

Programación de Sistemas y Concurrency

Tema 6: Comunicación y Sincronización en Memoria Compartida

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores



Índice

- Semáforos
 - Definición
 - Implementación
- Exclusión mutua
- El problema del productor/consumidor
- Implementación de semáforos generales con binarios
- El problema del barbero dormilón
- Lectores/escritores
- El problema de los filósofos

Definición de Semáforo

- Un **semáforo** es un objeto del tipo **Semaphore**, predefinido en Java (a partir de la versión 5) que se encuentra en el paquete **java.util.concurrent**
- Un semáforo tiene como valor un **número natural**, que representa el número permisos aún disponibles para acceder a un recurso compartido.
 - El **semáforo** se llama **general**, si puede tomar valores superiores a uno,
 - o **binario**, si solo toma los valores 0, 1.

Definición de Semáforo

- Sobre un semáforo s , pueden realizarse dos operaciones principales.
 - **$s.acquire()$** : Decrementa el valor de s en cuanto el resultado sea no negativo.
 - **$s.release()$** : Incrementa el valor de s .
- Por definición, **acquire** y **release** se ejecutan de *forma atómica*, es decir, su ejecución paralela nunca produce valores inconsistentes para el semáforo.

Definición de Semáforo

- La operación **s.acquire()** *suspende* a la hebra que la ejecuta si el semáforo vale 0. La hebra espera a que el valor del semáforo sea positivo para realizar el decremento.
- La operación **s.release()** puede *despertar* a alguna hebra, si está esperando para completar **s.acquire()**

Definición de Semáforo

```
Semaphore s;  
// s = 0
```

```
public class P1 extends Thread{  
    public void run(){  
        s.acquire();  
    }  
}
```

```
public class P2 extends Thread{  
    public void run(){  
        s.release();  
    }  
}
```

```
P1 p1 = new P1();  
P2 p2 = new P2();  
p1.start();p2.start();
```

Hay dos posibles ejecuciones para p1 | p2:

```
p1:s.acquire()  
{p1 suspende}  
p2:s.release()  
{p1 despierta}  
{s = 0}
```

```
p2:s.release()  
{s = 1}  
p1:s.acquire()  
{s = 0}
```

En ambas ejecuciones, el valor final de s es 0.

Implementación de Semáforos

```
public class Semaphore{
    private int valor;
    private Collection<Thread> hSuspendidas;
    ....

    public void acquire(){
        if (valor > 0) valor--;
        else bloquear a la hebra en hSuspendidas
    }
    public void release(){
        if (hSuspendidas.empty()) valor++;
        else despertar una hebra de hSuspendidas
    }
}
```

Esta es una posible implementación de la clase semáforo de java

Implementación de Semáforos

```
public class Semaphore{  
    private int valor;  
    private Collection<Thread> hSuspendidas;  
    ....  
  
    public void acquire(){  
        if (valor > 0) valor--;  
        else bloquear a la hebra en hSuspendidas  
    }  
    public void release(){  
        if (hSuspendidas.empty()) valor++;  
        else despertar una hebra de hSuspendidas  
    }  
}
```

Estas dos operaciones se compensan: cuando la hebra se despierta no decrementa valor, y cuando release despierta a una hebra no incrementa valor. De esta forma se ahorran un par de operaciones.

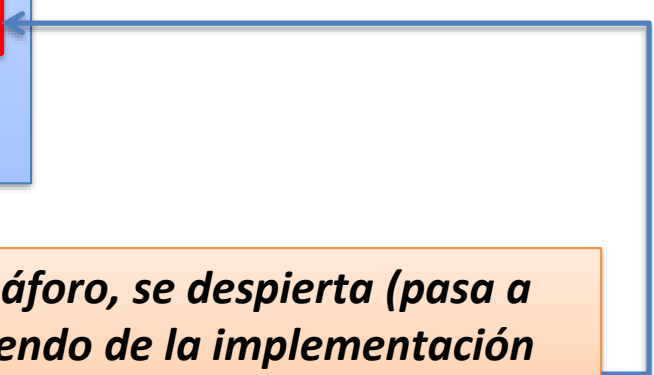
Implementación de Semáforos

```
public class Semaphore{  
    private int valor;  
    private Collection<Thread> hSuspendidas;  
    ....  
  
    public void acquire(){  
        if (valor > 0) valor--;  
        else bloquear a la hebra en hSuspendidas  
    }  
    public void release(){  
        if (hSuspendidas.empty()) valor++;  
        else despertar una hebra de hSuspendidas  
    }  
}
```

La suspensión de la hebra que hace acquire no es activa, es decir, la hebra pasa a estado bloqueado, y no vuelve a competir por el uso del procesador hasta que vuelva a estado ejecutable.

Implementación de Semáforos

```
public class Semaphore{  
    private int valor;  
    private Collection<Thread> hSuspendidas;  
    ....  
  
    public void acquire(){  
        if (valor > 0) valor--;  
        else bloquear a la hebra en hSuspendidas  
    }  
    public void release(){  
        if (hSuspendidas.empty()) valor++;  
        else despertar una hebra de hSuspendidas  
    }  
}
```



Si hay varias hebras suspendidas en el semáforo, se despierta (pasa a estado ejecutable) a una de ellas. Dependiendo de la implementación del semáforo, puede utilizarse cualquier criterio para seleccionar la hebra a despertar.

Implementación de Semáforos

```
public class Semaphore{  
    private int valor;  
    private Collection<Thread> hSuspendidas;  
    ....  
  
    public void acquire(){  
        if (valor > 0) valor--;  
        else bloquear a la hebra en hSuspendidas  
    }  
    public void release(){  
        if (hSuspendidas.empty()) valor++;  
        else despertar una hebra de hSuspendidas  
    }  
}
```

La única forma de que un semáforo tenga procesos suspendidos es que su valor sea cero.

Podemos imaginarnos un semáforo como un registro $s = (\text{valor}, h\text{Suspendidas})$, tal que si la componente $h\text{Suspendidas}$ no es vacía, entonces $\text{valor} = 0$.

Definición de Semáforo

```
public class Semaphore {  
  
    public Semaphore(int permits);  
    public Semaphore(int permits, boolean fair);  
  
    public void acquire() throws InterruptedException;  
    public void acquire(int permits) throws InterruptedException;  
  
    public void release();  
    public void release(int permits) ;  
  
    public int availablePermits();  
  
    ...  
}
```

valor inicial del semáforo

si fair es true el semáforo es justo

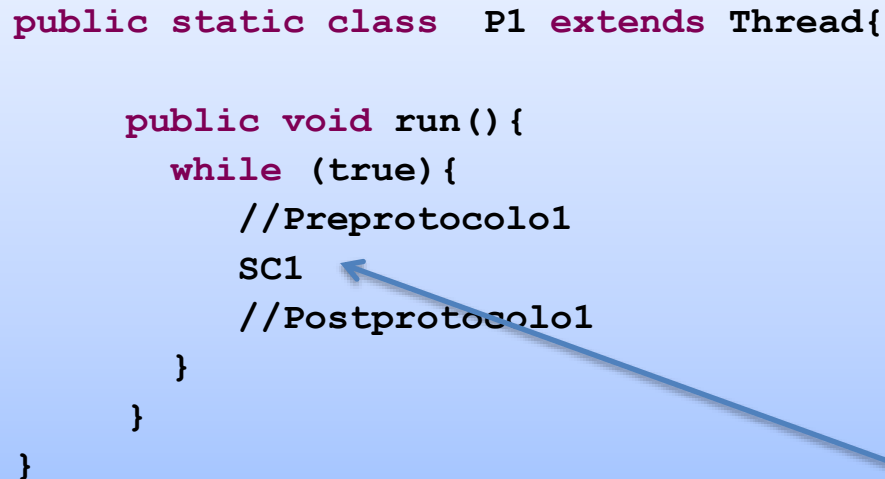
pide uno o varios permisos al semáforo. Pueden lanzar la excepción comprobada InterruptedException

devuelve uno o varios permisos al semáforo

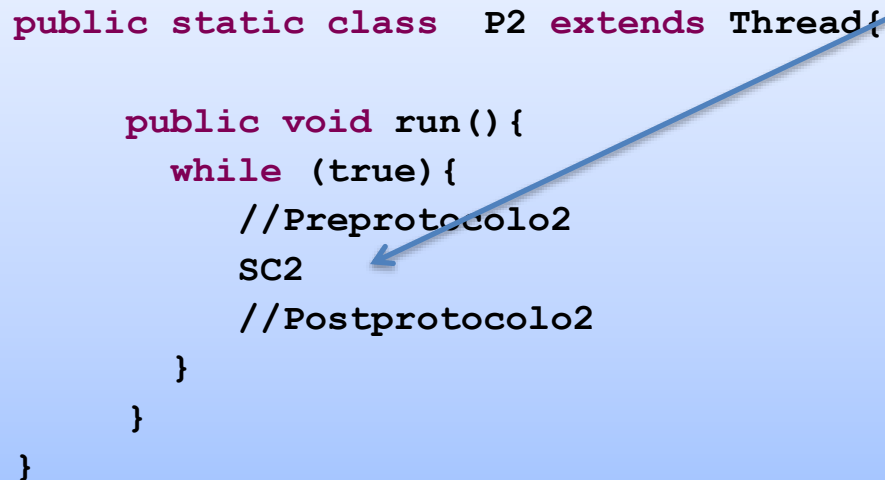
devuelve el valor del semáforo

Exclusión Mutua

```
public static class P1 extends Thread{  
  
    public void run(){  
        while (true){  
            //Preprotocolo1  
            SC1  
            //Postprotocolo1  
        }  
    }  
}
```



```
public static class P2 extends Thread{  
  
    public void run(){  
        while (true){  
            //Preprotocolo2  
            SC2  
            //Postprotocolo2  
        }  
    }  
}
```



RI: en cada momento hay a lo sumo un proceso ejecutando su sección crítica

Exclusión Mutua

```
public static class P1 extends Thread{
    public void run(){
        try{
            while (true){
                s.acquire();
                SC1
                s.release();
            }
        } catch (InterruptedException ie){}
    }
}
```

```
static Semaphore s = new Semaphore(1,true);
// semáforo binario
```

Cuando una hebra entra en su sección crítica cierra el semáforo y así impide que el otro proceso entre.

```
public static class P2 extends Thread{
    public void run(){
        try{
            while (true){
                s.acquire();
                SC2
                s.release();
            }
        } catch (InterruptedException ie){}
    }
}
```

Cuando la hebra sale de su sección crítica abre el semáforo para permitir que la otra hebra ejecute su SC si quiere hacerlo.

