) y entra en su SC	0	0	1	0	0	c[2] en SC2

Estructura del Código: Traza de error

```

public class Panaderia {
    private int[] turno;
    public Panaderia(int N){
        turno = new int[N];
    }
    public void cogeTurno(int id){
        int max = 0;
        for (int i = 0; i < turno.length; i++)
            if (max < turno[i]) max = turno[i];
        turno[id] = max + 1;
    }
    private boolean meToca(int id, int i){
        // devuelve true si el turno de id es anterior al de i
        if (turno[i] > 0 && turno[i] < turno[id])
            return false;
        else if (turno[i] == turno[id] && i < id)
            return false;
        else return true;
    }
    public void esperoTurno(int id){
        for (int i = 0; i < turno.length; i++)
            while (!meToca(id, i)) Thread.yield();
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
    
```

```

class Cliente extends Thread{
    private int id;
    private Panaderia pan;
    public Cliente(int id, Panaderia pan){
        this.id = id; this.pan = pan;
    }
    public void run(){
        pan.cogeTurno(id);
        pan.esperaTurno(id);
        // el cliente id es atendido por el dependiente
        pan.salePanaderia(id);
        // el cliente id sale de la panadería
    }
}
    
```

**

*

Instrucción	turno					Acción
c[2] llama a meToca(2,0), ... meToca(2,4) y entra en su SC	0	0	1	0	0	c[2] en SC2
C[0] termina cogeTurno, almacena 1 en turno[0]	1	0	1	0	0	
c[0] llama a meToca(0,0), meToca(0,1), meToca(0,2) ... y entra en su SC	1	0	1	0	0	c[0] en SC0 ERROR

Estructura del Código: pidiendoTurno

```
public class Panaderia {
    private int[] turno;
    private boolean[] pidiendoTurno;
    public Panaderia(int N){
        turno = new int[N];
        pidiendoTurno = new boolean[N]
    }
    public void cogeTurno(int id){
        pidiendoTurno[id] = true;
        int max = 0;
        for (int i = 0; i<turno.length; i++){
            if (max<turno[i]) max=turno[i];
            turno[id] = max + 1;
            pidiendoTurno[id] = false;
        }
    }
    private boolean meToca(int id,int i){
        ....
    }
    public void esperoTurno(int id){
        for (int i = 0; i<N; i++){
            while (pidiedoTurno[i])Thread.yield();
            while (!meToca(id,i)) Thread.yield();
        }
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

- Declaramos un array *pidiendoTurno* que guarda en cada momento si un proceso está escogiendo su turno o no.
- Inicialmente, todas sus componente están a *false*.
- Cada proceso indica, modificando este array, si está cogiendo su turno
- Antes de comprobar si otro proceso va antes o después que él, espera hasta que ha terminado de escoger su turno.

Panadería: Código definitivo

```
public class Panaderia {
    private int[] turno;
    private boolean[] pidiendoTurno;
    public Panaderia(int N){
        turno = new int[N];
        pidiendoTurno = new boolean[N]
    }
    public void cogeTurno(int id){
        pidiendoTurno[id] = true;
        int max = 0;
        for (int i = 0; i < turno.length; i++)
            if (max < turno[i]) max = turno[i];
        turno[id] = max + 1;
        pidiendoTurno[id] = false;
    }
    private boolean meToca(int id, int i){
        if (turno[i] > 0 && turno[i] < turno[id]) return false;
        else if (turno[i] == turno[id] && i < id) return false;
        else return true;
    }
    public void esperoTurno(int id){
        for (int i = 0; i < N; i++){
            while (pidiendoTurno[i]) Thread.yield();
            while (!meToca(id, i)) Thread.yield();
        }
    }
    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

Corrección de un programa concurrente

- **Propiedades de seguridad:** las que afirman que el sistema “nunca” va a entrar en un estado “malo” o de error
 - Exclusión mutua
 - Condiciones de sincronización para el productor/consumidor
 - Ausencia de bloqueo (deadlock)
 - Deadlock es el estado del sistema en el que todos los procesos están bloqueados esperando algún evento.
- **Propiedades de viveza:** las que afirman que “en algún momento” ocurre algo “bueno” en el sistema
 - R3: Si sólo una hebra quiere entrar en su SC, en algún momento debe poder hacerlo
 - R2: Si las dos hebras quieren entrar en su sección crítica, en algún momento alguna de ellas debe poder hacerlo (ausencia de livelock)
 - R4: Ausencia de posposición indefinida (starvation): todos los procesos del sistema tienen la oportunidad de evolucionar en su código

