



---

CONCURRENCIA  
*Condiciones de Carrera*

**Guillermo Román Díez**  
**[groman@fi.upm.es](mailto:groman@fi.upm.es)**

Universidad Politécnica de Madrid

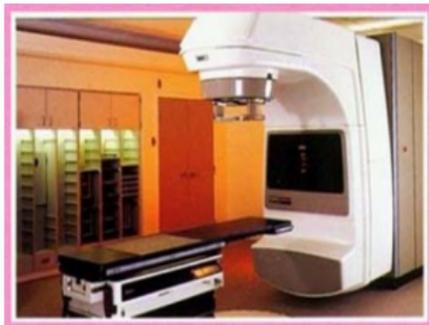
Curso 2018-2019

### *Condición de carrera (race condition)*

*“Una condición de carrera es un comportamiento del software en el cual la salida depende del orden de ejecución de eventos que no se encuentran bajo control”*

- ▶ Una condición de carrera se pueden producir cuando varios threads NO acceden en exclusión mutua a un recurso compartido
- ▶ Se convierte en un fallo siempre que el orden de ejecución no sea el esperado
- ▶ El nombre viene de la idea de dos procesos compiten en una *carrera* para acceder a un recurso compartido

## Therac 25



- ▶ Década 1980
- ▶ Sobreexposición de radiación (125x)
- ▶ Murieron 3 personas

## Apagón 2003



- ▶ 2003
- ▶ Las alarmas no notifican correctamente
- ▶ Efecto apagón en cascada

- ▶ El comportamiento de un programa con una condición de carrera es **imprevisible**
  - ▶ El comportamiento depende de la ejecución simultánea de múltiples threads accediendo a recursos compartidos
- ▶ **Difíciles de detectar**
- ▶ **Difíciles de reproducir**
- ▶ **Difíciles de depurar**

### *Solución*

La única solución es **prevenirlas** mediante una sincronización adecuada!

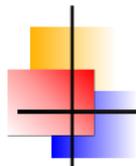
- ▶ Comprobar y entonces actuar (check-then-act)

```
public Singleton getInstance(){
    if(instance == null){
        instance = new Singleton();
    }
}
```

- ▶ Leer-modificar-escribir (read-modify-write)

```
if (x == 0) {
    x++;
}
```

- ▶ Las **condiciones de carrera** se producen al acceder o modificar la **memoria compartida**



## CARRERA: A NIVEL DE SENTENCIAS

---

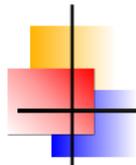
```
int [] a = {1,2,4}
```

Thread 1		Thread 2
$a[0] = a[1] + a[2]$		$a[2] = a[1] + a[0]$

```
int [] a = {1,2,4}
```

Thread 1		Thread 2
$a[0] = a[1] + a[2]$		$a[2] = a[1] + a[0]$

- ▶ Podemos “ignorar” sentencias ...
  - ▶ Thread1; Thread 2  $\Rightarrow \{6,2,8\}$
  - ▶ Thread2; Thread 1  $\Rightarrow \{5,2,3\}$
  - ▶ Entrelazados  $\Rightarrow \{6,2,3\}$



## CARRERA: A NIVEL DE INSTRUCCIONES

---

```
int contador = 0;
```

```
Thread 1    ||    Thread 2
```

```
contador ++; || contador --;
```



## CARRERA: A NIVEL DE INSTRUCCIONES

---

```
int contador = 0;
```

Thread 1

Thread 2

```
contador ++;
```

```
contador --;
```

```
i0: load x  
i1: const 1  
i2: add  
i3: store x
```

```
i'0: load x  
i'1: const 1  
i'2: sub  
i'3: store x
```

x = 0



## CARRERA: A NIVEL DE INSTRUCCIONES

---

```
int contador = 0;
```

Thread 1	Thread 2
contador ++;	contador --;

i <sub>0</sub> : load x	i' <sub>0</sub> : load x
i <sub>1</sub> : const 1	i' <sub>1</sub> : const 1
i <sub>2</sub> : add	i' <sub>2</sub> : sub
i <sub>3</sub> : store x	i' <sub>3</sub> : store x

x = -1

```
int contador = 0;
```

```
Thread 1
```

```
Thread 2
```

```
contador ++;
```

```
contador --;
```

```
i0: load x
```

```
i'0: load x
```

```
i1: const 1
```

```
i'1: const 1
```

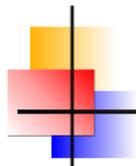
```
i2: add
```

```
i'2: sub
```

```
i3: store x
```

```
i'3: store x
```

```
x = 1
```



Asumiendo que tenemos una arquitectura de 16 bits  
que necesita 2 operaciones para escribir una palabra en memoria

```
int x = 0;
```

Thread 1		Thread 2
x = 987152;		x = 1;



