

# Sesión 10: Sincronización por condición

Concurrencia

---

Ángel Herranz

2022-2023

Universidad Politécnica de Madrid

# Solución ejercicio mutex con semáforos

```
public static Semaphore s = new Semaphore(1);
```

proceso  $A$

```
 $A_1$ ;  
s.await();  
 $SC_A$ ;  
s.signal();  
 $A_2$ ;
```

proceso  $B$

```
 $B_1$ ;  
s.await();  
 $SC_B$ ;  
s.signal();  
 $B_2$ ;
```

proceso  $C$

```
 $C_1$ ;  
s.await();  
 $SC_C$ ;  
s.signal();  
 $C_2$ ;
```

# Análisis

- 👍 Corrección: mutex, ausencia de esperas, ausencia de interbloqueos, ausencia de inanición<sup>1</sup>
- 👍 Escalable:  $n$  procesos
- 👍 Escalable: consumo de CPU 0 en sincronización
- 👍 Escalable: resuelve cualquier problema de mutex

---

<sup>1</sup>Depende de la equitatividad de la implementación de semáforos

# Análisis

- 👍 Corrección: mutex, ausencia de esperas, ausencia de interbloqueos, ausencia de inanición<sup>1</sup>
- 👍 Escalable:  $n$  procesos
- 👍 Escalable: consumo de CPU 0 en sincronización
- 👍 Escalable: resuelve cualquier problema de mutex
- 👎 ¡Es más lento que la espera activa!



---

<sup>1</sup>Depende de la equitatividad de la implementación de semáforos

# ¿Ausencia de inanición?

proceso $A$	proceso $B$	proceso $C$
$A_1$ ;	$B_1$ ;	$C_1$ ;
$s.await()$ ;	$s.await()$ ;	$s.await()$ ;
$SC_A$ ;	$SC_B$ ;	$SC_C$ ;
$s.signal()$ ;	$s.signal()$ ;	$s.signal()$ ;
$A_2$ ;	$B_2$ ;	$C_2$ ;

$A_1 \rightarrow SC_A \rightarrow B_1 \rightarrow C_1 \rightarrow A_2 \rightarrow SC_B \rightarrow A_1 \rightarrow B_2 \rightarrow SC_A \rightarrow A_2 \rightarrow SC_B \rightarrow A_1 \rightarrow B_2 \rightarrow SC_A \dots$

# ¿Ausencia de inanición?

proceso $A$	proceso $B$	proceso $C$
$A_1$ ;	$B_1$ ;	$C_1$ ;
$s.await()$ ;	$s.await()$ ;	$s.await()$ ;
$SC_A$ ;	$SC_B$ ;	$SC_C$ ;
$s.signal()$ ;	$s.signal()$ ;	$s.signal()$ ;
$A_2$ ;	$B_2$ ;	$C_2$ ;

$A_1 \rightarrow SC_A \rightarrow B_1 \rightarrow C_1 \rightarrow A_2 \rightarrow SC_B \rightarrow A_1 \rightarrow B_2 \rightarrow SC_A \rightarrow A_2 \rightarrow SC_B \rightarrow A_1 \rightarrow B_2 \rightarrow SC_A \dots$

¡ $C$  puede sufrir de inanición!

# Equitatividad (*fairness*)

Cuando hay varios procesos bloqueados. . .

Q: ¿Qué proceso se *desbloquea*?

A: Uno *cualquiera*, quizás de forma *no determinista*

Q: Entonces, ¿podría un proceso *perder* siempre?

A: Depende de la *política de equitatividad (fairness)*

# Políticas de equitatividad (*fairness*)

No equitativa ( <i>Unfair</i> )	Debilmente equitativa ( <i>Weak fairness</i> )	Fuertemente equitativa ( <i>Strong fairness</i> )
Problema de Inanición		
ej. LIFO	ej. aleatorio	ej. FIFO

h



# Equitatividad en otros mecanismos

- Conocer la **política de tu mecanismo de concurrencia** es fundamental, por ejemplo:
- `java.util.concurrent.Semaphore`  
***public** Semaphore(**int** permits) Creates a Semaphore with the given number of permits and **non-fair fairness** setting.*  
***public** Semaphore(**int** permits, **boolean** fair) Creates a Semaphore with the given number of permits and the **given fairness setting** (FIFO).*
- `es.upm.babel.cclib.Semaphore`: **strong fairness** (FIFO)



# Concurrencia

 Simultaneidad

+

Sincronización<sup>2</sup> +  Comunicación<sup>3</sup>

---

<sup>2</sup>Sólo exclusión mutua.

