# Sesión 8: Semáforos

## Concurrencia

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid

# Terminología[1] 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

[1] En inglés: *atomic action*, *interleaving*, *race conditions*, *critical section*

# Terminología[3] 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*[2] basado en un bucle de comprobación continua

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

[2] No requiere ayuda del sistema operativo o de bibliotecas

[3] *mutual exclussion* (*mutex*), *busy-waiting*

# Terminología[4] iii

- Propiedades de Seguridad:

  Siempre se cumple $P$

  Nunca se cumple $N$

- Propiedades de Vivacidad:

  Alguna vez se cumple $P$

[4] *safety*, *liveness*

# Terminología[4] 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

[4] *safety*, *liveness*

# Concurrencia

👍 Simultaneidad

+

Sincronización[5] + 👍 Comunicación[6]

[5] Sólo con espera activa para exclusión mutua.

[6] Sólo con memoria compartida.

# Concurrencia

Simultaneidad

+

Sincronización[5] + Comunicación[6]

[5] Sólo con espera activa para exclusión mutua.

[6] 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;
```

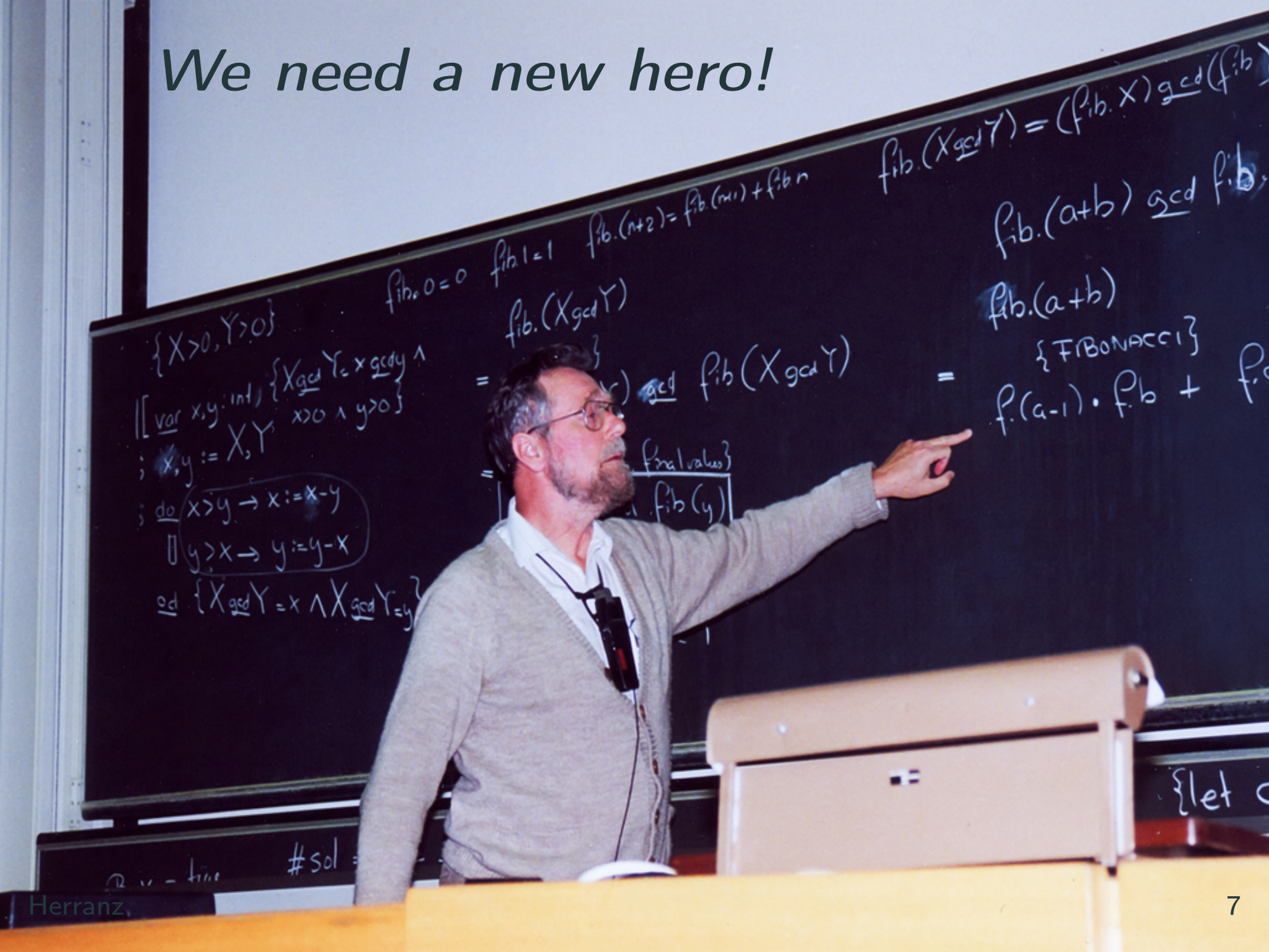
# Sincronización con espera activa

- 👍 Ejercicio conceptual extraordinario[7].
  - Garantiza exclusión mutua
  - Ausencia de esperas innecesarias
  - Ausencia de interbloqueo
  - Ausencia de inanición
- 👎 Problemas de *escalabilidad*:
  - Sólo dos procesos[8]
  - Consumo improductivo de recursos (CPU)
  - Complejidad conceptual: difícil de generalizar a otras situaciones que requieren sincronización

[7] Otros algoritmos: Dekker, Bakery (*Lamport*).

[8] Hay otros algoritmos pero son aún más complejos, ej. *Bakery*.

