

# 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

- Procesos vs. Recursos
- Problema de los Jardines (Exclusión Mutua)
- Sincronización y modelo de memoria de Java. Variables volátiles
- Problema del Productor/Consumidor (Condiciones de Sincronización)
- Solución al problema del productor consumidor con espera activa
- Exclusión mutua con espera activa para 2 procesos
  - Solución de Peterson
  - Solución de Dekker
- Exclusión mutua con espera activa para  $N \geq 2$  - Algoritmo de la Panadería (Lamport)
- Corrección de un programa concurrente
- Justicia
- El problema de los lectores/escritores
- El problema de los filósofos

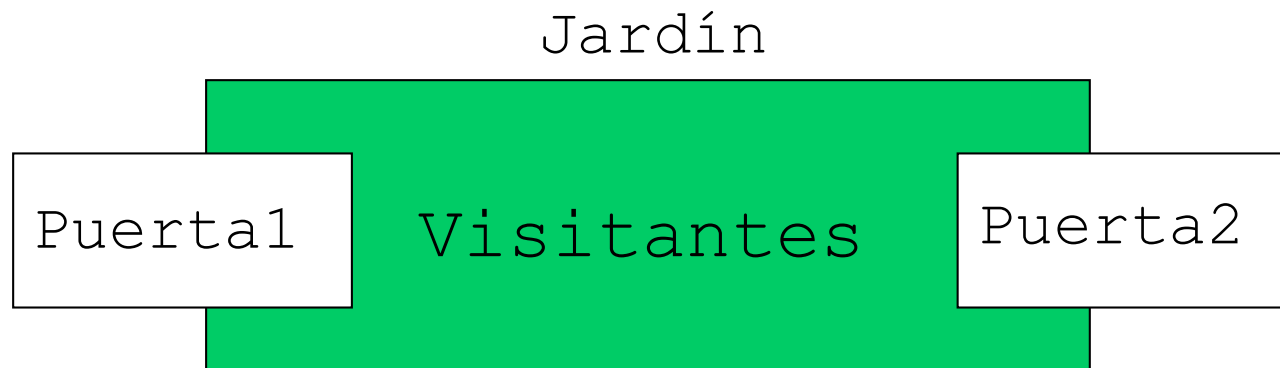
# Procesos vs Recursos

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. Los procesos pueden compartir los recursos:
  - **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

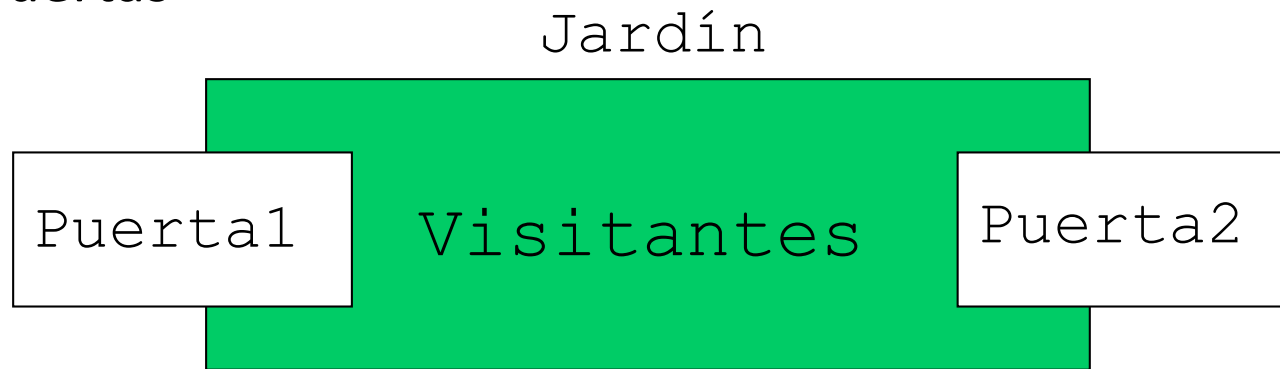
# El Problema de los Jardines

- Supón que hay un jardín 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 representa en cada momento el número total de personas que han entrado por alguna de las dos puertas

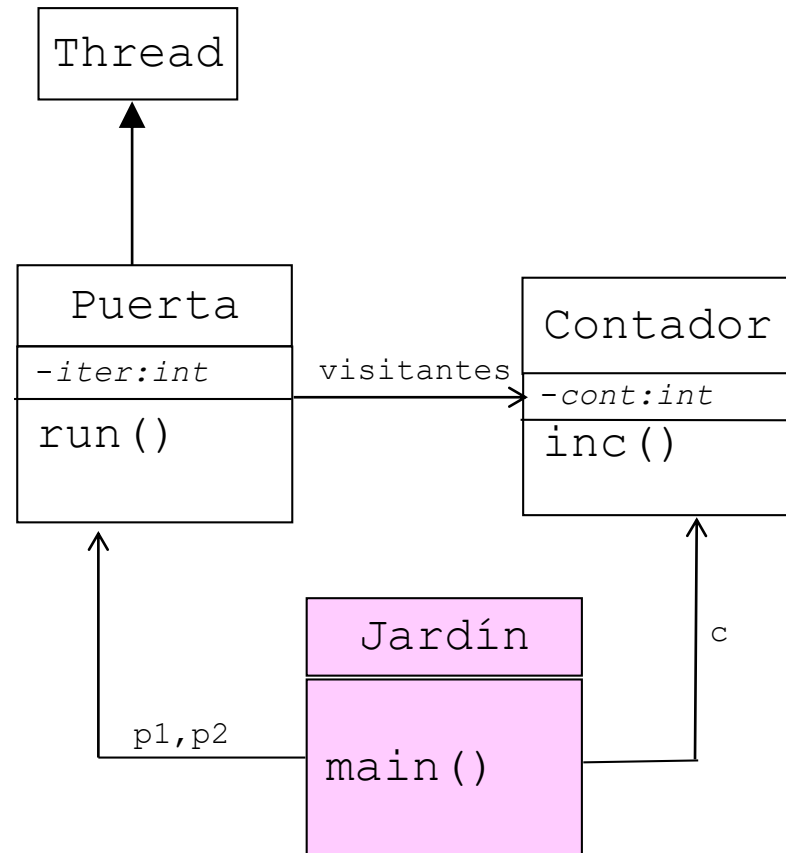


# El Problema de los Jardines

Una posible implementación en Java

- El **Jardín** está compuesto de dos objetos Puerta que comparten un objeto Contador:
- **Contador** almacena el número total de visitantes. El método `inc()` incrementar en 1 el contador de visitantes
- **Puerta** es una hebra que modela la llegada periódica de visitantes:
  - En el método `run`, cada cierto tiempo llama al método `inc()` del objeto **Contador**