<sup>3</sup>Sólo con memoria compartida.



# Concurrencia

 Simultaneidad

+

 Sincronización<sup>2</sup> +  Comunicación<sup>3</sup>

---

<sup>2</sup>Sólo exclusión mutua.

<sup>3</sup>Sólo con memoria compartida.

# Sincronización



- En la sincronización por exclusión mutua, un proceso debe esperar porque pretende ejecutar una sección crítica y otro proceso está ejecutando su sección crítica y así evitar una condición de carrera
- Pero hay muchas otras razones por las que un proceso puede querer esperar

# Problema de la cena de los filósofos<sup>4</sup>

- Diseñado por Edsger W. Dijkstra en 1965 para un examen de concurrencia ;)
- Posteriormente decorado como lo conocemos por Tony Hoare en *Communicating Sequential Processes* (1985).

---

<sup>4</sup>The Dining Philosophers

# *The Dining Philosophers* i

*Tony Hoare decoration, 1985*

*In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which they could engage in their professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. They sat anticlockwise around the table. To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished.*

# *The Dining Philosophers* ii



# The Dining Philosophers iii

A philosopher was expected to spend most of their time *thinking*; but when they felt *hungry*, they went to the dining room, *sat down* in their own chair, *picked up their own fork* on their left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that *a second fork is required* to carry it to the mouth. The philosopher therefore had also to *pick up the fork on their right*. When they were finished they would *put down both their forks*, get up from their chair, and *continue thinking*. Of course, *a fork can be used by only one philosopher at a time*. If the other philosopher wants it, they just have to *wait until the fork is available* again.



# *The Dining Philosophers*: modelización

- Cada filósofo *es un bucle infinito*:  
pensar, comer, pensar, comer, ...
- Cada filósofo tiene un identificador *id*<sup>5</sup>:  
0, 1, 2, 3, 4
- Antes de comer es necesario *coger* los tenedores
- Cada *tenedor* tiene un *identificador*
- Primero se coge el de la *izquierda*: *id*
- y luego el de la *derecha*:  $(id + 1) \bmod 5$

---

<sup>5</sup>Sentido antiohorario, Descartes es el 0

# *The Dining Philosophers: sincronización*

¿Cómo simular que el filósofo *id*  
coge el tenedor de la izquierda?

# *The Dining Philosophers*: sincronización

¿Cómo simular que el filósofo *id*  
coge el tenedor de la izquierda?

```
tenedor[id].await();
```

# *The Dining Philosophers*: sincronización

¿Cómo simular que el filósofo *id*  
coge el tenedor de la izquierda?

```
tenedor[id].await();
```

¡Perfecto!

# Sincronización por condición<sup>6</sup>



- En la sincronización por exclusión mutua, un proceso debe esperar porque pretende ejecutar una sección crítica y otro proceso está ejecutando su sección crítica y así evitar una condición de carrera
- En la sincronización por condición, un proceso debe esperar por una condición **arbitraria** que otro proceso puede hacer cierta

---

<sup>6</sup>*Condition synchronization*

# *The Dining Philosophers: talk is easy i*

```
public class Philosopher extends Thread {  
    private int id;  
  
    public Philosopher(int i) {  
        id = i;  
    }  
  
    public void think() {  
        ConcIO.printfnl("%d thinking", id);  
        sleep(0);  
    }  
  
    public void eat() {  
        ConcIO.printfnl("%d eating", id);  
        sleep(0);  
    }  
  
    ...  
}
```

# *The Dining Philosophers: talk is easy ii*

```
public static Semaphore[] tenedor = new Semaphore[5];

public void run() {
    int left = id, right = (id + 1) % 5;
    while (true) {
        think();
        ConcIO.printfnl("%d is hungry", id);
        tenedor[left].await();
        ConcIO.printfnl("%d picks up fork %d", id, left);
        tenedor[right].await();
        ConcIO.printfnl("%d picks up fork %d", id, right);
        eat();
        tenedor[left].signal();
        ConcIO.printfnl("%d puts down fork %d", id, left);
        tenedor[right].signal();
        ConcIO.printfnl("%d puts down fork %d", id, right);
    }
}
}
```

