

L10: Semaphores I

César Sánchez

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
Universidad Politécnica de Madrid

Fri, 6-March-2015

Este texto se distribuye bajo los términos de la Creative Commons License

Under construction! Do **not** print

Mapa Conceptual

Concurrency = Simultaneous + Nondeterminism + Interaction

Interaction = Communication | Synchronization

Synchronization = Mutual Exclusion | Conditional Synchronization

■ Terminology:

atomic

interleaving

mutual exclusion

deadlock

liveness

race condition

busy-wait

critical section

livelock

semaphores

HW4: Garantizar exclusión mutua con semaforos

Homework:

- HW1: Creación de threads en Java
- HW2: Provocar una condición de carrera
- HW3: Garantizar la exclusión mutua con espera activa
- **HW4: Garantizar la exclusión mutua con semáforos**

Fecha de Cierre:

Miércoles 11-Marzo-2015 11am

Entrega online:

`http://lml.ls.fi.upm.es/~entrega`

HW4: Garantizar exclusión mutua con semáforos

Hoja de ejercicios cortos (r1638)

Concurrencia

Segundo semestre 2013-2014

4. Garantizar exclusión mutua con semáforos

Este ejercicio, al igual que el anterior, consiste en evitar una condición de carrera. En esta ocasión tenemos el mismo número de procesos incrementadores que decrementadores que incrementan y decrementan, respectivamente, en un mismo número de pasos una variable compartida. El objetivo es asegurar la exclusión mutua en la ejecución de los incrementos y decrementos de la variable y el objetivo es hacerlo utilizando exclusivamente un semáforo de la clase `es.upm.babel.clib.Semaphore` (está prohibido utilizar cualquier otro mecanismo de concurrencia). La librería de concurrencia `cclib.jar` puede descargarse de la página web de la asignatura.

Material a entregar

El fichero fuente a entregar debe llamarse `CC_04_MutexSem.java`.

Material de apoyo

Se ofrece el fichero `CC_04_MutexSem.todo.java` con un esqueleto que debe respetarse y que muestra una condición de carrera:

Problems with Busy-Wait

Busy-wait algorithms (Peterson, Dekker, Bakery) are correct, but
problems

- hard to prove correctness
- waste of CPU
- atomic actions are too fine grain and depend on the architecture

Problems with Busy-Wait

Busy-wait algorithms (Peterson, Dekker, Bakery) are correct, but
problems

- hard to prove correctness
- waste of CPU
- atomic actions are too fine grain and depend on the architecture

solution

Problems with Busy-Wait

Busy-wait algorithms (Peterson, Dekker, Bakery) are correct, but
problems

- hard to prove correctness
- waste of CPU
- atomic actions are too fine grain and depend on the architecture

solution

semaphores

- stop threads (like join) ← no busy-wait
- based on atomic test and set
- higher level, easier to reason about

Semaphores

- **stop** threads (like join)
no busy-wait
- based on atomic **test and set**
supported by most hardware
- higher level, easier to reason about
assertion based reasoning feasible

Semaphores

- **stop** threads (like join)
no busy-wait
- based on atomic **test and set**
supported by most hardware
- higher level, easier to reason about
assertion based reasoning feasible



Architecture independent
Present in many OSs and languages

Semaphores

- **stop** threads (like join)
no busy-wait
- based on atomic **test and set**
supported by most hardware
- higher level, easier to reason about
assertion based reasoning feasible



Architecture independent
Present in many OSs and languages

- Allow to solve easily **mutual-exclusion** by n processes
- We will learn how to use them, and their limitations

What is a semaphore

Resembles (only remotely) traffic lights:

- if allowed, you pass
- if not allowed, you wait

What is a semaphore

Resembles (only remotely) traffic lights:

- if allowed, you pass
- if not allowed, you wait

A **semaphore** is a data-type with two operations:

- $P()$
- $V()$

What is a semaphore

Resembles (only remotely) traffic lights:

- if allowed, you pass
- if not allowed, you wait

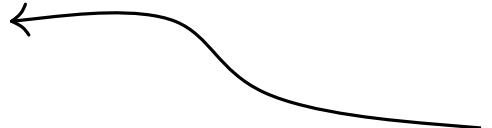
A **semaphore** is a data-type with two operations:

- $P()$
- $V()$

Internally, a **semaphore** maintains a counter, which is initialize at creation.

```
semaphore s(3);  
...  
s.P();  
...  
s.P();
```

$s.count \leftarrow 3$



Semantics of semaphores

Semantics of $s.P()$:


$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Semantics of semaphores

Semantics of $s.P()$:

$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Example 1:

$s.\text{count} == 0$ 
 $s.P()$

Semantics of semaphores

Semantics of $s.P()$:

$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Example 1:


$$s.\text{count} == 0 \quad \begin{array}{c} \sim \\ s.P_x() \end{array}$$

Semantics of semaphores

Semantics of $s.P()$:

$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Example 2:

$s.\text{count} == 1$ 
 $s.P()$

Semantics of semaphores

Semantics of $s.P()$:

$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Example 2:

$s.\text{count} == 1$ $\left\{ \begin{array}{l} s.P() \\ s.\text{count} == 0 \end{array} \right.$

Semantics of semaphores

Semantics of $s.P()$:

$$\begin{cases} \langle \text{if } s.\text{count} == 0 & \text{then } \textit{block} \rangle \\ \langle \text{if } s.\text{count} != 0 & \text{then } s.\text{count}-- \rangle \end{cases}$$

Example 2:

Atomically! \downarrow

$s.\text{count} == 1$
 $s.\text{count} == 0$

$s.P()$

Semantics of semaphores

Semantics of $s.V()$:

$$\begin{cases} \langle \text{if } \mathbf{no} \text{ blocked} & \text{then } s.count++ \rangle \\ \langle \text{if } \mathbf{some} \text{ blocked} & \text{then unblock } \textit{one} \text{ thread} \rangle \end{cases}$$

Semantics of semaphores

Semantics of $s.V()$:

$$\begin{cases} \langle \text{if } \mathbf{no} \text{ blocked} & \text{then } s.count++ \rangle \\ \langle \text{if } \mathbf{some} \text{ blocked} & \text{then unblock } \textit{one} \text{ thread} \rangle \end{cases}$$

Example 3:

$s.count == 0$
 $s.P_x()$

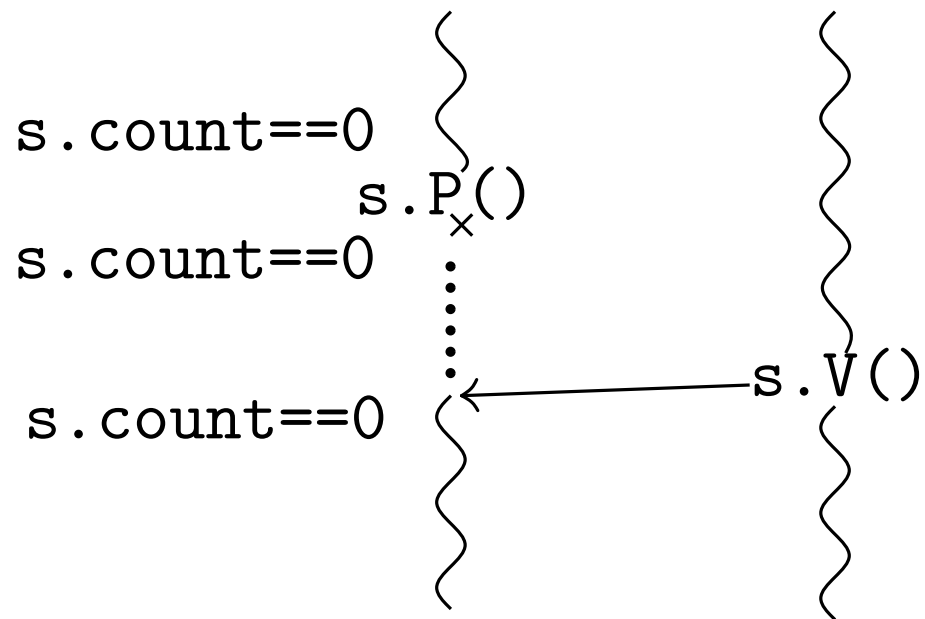
$s.V()$

Semantics of semaphores

Semantics of $s.V()$:

$$\begin{cases} \langle \text{if } \mathbf{no} \text{ blocked} & \text{then } s.count++ \rangle \\ \langle \text{if } \mathbf{some} \text{ blocked} & \text{then unblock } \textit{one} \text{ thread} \rangle \end{cases}$$