Justicia (fairness)

- La **justicia del planificador** afecta algunas propiedades del sistema (las de viveza)
- **Planificador justo**: aquél que asegura que cualquier proceso que está en estado listo es alguna vez seleccionado para continuar con su ejecución.

```
public class Justicia1;
    private static boolean fin1, fin2;
    public static Uno extends Thread{
        public void run(){
            fin1 = true;
            while (!fin2) Thread.yield();
        }
    }
    ....
```

```
...
    public static Dos extends Thread{
        public void run(){
            while (!fin1) Thread.yield();
            fin2 = true;
        }
    }
```

```
public static void main(String[] args){
    Uno uno = new Uno();
    Dos dos = new Dos();
    uno.start(); dos.start();
}
```

Con un planificador justo este programa terminaría siempre.

Justicia

- **Planificador débilmente justo:** aquél que asegura que si un proceso hace una petición de forma continua, en algún momento será atendida.
- **Planificador fuertemente justo:** aquél que asegura que si un proceso hace una petición con infinita frecuencia, en algún momento será atendida.

```
public class Justicia2;  
    private static boolean fin1, fin2;  
    public static Uno extends Thread{  
        public void run(){  
            while (!fin2) {  
                fin1 = true;  
                fin1 = false;  
            }  
        }  
    }  
    ....
```

```
...  
    public static Dos extends Thread{  
        public void run(){  
            while (!fin1) Thread.yield();  
            fin2 = true;  
        }  
    }
```

Con un planificador **débilmente justo** este programa podría no terminar.
Con un planificador **fuertemente justo**, siempre termina.

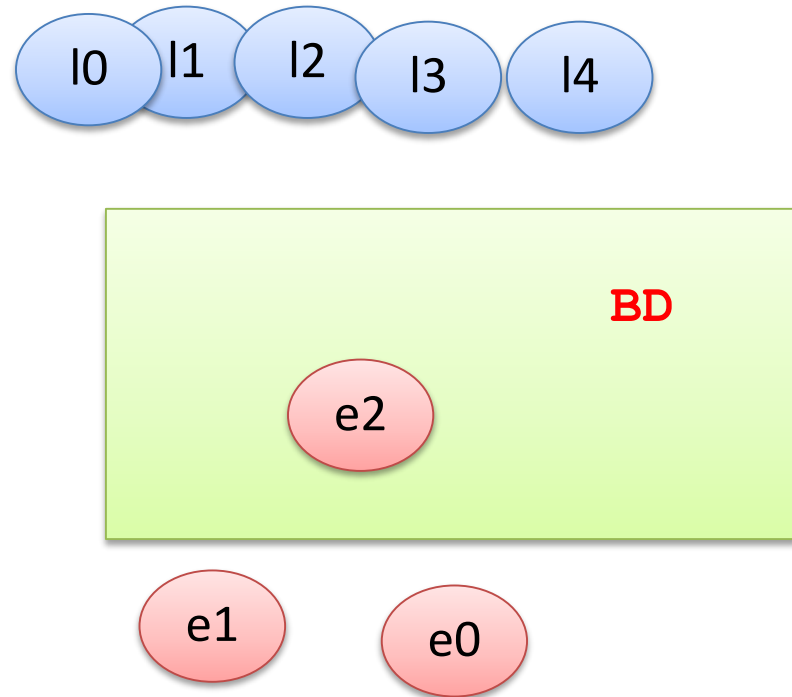
Lectores/Escritores

- El problema de los lectores/escritores representa un modelo de sincronización entre dos tipos de procesos (los lectores y los escritores) que acceden a un recurso compartido, típicamente una base de datos (BD).
- Los procesos escritores acceden a la BD para actualizarla.
- Los procesos lectores leen los registros de la BD.