# We need a new hero!



# *We need a new hero!*

## Edsger W. Dijkstra

# *Semaphore*

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

[9] 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`)

[9] 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

[9] 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"

[9] 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**[9] P, *passering*, "pasar"

  **Signal** V, *vrijgave*, "soltar"

[9] 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?

[10] *We specify that a statement is executed as a single atomic action by enclosing it in angle brackets.*, 1980 Lamport and Schneider.

# `Semaphore s = new Semaphore(n)`

¿Semántica?

$$\langle \texttt{ s = n } \rangle$$

- Los símbolos "$\langle$" y "$\rangle$" denotan atomicidad[10]

[10] *We specify that a statement is executed as a single atomic action by enclosing it in angle brackets.*, 1980 Lamport and Schneider.

# `s.signal()`

$$\langle \texttt{ s = s + 1 } \rangle$$

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

# s.await()

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

- $\langle$ **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 $C$ |
|---|---|---|
| $A_1$; | $B_1$; | $C_1$; |
| $A_2$; | $B_2$; | $C_2$; |

$$A_1 A_2 B_1 B_2 C_1 C_2$$

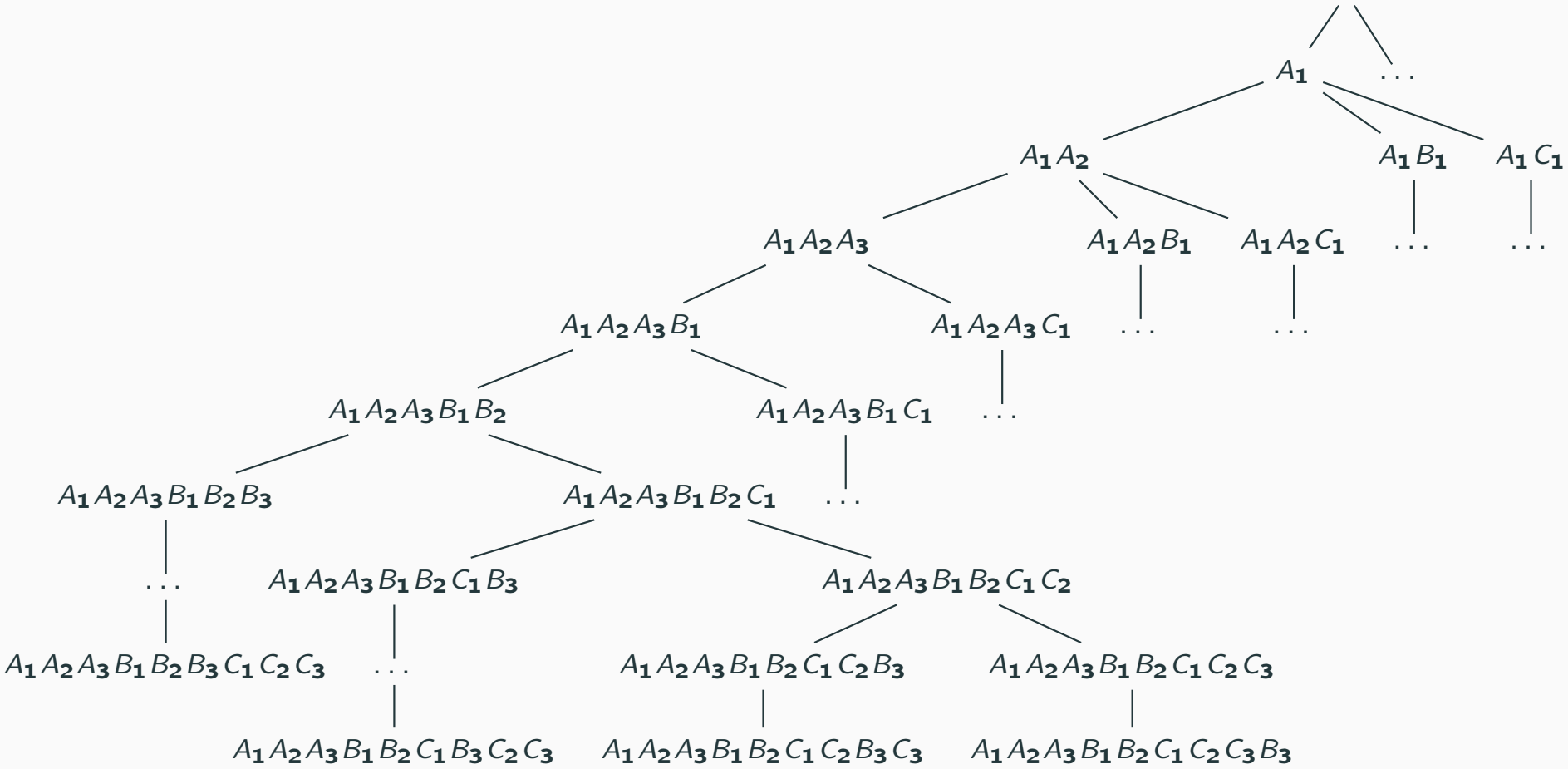
$$\ldots$$

$$A_1 B_1 C_1 A_2 B_2 C_2$$

$$\ldots$$

$$C_1 C_2 B_1 B_2 A_1 A_2$$

# Semántica: todos los entrelazados



# Escenario i

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

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

[11] **Notación:** $|s|$ es el valor del contador interno del semáforo s

# Escenario i

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

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

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

Nunca se ejecuta $A_2$

$A$ no termina (¡y `main` tampoco!)

[11] **Notación:** $|s|$ es el valor del contador interno del semáforo s

# Escenario ii

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

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

# Escenario ii

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

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

$$|s| = 0$$

Misma semántica que escenario 0

# Escenario iii

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

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

# Escenario iii

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

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

$$|s| = 1$$

Misma semántica que escenario 0

# Escenario iv

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

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

# Escenario iv

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

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

$$|s| = 0$$

Nunca $\ldots B_2 \ldots A_1 \ldots$

# 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();` |
| $A_2$; | $B_2$; | $C_2$; |

# 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();` |
| $A_2$; | $B_2$; | $C_2$; |

$$|s| = 0$$

$$\text{Nunca } \ldots B_2 \ldots \vee \text{Nunca } \ldots C_2 \ldots$$

$$B \text{ no termina} \vee C \text{ no termina}$$

$$\text{Nunca } \ldots B_2 \ldots A_1 \ldots \vee \text{Nunca } \ldots C_2 \ldots A_1 \ldots$$

# Ejercicio obligatorio semanal

Hoja de ejerccios en:

`http://babel.ls.fi.upm.es/teaching/concurrencia`

Ejerccio 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).