Diagrama de clases



# El Problema de los Jardines

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

Creación del  
recurso compartido

Creación de los  
objetos p1 y p2

Comienzo ejecución  
de las hebras p1 y p2

La hebra main espera a  
que termine la ejecución  
de p1 y p2

```
public class Jardines {  
    public static void main(String[] args){  
        Contador visitantes = new Contador(); //Entidad pasiva  
        Puerta p1 = new Puerta(visitantes,10000000);  
        Puerta p2 = new Puerta(visitantes,10000000);  
  
        p1.start(); //Entidad activa  
        p2.start(); //Entidad activa  
  
        try{  
            p1.join();  
            p2.join();  
        } catch (InterruptedException e){  
            System.out.println("La hebra ha sido interrumpida");  
        }  
        System.out.println(visitantes.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 por tanto la hebra) cuando han entrado **iter** visitantes por esa puerta



# El Problema de los Jardines

P1	P2	Suma	Suma Correcta	Diferencia
10000	10000	17439	20000	2561

Iteraciones? 10000

Inicio    Calcula Valores Correctos



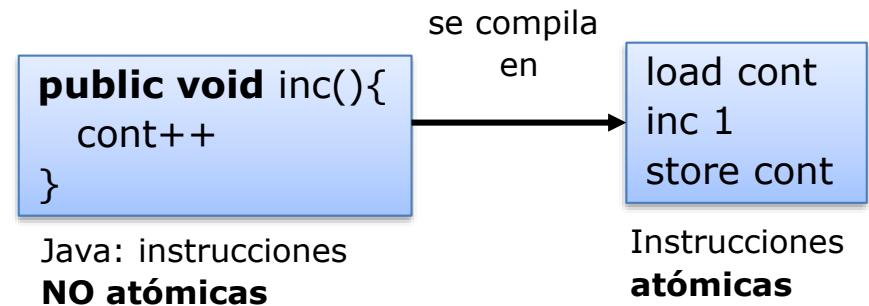
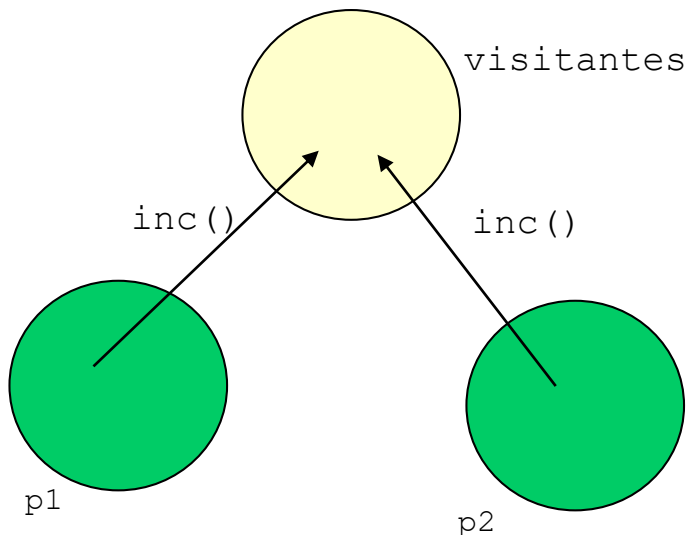
Ejemplo de ejecución:

Entran 10.000 visitantes por cada puerta, y la suma es 17.439 en lugar de 20.000

**¿Por qué?**

# El Problema de los Jardines

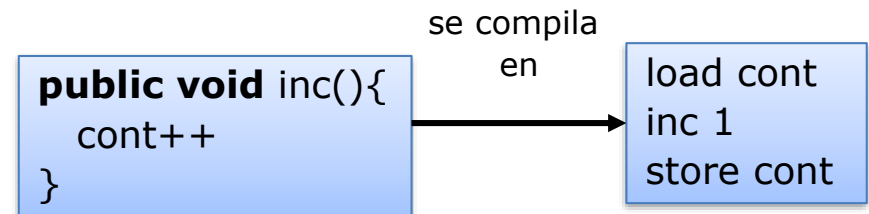
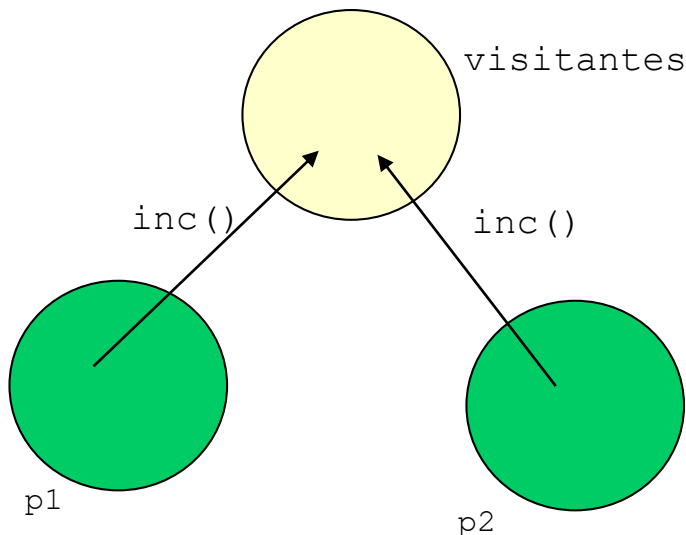
- El objeto **visitantes** es compartido por las hebras **p1** y **p2**
- Las llamadas al método **inc** de visitantes por parte de **p1** y **p2** se interfieren
- Las únicas instrucciones **atómicas** son las instrucciones máquina



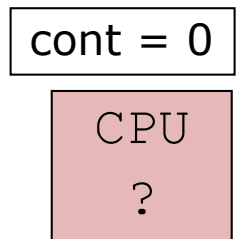
**¡IMPORTANTE!** Cada vez que un proceso deja el procesador y entra otro hay un cambio de contexto

# 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

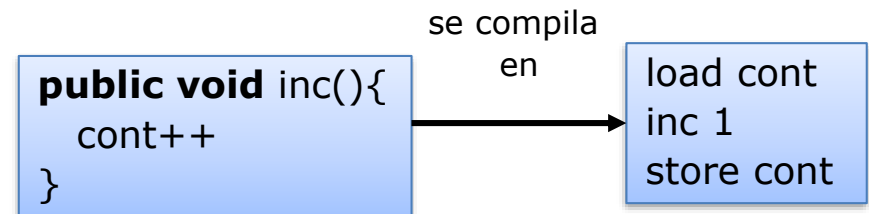
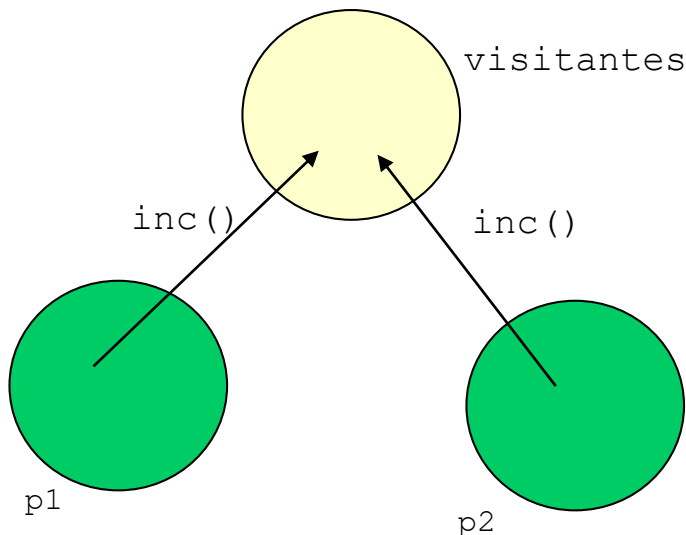


Ejemplo de traza de ejecución

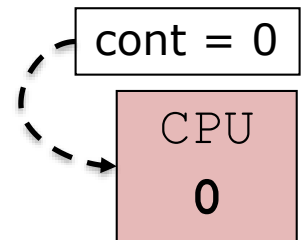


# El Problema de los Jardines

- 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

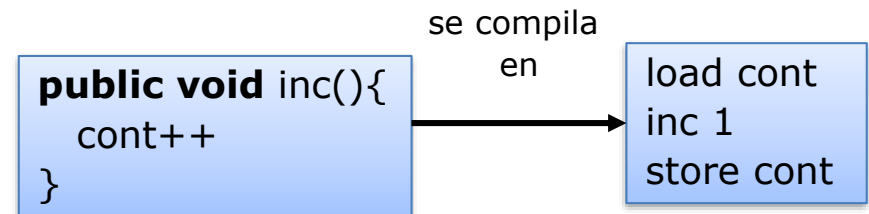
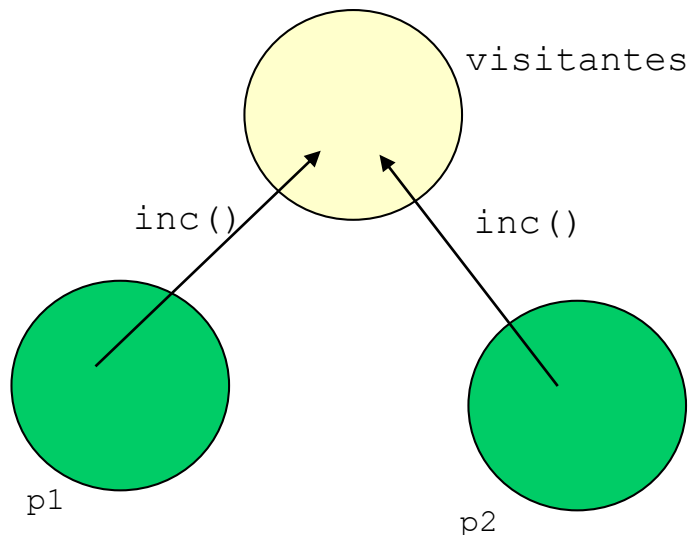


Ejemplo de traza de ejecución  
p1: load cont



# 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

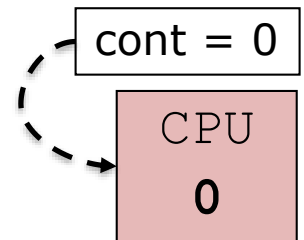


Ejemplo de traza de ejecución

p1: load cont

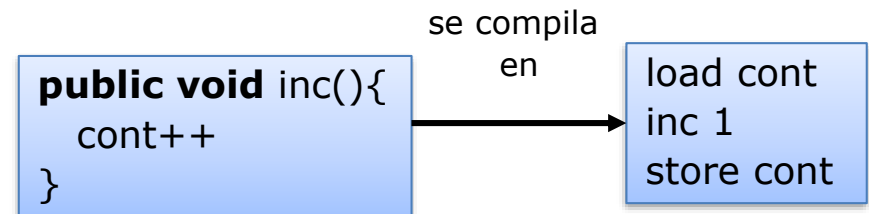
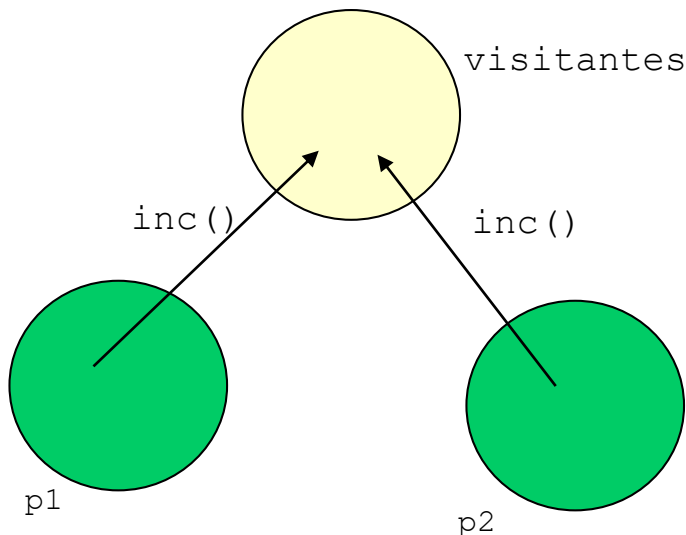
p1 sale del procesador  
p2 entra al procesador  
→ Cambio de contexto

p2: load cont



# 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



Ejemplo de traza de ejecución

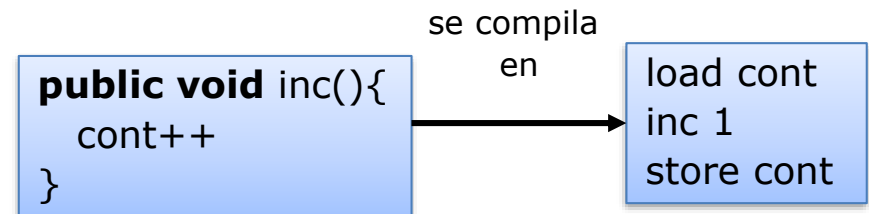
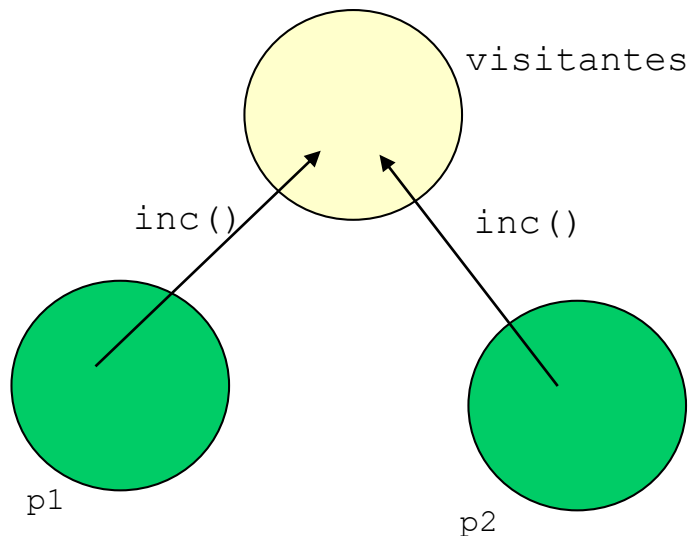
p1: load cont  
p2: load cont  
p1: inc 1

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



Ejemplo de traza de ejecución

