

# Sesión 9: Semáforos

Concurrencia

---

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid



# Terminología<sup>1</sup> i

- Acción atómica: instrucción *mínima* que no pueden ser *interrumpida* por otro proceso
- Entrelazado: intercalado posible de acciones atómicas de diferentes procesos (*semántica*)
- Condición de carrera: resultados *indeseados* por *interacción* de dos o más procesos que *leen y modifican* datos *compartidos*
- Sección crítica: porción de código que puede dar lugar a una condición de carrera

---

<sup>1</sup>En inglés: *atomic action*, *interleaving*, *race conditions*, *critical section*



# Terminología<sup>3</sup> ii

- Exclusión mutua: propiedad deseable de nuestros programas que dice que **nunca** hay dos procesos ejecutando una **sección crítica** *al mismo tiempo*
- Espera activa: mecanismo de sincronización *autónomo*<sup>2</sup> basado en un bucle de comprobación continua

```
while (!C) { // no hacer nada }  
// ¡Aquí se cumple C!
```

---

<sup>2</sup>No requiere ayuda del sistema operativo o de bibliotecas

<sup>3</sup>mutual exclusion (mutex), busy-waiting



# Terminología<sup>4</sup> iii

- Propiedades de Seguridad:

Siempre se cumple  $P$

Nunca se cumple  $N$

- Propiedades de Vivacidad:

Alguna vez se cumple  $P$

---

<sup>4</sup>safety, liveness



# Terminología<sup>4</sup> iii

- Propiedades de Seguridad:

Siempre se cumple  $P$

Nunca se cumple  $N$

- Garantizar exclusión mutua
- Ausencia de esperas innecesarias
- Ausencia de interbloqueo

- Propiedades de Vivacidad:

Alguna vez se cumple  $P$

- Ausencia de inanición

---

<sup>4</sup>safety, liveness



 Simultaneidad

+

Sincronización<sup>5</sup> +  Comunicación<sup>6</sup>

---

<sup>5</sup>Sólo con espera activa para exclusión mutua.

<sup>6</sup>Sólo con memoria compartida.



# Concurrencia

 Simultaneidad

+

 Sincronización<sup>5</sup> +  Comunicación<sup>6</sup>

---

<sup>5</sup>Sólo con espera activa para exclusión mutua.

<sup>6</sup>Sólo con memoria compartida.



# Algoritmo de Peterson

```
public static volatile boolean a = false;  
public static volatile boolean b = false;  
public static volatile boolean turnoA = false;
```

```
a = true;  
turnoA = false;  
while (b && !turnoA) {}  
SC;  
a = false;
```

```
b = true;  
turnoA = true;  
while (a && turnoA) {}  
SC;  
b = false;
```



# Sincronización con espera activa

👍 Ejercicio conceptual **extraordinario**<sup>7</sup>.

- Garantiza exclusión mutua
- Ausencia de esperas innecesarias
- Ausencia de interbloqueo
- Ausencia de inanición

👎 Problemas de **escalabilidad**:

- Sólo **dos procesos**<sup>8</sup>
- Consumo **improductivo** de recursos (CPU)
- **Complejidad** conceptual: difícil de generalizar a otras situaciones que requieren sincronización

---

<sup>7</sup>Otros algoritmos: Dekker, Bakery (*Lamport*).

<sup>8</sup>Hay otros algoritmos pero son aún más complejos, ej. *Bakery*.