# *The Dining Philosophers: talk is easy iii*

```
public static void main(String[] argv) {  
    for(int i = 0; i < 5; i++) {  
        tenedor[i] = new Semaphore(1);  
    }  
    for(int i = 0; i < 5; i++) {  
        new Philosopher(i).start();  
    }  
}
```



# *The Dining Philosophers: talk is easy iv*

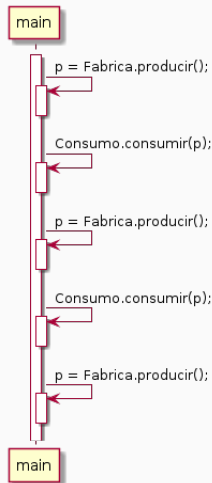
```
$ javac -cp .:cclib-0.4.9.jar Philosopher.java
$ java -cp .:cclib-0.4.9.jar Philosopher
1: [10] -> 1 thinking
4:    [9] -> 0 thinking
4:    [13] -> 4 thinking
4:    [11] -> 2 thinking
4:    [12] -> 3 thinking
...
```

# *The Dining Philosophers: talk is easy v*

```
1004: [10] -> 1 is hungry
1004:  [9] -> 0 is hungry
1005:  [9] -> 0 picks up fork 0
1006:    [13] -> 4 is hungry
1006:    [13] -> 4 picks up fork 4
1006: [10] -> 1 picks up fork 1
1007: [10] -> 1 picks up fork 2
1007:    [11] -> 2 is hungry
1007:    [12] -> 3 is hungry
1007:    [12] -> 3 picks up fork 3
1008: [10] -> 1 eating
2008: [10] -> 1 puts down fork 1
2009: [10] -> 1 puts down fork 2
2010: [10] -> 1 thinking
2011:  [9] -> 0 picks up fork 1
2012:  [9] -> 0 eating
2012:    [11] -> 2 picks up fork 2
3011: [10] -> 1 is hungry
3013:  [9] -> 0 puts down fork 0
3014:  [9] -> 0 puts down fork 1
3014:  [9] -> 0 thinking
3015:    [13] -> 4 picks up fork 0
3017:    [13] -> 4 eating
3017: [10] -> 1 picks up fork 1
4015:  [9] -> 0 is hungry
4017:    [13] -> 4 puts down fork 4
4018:    [13] -> 4 puts down fork 0
4018:    [13] -> 4 thinking
4019:        [12] -> 3 picks up fork 4
4020:        [12] -> 3 eating
4020:  [9] -> 0 picks up fork 0
5019:    [13] -> 4 is hungry
5020:        [12] -> 3 puts down fork 3
5021:    [11] -> 2 picks up fork 3
```

# Producir/consumir

```
Producto p;  
while (true) {  
    p = Fabrica.producir();  
    Consumo.consumir(p);  
}
```



# Productor/consumidor (sin sincronizar)

```
volatile public static Producto p = null;
```

Productor

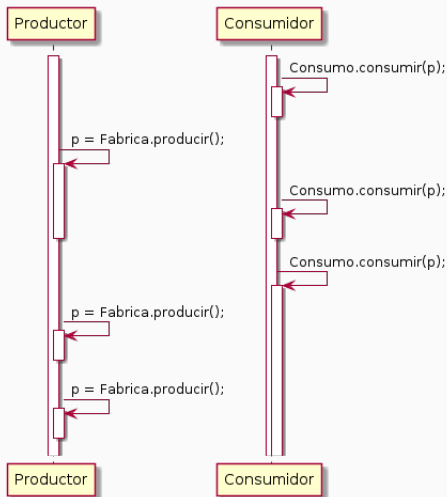
```
while (true) {  
    p = Fabrica.fabricar();  
}
```

Consumidor

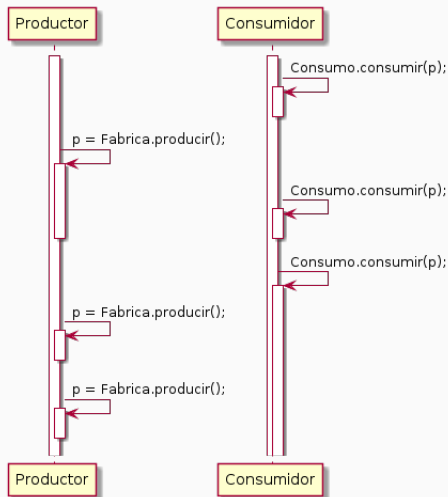
```
while (true) {  
    Consumo.consumir(p);  
}
```

¿Problemas?