p1: load cont

p2: load cont

p1: inc 1

p1 sale del procesador  
p2 entra al procesador  
→ Hay un cambio de contexto

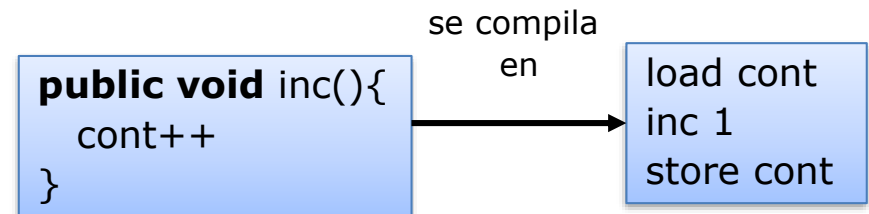
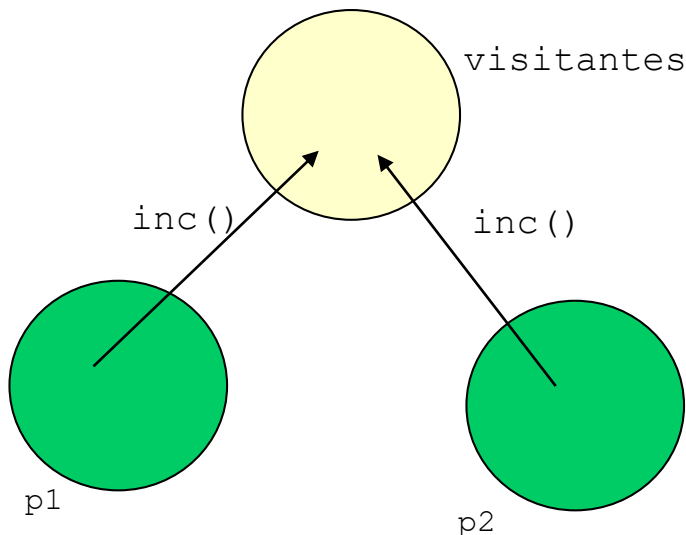
cont = 0

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



Ejemplo de traza de ejecución

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

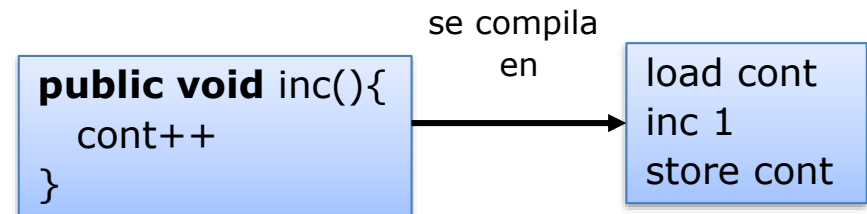
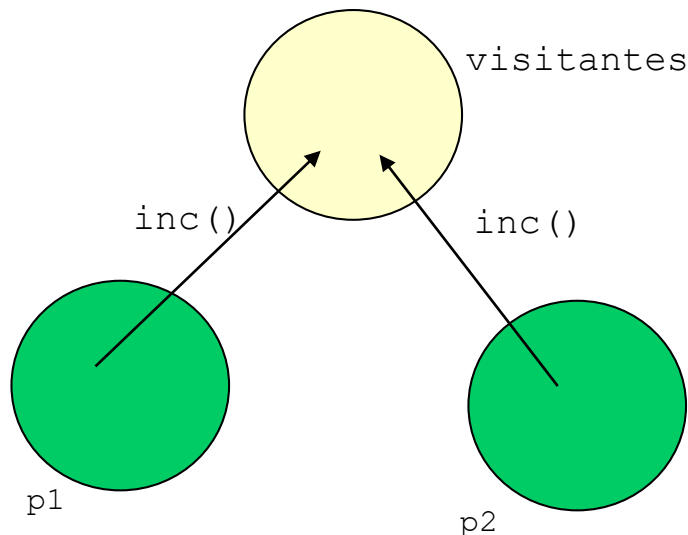
cont = 0

CPU  
1



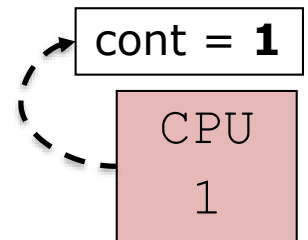
# El Problema de los Jardines

- 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



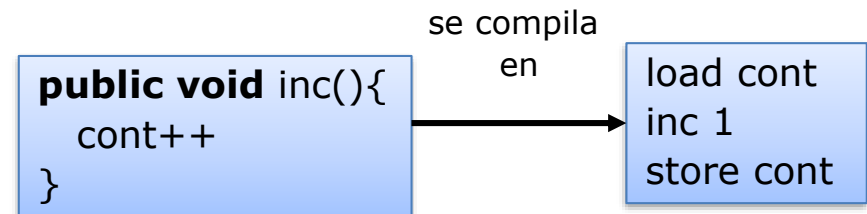
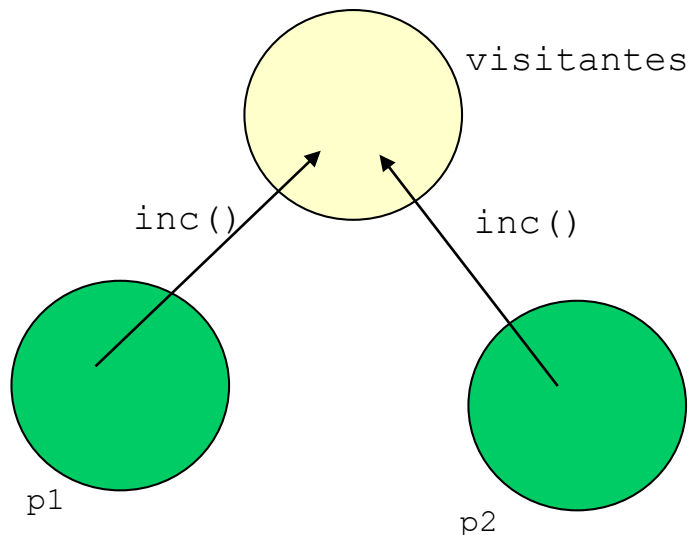
Ejemplo de traza de ejecución

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



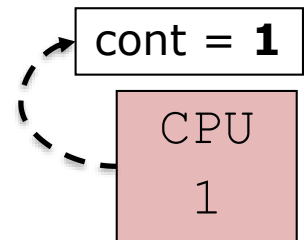
# El Problema de los Jardines

- 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



Ejemplo de traza de ejecución

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



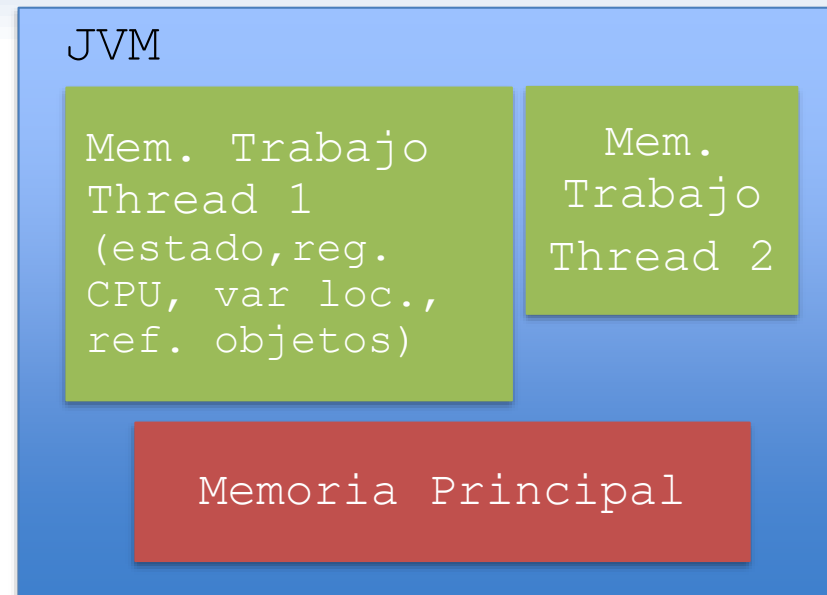
# El Problema de los Jardines

Este ejemplo muestra uno de los grandes problemas de la programación concurrente

- Detectar cuándo un recurso compartido (**Contador**) 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** los procesos involucrados.
  - Si P1 quiere ejecutar su sección crítica cuando P2 la está ejecutando, **P1 debe esperar** a que P2 termine y viceversa.

# Sincronización y el modelo de memoria de Java (JMM)

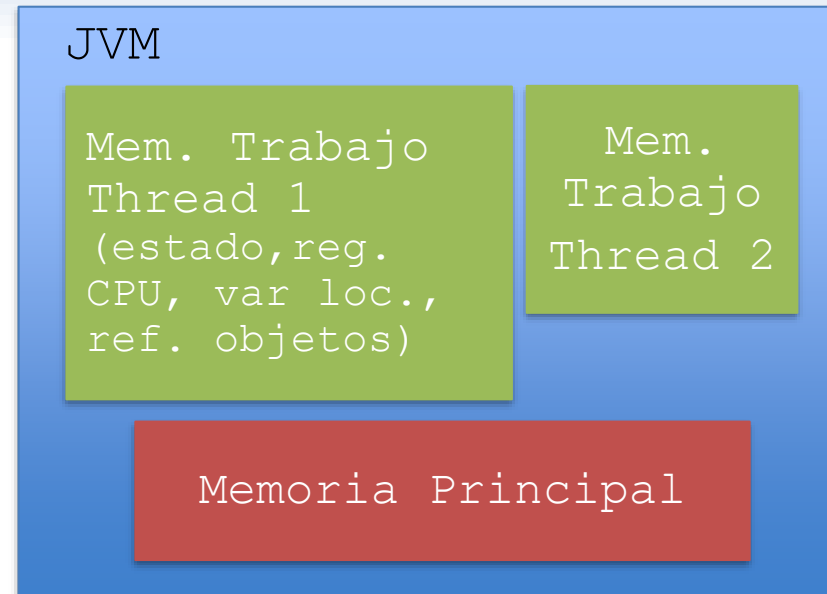
- En el JMM, cada hebra tiene acceso a dos zonas de memoria:
  - La memoria de trabajo, local a la hebra
  - La memoria principal, compartida por todas las hebras



- Todas las variables locales de tipos primitivos (boolean, char, int, etc.) se almacenan en la memoria de trabajo del thread.
- La memoria principal almacena todos los objetos creados por la aplicación Java, independientemente del thread que lo creó.
- Una variable local puede ser una referencia a un objeto.

# Sincronización y el modelo de memoria de Java (JMM)

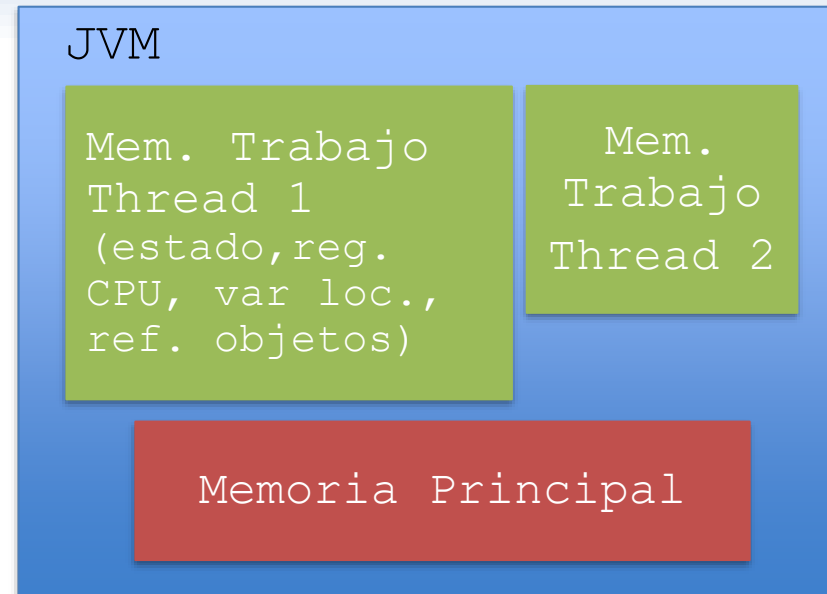
- La memoria de trabajo se utiliza también como caché para optimizar el acceso a los datos y puede contener copias de los datos almacenados en la memoria principal
- La máquina virtual JVM transfiere datos entre la memoria principal y la memoria de trabajo en los siguientes casos:
  - La memoria de trabajo 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



Los locks se verán en el siguiente tema

# Sincronización y el modelo de memoria de Java (JMM)

- Si no estamos trabajando con lock podemos tener **problemas de visibilidad** de los cambios porque no se actualizan las cachés
- Java permite definir atributos volátiles (**volatile**)
  - Los campos volátiles no se almacenan 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



# Sincronización y el modelo de memoria de Java (JMM)