Asumiendo que tenemos una arquitectura de 16 bits que necesita 2 operaciones para escribir una palabra en memoria

```
int x = 0;
```

Thread 1	Thread 2
<pre>x = 987152; // 0x000F1010</pre>	<pre>x = 1; // 0x00000001</pre>



Asumiendo que tenemos una arquitectura de 16 bits que necesita 2 operaciones para escribir una palabra en memoria

```
int x = 0;
```

Thread 1	Thread 2
<pre>x = 987152; // 0x000F1010</pre>	<pre>x = 1; // 0x00000001</pre>

```
x = {1,987152,983041,65793}
```

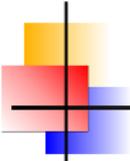
Asumiendo que tenemos una arquitectura de 16 bits que necesita 2 operaciones para escribir una palabra en memoria

```
int x = 0;
```

Thread 1	Thread 2
<pre>x = 987152; // 0x000F1010</pre>	<pre>x = 1; // 0x00000001</pre>

```
x = {1,987152,983041,65793}
```

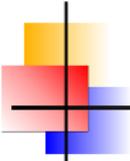
```
x = {0x00000001,0x000F1010,0x000F0001,0x00001010}
```



## ¿DÓNDE PUEDE HABER CONDICIONES DE CARRERA?

---

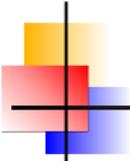
- ▶ Las condiciones de carrera se producen cuando dos o más threads acceden a memoria compartida
- ▶ En Java esto puede ocurrir de dos formas:



## ¿DÓNDE PUEDE HABER CONDICIONES DE CARRERA?

---

- ▶ Las condiciones de carrera se producen cuando dos o más threads acceden a memoria compartida
- ▶ En Java esto puede ocurrir de dos formas:
  - ▶ **Variables estáticas**
    - ▶ Si el código de dos o más threads accede (lectura y escritura) a la misma variable estática
    - ▶ Si hay dos instancias de la misma clase thread que accede a una variable estática



## ¿DÓNDE PUEDE HABER CONDICIONES DE CARRERA?

---

- ▶ Las condiciones de carrera se producen cuando dos o más threads acceden a memoria compartida
- ▶ En Java esto puede ocurrir de dos formas:
  - ▶ **Variables estáticas**
    - ▶ Si el código de dos o más threads accede (lectura y escritura) a la misma variable estática
    - ▶ Si hay dos instancias de la misma clase thread que accede a una variable estática
  - ▶ Con **referencias** al mismo objeto **compartidas** entre threads
    - ▶ En Java cualquier objeto o array se pasa por referencia
    - ▶ Si se comparte una referencia varios threads pueden estar accediendo al mismo objeto
    - ▶ Ojo! Puede se puede “atravesar” más de una “referencia”, es decir, un objeto que apunta a otro objeto. . .

- ▶ Cuando un proceso accede a un recurso compartido, decimos que está accediendo a una **sección crítica**
- ▶ Un mismo proceso puede acceder a más de una sección crítica
- ▶ La ejecución de la sección crítica debe ser en **exclusión mutua**
  - ▶ Únicamente **un proceso** puede estar el código de la sección crítica
  - ▶ Es necesario un *protocolo* para controlar el acceso a la SC
    - ▶ Es necesario algún mecanismo para **pedir permiso** para **entrar** en la sección crítica
    - ▶ También es necesario **avisar** de que hemos **salido** para que otros puedan entrar

## *Sincronización*

### **Exclusión Mutua** || Sincronización por condición

#### *Exclusión Mutua*

*“es la comunicación requerida entre dos o más procesos que necesitan acceder a la vez a un recurso compartido para que únicamente acceda uno de procesos simultáneamente a dicho recurso”*

- ▶ Asegura que una secuencia de instrucciones sea tratada como una operación indivisible
- ▶ Permite el acceso *seguro* a una sección crítica

## *Sincronización*

Exclusión Mutua || **Sincronización por condición**

### *Sincronización por condición*

*“Consiste en retrasar un proceso hasta que una de las variables compartidas cumpla las condiciones necesarias para ejecutar la siguiente operación”*

- ▶ Asegura que se cumplen las condiciones necesarias para poder llevar a cabo una determinada acción
- ▶ En caso de que no se cumplan estas condiciones, se retrasa la acción
- ▶ Por ejemplo: antes de sacar elementos de un buffer es necesario que haya elementos

### *Propiedades de seguridad*

*“Aseguran que nada malo pasa en el programa concurrente”*

- ▶ **Exclusión mutua**

- ▶ En el acceso a la sección crítica debe realizarse en exclusión mutua

- ▶ **Sincronización condicional**

- ▶ Se cumplen las condiciones necesarias para que un cierto código se pueda ejecutar