# Productor/consumidor (sin sincronizar)

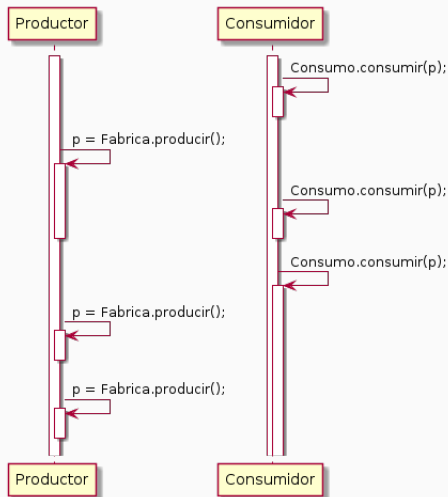


# Productor/consumidor (sin sincronizar)



- Se consume el mismo producto al menos dos veces (quizás **null**)

# Productor/consumidor (sin sincronizar)



- Se consume el mismo producto al menos dos veces (quizás **null**)
- Se pierde al menos un producto

# Productor/consumidor (mutex *total*)

```
volatile public static Producto p = null;
```

Productor

```
while (true) {  
    < p = Fabrica.fabricar() >;  
}
```

Consumidor

```
while (true) {  
    < Consumo.consumir(p) >;  
}
```

 < S >: el proceso ejecuta S **atómicamente**



# Productor/consumidor (mutex *total*)

```
volatile public static Producto p = null;
```

Productor

```
while (true) {  
    < p = Fabrica.fabricar() >;  
}
```

Consumidor

```
while (true) {  
    < Consumo.consumir(p) >;  
}
```

 < S >: el proceso ejecuta S **atómicamente**


## ¿Problemas?

# Productor/consumidor (deseable)

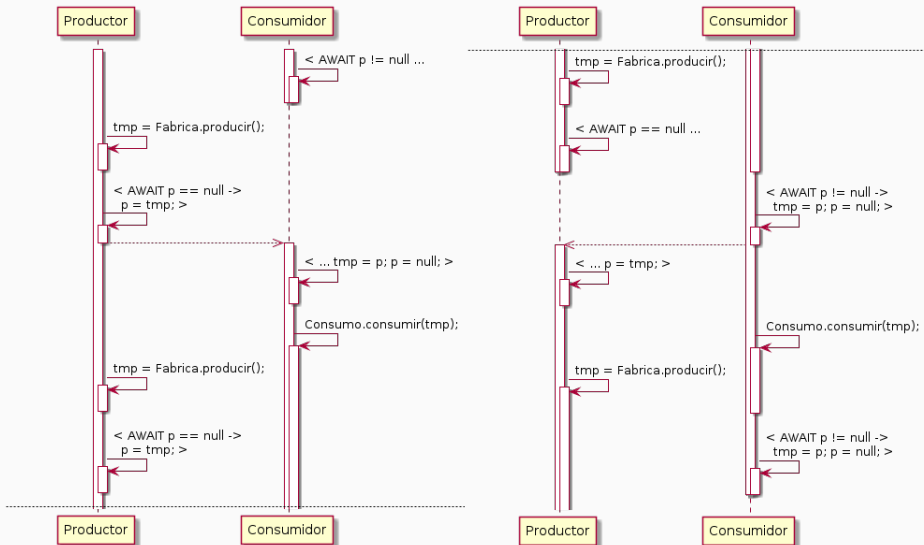
```
volatile public static Producto p = null;
```

```
Producto tmp;  
while (true) {  
    tmp = Fabrica.fabricar();  
    < AWAIT p == null →  
        p = tmp; >;  
}
```

```
Producto tmp;  
while (true) {  
    < AWAIT p != null →  
        tmp = p; p = null; >;  
    Consumo.consumir(tmp);  
}
```

  $\langle \text{AWAIT } C \rightarrow S \rangle$ : el proceso **comprueba**  $C$ , si  $C$  se cumple entonces **ejecuta**  $S$  **atómicamente**, si no se cumple **espera a que se cumpla** y ejecuta  $S$  **atómicamente**

# Productor/consumidor (deseable)





# Ejercicios obligatorio semanal

Hoja de ejercicios en Moodle

## **E5: Almacen de un dato con semáforos**

`Almacen1.java`

## **E6: Almacen de varios datos con semáforos**

`AlmacenN.java`

Sistema de entrega: *deliverit*

<https://deliverit.fi.upm.es>