```
public class TestVolatile {  
    private volatile static int myInt =0;  
    public static void main(String[] args){  
        ThreadListener t1 = new ThreadListener();  
        ThreadModifier t2 = new ThreadModifier();  
        t1.start();  
        t2.start();  
    }  
    static class ThreadListener extends Thread{  
        public void run(){  
            int local_value = myInt;  
            while (local_value < 5) {  
                if (local_value!=myInt) {  
                    System.out.println("Listener:"+myInt);  
                    local_value = myInt;  
                }  
            }  
        }  
    }  
}
```

...

```
static class ThreadModifier extends Thread{  
    public void run() {  
        int local_value = myInt;  
        while (local_value < 5) {  
            local_value++;  
            myInt=local_value;  
            System.out.println("Modiyer:"+myInt);  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```


# Sincronización y el modelo de memoria de Java (JMM)

## Ejecución sin volatile

```
Modifier: myInt= 1  
Listener: myInt = 1  
Modifier: myInt= 2  
Modifier: myInt= 3  
Modifier: myInt= 4  
Modifier: myInt= 5
```

## Ejecución con volatile

```
Modifier: myInt= 1  
Listener: myInt = 1  
Modifier: myInt= 2  
Listener: myInt = 2  
Modifier: myInt= 3  
Listener: myInt = 3  
Modifier: myInt= 4  
Listener: myInt = 4  
Listener: myInt = 5  
Modifier: myInt= 5
```

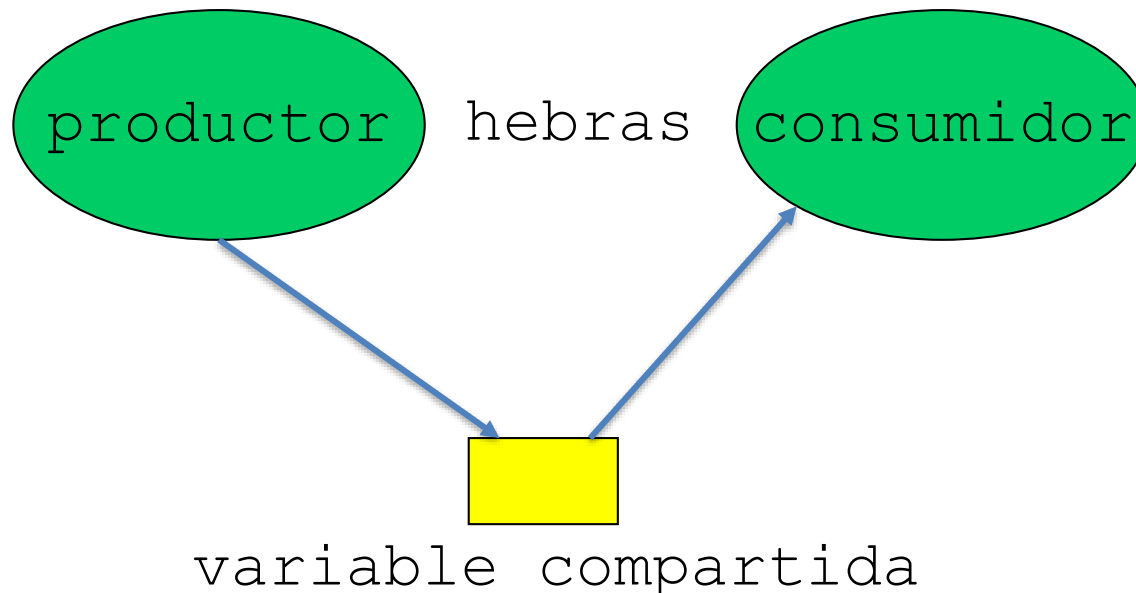


La hebra Listener se queda  
bloqueada porque no detecta los  
5 cambios



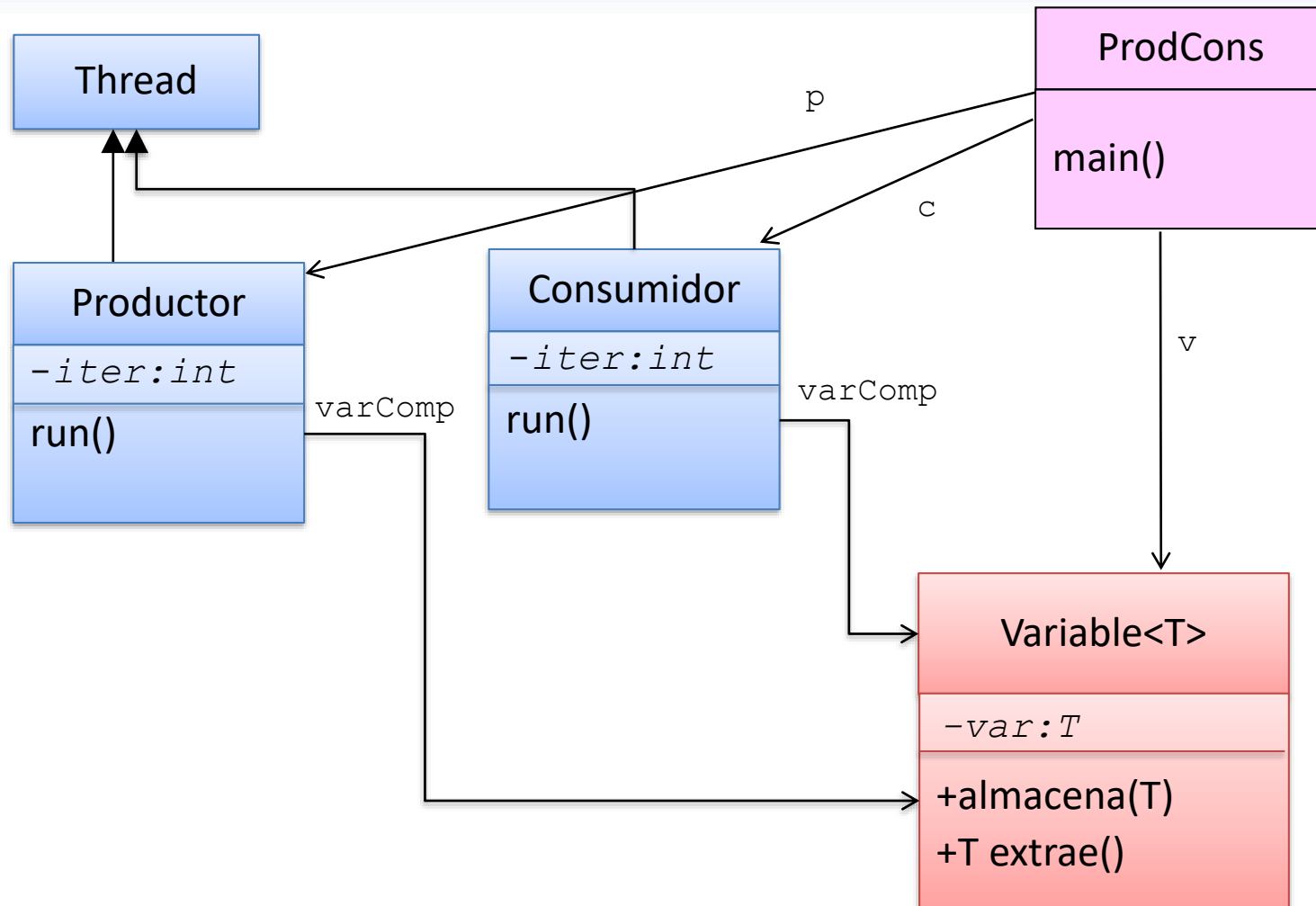
# El Problema del Productor/Consumidor

- 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



# El Problema del Productor/Consumidor

Una posible implementación en Java – Diagrama de Clases



# El Problema del Productor/Consumidor

## Clase principal

Se crean la variable compartida **v** y las hebras Productor **p** y Consumidor **c**

```
public class ProdCons {  
    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();  
    }  
}
```

## Clase Variable<T>

Modela la variable compartida

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

# El Problema del Productor/Consumidor

## Clase Productor

hebra que almacena valores aleatorios en la variable compartida

```
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);
        }
    }
}
```

## Clase Consumidor

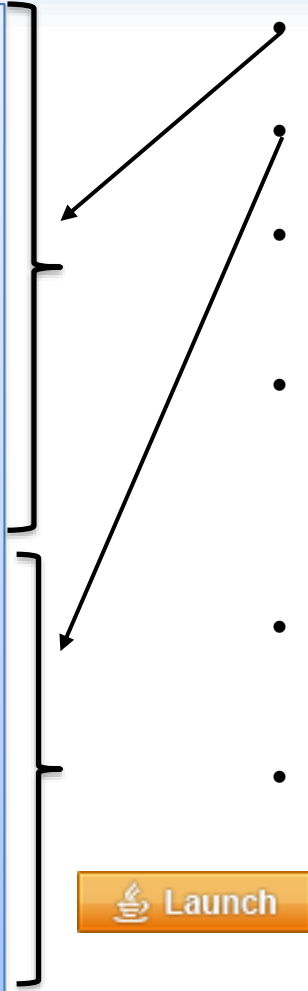
hebra que extrae el valor de la variable compartida

```
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);
        }
    }
}
```

# El Problema del Productor/Consumidor

## Ejemplo de ejecución (no deseada)

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** la variable

- 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**



# Solución al problema del Productor/Consumidor con espera activa

Mantenemos la misma implementación del Productor, Consumidor y el programa principal.

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;  
        hayDato = true;  
    }  
  
    public T extrae(){  
        while (!hayDato) Thread.yield();  
        T 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 nuevo para extraerlo (condición de sincronización 1)

# Solución al problema del Productor/Consumidor con espera activa

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



## Ejecución de la implementación revisada

- 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** en vez de una variable para desacoplar a los procesos
- En la espera activa las hebras están continuamente en el estado Ejecutable “compitiendo” por usar el procesador

# Exclusión mutua con espera activa para dos procesos

Tipo de solución que buscamos

```
//Hebra h0
run(){
    while (true){
        preProtocolo0
        SC0
        postProtocolo0
        SNC0
    }
}
```

SC: Sección Crítica

preProtocolo i: que debe hacer la hebra i antes de entrar a la SCi

postProcolo i: que debe hacer la hebra i después de salir de la SCi

```
//Hebra h1
run(){
    while (true){
        preProtocolo1
        SC1
        postProtocolo1
        SNC1
    }
}
```

SNC: Sección No Crítica

Requisito 1: **Ejecución en exclusión mutua**

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



# Exclusión mutua con espera activa para dos procesos

## 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 para dos procesos

## Primer Intento

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

```
public class Sinc{  
    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;  
    }  
}
```

# Exclusión mutua con espera activa para dos procesos

**Primer Intento:** Esta solución satisface el Requisito 1

Requisito 1: **Ejecución en exclusión mutua**

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

Para probarlo basta observar que

“la hebra  $h_i$  está ejecutando  $SC_i$  sii  $f_i$  es true”

Demostración 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 para dos procesos

## Primer Intento

Requisito 1: **Ejecución en Exclusión Mutua**. 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**:

Las hebras están vivas (en estado ejecutable) pero no pueden avanzar porque que no satisface la condición

La 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 para dos procesos

## Segundo Intento

- Cambiamos el orden de las instrucciones en el preProtocolo para evitar el livelock

Esta solución no es válida porque se viola la exclusión mutua

Ejemplo de ejecución:

h0: ¿f1 == true? No

h1: ¿f0 == true? No

h0: f0 = true

h1: h1 = true

h0 está en SC0

h1 está en SC1

**ERROR!!**

.....

```
public class Sinc{  
    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 para dos procesos

## Tercer Intento

- Para evitar el livelock, usamos una variable **turno**
  - turno toma los valores 0 o 1
- Antes de entrar en su SC, cada hebra comprueba si es su turno, 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

```
public class Sinc {  
    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;  
    }  
}
```

### Requisito 3: Progreso en la ejecución

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

- Peterson permite a dos o más hebras compartir un recurso sin conflicto
- Es una combinación de los intentos 1 y 3
- Esta es la versión para 2 hebras

Cuando h0 quiere entrar en SC0:

1. lo indica poniendo f0 a true y le pasa el turno a h1
2. espera si h1 quiere entrar en SC1 y es su turno

h1 hace las operaciones equivalentes

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

- Peterson permite a dos o más hebras compartir un recurso sin conflicto
- Es una combinación de los intentos 1 y 3
- Esta es la versión para 2 hebras

Cuando sale de SC0, h0 pone f0 a false para indicar que ha salido

h1 hace la operación equivalente



# Exclusión mutua con espera activa: Solución de Peterson

Las hebras comparten un objeto de la clase 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

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 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
        }
    }
}
```

...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

# Exclusión mutua con espera activa: Solución de Peterson

**R1: Ejecución en Exclusión mutua.**

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

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

```
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 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

**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.