Exclusión Mutua: extensión a N hebras

```
static Semaphore s = new Semaphore(1,true);  
// semáforo binario
```

La extensión a N procesos es directa

```
public static class P extends Thread{  
    public void run(){  
        try{  
            while (true){  
                s.acquire();  
                SC  
                s.release();  
            }  
        } catch (InterruptedException ie){}  
    }  
}
```

```
public static void main(String[] args){  
    P[] proc = new P[N];  
    for (int i = 0; i < N; i++)  
        proc[i] = new P();  
    for (int i = 0; i < N; i++)  
        proc[i].start();  
}
```

Exclusión Mutua: Generalización

```
static Semaphore s = new Semaphore(m,true);  
// semáforo general
```

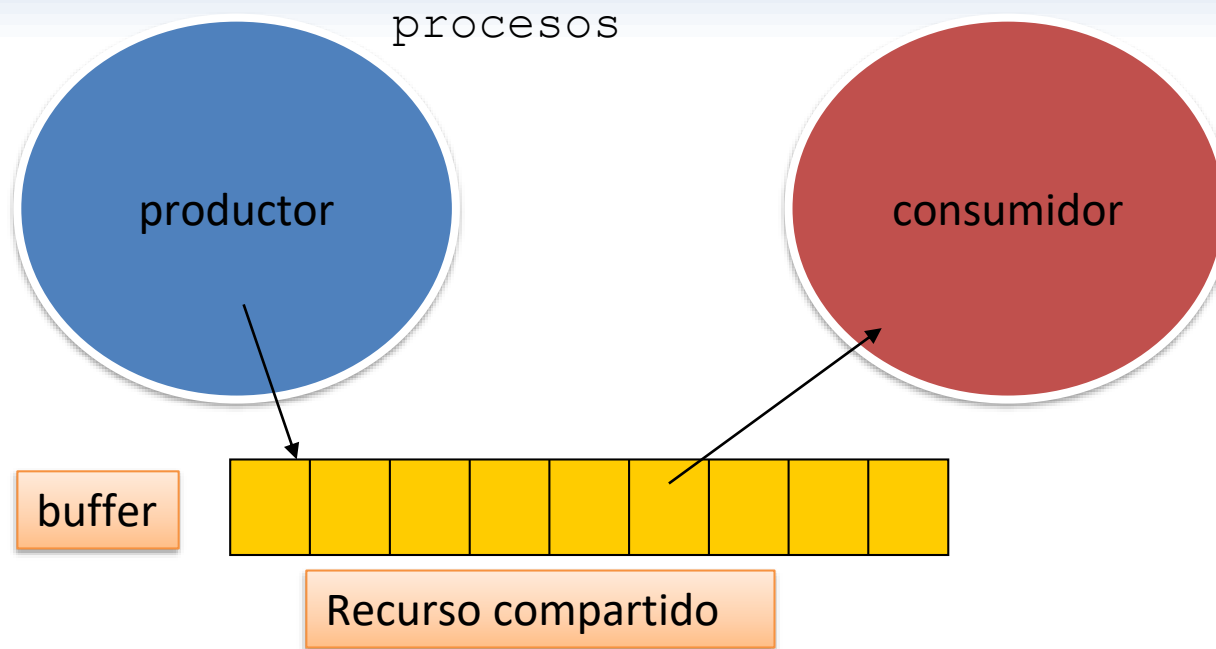
```
public static class P extends Thread{  
    public void run(){  
        try{  
            while (true){  
                s.acquire();  
                SC  
                s.release();  
            }  
        } catch (InterruptedException ie){}  
    }  
}
```

R1': en cada momento hay a lo sumo $m < N$ hebras ejecutando su sección crítica

La generalización para que se satisfaga R1' es directa

```
public static void main(String[] args){  
    P[] proc = new P[N];  
    for (int i = 0; i < N; i++)  
        proc[i] = new P();  
    for (int i = 0; i < N; i++)  
        proc[i].start();  
}
```


El problema del productor/consumidor



Propiedades

- C1: El productor no puede almacenar en el buffer, si está lleno
- C2: El consumidor no puede extraer del buffer, si está vacío
- R1: El buffer siempre se utiliza en exclusión mutua

El problema del productor/consumidor

```
public class Buffer { //Recurso
    private int[] b;
    public Buffer(int tam){
        b=new int[tam]
    }
    public void almacena {...}
    public int extrae(){.....}
}
```

Hemos eliminado las instrucciones try/catch en el código de las hebras para simplificar el código

```
public class Productor extends Thread{
    private Buffer b;
    public Productor(Buffer b){
        this.b = b;
    }
    public void run(){
        while (true){
            //produce(dato): tarea asíncrona
            b.almacena(dato);
        }
    }
}
```

```
public class consumidor extends Thread{
    private Buffer b;
    public Productor(Buffer b){
        this.b = b;
    }
    public void run(){
        while (true){
            int dato = b.extrae();
            //consume(dato): tarea asíncrona
        }
    }
}
```

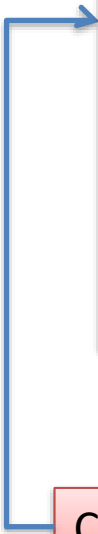
El consumidor debe leer todos los datos producidos por el productor y en el mismo orden

El problema del productor/consumidor

```
public class Buffer { //Recurso
    private int[] b;
    public Buffer(int tam){
        b=new int[tam];
    }
    public void almacena(){}

    }
    public int extrae(){}

    }
}
```



C1: El productor no puede almacenar en el buffer, si está lleno

Necesitamos un semáforo que valga 0 cuando el buffer está lleno

El problema del productor/consumidor

```
import java.util.concurrent.*  
public class Buffer { //Recurso  
    private int[] b;  
    private Semaphore hayEspacio;  
    public Buffer(int tam){  
        b=new int[tam];  
        hayEspacio = new Semaphore(tam);  
    }  
    public void almacena () throws InterruptedException{  
        hayEspacio.acquire();  
    }  
    public int extrae(){  
        hayEspacio.release();  
    }  
}
```

El semáforo hayEspacio representa los huecos que hay en el buffer. Cuando no haya huecos su valor será cero. Se inicializa a tam, porque al principio todos los componentes del buffer están vacíos

C1: el productor espera si no hay espacio

C1: el consumidor indica que hay un nuevo hueco en el buffer

C1: El productor no puede almacenar en el buffer, si está lleno

Necesitamos un semáforo que valga 0 cuando el buffer está lleno

El problema del productor/consumidor

```
import java.util.concurrent.*;
public class Buffer { //Recurso
    private int[] b;
    private Semaphore hayEspacio;
    private Semaphore hayDatos = new Semaphore(0);
    public Buffer(int tam){
        b=new int[tam];
        hayEspacio = new Semaphore(tam);
    }
    public void almacena () throws InterruptedException{
        hayEspacio.acquire();

        hayDatos.release();
    }
    public int extrae() throws InterruptedException{
        hayDatos.acquire();

        hayEspacio.release();
    }
}
```

El semáforo hayDatos representa los datos que hay en el buffer. Cuando no haya datos su valor será cero. Se inicializa a 0, porque al principio el buffer está vacío

C2: el productor indica que hay un nuevo dato en el buffer

C2: el consumidor espera si no hay datos

C2: El consumidor no puede almacenar en el buffer, si está vacío

Necesitamos un semáforo que valga 0 cuando el buffer está vacío

El problema del productor/consumidor

```
import java.util.concurrent.*  
public class Buffer { //Recurso  
    private int[] b;  
    private Semaphore hayEspacio;  
    private Semaphore hayDatos = new Semaphore(0);  
    public Buffer(int tam){  
        b=new int[tam];  
        hayEspacio = new Semaphore(tam);  
    }  
    public void almacena () throws InterruptedException{  
        hayEspacio.acquire();  
  
        hayDatos.release();  
    }  
    public int extrae() throws InterruptedException{  
        hayDatos.acquire();  
  
        hayEspacio.release();  
    }  
}
```

Los semáforos **hayEspacio** y **hayDatos** son semáforos generales. Toman valores mayores que 1. Además satisfacen la relación:
$$\text{tam} = \text{hayDatos} + \text{hayEspacio}$$

El problema del productor/consumidor

```
Import java.util.concurrent.*
public class Buffer { //Recurso
    private int[] b;
    private Semaphore hayEspacio;
    private Semaphore hayDatos = new Semaphore(0);
    private Semaphore mutex = new Semaphore(1);
    public Buffer(int tam){
        b=new int[tam];
        hayEspacio = new Semaphore(tam);
    }
    public void almacena () throws InterruptedException{
        hayEspacio.acquire();
        mutex.acquire();

        mutex.release();
        hayDatos.release();
    }
    ....
}
```

La exclusión mutua se garantiza siempre con un semáforo binario inicializado a 1.

```
public int extrae() throws InterruptedException{
    hayDatos.acquire();
    mutex.acquire();

    mutex.release();
    hayEspacio.release();
}
```

R1: El buffer se utiliza en exclusión mutua

Necesitamos un semáforo para la exclusión mutua de buffer

El problema del productor/consumidor

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

public class Buffer { //Recurso
    private int[] b;
    private Semaphore hayEspacio;
    private Semaphore hayDatos = new Semaphore(0);
    private Semaphore mutex = new Semaphore(1);
    private int i = 0, j = 0; //i-indice productor, j-indice consumidor
    public Buffer(int tam){
        b=new int[tam];
        hayEspacio = new Semaphore(tam);
    }
    public void almacena () throws InterruptedException{
        hayEspacio.acquire();
        mutex.acquire();
        buffer[i] = dato;
        i = (i+1) % b.length; //el buffer se utiliza de forma circular
        mutex.release();
        hayDatos.release();
    }
    ....
}
```

La exclusión mutua se garantiza siempre con un semáforo binario inicializado a 1.

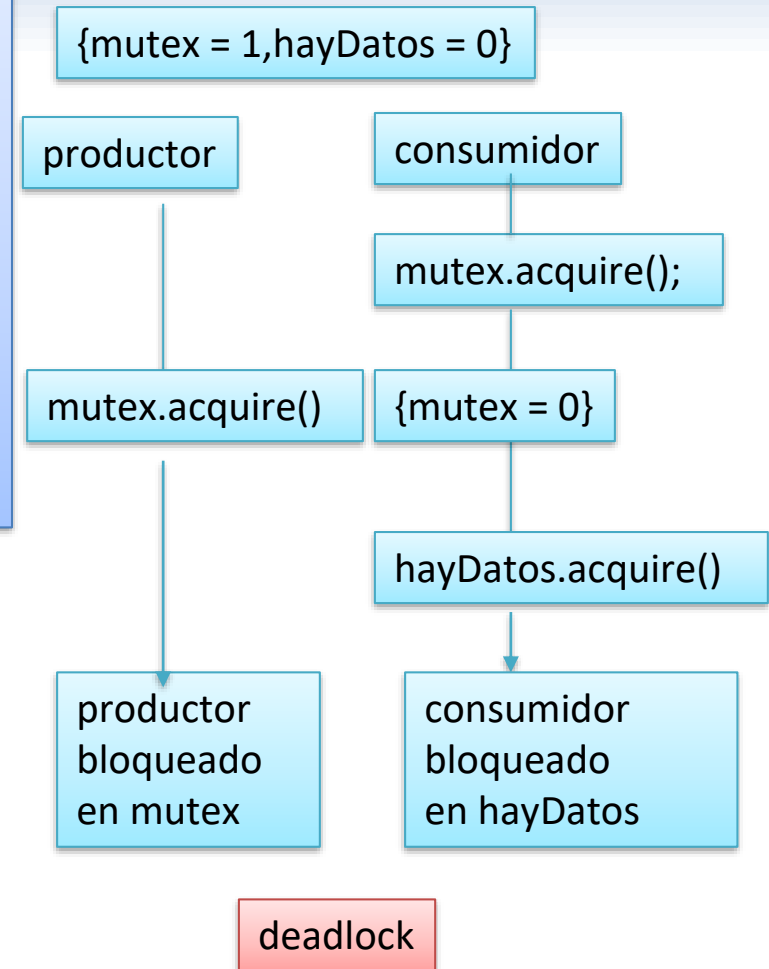
```
public int extrae() throws InterruptedException{
    hayDatos.acquire();
    mutex.acquire();
    int dato = buffer[j];
    j = (j+1) % b.length; //el buffer se utiliza de
                        // forma circular
    mutex.release();
    hayEspacio.release();
}
}
```


El problema del productor/consumidor

No podemos intercambiar las instrucciones del preprotocolo