Lectores/Escritores

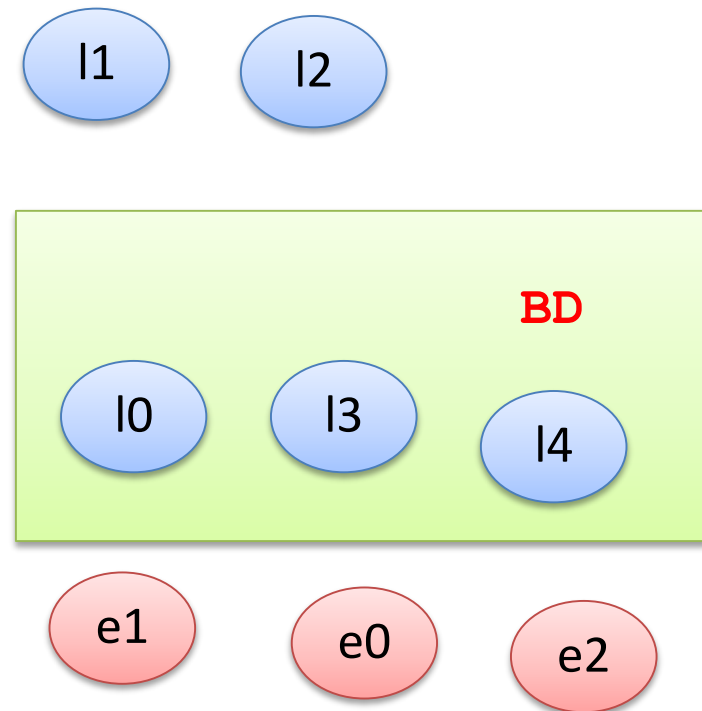
- Condición de sincronización para los escritores:

- Acceden a la BD en **exclusión mutua** con cualquier otro proceso de tipo lector o escritor



Lectores/Escritores

- Condición de sincronización para los lectores:
 - **Cualquier número** de lectores puede acceder simultáneamente a la BD.



Lectores/Escritores: Código incompleto

```
class Lector extends Thread{
    private int id;
    private GestorBD ge;
    public Lector(int id,GestorBD g){
        this.g = g;
        this.id = id; start();
    }
    public void run(){
        while (true){
            g.entraLector(id);
            //lector id en la BD
            g.saleLector(id);
        }
    }
}
```

```
class GestorBD {
    public void entraLector(int id){...}
    public void entraEscritor(int id){...}
    public void saleLector(int id){...}
    public void saleEscritor(int id){...}
}
```

```
class Escritor extends Thread{
    private int id;
    private GestorBD g;
    public Escritor(int id,GestorBD g){
        this.g = g;
        this.id = id; start();
    }
    public void run(){
        while (true){
            g.entraEscritor(id);
            //escritor id en la BD
            g.saleEscritor(id);
        }
    }
}
```

```
public static void main(String[] args){
    GestorBD gestor = new GestorBD();
    Lector[] lec = new Lectores[NL];
    Escritor[] esc = new Escritor[NE];
    for (int i = 0; i<NL; i++)
        lec[i] = new Lector(i,gestor);
    for (int i = 0; i<NE; i++)
        esc[i] = new Escritor(i,gestor);
}
```