```
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

**R3: Progreso en la ejecución.** 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  $f1 \ \&\&(\text{turno} == 1)$  es falsa, y, por lo tanto, h0 puede entrar en SC0.

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

            //SC0

            s.postProt0();

            //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 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

La solución de Peterson satisface un requisito adicional

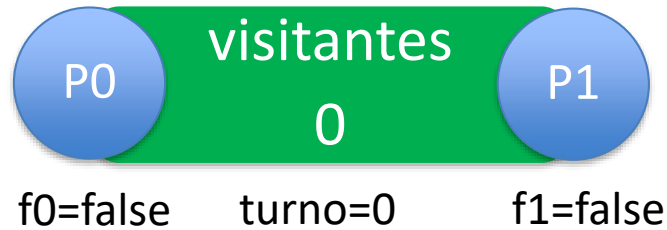
**R4: Justicia.** Si ambos procesos quieren entrar simultáneamente, primero lo hace uno (h0, por ejemplo) y luego lo hace el otro (h1, en este caso) .

- 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 la hebra que puede continuar ejecutando SC1.

# Solución de Peterson para el problema de los Jardines

Las puertas dejan pasar visitantes de uno en uno

Estado inicial



PreProt0



P0 indica que tiene visitantes y cede el turno a P1



P1 no tiene visitantes -> P0 deja pasar un visitante  
SC0



P0 indica que ha pasado un visitante



PostPro0

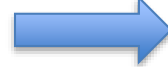


# Solución de Peterson para el problema de los Jardines

P0 y P1 tienen visitantes



PreProt0



P0 indica que tiene visitantes y cede el turno a P1



**f0=true** turno=1 f1=false

PreProt1



P1 indica que tiene visitantes y cede el turno a P0



f0=true turno=0 f1=true

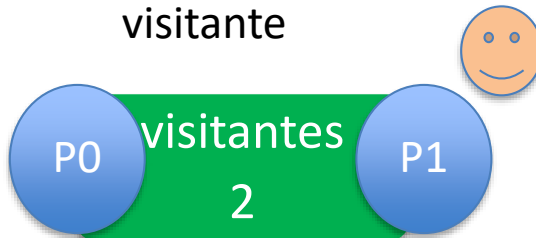
SC0



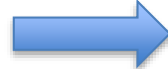
**f0=true** turno=0 **f1=true**

PostProt0

P0 indica que ha pasado el visitante



SC1



f0=false turno=0 f1=true

PostProt1



f0=false turno=0 **f1=false**



# Exclusión mutua con espera activa: Solución de Dekker

Utiliza los mismos recursos que la solución de Peterson:  
turno y dos flags f0 y f1

```
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

Utiliza los mismos recursos que la solución de Peterson:  
turno y dos flags f0 y f1

```
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;  
    }  
    ....  
}
```

El comportamiento de h1 es equivalente

Cuando h0 quiere entrar a SC0 lo indica con f0=true

Mientras h1 quiera entrar en SC1 y es el turno de h1 -> h0 espera a que sea el turno de h0

Cuando h1 no quiera entrar en SC1 entrará h0 a SC0

Cuando h0 sale de SC0 le pasa el turno a h1 e indica que no quiere entrar (f0=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;
        }
    }
}

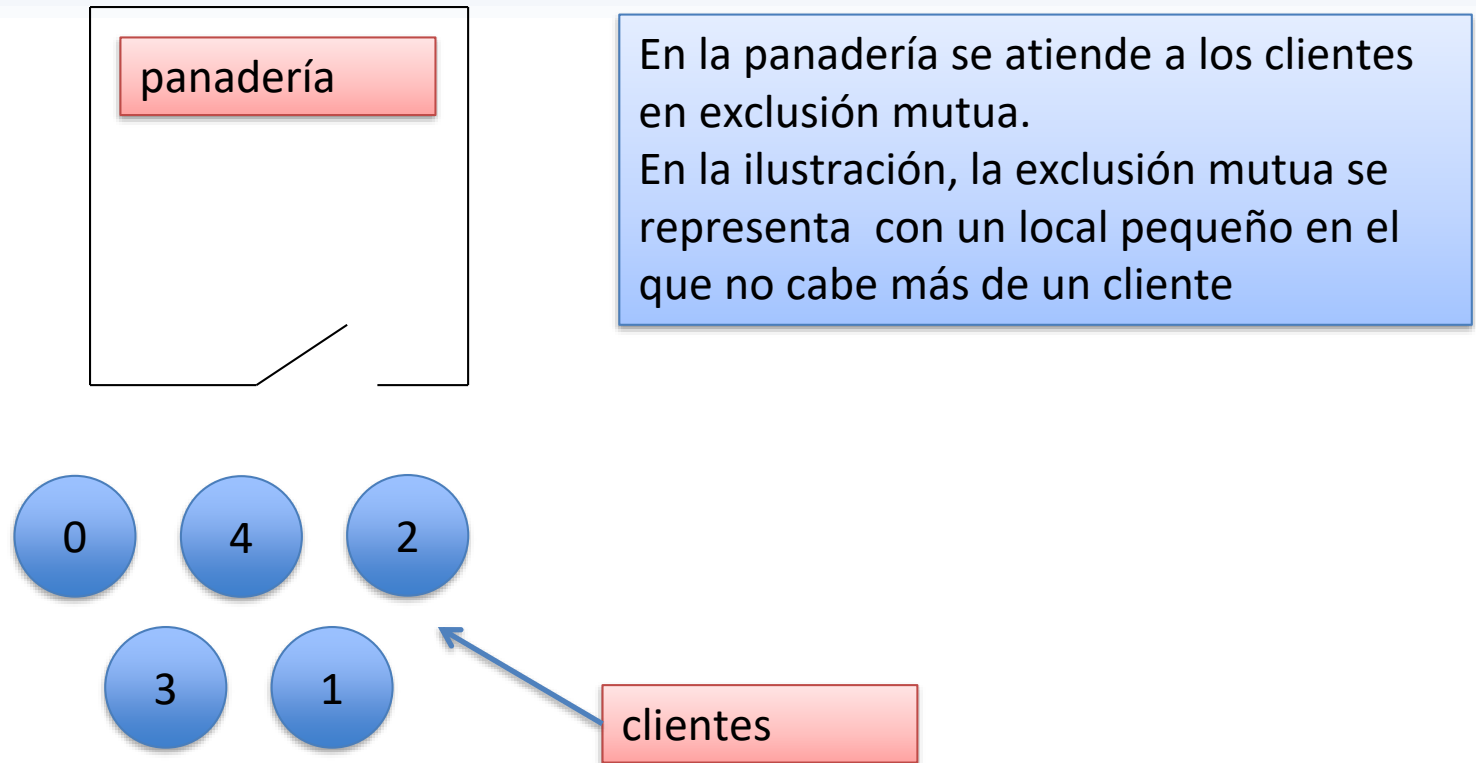
public void postProt1(){
    turno = 0;
    f1 = false;
}
}
```

Satisface R1, R2, R3 y R4

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

- El algoritmo de la panadería (Lamport) 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:
  - 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)

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería



# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## Estructura del Código

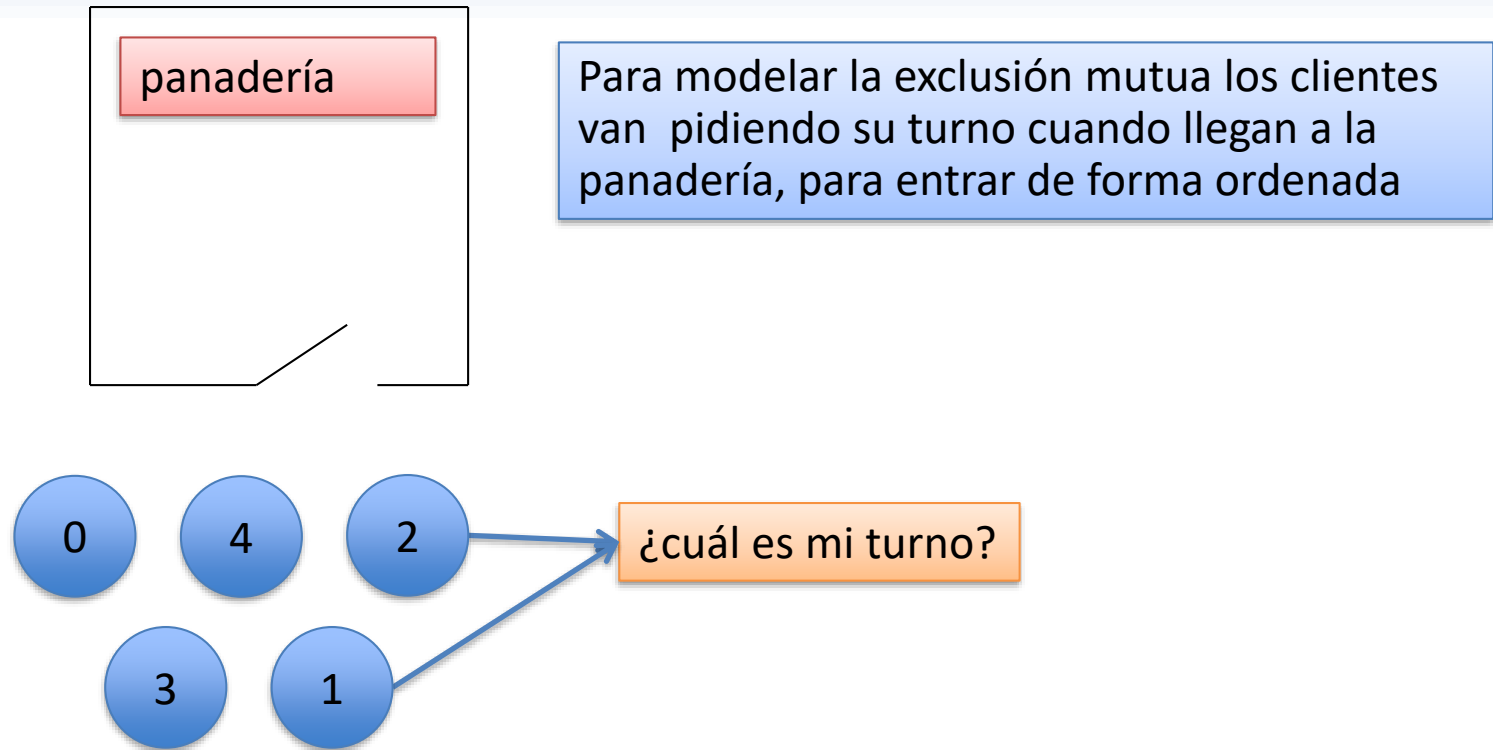
```
public class Panaderia {  
    public Panaderia(int N){  
        ....  
    }  
    ....  
}
```

- La Panadería es el recurso compartido entre todos los Clientes
- Los Clientes son hebras
- En el programa principal, N (=15) clientes llegan a la panadería y son atendidos en exclusión mutua.

```
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*/  
        //SC: el cliente id es atendido por el dependiente  
        /*Postprotocolo*/  
        //SNC: 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();  
    }  
}
```

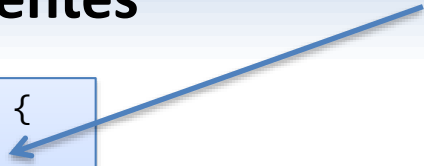
# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería



# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## Turno de los Clientes

```
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** si **no quiere acceder a su SC**. 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*/  
        //SC: el cliente id es atendido por el dependiente  
        /*Postprotocolo*/  
        //SNC: el cliente id sale de la panadería  
    }  
}
```



# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

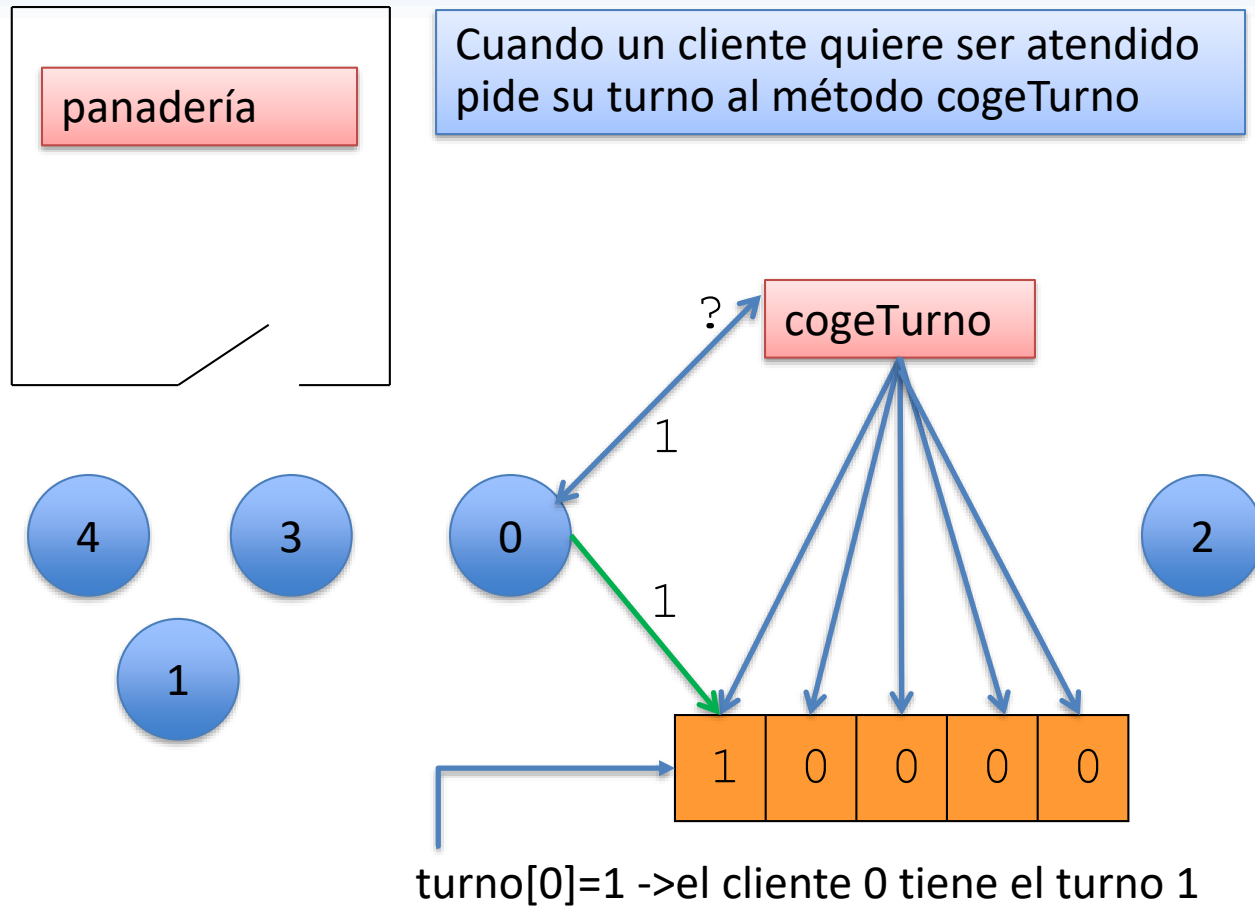
## cogeTurno

```
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.

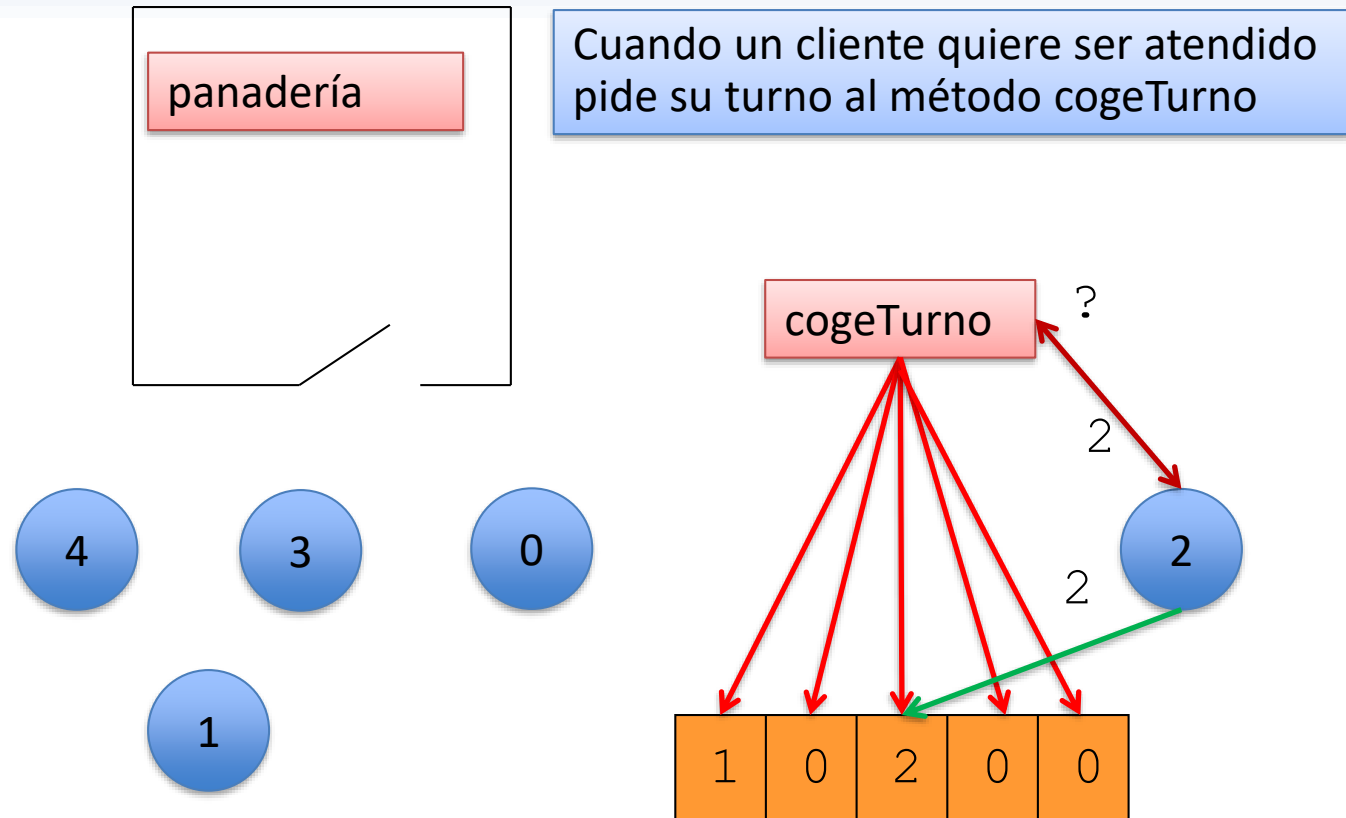
```
class Cliente extends Thread{  
    private int id;  
    private Panaderia pan;  
    public Cliente(int id, Panaderia pan){  
        this.id = id; this.pan = pan;  
    }  
    public void run(){  
        pan.cogeTurno(id);  
        //SC: el cliente id es atendido por el dependiente  
     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

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería



turno[i] almacena el turno del cliente i

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería



turno[0]=1 -> el cliente 0 tiene el turno 1

turno[2]=2 -> el cliente 2 tiene el turno 2

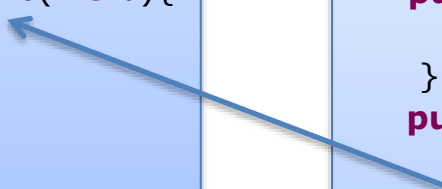
# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## 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 el cliente tiene su turno, 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);  
        //SC: el cliente id es atendido por el dependiente  
        /*Posprotocolo*/  
        //SNC: el cliente id sale de la panadería  
    }  
}
```

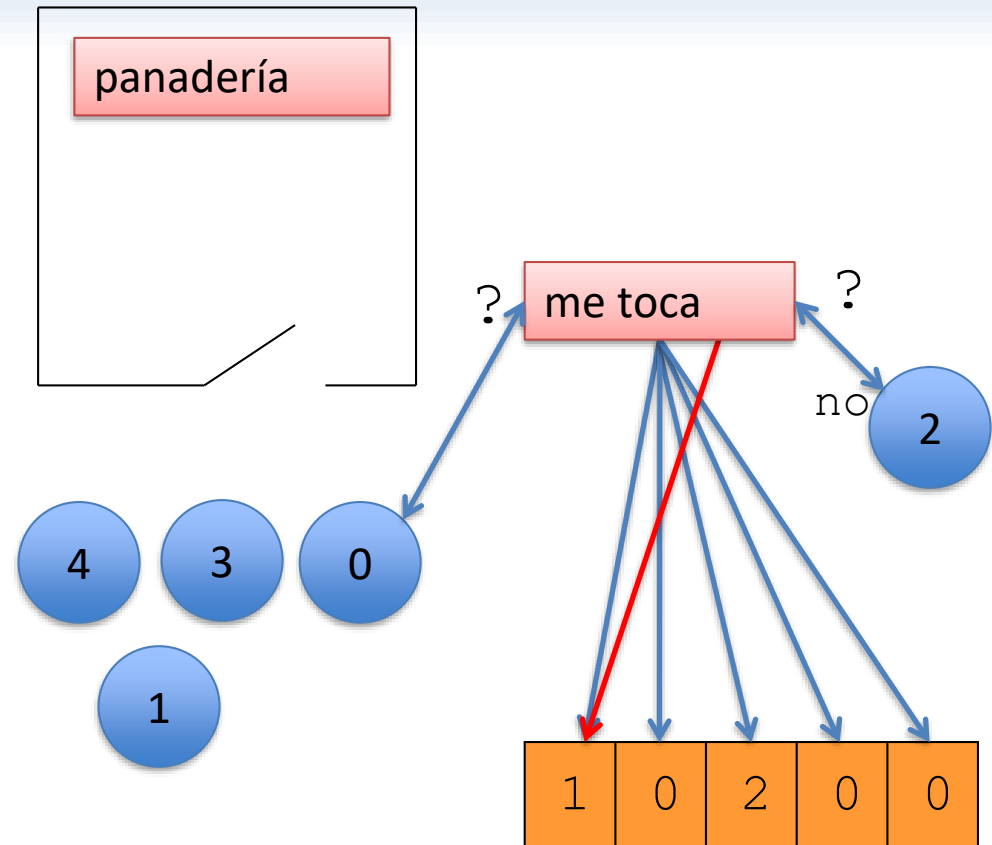


# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

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 cliente ve que hay otro al que le toca antes, espera a que llegue su turno



# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## esperoTurno -- 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

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## salePanaderia