```
...  
public void almacena () throws InterruptedException{  
    mutex.acquire();  
    hayEspacio.acquire();  
    buffer[i] = dato;  
    i = (i+1) % b.length;  
    mutex.release();  
    hayDatos.release();  
}  
...  
}
```

```
public int extrae() throws InterruptedException{  
    mutex.acquire();  
    hayDatos.acquire();  
    int dato = buffer[j];  
    j = (j+1) % b.length;  
    mutex.release();  
    hayEspacio.release();  
}  
}
```

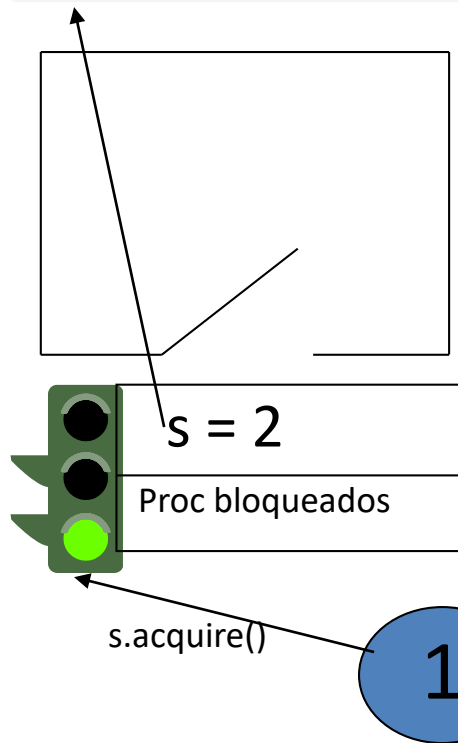


Semáforos generales con binarios

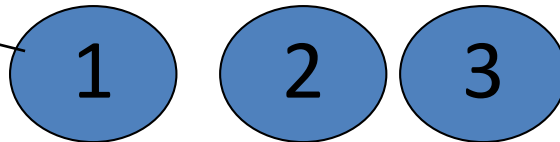
- La implementación de semáforos generales mediante semáforos binarios significa que los semáforos generales no añaden expresividad al lenguaje.
- Un semáforo general se caracteriza porque
 - puede tener valores superiores a 1.
 - el valor del semáforo representa el número de operaciones **acquire()** sucesivas que admite antes de suspender a la hebra.

Semáforos generales con binarios

s es el guardián de la sección crítica

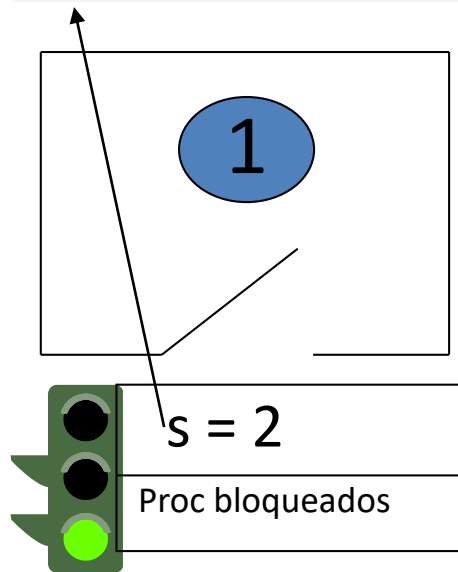


Acción	P1	P2	P3	S
P1 hace s.acquire()				2

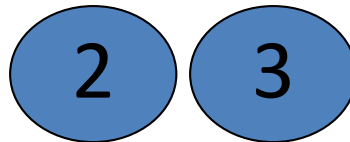


Semáforos generales con binarios

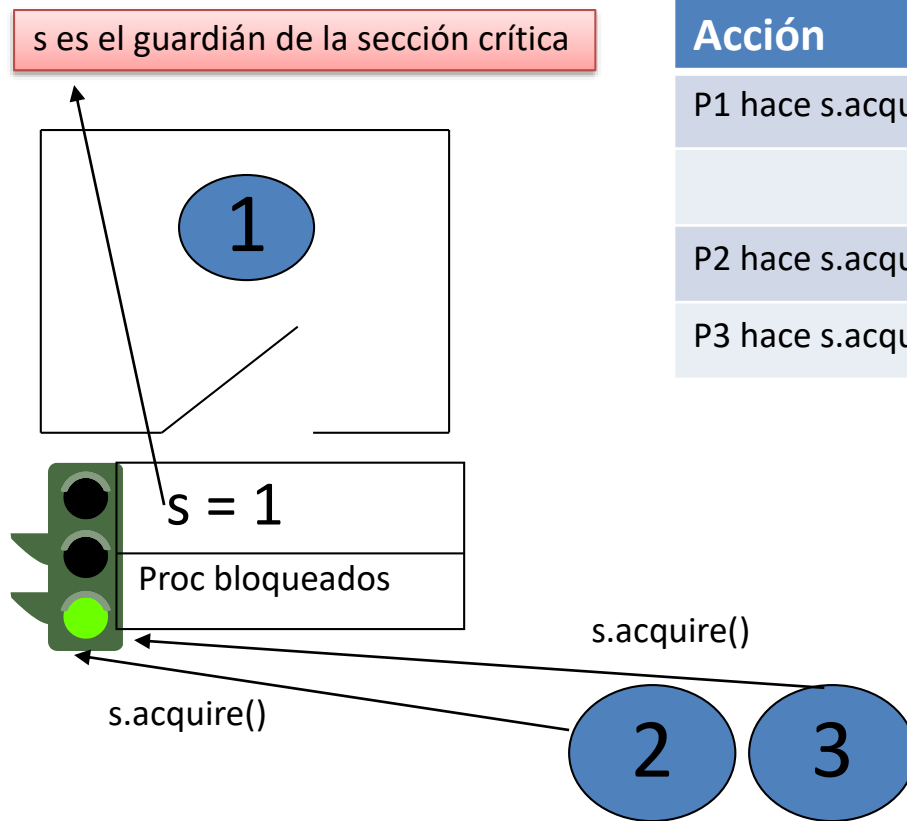
s es el guardián de la sección crítica



Acción	P1	P2	P3	S
P1 hace s.acquire()				2
	En SC1			1

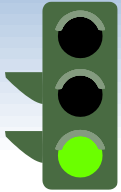


Semáforos generales con binarios

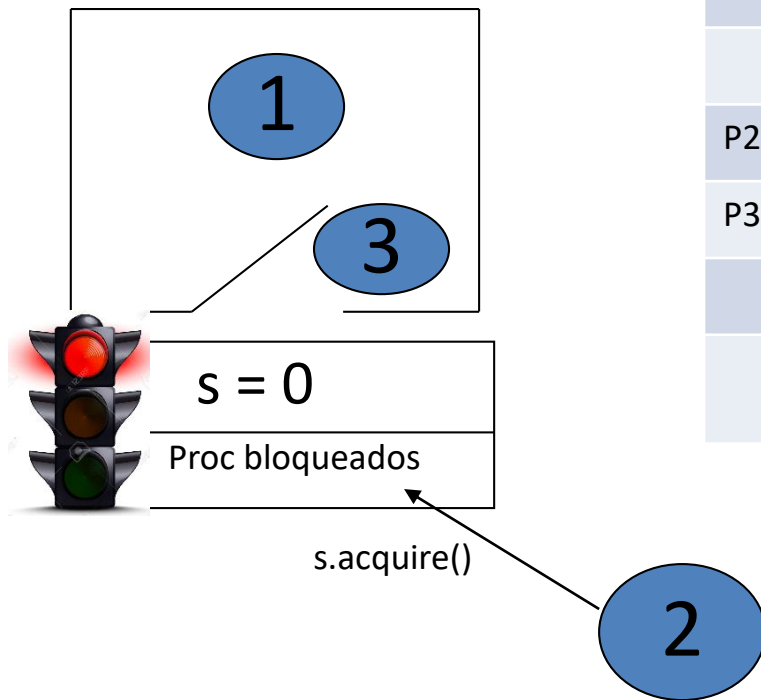


Acción	P1	P2	P3	S
P1 hace s.acquire()				2
	En SC1			1
P2 hace s.acquire()				1
P3 hace s.acquire()				1

Semáforos generales con binarios



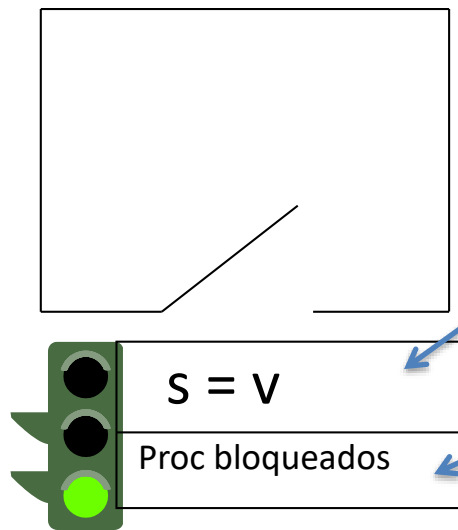
s es el guardián de la sección crítica



Acción	P1	P2	P3	S
P1 hace s.acquire()				2
	En SC1			1
P2 hace s.acquire()				1
P3 hace s.acquire()				1
			En SC3	0
		Bloqueado en s		0

Semáforos generales con binarios

- Para representar este comportamiento con semáforos binarios necesitamos



- Un número que indique el número de `acquire()` que pueden realizarse sin suspender a la hebra
- Una estructura de procesos suspendidos, que puede estar vacía o no.

Semáforos generales con binarios