La BD no hace falta modelarla

Todos los lectores ejecutan los mismos protocolos de entrada y salida

Todos los escritores ejecutan los mismos protocolos de entrada y salida

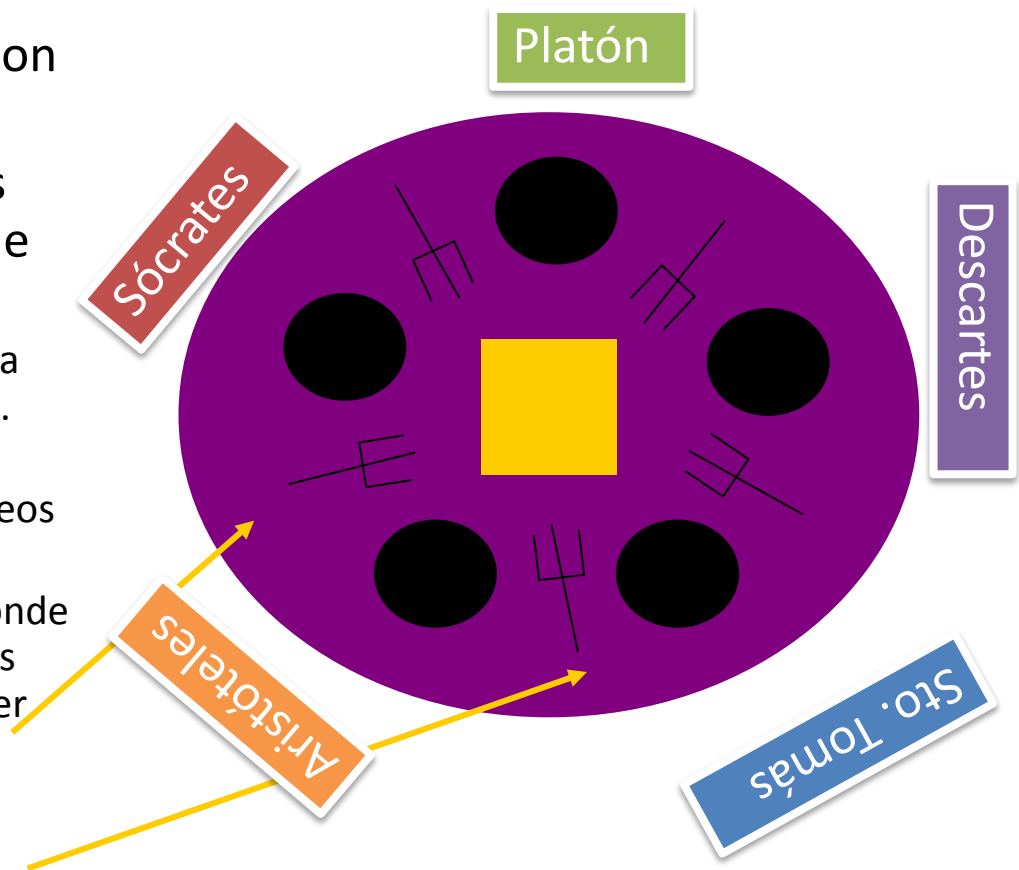
El problema de los filósofos

- $N = 5$ procesos filósofos dedican su vida a dos únicas tareas:
 - pensar, la mayor parte del tiempo
 - comer, de vez en cuando

```
public class Filosofo extends Thread{  
  
    public void run(){  
        while (true){  
            //pensar  
            //comer  
        }  
    }  
}
```

El problema de los filósofos

- La tarea “pensar” representa la actividad que cada proceso puede hacer sin necesidad de sincronizarse ni comunicarse con los demás.
- Sin embargo, para “comer” los filósofos tienen que ponerse de acuerdo:
 - En el comedor hay una mesa en la que cada filósofo tiene su puesto.
 - En el centro de la mesa hay una cantidad ilimitada de comida (fideos chinos, espaguetis,...)
 - Adyacentes al plato que corresponde a cada filósofo, hay dos tenedores que el filósofo necesita para poder comer