```
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;
    }

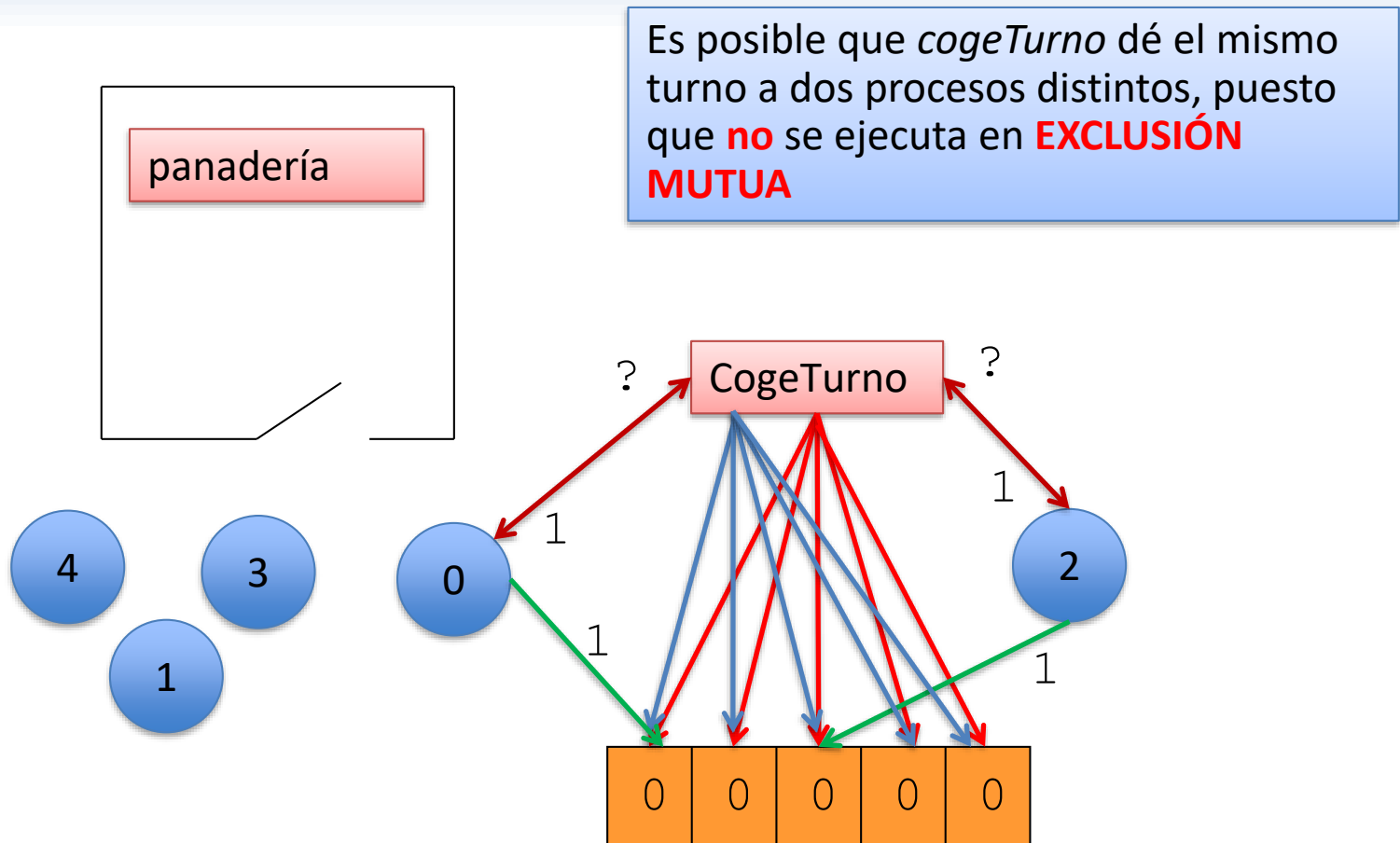
    public void esperoTurno(int id){
        for (int i = 0; i < turno.length; i++)
            while (!meToca(id,i)) Thread.yield();
    }

    public void salePanaderia(int id){
        turno[id] = 0;
    }
}
```

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

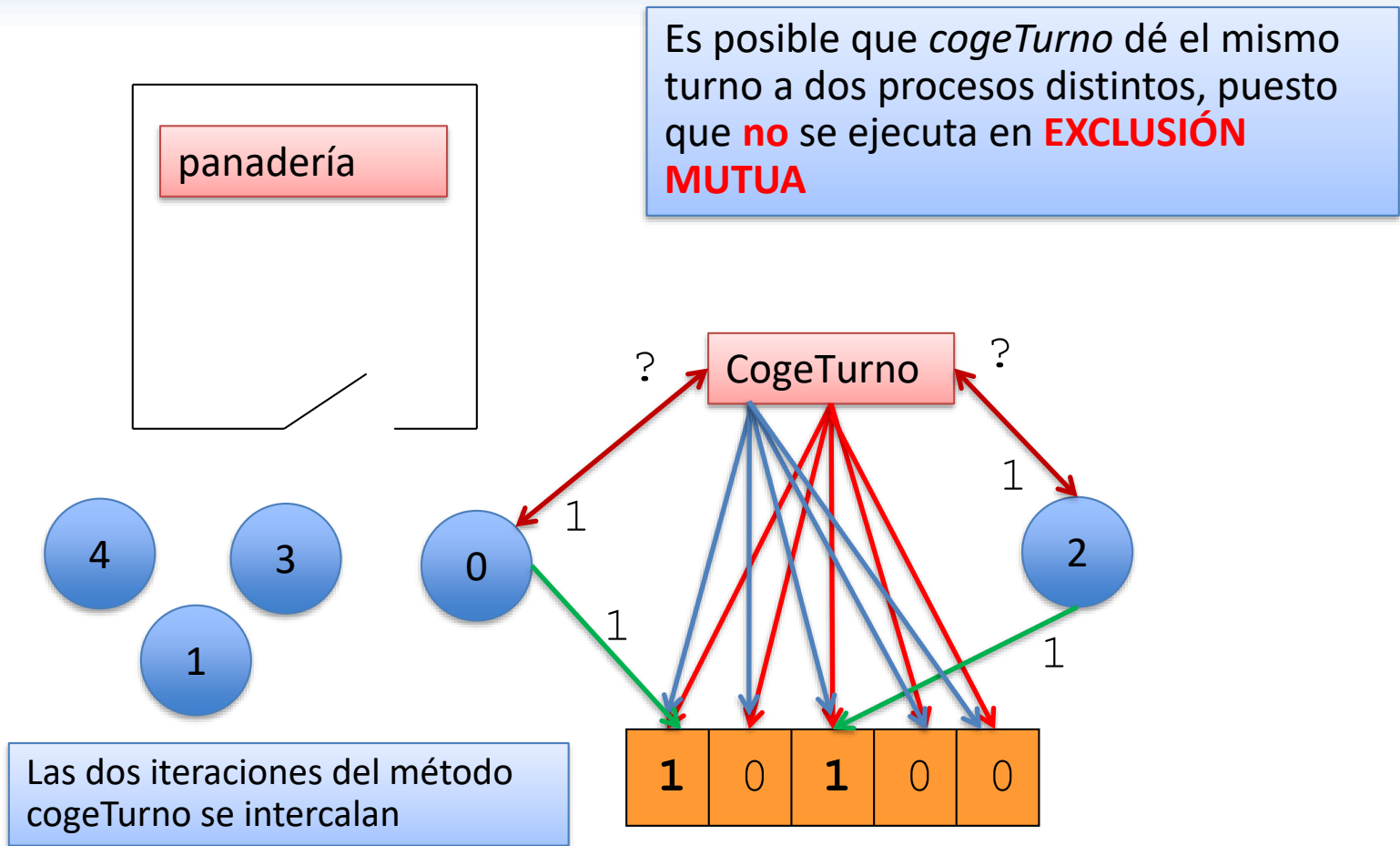
```
class Cliente extends Thread{
    private int id;
    private Panaderia pan;
    public Cliente(int id,Panaderia pan){
        this.id = id; this.pan = pan;
    }
    public void run(){
        pan.cogeTurno(id);
        pan.esperaTurno(id);
        // SC: el cliente id es atendido por el dependiente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

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería





# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería



# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

```
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...

Veamos una posible traza de ejecución

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## 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.esperoTurno(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

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

## 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
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

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la Panadería

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

## Mejoramos la implementación

- El 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 haya terminado de escoger su turno.

# Exclusión mutua con espera activa para más de dos procesos: Algoritmo de la 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 < turno.length; 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
  - Progreso en la ejecución (R3): Si sólo una hebra quiere entrar en su SC, en algún momento debe poder hacerlo
  - Ausencia de livelock (R2): Si dos hebras quieren entrar en su sección crítica, en algún momento alguna de ellas debe poder hacerlo
  - Ausencia de posposición indefinida (inanición o 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 (fairness)

- **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;
    }
}
```

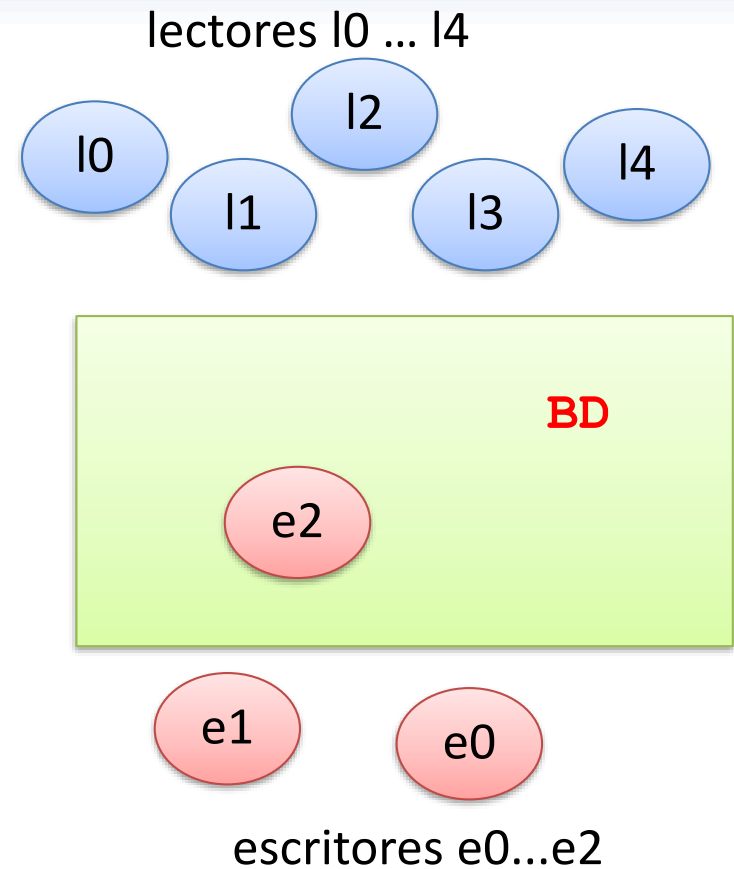
Con un planificador **débilmente justo** este programa podría no terminar.  
Con un planificador **fuertemente justo**, siempre termina.