```
import java.util.concurrent.*;  
public class SemGen {  
    private Semaphore espera;//binario  
    private int valor;  
    private Semaphore mutex; //binario  
}
```

- ▶ Las operaciones sobre los objetos de tipo SemGen deben asegurar que

espera = 0 sii **valor = 0**

Hace falta **mutex** para asegurar que la variable entera **valor** se utilice siempre en exclusión mutua

Semáforos generales con binarios

- ▶ Las operaciones sobre los objetos de tipo SemGen deben asegurar que

espera = 0 sii valor = 0

```
import java.util.concurrent.*;
public class SemGen {
    private Semaphore espera; //binario
    private int valor;
    private Semaphore mutex; //binario

    public SemGen(int valor){
        this.valor = valor;
        mutex = new Semaphore(1,true);
        if (valor == 0) espera = new Semaphore(0,true);
        else espera = new Semaphore(1,true);
    }
    ...
}
```

Semáforos generales con binarios

- ▶ Las operaciones sobre los objetos de tipo SemGen deben asegurar que

espera = 0 si valor = 0

```
import java.util.concurrent.*;  
public class SemGen {  
    private Semaphore espera; //binario  
    private int valor;  
    private Semaphore mutex; //binario
```

```
    public void acquireGen() throws InterruptedException {  
        espera.acquire();  
        mutex.acquire();  
        valor--;  
        if (valor > 0) espera.release();  
        mutex.release();  
    }  
}
```

espero, si el semáforo está cerrado

Ojo, después de completar acquire() sobre un semáforo binario, éste siempre vale 0!!

Decremento el valor del semáforo

Si después del decremento, el valor del semáforo es mayor que 0, abro el semáforo (que estaba a 0)

Semáforos generales con binarios

- ▶ Las operaciones sobre los objetos de tipo SemGen deben asegurar que

espera = 0 sii valor = 0

```
import java.util.concurrent.*;
public class SemGen {
    private Semaphore espera; //binario
    private int valor;
    private Semaphore mutex; //binario
    ....
    public void releaseGen() throws InterruptedException{
        mutex.acquire();
        valor++;
        if (valor == 1) espera.release();
        mutex.release();
    }
}
```

Incremento el valor del semáforo

Si después del incremento, el valor del semáforo es 1, abro el semáforo (es necesario ya que antes valor = 0, y espera = 0)

El barbero dormilón

- El problema del barbero dormilón es una representación del problema del *productor/consumidor* con un buffer no acotado (que nunca se llena) resuelto sólo con *semáforos binarios*.
- Sigue la filosofía de la implementación de semáforos generales con binarios vista antes.
- Enunciado:
 - Supongamos que tenemos una barbería de las antiguas, en la que hay un solo barbero (proceso consumidor) que está pelando continuamente a sus clientes (datos).
 - Pero como el barbero se cansa de trabajar, cuando ve que no tiene clientes se echa un ratito a dormir....

El barbero dormilón



La barbería consta de una **sala de espera** (el buffer) con una capacidad ilimitada (puede albergar a cualquier número de clientes) y de una **sala de pelar** donde pela el barbero.

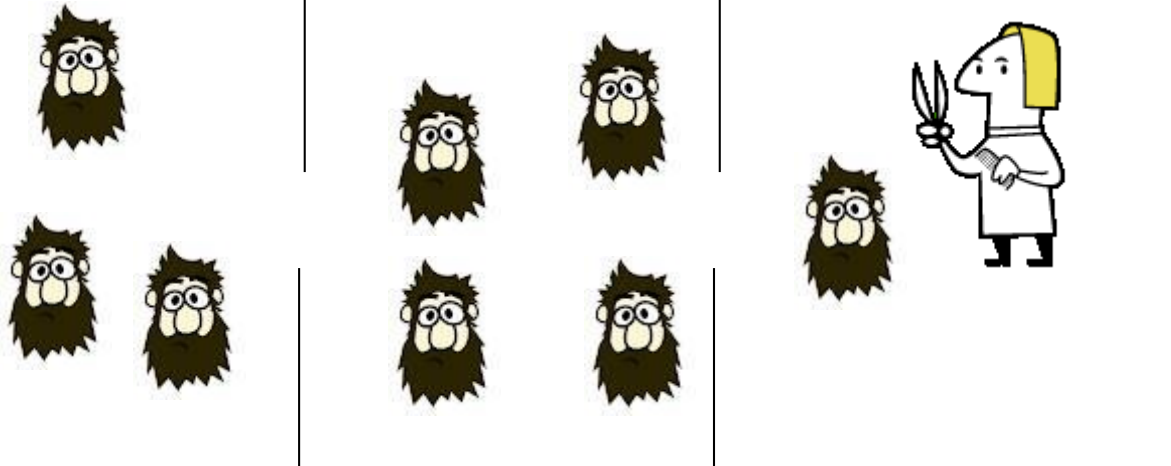
Lo **datos** son los clientes, que van a la barbería a arreglarse un poco

El barbero dormilón

Sala de espera

Sala de pelar

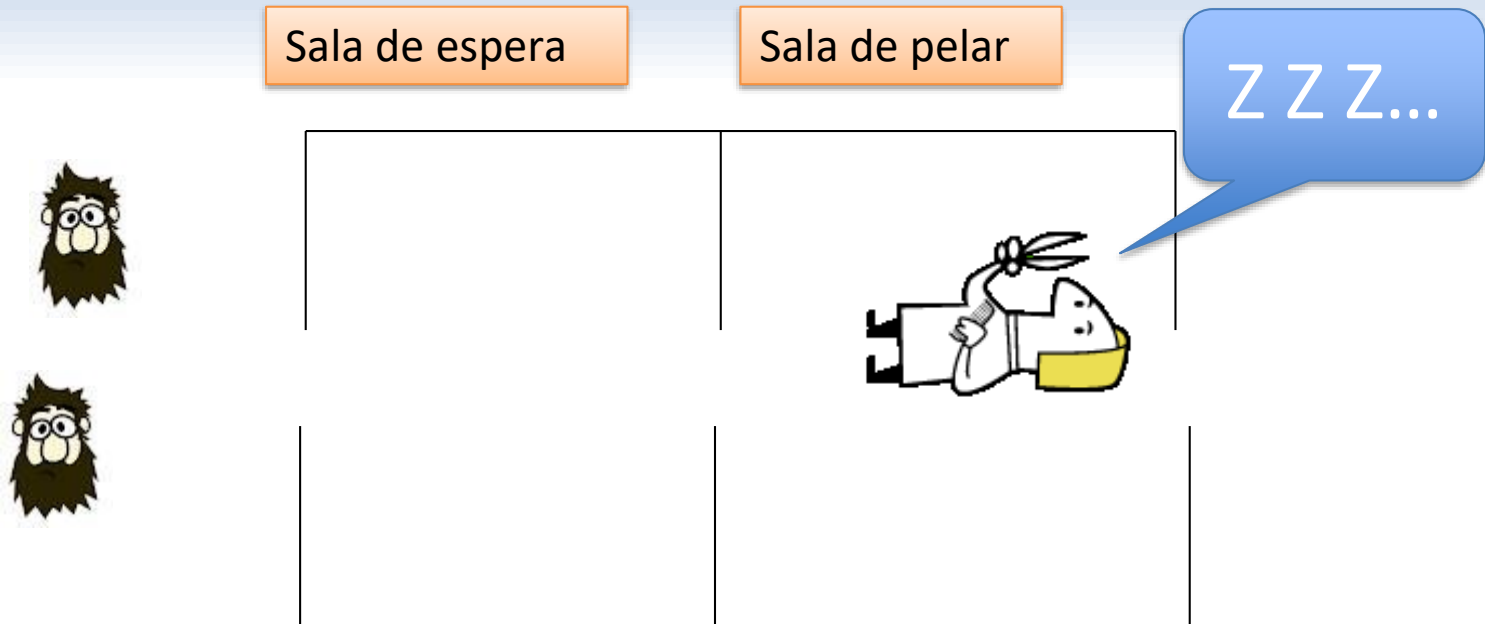
Es un sistema que no termina nunca



El **barbero** (el consumidor) está pelando continuamente.

El **entorno** es el productor que suministra continuamente clientes a la barbería. Como el buffer es infinito, el entorno no tiene que bloquearse nunca.

El barbero dormilón



La condición de sincronización es que el barbero no puede pelar un cliente que no existe (o en la versión productor/consumidor, que el consumidor no puede extraer de un buffer vacío

El **barbero** siempre tiene sueño y si no hay clientes está durmiendo

Barbero dormilón

```
import java.util.concurrent.*;
public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);

    public void nuevoCliente(){
        //llega un nuevo cliente a la sala de espera
    }

    public void pelar(){
        //el barbero pela a un cliente, si hay alguien esperando
        //en otro caso sigue durmiendo
    }
}
```

- El buffer se representa con una variable entera **n**. Sólo nos interesa el número de clientes que hay esperando. La variable **n** puede tomar los valores 0, 1, 2, ...
- Inicialmente vale 0 puesto que no hay ningún cliente en la sala de espera

Barbero dormilón

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);

    public void nuevoCliente(){
        //llega un nuevo cliente a la sala de espera
    }

    public void pelar(){
        //el barbero pela a un cliente, si hay alguien esperando
        //en otro caso sigue durmiendo
    }
}
```

- Necesitamos un **mutex** para asegurar que la variable **n** se utiliza siempre en exclusión mutua

Barbero dormilón

```
import java.util.concurrent.*;
public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);

    public void nuevoCliente(){
        //llega un nuevo cliente a la sala de espera
    }

    public void pelar(){
        //el barbero pela a un cliente, si hay alguien esperando
        //en otro caso sigue durmiendo
    }
}
```

- Necesitamos un semáforo de sincronización, que llamamos **espera**, y que valga 0 si no hay ningún cliente en la sala de espera. Por eso se inicializa a 0.