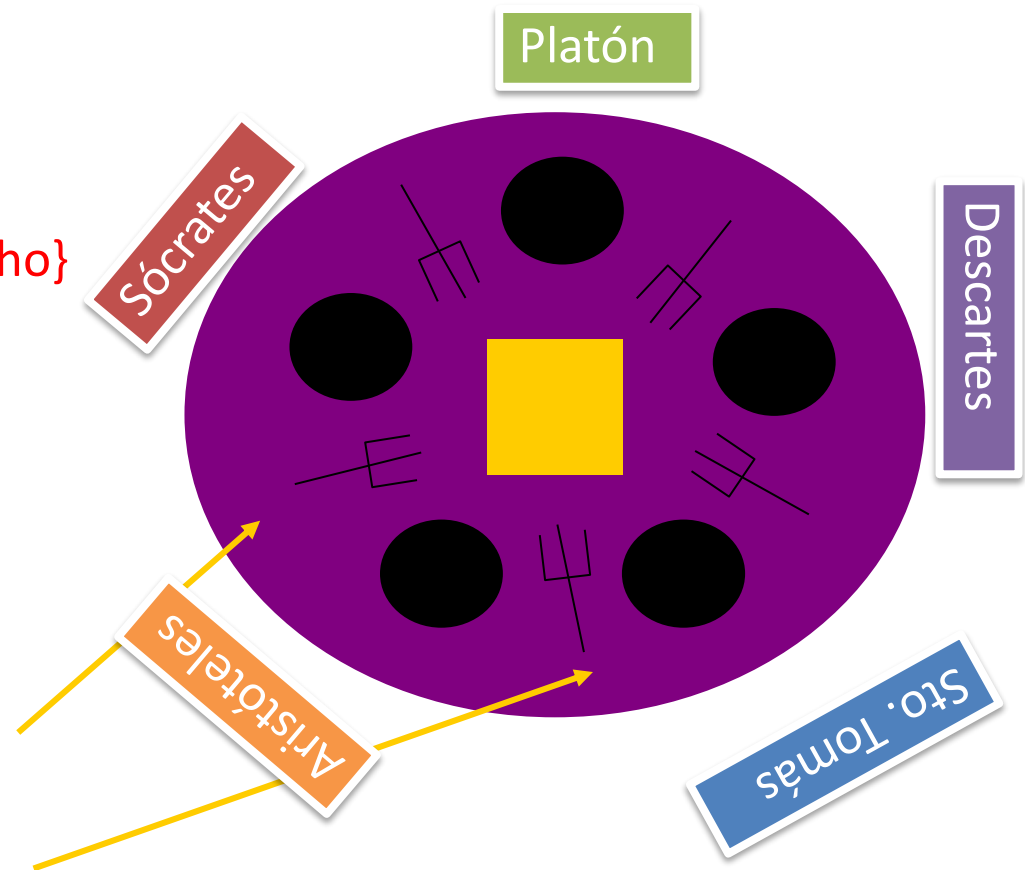
El problema de los filósofos

- Así que el código para comer para cada filósofo es:

{coge tenedores izdo y dcho}

{come}

{devuelve tenedores izdo y dcho}



El problema de los filósofos

- Así que el código para comer para cada filósofo es:

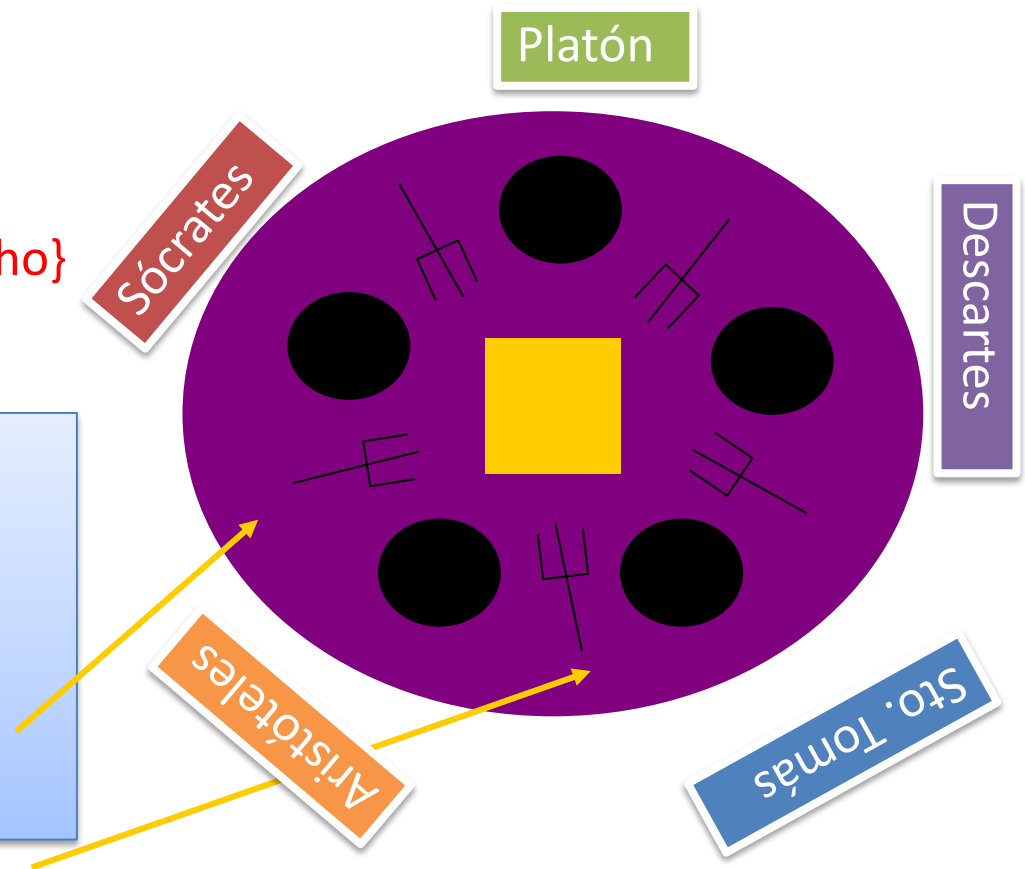
{coge tenedores izdo y dcho}

{come}

{devuelve tenedores izdo y dcho}

Este sistema nos sirve para representar

- La exclusión mutua
- Condiciones de sincronización
- Deadlock
- PostPosición Indefinida
-



Referencias

- Concurrency: State Models & Java Programs
Jeff Magee, Jeff Kramer, Ed. Willey
- Concurrent Programming
Alan Burns, Geoff Davies, Ed. Addison Wesley