Example 3:



Questions

1. Could a semaphore become negative?

Questions

1. Could a semaphore become negative?
2. Could a semaphore (initialized to ≥ 0) become negative?

Questions

1. Could a semaphore become negative?
2. Could a semaphore (initialized to ≥ 0) become negative?
3. Let s be a semaphore be initialized to ≥ 0 . Is the following true?:
If a thread is blocked then $s.count == 0$

Questions

1. Could a semaphore become negative?
2. Could a semaphore (initialized to ≥ 0) become negative?
3. Let s be a semaphore be initialized to ≥ 0 . Is the following true?:
If a thread is blocked then $s.count == 0$
4. Let s be a semaphore be initialized to ≥ 0 . Is the following true?:
If $s.count == 0$ then a thread is blocked

Semaphores in Java

In HW4 you must use

```
es.upm.babel.cclib.Semaphore
```

Semaphores in Java

In HW4 you must use

```
es.upm.babel.cclib.Semaphore
```

You must import with:

```
import es.upm.babel.cclib.Semaphore;
```

Semaphores in Java

In HW4 you must use

```
es.upm.babel.cclib.Semaphore
```

You must import with:

```
import es.upm.babel.cclib.Semaphore;
```

Create a semaphore:

```
Semaphore s(1);
```

Semaphores in Java

In HW4 you must use

```
es.upm.babel.cclib.Semaphore
```

You must import with:

```
import es.upm.babel.cclib.Semaphore;
```

Create a semaphore:

```
Semaphore s(1);
```

Acquire a semaphore:

```
s.await();
```

Semaphores in Java

In HW4 you must use

```
es.upm.babel.cclib.Semaphore
```

You must import with:

```
import es.upm.babel.cclib.Semaphore;
```

Create a semaphore:

```
Semaphore s(1);
```

Acquire a semaphore:

```
s.await();
```

Release a semaphore:

```
s.signal();
```

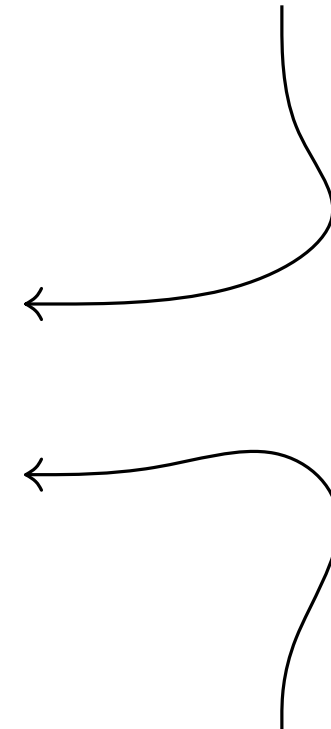
Operations Names

Original	cclib	java
P()	await()	acquire()
V()	signal()	release()

Operations Names

`decrement_or_block_if_the_result_is_negative()`

Original	cclib	java
P()	await()	acquire()
V()	signal()	release()



`wake_a_waiting_process_if_any_or_increment()`

Fairness

Q: Which **blocked** thread is awoken?

Fairness

Q: Which **blocked** thread is awoken?

A: *non-deterministic*

Fairness

Q: Which **blocked** thread is awoken?

A: non-deterministic

Q: Can a **blocked** thread always loose against others?

Fairness

Q: Which **blocked** thread is awoken?

A: non-deterministic

Q: Can a **blocked** thread always loose against others?

A: It depends

Fairness

Q: Which **blocked** thread is awoken?

A: *non-deterministic*

Q: Can a **blocked** thread always loose against others?

A: *It depends*

Fairness: a thread cannot be blocked forever
(if its semaphore is ready repeatedly)

Fairness

Q: Which **blocked** thread is awoken?

A: *non-deterministic*

Q: Can a **blocked** thread always loose against others?

A: *It depends*

Fairness: a thread cannot be blocked forever
(if its semaphore is ready repeatedly)

Typically, threads are awoken in the order they were blocked.

- The Java implementation has a flag to force fairness (by default it need not be)
- `cclib` is fair