Barbero dormilón

```
public class Barbero extends Thread{
    private Barberia b;
    public Barbero(Barberia b){
        this.b = b;
    }
    public void run(){
        while (true)
            b.pelar();
    }
}

public class Entorno extends Thread{
    private Barberia b;
    public Entorno(Barberia b){
        this.b = b;
    }
    public void run(){
        while (true)
            b.nuevoCliente();
    }
}
```

```
public static void main(String[] args){
    Barberia b = new Barberia();
    Barbero barbero = new Barbero(b);
    Entorno entorno = new Entorno(b);
    entorno.start();
    barbero.start();
}
```

Hemos eliminado las instrucciones try/catch en el código de las hebras para simplificar el código

Barbero dormilón: comportamiento del barbero y del entorno

- El barbero está continuamente pelando...
 - Cuando se queda desocupado, mira en la sala de espera, y
 - si hay clientes, llama al siguiente cliente, lo pela, y cuando termina, comienza de nuevo el ciclo..
 - pero si la sala de espera está vacía, el barbero aprovecha para descansar en la sala de pelar ...
- Cuando llega un cliente a la sala de espera
 - si hay clientes esperando, se pone en la cola...
 - pero si se encuentra la sala de espera vacía, despierta al barbero

Barbero dormilón

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);

    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }

    public void pelar(){
        //el barbero pela a un cliente, si hay alguien esperando
        //en otro caso sigue durmiendo
    }
}
```

- Cuando llena un nuevo cliente, se actualiza el buffer en exclusión mutua.
- Si es el primer cliente, despierta al barbero

Barbero dormilón

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);
    private boolean primera = true;
    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }
    public void pelar() throws InterruptedException {
        if (primera) { espera.acquire(); primera=false; }
        mutex.acquire();
        n--;
        if (n == 0) espera.acquire();
        mutex.release();
    }
}
```

- El barbero está dormido inicialmente. Utilizamos la variable **primera** para este comportamiento solo ocurra al principio del sistema
- Cuando lo despierta un cliente, pasa a pelarlo
- Si la sala de espera se queda vacía se vuelve a dormir
- Si hay más clientes, los sigue pelando...
- Cuando el barbero decrementa **n**, la variable **vale al menos 0...**

Barbero dormilón: primer intento

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0, true);
    private Semaphore mutex = new Semaphore(1, true);
    private boolean primera = true;
    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }
    public void pelar() throws InterruptedException {
        if (primera) { espera.acquire(); primera = false; }
        mutex.acquire();
        n--;
        if (n == 0) espera.acquire();
        mutex.release();
    }
}
```

- La idea de la solución es que el productor (entorno) sólo despierta al consumidor (**espera.release()**) cuando produce sobre un buffer que está vacío.
- De forma simétrica, el barbero (consumidor) solo ejecuta **espera.acquire()** cuando se encuentra el buffer (**n**) vacío. El resto del tiempo simplemente va al buffer, coge el dato, y lo consume.
- De esta forma, se minimiza el número de operaciones **acquire()/release()** ejecutadas, con respecto a la solución que utiliza semáforos generales

Barbero dormilón: primer intento

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0, true);
    private Semaphore mutex = new Semaphore(1, true);
    private boolean primera = true;
    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }
    public void pelar() throws InterruptedException {
        if (primera) { espera.acquire(); primera= false; }
        mutex.acquire();
        n--;
        if (n == 0) espera.acquire();
        mutex.release();
    }
}
```

- Esta solución *puede bloquear*.
- El barbero no puede bloquearse en la sección crítica!!!

Barbero dormilón: primer intento

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0, true);
    private Semaphore mutex = new Semaphore(1, true);
    private boolean primera = true;

    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }

    public void pelar() throws InterruptedException {
        if (primera) {espera.acquire(); primera=false;}
        mutex.acquire();
        n--;
        mutex.release();
        if (n == 0) espera.acquire();
    }
}
```

- ¿y si sacamos la instrucción de la sección crítica?

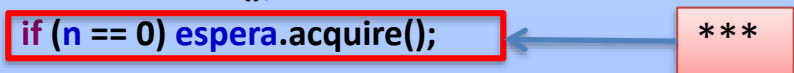
Barbero	Entorno	Variables
pc inicial	pc inicial	n=0,espera=0
	1 iteración	n=1,espera=1
Hasta ***		n=0,espera=0
	1 iteración	n=1,espera=1
1 iteración hasta ***		n=0,espera=1
1 iteración hasta ***		n=-1,espera=0

ERROR!! El barbero ha consumido de un buffer vacío

Barbero dormilón: segundo intento

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0, true);
    private Semaphore mutex = new Semaphore(1, true);
    private boolean primera = true;
    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }
    public void pelar() throws InterruptedException {
        if (primera) {espera.acquire(); primera=false;}
        mutex.acquire();
        n--;
        mutex.release();
        if (n == 0) espera.acquire();
    }
}
```



- El error se produce porque en *** se intercala el proceso entorno, y modifica el valor de *n*
- El valor de *n* que nos interesa es el que *tenía* cuando el barbero ejecutó la sección crítica

Barbero dormilón: segundo intento

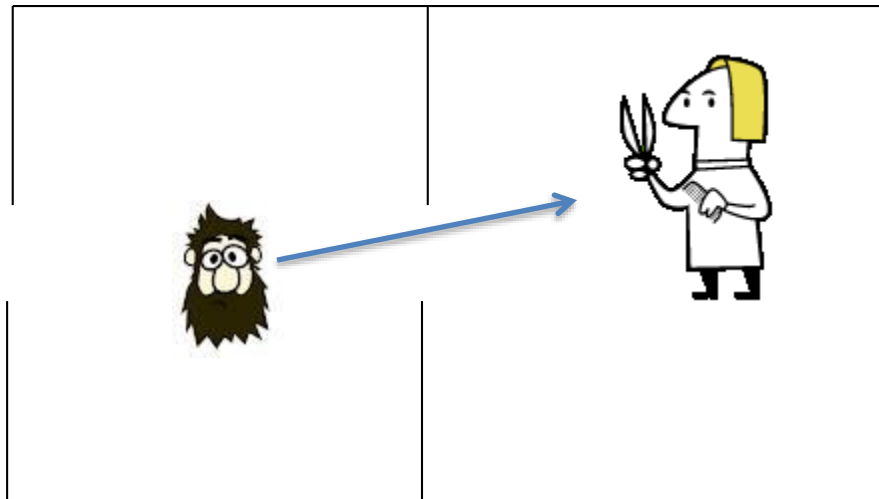
```
import java.util.concurrent.*;
public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);
    private boolean primera = true;
    public void nuevoCliente() throws InterruptedException {
        mutex.acquire();
        n++;
        if (n==1) espera.release();
        mutex.release();
    }
    public void pelar() throws InterruptedException {
        if (primera) {espera.acquire();primera=false}
        mutex.acquire();
        n--;
        int m = n;
        mutex.release();
        if (m == 0) espera.acquire();
    }
}
```

- Una posible solución es guardar el valor de n en una variable local m , y ..
- luego hacer el test con m

Barbero dormilón: un caso extremo

Sala de espera

Sala de pelar



Supongamos que tenemos el siguiente escenario:

Inicialmente ($\text{espera} = 0, n=0$), el barbero está durmiendo... $\rightarrow \text{espera.acquire}()$

Llega el primer cliente y lo despierta $\rightarrow \text{espera.release}()$ ($n=1, \text{espera} = 0$)

Mientras el barbero está pelando ($n=0, \text{espera} = 0$) llega un nuevo cliente.

Ve la sala de espera vacía ($n = 0$), y supone que el barbero está durmiendo, por lo que intenta despertarlo $\rightarrow \text{espera.release}()$ ($n=1, \text{espera} = 1$)

Barbero dormilón: un caso extremo



Cuando se va el cliente que estaba pelando, el barbero pasa a pelar al nuevo cliente, pero como cuando pasó al anterior, la sala de espera quedó vacía ($m = 0$) ejecuta `espera.acquire()` para volver a poner `espera = 0`. Este escenario puede repetirse indefinidamente....

Barbero dormilón: un caso extremo

- En cada iteración, se ejecutan un par de instrucciones `acquire()/release()`, pero ¿es esto necesario?
 - Los clientes siempre llegan a una sala de espera vacía, y
 - el barbero siempre consume de un buffer que NO está vacío
- El problema está en que cuando llega un cliente, y ve la sala de espera vacía ejecuta `espera.release()`, pero no sabe si
 - el barbero está durmiendo (ha ejecutado `espera.acquire()`),
 - o está pelando a otro cliente (no le ha dado tiempo a hacer `espera.acquire()`)

Barbero dormilón: un caso extremo

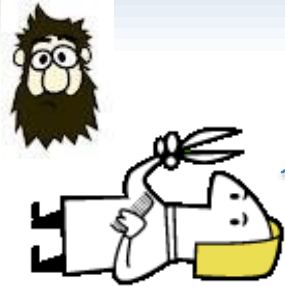
- Una solución es distinguir entre esos dos casos, obligando al barbero a **dormir en la sala de espera**.
- De esta forma cuando llega un cliente si:
 - Hay más clientes, espera con ellos
 - Si está vacía, el barbero está pelando un cliente en la sala de pelar
 - Si el barbero está durmiendo (ha ejecutado `espera.acquire()`), lo despierta (ejecuta `espera.release()`)

Barbero dormilón: un caso extremo

Sala de espera

Sala de pelar

ZZZ...



Sala de espera

Sala de pelar



Sala de espera

Sala de pelar



Barbero dormilón: un caso extremo

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

public class Barberia {
    private int n = 0;
    private Semaphore espera = new Semaphore(0,true);
    private Semaphore mutex = new Semaphore(1,true);
    private boolean primera = true;
    public void pelar() throws InterruptedException {
        mutex.acquire();
        n--;
        if (n == -1){
            mutex.release();
            espera.acquire();
            mutex.acquire();
        }
        mutex.release();
    }
}
```

Ojo, aquí hay otra forma de evitar bloquearse en la sección crítica

Distinguimos los casos
 $n = -1$ (barbero dormido)
 $n = 0$ (sala vacía)
 $n > 0$ (sala con clientes)

```
public void nuevoCliente() throws InterruptedException {
    mutex.acquire();
    n++;
    if (n==0) espera.release();
    mutex.release();
}
}
```

Si el barbero se adelanta, e intenta consumir de un buffer vacío, se bloquea

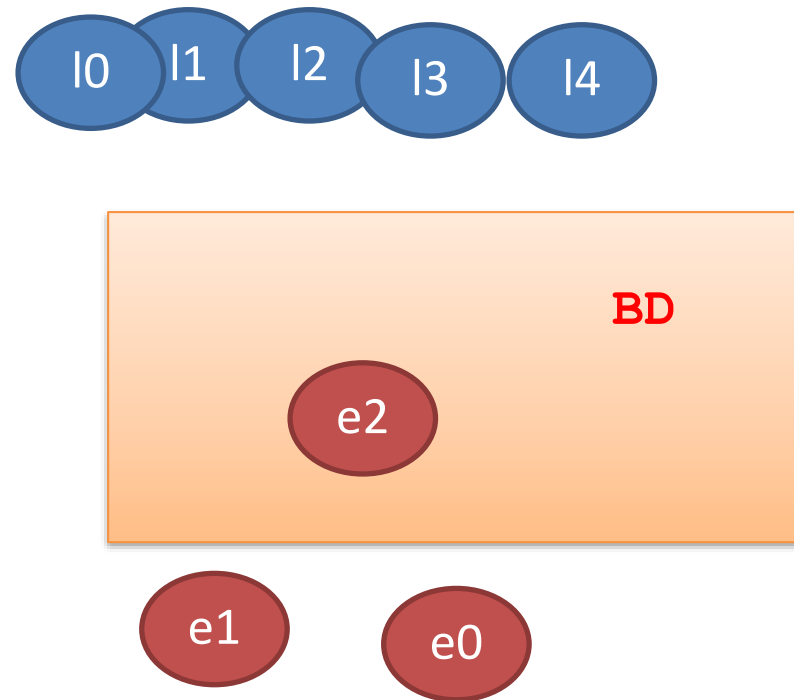
Si el productor detecta que el barbero duerme, lo despierta

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.

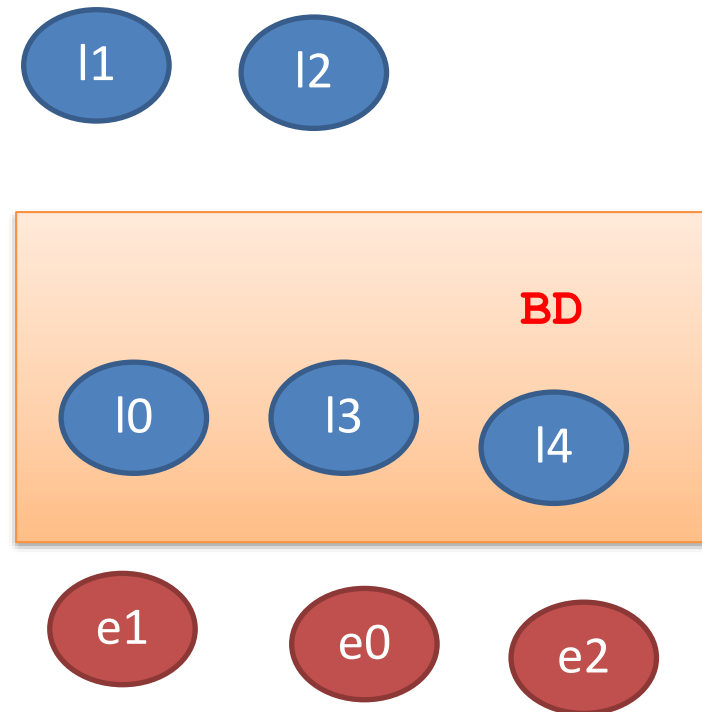
Lectores/Escritores

- 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 g;
    public Lector(int id,GestorBD g){
        this.id = id; this.g = g;start(); }
    public void run(){
        while (true){
            try{
                g.entraLector(id);
                //lector id en la BD
                g.saleLector(id);
            }catch(InterruptedException ie){}
        }
    }
}
```