# We need a new hero!

$\{X > 0, Y > 0\}$   
 $\llbracket \text{var } x, y: \text{int}; \{X_{\text{gcd}} Y = x \text{ gcd } y \wedge x > 0 \wedge y > 0\}$   
 $\text{; } x, y := X, Y$   
 $\text{do } x > y \rightarrow x := x - y$   
 $\quad \parallel y > x \rightarrow y := y - x$   
 $\text{od } \{X_{\text{gcd}} Y = x \wedge X_{\text{gcd}} Y = y\}$

$\text{fib}.0 = 0$     $\text{fib}.1 = 1$     $\text{fib}.(n+2) = \text{fib}.(n+1) + \text{fib}.n$

$\text{fib}.(X_{\text{gcd}} Y)$

$= \text{fib}.(X_{\text{gcd}} Y)$

$\frac{\text{fib}.a \text{ value}}{\text{fib}.(y)}$

$\text{fib}.(X_{\text{gcd}} Y) = (\text{fib}.X)_{\text{gcd}}(\text{fib}.Y)$

$\text{fib}.(a+b)_{\text{gcd}} \text{fib}.b$

$\text{fib}.(a+b)$

$\{ \text{FIBONACCI} \}$

$\text{fib}.(a-1) \cdot \text{fib}.b + \text{fib}.a$

We need a new hero!

Edsger W. Dijkstra

$\{X > 0, Y > 0\}$   
[ var  $x, y: \text{int}$  ;  $\{X_{\text{gcd}} Y = x \text{ gcd } y \wedge x > 0 \wedge y > 0\}$   
;  $x, y := X, Y$   
do  $x > y \rightarrow x := x - y$   
||  $y > x \rightarrow y := y - x$   
od  $\{X_{\text{gcd}} Y = x \wedge X_{\text{gcd}} Y = y\}$

$\text{fib}.0 = 0$   $\text{fib}.1 = 1$   $\text{fib}.(n+2) = \text{fib}.(n+1) + \text{fib}.n$

$\text{fib}.(X_{\text{gcd}} Y)$

$= \text{fib}.(X_{\text{gcd}} Y)$

$\text{fib}.n$

$\text{fib}.(y)$

$\text{fib}.(X_{\text{gcd}} Y) = (\text{fib}.X)_{\text{gcd}} (\text{fib}.Y)$

$\text{fib}.(a+b)_{\text{gcd}} \text{fib}.a$

$\text{fib}.(a+b)$

$\{ \text{FIBONACCI} \}$

$\text{fib}.(a-1) \cdot \text{fib}.b + \text{fib}.a$

$\{\text{let } c\}$

# Semaphore

- Edsger W. Dijkstra
- ¿Qué son los semáforos?

---

<sup>9</sup>En la literatura el nombre de la operación en inglés suele ser *wait* pero en Java ese nombre ya es para un método de `Object`.

# Semaphore

- Edsger W. Dijkstra
- ¿Qué son los semáforos?
- Tipo Abstracto de Datos (**class** Semaphore)

---

<sup>9</sup>En la literatura el nombre de la operación en inglés suele ser *wait* pero en Java ese nombre ya es para un método de `Object`.

# Semaphore

- Edsger W. Dijkstra
- ¿Qué son los semáforos?
- Tipo Abstracto de Datos (**class** Semaphore)  
contador interno

---

<sup>9</sup>En la literatura el nombre de la operación en inglés suele ser *wait* pero en Java ese nombre ya es para un método de `Object`.

# Semaphore

- Edsger W. Dijkstra
- ¿Qué son los semáforos?
- Tipo Abstracto de Datos (**class** Semaphore)  
contador interno
- API:

P, *passering*, “pasar”

V, *vrijgave*, “soltar”

---

<sup>9</sup>En la literatura el nombre de la operación en inglés suele ser *wait* pero en Java ese nombre ya es para un método de `Object`.

# Semaphore

- Edsger W. Dijkstra
- ¿Qué son los semáforos?
- Tipo Abstracto de Datos (**class** Semaphore)  
contador interno
- API:

**Await**<sup>9</sup> P, *passering*, “pasar”

**Signal** V, *vrijgave*, “soltar”

---

<sup>9</sup>En la literatura el nombre de la operación en inglés suele ser *wait* pero en Java ese nombre ya es para un método de `Object`.



Semaphore s = new Semaphore(n)

¿Semántica?

---

<sup>10</sup> We specify that a statement is executed as a single atomic action by enclosing it in angle brackets., 1980 Lamport and Schneider.