- ▶ El programa está **libre** de posibles **deadlocks** o **livelocks**

- ▶ Los procesos que conforman el programa progresan

### *Propiedades de viveza*

*“garantizan que en algún momento ocurre algo bueno en el programa concurrente”*

#### ▶ **Inanición (starvation)**

- ▶ El tiempo de espera para acceder a la sección crítica no está acotado

#### ▶ **Elección Justa (fairness)**

- ▶ Si hay varios procesos intentando entrar en la sección crítica, éstos deberían hacerlo con frecuencias similares

#### ▶ **Esperas innecesarias**

- ▶ Ningún proceso fuera de la sección crítica debe impedir el acceso a la sección crítica



## PROPIEDADES DE VIVEZA (LIVENESS)

---

- ▶ **No Interbloqueo (deadlock)**

### ▶ No Interbloqueo (deadlock)

#### *Deadlock*

*“Describe una situación en la cual hay dos o más procesos bloqueados esperándose entre sí indefinidamente”*

### ▶ **No Interbloqueo (deadlock)**

#### *Deadlock*

*“Describe una situación en la cual hay dos o más procesos bloqueados esperándose entre sí indefinidamente”*

### ▶ **No Livelock**

### ▶ **No Interbloqueo (deadlock)**

#### *Deadlock*

*“Describe una situación en la cual hay dos o más procesos bloqueados esperándose entre sí indefinidamente”*

### ▶ **No Livelock**

#### *Livelock*

*“Es similar a un deadlock, excepto que el estado de los procesos involucrados cambia constantemente, pero ninguno progresa”*



Concurrencia = Ej. Simultánea + Indeterminismo + Interacción

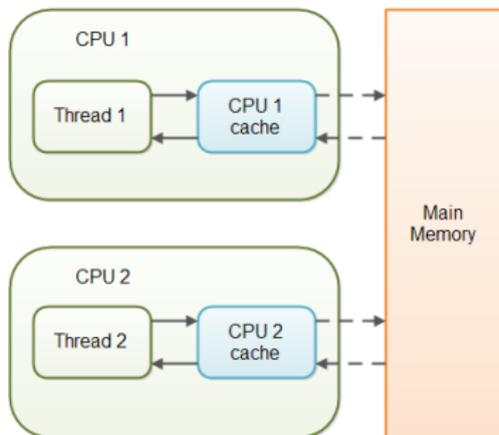
Interacción = Comunicación | Sincronización

Sincronización = Exclusión Mutua | Sincronización condicional

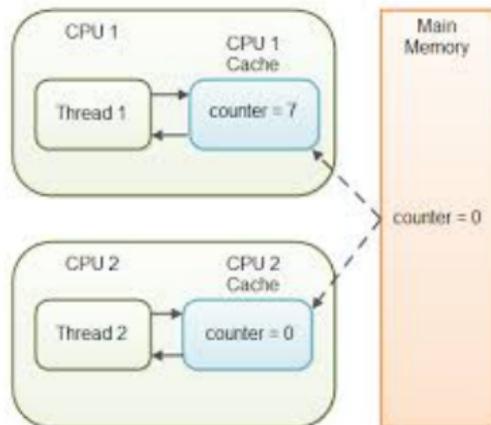
Terminología:

- Operación Atómica
- Sección crítica
- Exclusión mutua
- Sincronización condicional
- Deadlock
- Livelock
- Justicia

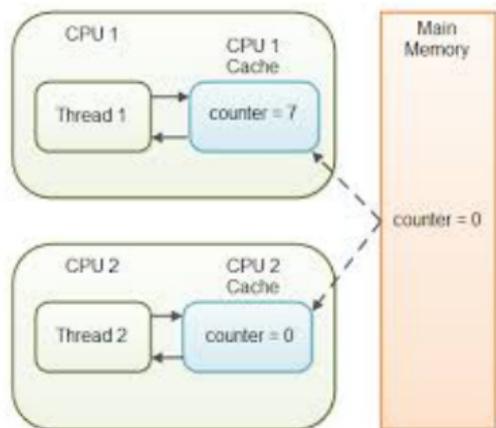
# VARIABLES “VOLATILE”



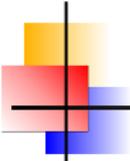
# VARIABLES “VOLATILE”



## VARIABLES “VOLATILE”



- ▶ Una variable declarada como **volatile** fuerza a que el thread se sincronice siempre con la memoria principal



## VARIABLES “VOLATILE”

---

- ▶ Una variable declarada como **volatile** fuerza a que el thread se sincronice siempre con la memoria principal

```
class ClaseCualquiera {  
    volatile int dato;  
}
```

- ▶ La lectura en memoria está sincronizada
- ▶ La escritura propaga la información a todos los threads
- ▶ Impide la reordenación para optimización
- ▶ Puede afectar al rendimiento (no hay caché)
- ▶ Ojo, porque sólo afecta a lecturas y escrituras (operaciones atómicas)

### NOTA!!

Los incrementos (i++) NO son operaciones atómicas!!!