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

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

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

Lectores/Escritores

- Clase GestorBD
- La condición de sincronización de los escritores puede modelarse con un semáforo binario *escribiendo* que garantice la exclusión mutua de los escritores en la BD

```
private Semaphore escribiendo = new Semaphore(1,true);
```

```
public void entraEscritor(int id) throws InterruptedException{
    escribiendo.acquire();
}
public void saleEscritor(int id){
    escribiendo.release();
}
```

Lectores/Escritores

- Clase GestorBD
- La condición de sincronización de los lectores puede modelarse con una variable entera *nLectores*, que cuenta el número de lectores que hay en la BD, y un semáforo *mutex* que garantiza el uso en exclusión mutua de *nLectores*.

```
private int nLectores= 0;  
private Semaphore mutex= new Semaphore(1,true);
```

```
public void entraLector(int id) throws InterruptedException{  
    mutex.acquire();  
    nLectores++;  
    if (nLectores==1) escribiendo.acquire();  
    mutex.release();  
}
```

Lectores/Escritores

```
public void entraLector(int id) throws InterruptedException{  
    mutex.acquire();  
    nLectores++;  
    if (nLectores==1) escribiendo.acquire();  
    mutex.release();  
}
```

- Cuando un lector quiere entrar en la BD,
 - si no hay ningún escritor en la BD, entonces escribiendo = 1
 - si es el primer lector que lo intenta, pone nLectores a 1, cierra el semáforo escribiendo y entra
 - Hay uno o más lectores en la BD sii escribiendo = 0
 - en otro caso, incrementa nLectores, y entra
 - Si hay un escritor en la BD, entonces escribiendo = 0
 - si es el primer lector que lo intenta, pone nLectores a 1, y se bloquea en escribiendo
 - si no es el primer lector, se bloquea en mutex

Lectores/Escritores

```
public void saleLector(int id) throws InterruptedException{  
    mutex.acquire();  
    nLectores--;  
    if (nLectores==0) escribiendo.release();  
    mutex.release();  
}
```

- Cuando un lector sale de la BD,
 - si es el último lector que queda en la BD, decrementa **nLectores** a 0, libera el semáforo **escribiendo** (**escribiendo = 1**) y sale
 - Si despierta a un proceso debe ser necesariamente un escritor
 - si no es el último lector que lo intenta sale, simplemente decrementa **nLectores**, y sale

Lectores/Escritores

```
public void entraEscritor(int id) throws InterruptedException{
    escribiendo.acquire();
}
public void saleEscritor(int id){
    escribiendo.release();
}
```

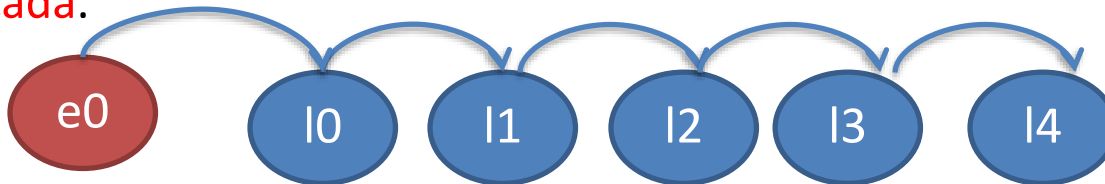
- Cuando un escritor quiere entrar en la BD
 - Si escribiendo = 1, entonces la BD está vacía. En este caso, pone escribiendo a 0, e impide que cualquier otro proceso (lector o escritor) entre en la BD
 - Si escribiendo = 0, hay alguien en la BD. En este caso, espera en el semáforo escribiendo, hasta que pueda entrar en solitario.

Lectores/Escritores

```
public void saleEscritor(int id){  
    escribiendo.release();  
}
```

```
public void entraLector(int id) throws InterruptedException{  
    mutex.acquire();  
    nLectores++;  
    if (nLectores==1) escribiendo.acquire();  
    mutex.release();  
}
```

- Cuando un escritor sale de la BD, libera el semáforo escribiendo
 - lo que puede despertar a un escritor,
 - o al primer lector, que espera en la sección crítica. En este caso, el lector despertado, hace release() sobre mutex, lo que puede despertar a otro lector, que hace lo mismo, y así sucesivamente. Esta forma de despertar a una serie indeterminada de procesos se denomina **despertado en cascada**.



Lectores/Escritores

```
public void entraLector(int id) throws InterruptedException{
    mutex.acquire();
    nLectores++;
    if (nLectores==1) escribiendo.acquire();
    mutex.release();
}

public void saleLector(int id) throws InterruptedException{
    mutex.acquire();
    nLectores--;
    if (nLectores==0) escribiendo.release();
    mutex.release();
}

public void entraEscritor(int id) throws InterruptedException{
    escribiendo.acquire();
}

public void saleEscritor(int id){
    escribiendo.release();
}
```

Es fácil ver que con esta solución los **lectores pueden hacerse con el control de la BD**, impidiendo el acceso de forma indefinida a los escritores.

Lectores/Escritores: versión justa para los escritores

- La idea de esta solución es la siguiente:
 - Aunque haya lectores en la BD, si hay algún escritor esperando, cualquier lector que quiera entrar debe esperar a que entren los escritores que esperan
- Esta nueva versión se basa en la solución anterior, pero se complica un poco...

```
private int nLectores= 0;  
private Semaphore mutex1= new Semaphore(1,true);  
private Semaphore escribiendo = new Semaphore(1,true);  
private int nEscritores= 0;  
private Semaphore mutex2= new Semaphore(1,true);  
private Semaphore leyendo= new Semaphore(1,true);  
private Semaphore mutex3= new Semaphore(1,true);  
//todos los semáforos son binarios
```

Lectores/Escritores : versión justa para los escritores

```
public void entraEscritor(int id) throws InterruptedException{  
    mutex2.acquire();  
    nEscritores++;  
    if (nEscritores == 1) leyendo.acquire();  
    mutex2.release();  
    escribiendo.acquire();  
}
```

Cuando llega un escritor, incrementa la variable **nEscritores**.

- Si es el primero, cierra el semáforo **leyendo** (para bloquear a los lectores que lleguen a continuación)
- Si no es el primero no ejecuta **leyendo.acquire()** para no bloquearse.
- A continuación, accede, si es posible, a la BD en exclusión mutua (como en la versión anterior)

Lectores/Escritores : versión justa para los escritores

```
public void entraEscritor(int id) throws InterruptedException{  
    mutex2.acquire();  
    nEscritores++;  
    if (nEscritores == 1) leyendo.acquire();  
    mutex2.release();  
    escribiendo.acquire();  
}
```

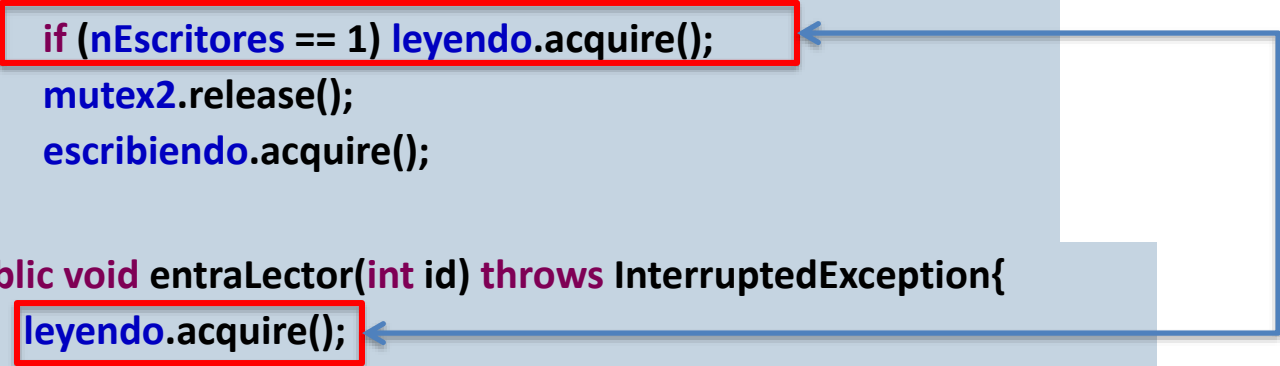
- El semáforo **leyendo** NO funciona igual que **escribiendo**.
- El primer escritor que llega **debe cerrarlo**, pero no bloquearse en él, es decir, en ese momento se espera que **leyendo** valga **1**
- Los escritores se deben bloquear en **escribiendo** (como en la versión anterior)

Lectores/Escritores :

versión justa para los escritores

```
public void entraEscritor(int id) throws InterruptedException{
    mutex2.acquire();
    nEscritores++;
    if (nEscritores == 1) leyendo.acquire();
    mutex2.release();
    escribiendo.acquire();
}

public void entraLector(int id) throws InterruptedException{
    leyendo.acquire();
    mutex1.acquire();
    nLectores++;
    if (nLectores==1) escribiendo.acquire();
    mutex1.release();
    leyendo.release();
}
```



Un lector que llega, si **leyendo** está cerrado, espera en **leyendo**.
Por lo tanto si hay **escritores esperando**, el lector no entra en la BD.