# Semaphore $s = \text{new Semaphore}(n)$

¿Semántica?

$\langle s = n \rangle$

- Los símbolos “ $\langle$ ” y “ $\rangle$ ” denotan **atomicidad**<sup>10</sup>

---

<sup>10</sup>We specify that a statement is executed as a single atomic action by enclosing it in angle brackets., 1980 Lamport and Schneider.

# s.signal()

$$\langle S = S + 1 \rangle$$

- $\langle S \rangle$ : el proceso ejecuta  $S$  **atómicamente**

## `s.await()`

$\langle \text{AWAIT } s > 0 \rightarrow s = s - 1 \rangle$

- $\langle \text{AWAIT } C \rightarrow S \rangle$ : el proceso **comprueba**  $C$ , si  $C$  se cumple entonces **ejecuta**  $S$  **todo ello atómicamente** (*test&set*), si  $C$  no se cumple, entonces **espera hasta que se cumpla** y entonces ejecuta  $S$  de forma atómica

# Escenarios con tres procesos: 0

proceso $A$	proceso $B$	proceso $C$
$A_1;$	$B_1;$	$C_1;$
$A_2;$	$B_2;$	$C_2;$

$A_1 A_2 B_1 B_2 C_1 C_2$

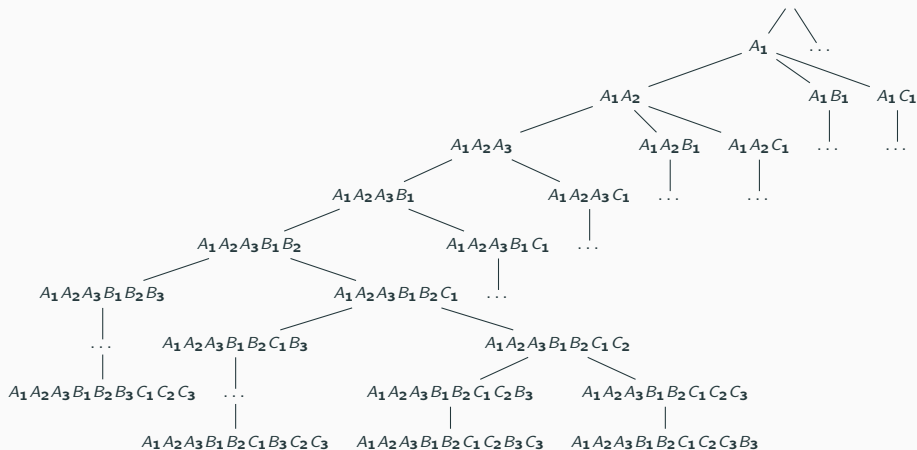
...

$A_1 B_1 C_1 A_2 B_2 C_2$

...

$C_1 C_2 B_1 B_2 A_1 A_2$

# Semántica: todos los entrelazados



# Escenario i

```
public static Semaphore s = new Semaphore(0);
```

proceso <i>A</i>		proceso <i>B</i>		proceso <i>C</i>
$A_1;$		$B_1;$		$C_1;$
$s.await();$		$B_2;$		$C_2;$
$A_2;$				

---

<sup>11</sup>Herranz **Notación:**  $|s|$  es el valor del contador interno del semáforo  $s$

# Escenario i

```
public static Semaphore s = new Semaphore(0);
```

proceso <i>A</i>		proceso <i>B</i>		proceso <i>C</i>
<i>A</i> <sub>1</sub> ;		<i>B</i> <sub>1</sub> ;		<i>C</i> <sub>1</sub> ;
<i>s.await()</i> ;		<i>B</i> <sub>2</sub> ;		<i>C</i> <sub>2</sub> ;
<i>A</i> <sub>2</sub> ;				

$$|s| = 0^{11}$$

Nunca se ejecuta *A*<sub>2</sub>

*A* no termina (¡y main tampoco!)