# Problema de los 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

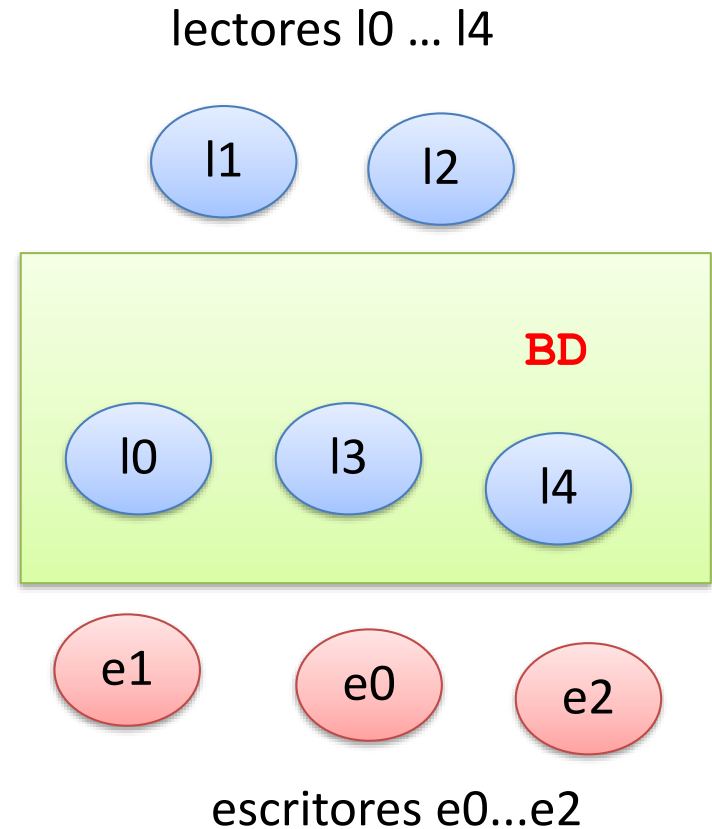
# Problema de los Lectores/Escritores

- Condición de sincronización para los escritores:
  - Un escritor accede a la BD en **exclusión mutua** con cualquier otro proceso de tipo lector o escritor



# Problema de los 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; this.start();
    }
    public void run(){
        while (true){
            g.entraLector(id);
            //lector id en la BD
            g.saleLector(id);
        }
    }
}
```

La BD no hace falta modelarla,  
el GestorDB se encarga de la  
sincronización

```
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; this.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 Lector[NL]; //NL número de lectores
    Escritor[] esc = new Escritor[NE]; //NE: número de escritores
    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);
}
```

Todos los lectores ejecutan los mismos  
protocolos de entrada y salida

Todos los escritores ejecutan los mismos  
protocolos de entrada y salida

# 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; this.start();
    }
    public void run(){
        while (true){
            g.entraLector(id);
            //lector id en la BD
            g.saleLector(id);
        }
    }
}
```

La BD no hace falta modelarla,  
el GestorDB se encarga de la  
sincronización

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

La implementación con  
espera activa es compleja,  
en el siguiente tema se  
verán otros mecanismos de  
sincronización

```
class Escritor extends Thread{
    private int id;
    private GestorBD g;
    public Escritor(int id, GestorBD g){
        this.g = g;
        this.id = id; this.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 Lector[NL]; //NL número de lectores
    Escritor[] esc = new Escritor[NE]; //NE: número de escritores
    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);
    }
}
```

Todos los lectores ejecutan los mismos  
protocolos de entrada y salida

Todos los escritores ejecutan los mismos  
protocolos de entrada y salida

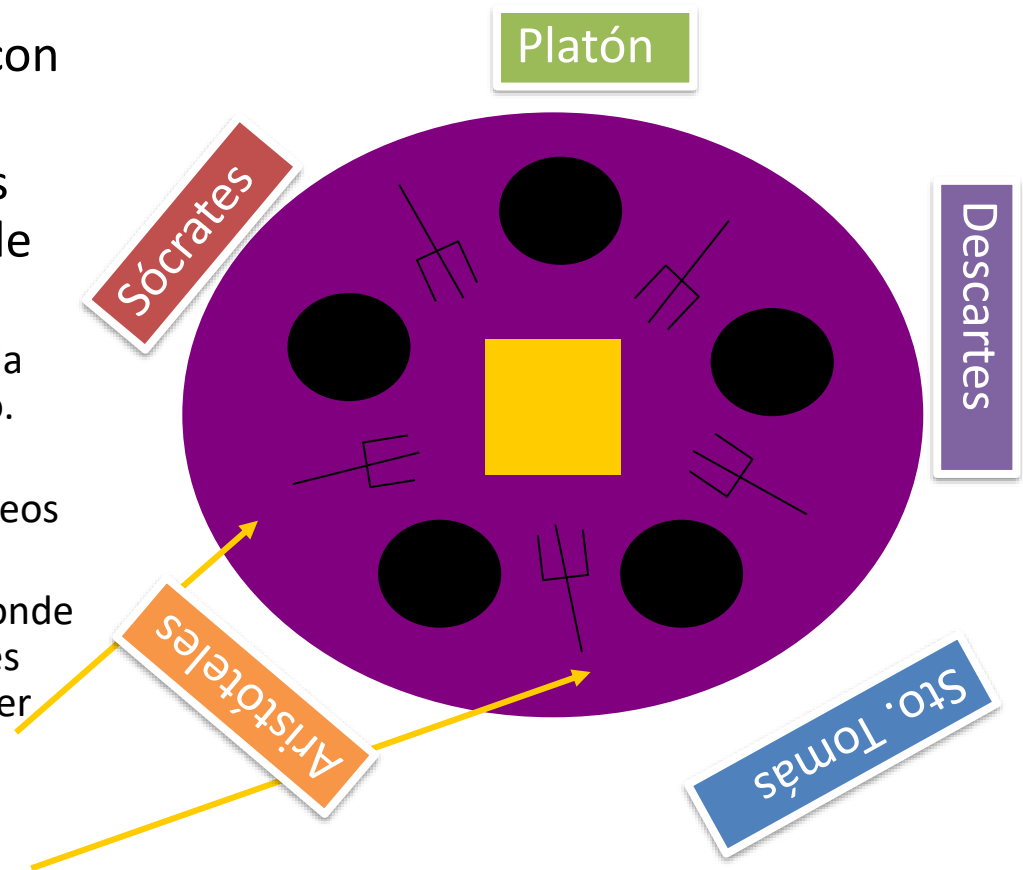
# El Problema de los Filósofos

- 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





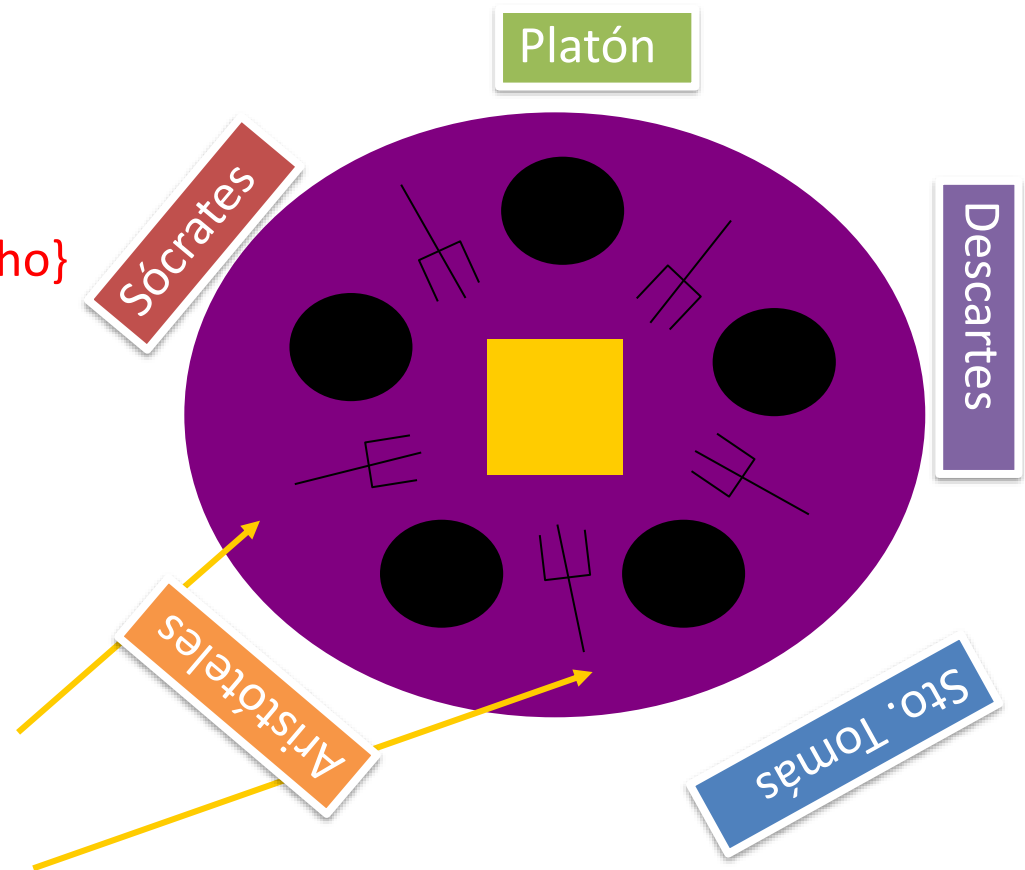
# El Problema de los Filósofos

- 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:

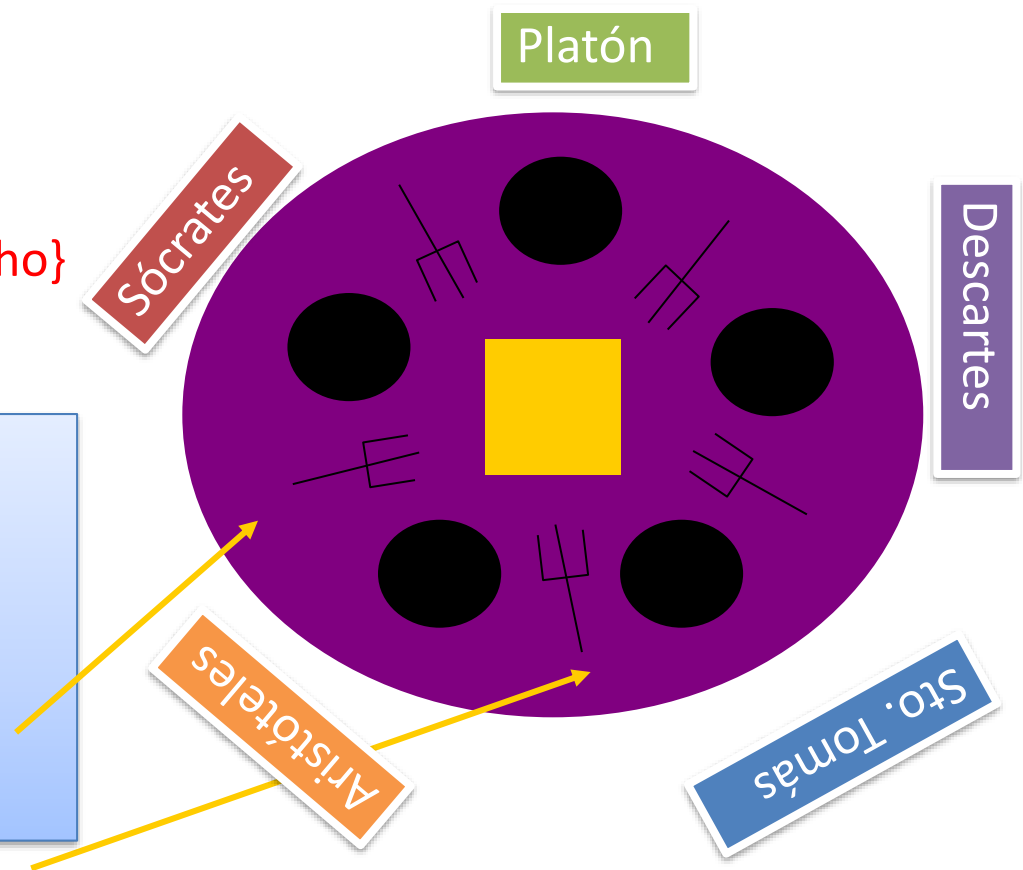
{coge tenedores izdo y dcho}

{come}

{devuelve tenedores izdo y dcho}

Este sistema nos sirve para representar

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



# El Problema de los Filósofos

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

{coge tenedores izquierdo y derecho}

{come}

{devuelve tenedores}

La implementación con espera activa es compleja, en el siguiente tema se verán otros mecanismos de sincronización

Este sistema nos sirve para:

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

Platón

Descartes

Aristóteles

Sto. Tomás

# Referencias

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