Lectores/Escritores : versión justa para los escritores

```
public void saleEscritor(int id) throws InterruptedException{  
    mutex2.acquire();  
    nEscritores--;  
    if (nEscritores == 0) leyendo.release();  
    mutex2.release();  
    escribiendo.release();  
}
```

- Un escritor que deja la BD decrementa **nEscritores**.
- Si ya no hay escritores esperando, abre el semáforo **leyendo**.
- En cualquier caso, libera la exclusión mutua de la BD


Lectores/Escritores : versión justa para los escritores

```
public void saleLector(int id) throws InterruptedException{  
    mutex1.acquire();  
    nLectores--;  
    if (nLectores==0) escribiendo.release();  
    mutex1.release();  
}
```


- El protocolo de salida para los lectores no cambia.

Lectores/Escritores : versión justa para los escritores

```
public void entraEscritor(int id) throws InterruptedException{  
    mutex2.acquire();  
    nEscritores++;  
    if (nEscritores == 1) leyendo.acquire();  
    mutex2.release();  
    escribiendo.acquire();  
}
```



```
public void entraLector(int id) throws InterruptedException{  
    leyendo.acquire();  
    mutex1.acquire();  
    nLectores++;  
    if (nLectores==1) escribiendo.acquire();  
    mutex1.release();  
    leyendo.release();  
}
```



Supongamos que un lector y el primer escritor están ejecutando simultáneamente sus protocolos de entrada. En este caso, como el semáforo leyendo está cerrado, y el primer escritor se bloquea en leyendo.

Si más lectores quieren ejecutar su protocolo de entrada, también esperan en leyendo. ¿cómo aseguramos que cuando el lector sale de su protocolo de entrada despierta al escritor y no a un lector?

Lectores/Escritores : versión justa para los escritores

```
public void entraEscritor(int id) throws InterruptedException{  
    mutex2.acquire();  
    nEscritores++;  
    if (nEscritores == 1) leyendo.acquire();  
    mutex2.release();  
    escribiendo.acquire();  
}
```

```
public void entraLector(int id) throws InterruptedException{  
    mutex3.acquire();  
    leyendo.acquire();  
    mutex1.acquire();  
    nLectores++;  
    if (nLectores==1) escribiendo.acquire();  
    mutex1.release();  
    leyendo.release();  
    mutex3.release();  
}
```

Si utilizamos otro semáforo **mutex3** se asegura que si hay un lector ejecutando su protocolo de entrada, el resto de los lectores esperan en **mutex3**, por lo que cuando el lector ejecuta **leyendo.release()** despierta al escritor, si está bloqueado. Este código resuelve el problema, aunque se utilicen semáforos que no sean justos.

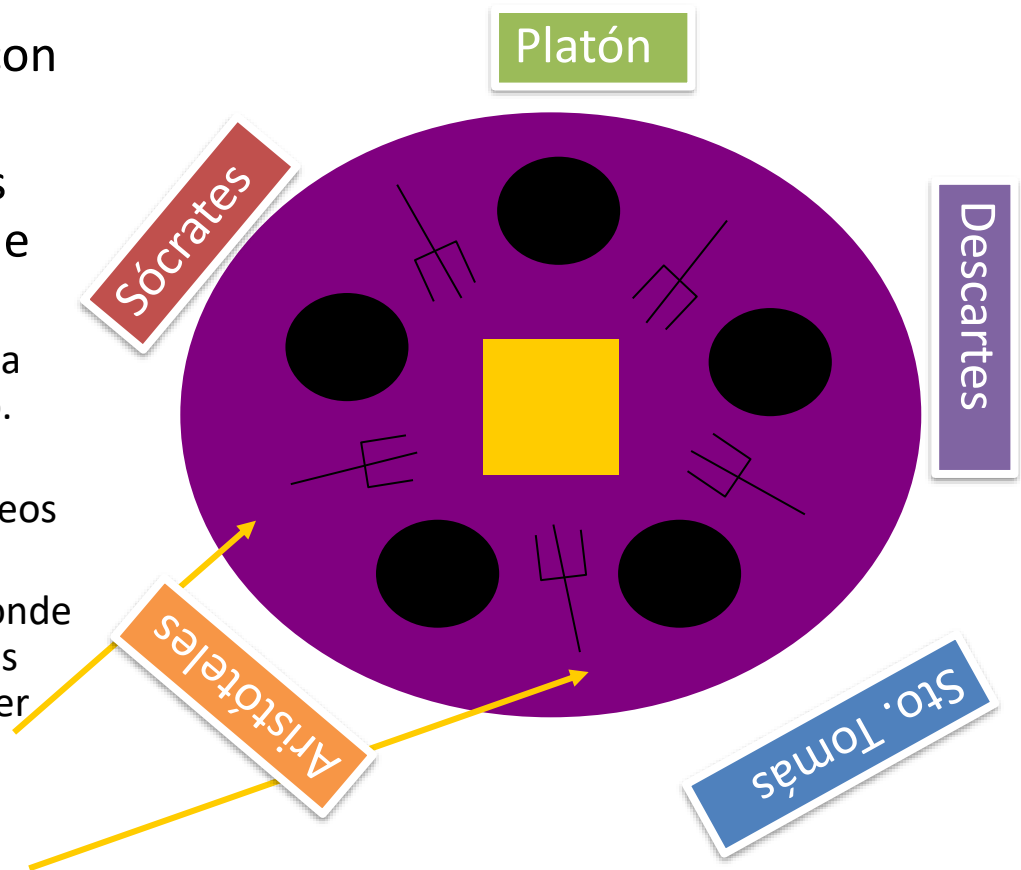
El problema de los filósofos

- $N = 5$ 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  
        }  
    }  
}
```

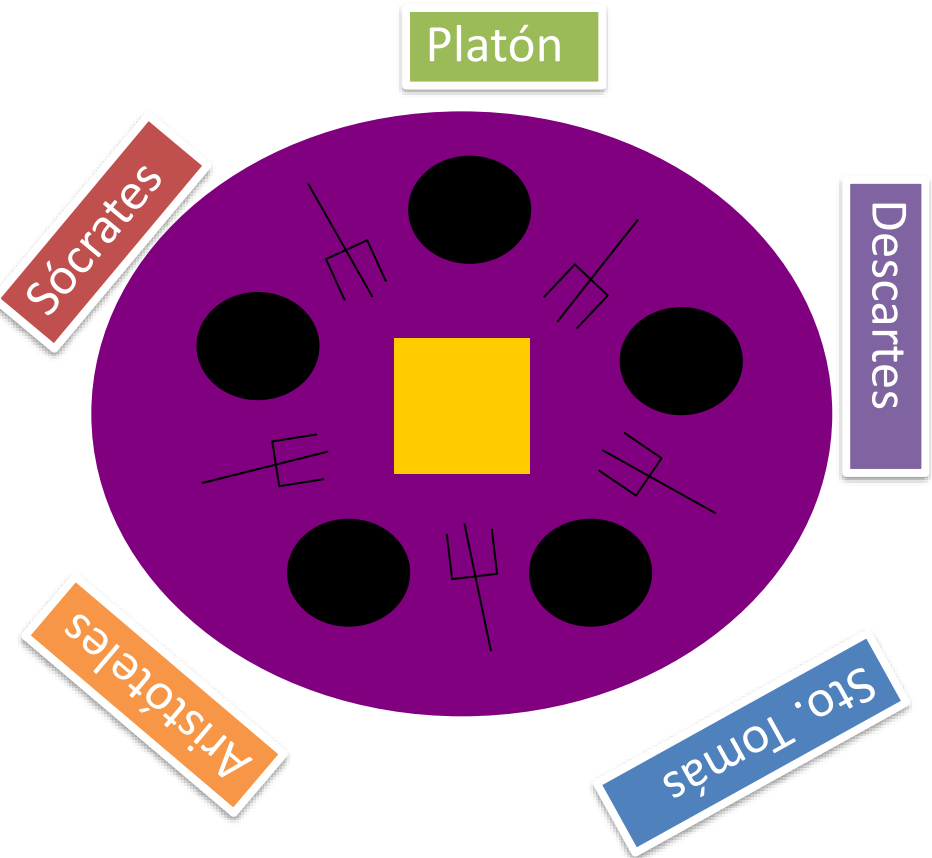
El problema de los filósofos

- 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

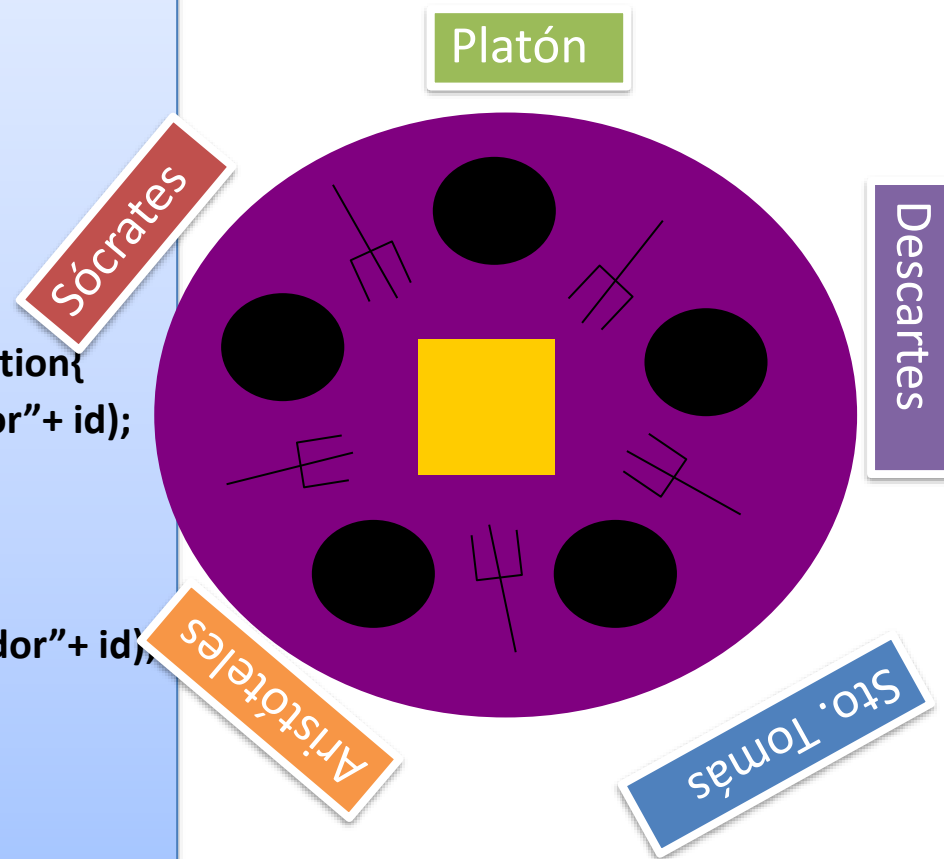
- Los tenedores son los **recursos** del sistema que los procesos deben utilizar en exclusión mutua (no puede haber dos filósofos utilizando simultáneamente el tenedor que comparten)
- Así que de forma natural cada tenedor puede representarse con un semáforo binario:
 - Si está abierto (1), el tenedor correspondiente está libre
 - Si está cerrado (0), el tenedor está siendo utilizado.



El problema de los filósofos

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

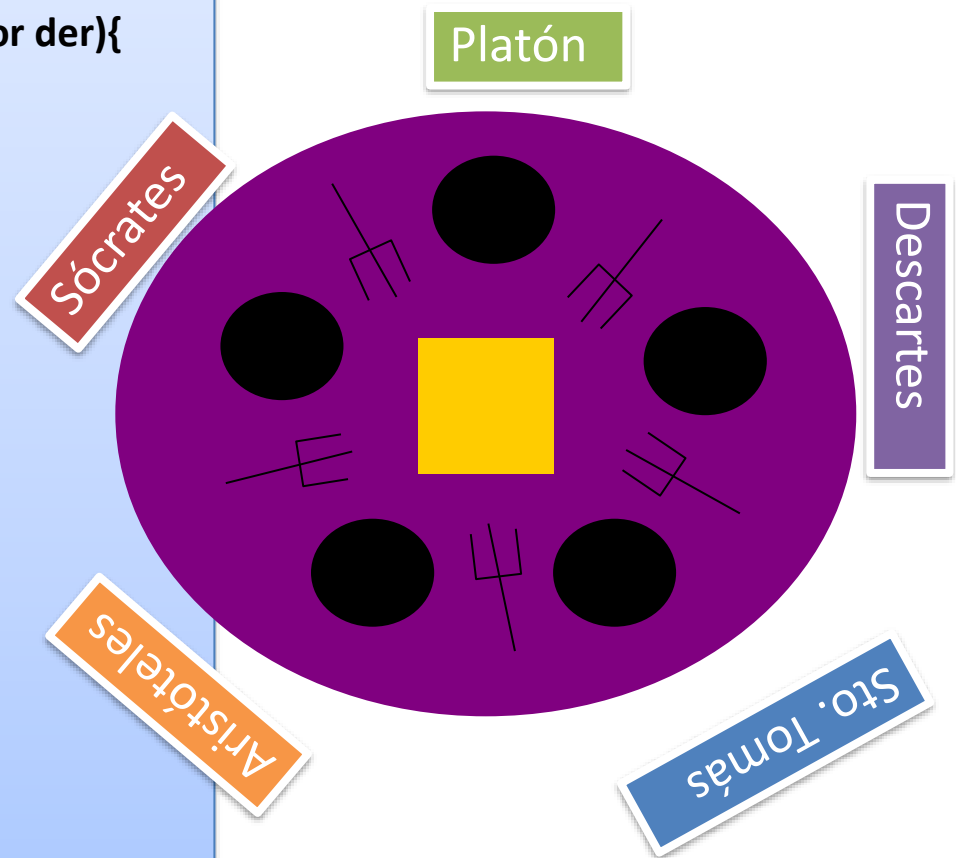
public class Tenedor{
    private int id;
    private Semaphore ten = new Semaphore(1);
    //semáforo binario
    public Tenedor(int id){
        this.id = id;
    }
    public cogeTenedor(int f) throws InterruptedException{
        System.out.println("Filósofo"+f+"coge el tenedor"+ id);
        ten.acquire()
    }
    public sueltaTenedor(int f) {
        System.out.println("Filósofo"+f+"suelta el tenedor"+ id);
        ten.release()
    }
}
```