---

<sup>11</sup>Herranz **Notación:**  $|s|$  es el valor del contador interno del semáforo *s*



## Escenario ii

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

proceso *A*

```
A1;  
s.await();  
A2;
```

proceso *B*

```
B1;  
B2;
```

proceso *C*

```
C1;  
C2;
```

## Escenario ii

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

proceso <i>A</i>		proceso <i>B</i>		proceso <i>C</i>
<i>A</i> <sub>1</sub> ;		<i>B</i> <sub>1</sub> ;		<i>C</i> <sub>1</sub> ;
s.await();		<i>B</i> <sub>2</sub> ;		<i>C</i> <sub>2</sub> ;
<i>A</i> <sub>2</sub> ;				

$$|s| = 0$$

Misma semántica que escenario 0

## Escenario iii

```
public static Semaphore s = new Semaphore(0);
```

proceso *A*

```
A1;  
s.signal();  
A2;
```

proceso *B*

```
B1;  
B2;
```

proceso *C*

```
C1;  
C2;
```

## Escenario iii

```
public static Semaphore s = new Semaphore(0);
```

proceso <i>A</i>		proceso <i>B</i>		proceso <i>C</i>
<i>A</i> <sub>1</sub> ;		<i>B</i> <sub>1</sub> ;		<i>C</i> <sub>1</sub> ;
s.signal();		<i>B</i> <sub>2</sub> ;		<i>C</i> <sub>2</sub> ;
<i>A</i> <sub>2</sub> ;				

$$|s| = 1$$

Misma semántica que escenario 0

# Escenario iv

```
public static Semaphore s = new Semaphore(0);
```

proceso *A*

```
A1;  
s.signal();  
A2;
```

proceso *B*

```
B1;  
s.await();  
B2;
```

proceso *C*

```
C1;  
C2;
```

# Escenario iv

```
public static Semaphore s = new Semaphore(0);
```

proceso <i>A</i>	proceso <i>B</i>	proceso <i>C</i>
<i>A</i> <sub>1</sub> ;	<i>B</i> <sub>1</sub> ;	<i>C</i> <sub>1</sub> ;
s.signal();	s.await();	<i>C</i> <sub>2</sub> ;
<i>A</i> <sub>2</sub> ;	<i>B</i> <sub>2</sub> ;	

$$|s| = 0$$

Nunca ... *B*<sub>2</sub> ... *A*<sub>1</sub> ...

# Escenario v

```
public static Semaphore s = new Semaphore(0);
```

proceso *A*

```
A1;  
s.signal();  
A2;
```

proceso *B*

```
B1;  
s.await();  
B2;
```

proceso *C*

```
C1;  
s.await();  
C2;
```

# Escenario v

```
public static Semaphore s = new Semaphore(0);
```

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

$$|s| = 0$$

Nunca ...  $B_2$  ...  $\vee$  Nunca ...  $C_2$  ...

$B$  no termina  $\vee$   $C$  no termina

Nunca ...  $B_2$  ...  $A_1$  ...  $\vee$  Nunca ...  $C_2$  ...  $A_1$  ...





# Ejercicio obligatorio semanal

Hoja de ejercicios en:

<http://babel.ls.fi.upm.es/teaching/concurrencia>

Ejercicio 4:

## **Garantizar exclusión mutua con semáforos**

Fichero a entregar:

`CC_04_MutexSem.java`

Sistema de entrega:

<http://vps142.cesvima.upm.es>

# class Semaphore

- En la asignatura vamos a emplear la clase  
`es.upm.babel.cclib.Semaphore`

- Forma parte de la biblioteca **cclib**  
implementada por **profesores de la UPM**

En la web de la asignatura y en  
<https://github.com/aherranz/cclib>

- Java tiene **su propia clase semáforo**:

`java.util.concurrent.Semaphore`

**Desaconsejamos** su uso por varias razones: nombres de los métodos (*acquire* y *release*) y métodos *problemáticos* (ej. *getter* para el contador interno).