El problema de los filósofos

```
public class Filosofo extends Thread{
    private static Random r = new Random();
    private int id; private Tenedor der,izq;
    public Filosofo(int id,Tenedor izq,Tenedor der){
        this.id = id; this.izq= izq; this.der = der
    }
    public void run(){....}
}

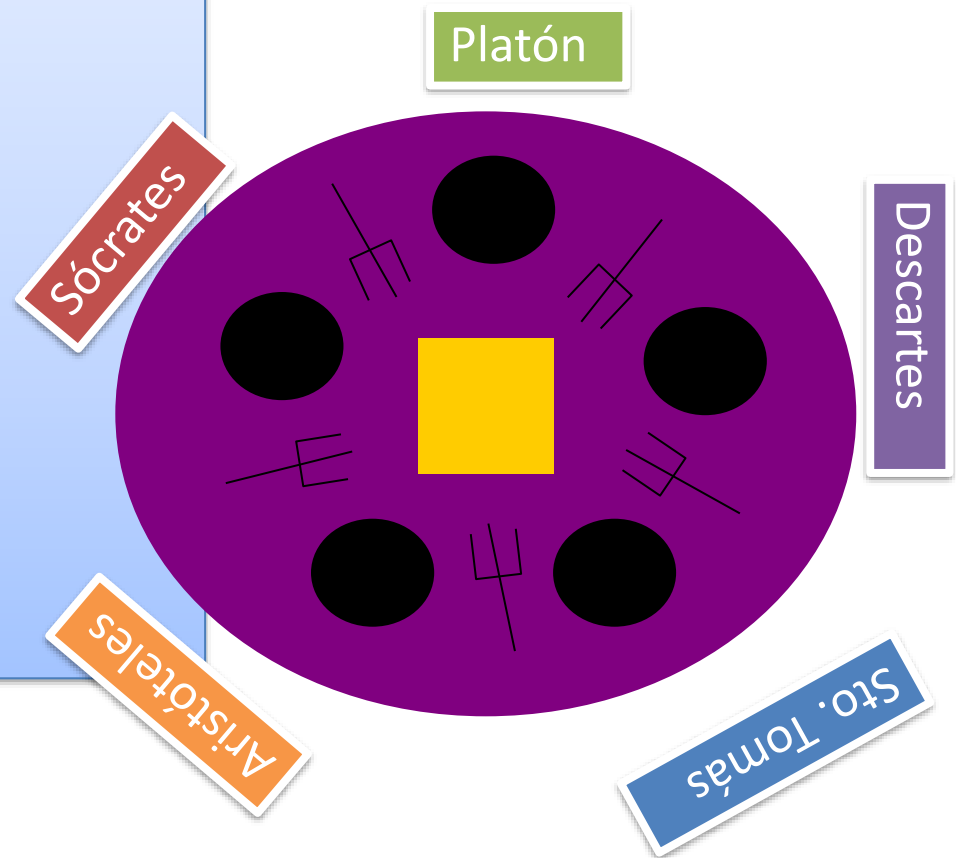
public static void main(String[] args){
    Tenedor t = new Tenedor[5];
    for (int i = 0; i<5; i++){
        t[i] = new Tenedor(i);
    }
    Filosofo[] f = new Filosofo[5];
    for (int i = 0; i<5; i++){
        f[i] = new Filosofo(i,t[i],t[(i+1)%5]);
    }
    for (int i = 0; i<5; i++){
        f[i].start();
    }
}
```



El problema de los filósofos

```
public void run(){  
    while (true){  
        try {  
            ten[izq].cogeTenedor();  
            ten[der].cogeTenedor();  
            Thread.sleep(r.nextInt(200));  
            ten[der].sueltoTenedor();  
            ten[izda].sueltoTenedor();  
            Thread.sleep(r.nextInt(500));  
        } catch (InterruptedException e) {  
        }  
    }  
}
```

Esta solución puede bloquear
si todos los filósofos intentan
comer simultáneamente y
cogen su primer tenedor



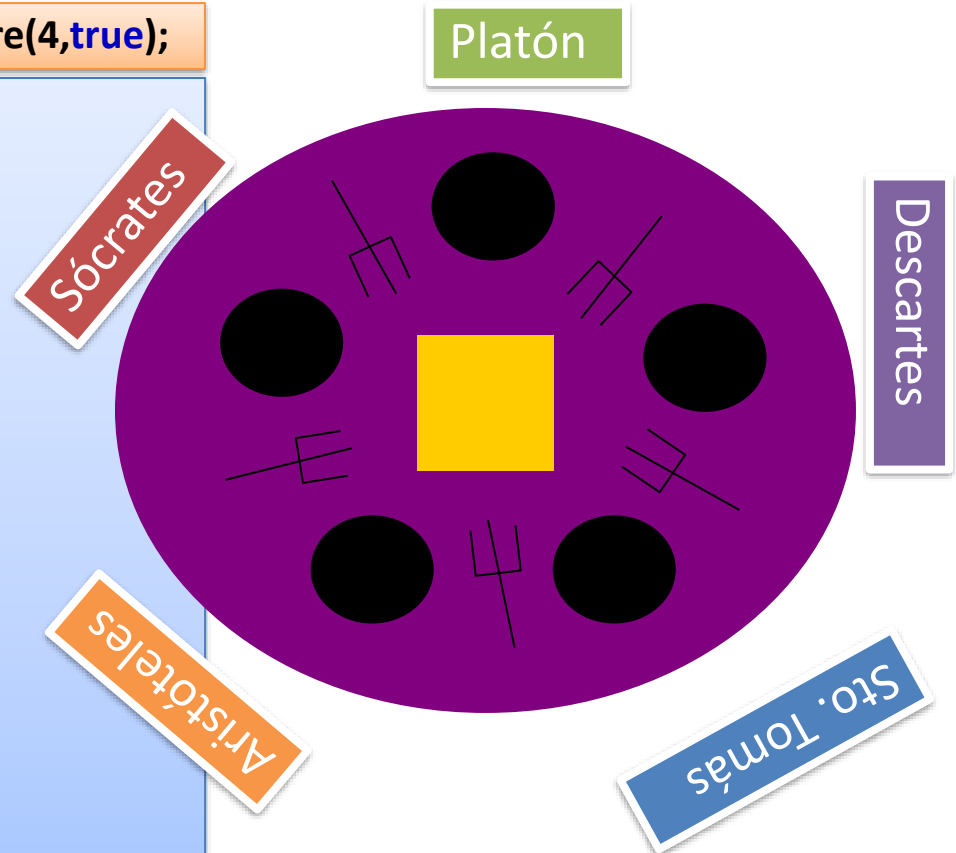
El problema de los filósofos

Por la disposición en la mesa, a lo sumo 2 filósofos pueden comer simultáneamente. Al siguiente le falta como mínimo un tenedor.

Una forma de evitar el bloqueo es utilizar un recurso **sillas** que deje pasar a los cuatro primeros filósofos que quieren comer, bloqueando al quinto, si también quiere comer.

```
private Semaphore sillasLibres = new Semaphore(4,true);
```

```
public void run(){  
    while (true){  
        try {  
            sillas.quieroSilla();  
            ten[izq].cogeTenedor();  
            ten[der].cogeTenedor();  
            Thread.sleep(r.nextInt(200));  
            ten[der].sueltoTenedor();  
            ten[izda].sueltoTenedor();  
            sillas.sueltoSillas();  
            Thread.sleep(r.nextInt(500));  
        } catch (InterruptedException e) {  
        }  
    }  
}
```



Referencias

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