



UD2. Sincronización de tareas



Concurrencia y Sistemas Distribuidos



Objetivos de la Unidad Didáctica

- ▶ Revisión del concepto de programación concurrente
 - ▶ Identificar los problemas que conlleva la programación concurrente
 - ▶ Conocer los mecanismos de cooperación necesarios para implantar una aplicación concurrente
 - ▶ Mecanismos de comunicación y de sincronización
 - ▶ Identificar las partes del código que pueden ocasionar problemas (secciones críticas) y cómo protegerlas
- ▶ Conocer un lenguaje de programación que da soporte a la programación concurrente: Conurrencia en JAVA
 - ▶ Introducir los mecanismos básicos del lenguaje Java para soportar la programación concurrente

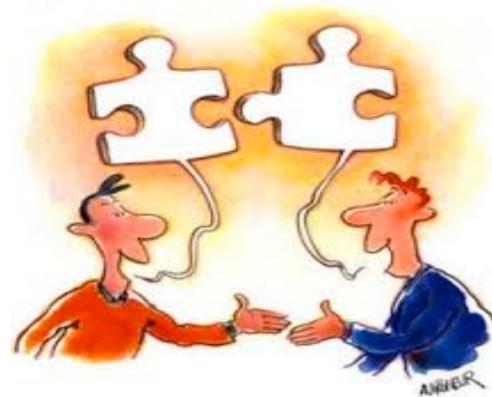


Unidad 2 - Contenido

- ▶ **Sincronización de tareas**
 - ▶ Mecanismos de comunicación
 - ▶ Memoria compartida
 - ▶ Intercambio de mensajes
 - ▶ Tipos de sincronización
 - ▶ Exclusión mutua
 - Sección Crítica
 - Locks
 - ▶ Sincronización condicional

Concurrencia entre hilos: requisitos

- ▶ La concurrencia implica parallelismo y cooperación de los hilos
- ▶ La **cooperación** requiere:
 - ▶ a) comunicación (intercambio de información)
 - ▶ b) sincronización (establecer determinado orden en casos concretos)





Unidad 2 - Contenido

► Sincronización de tareas

► Mecanismos de comunicación

- Memoria compartida
- Intercambio de mensajes

► Tipos de sincronización

- Exclusión mutua
 - Sección Crítica
 - Locks
- Sincronización condicional



Mecanismos de comunicación

- ▶ **Mecanismos de comunicación**
 - a) **Memoria compartida:** los hilos comparten espacio de memoria (variables u objetos compartidos)
 - b) **Intercambio de mensajes:** emisor/receptor potencialmente en espacios de memoria disjuntos
- ▶ Nos centramos aquí en **memoria compartida (opción a)**
 - ▶ Requiere mecanismos de sincronización para coordinar las tareas
 - ▶ Utilizado en lenguajes como Java y C#, y bibliotecas como pthreads
 - ▶ Veremos el modelo de concurrencia clásico
- ▶ La *opción b* la veremos en Sistemas Distribuidos
 - ▶ La tendencia actual es que las aplicaciones sean distribuidas



Memoria compartida: problemas que pueden aparecer

- ▶ **Condiciones de Carrera:** modificación inconsistente de la memoria compartida
 - ▶ Pueden aparecer errores cuando múltiples hilos acceden a datos compartidos.
 - ▶ Pueden aparecer errores de vistas inconsistentes de la memoria compartida.

Con la **Sincronización** se previenen estos problemas



Problemas por el uso de memoria compartida →

Ejemplo 1

- ▶ Ejemplo: **c++** se puede descomponer en 3 pasos
 1. Obtener el valor actual de **c**
 2. Incrementar el valor obtenido en 1
 3. Almacenar el valor resultante en **c**

```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

(lo mismo para **c--**, pero decrementando)

- ▶ Ojo! Incluso las sentencias simples pueden traducirse en múltiples pasos en la Máquina Virtual Java



Ejemplo 1

- ▶ EJEMPLO: tenemos un objeto c de tipo *Counter* y dos hilos A y B
 - ▶ c inicialmente vale 0.
 - ▶ A ejecuta *c.add(3)*
 - ▶ B ejecuta a la vez *c.add(2)*
- ▶ El valor final de c debería ser 5
- ▶ Pero el valor final depende del **intercalado exacto** (planificación) al ejecutar A y B
 - ▶ Se producen **interferencias** cuando dos operaciones, ejecutándose en diferentes hilos, pero actuando sobre los mismos datos, se **intercalan**.

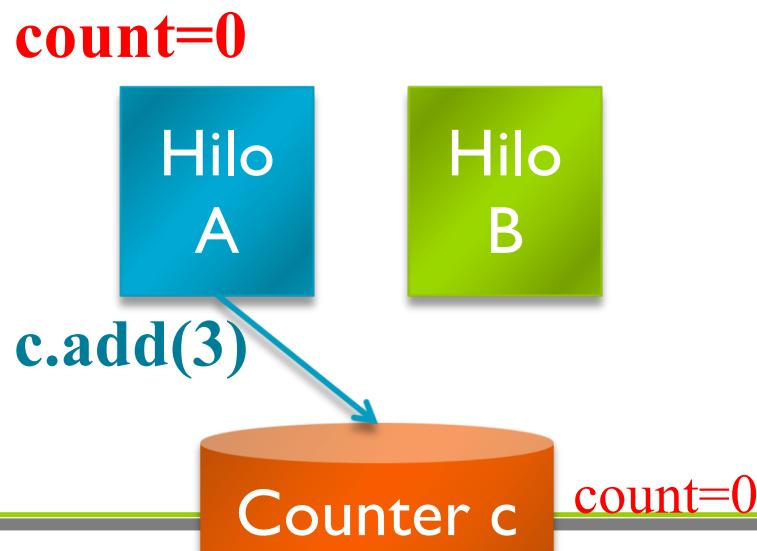
```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A cargo `c.count` (copia local=0)



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

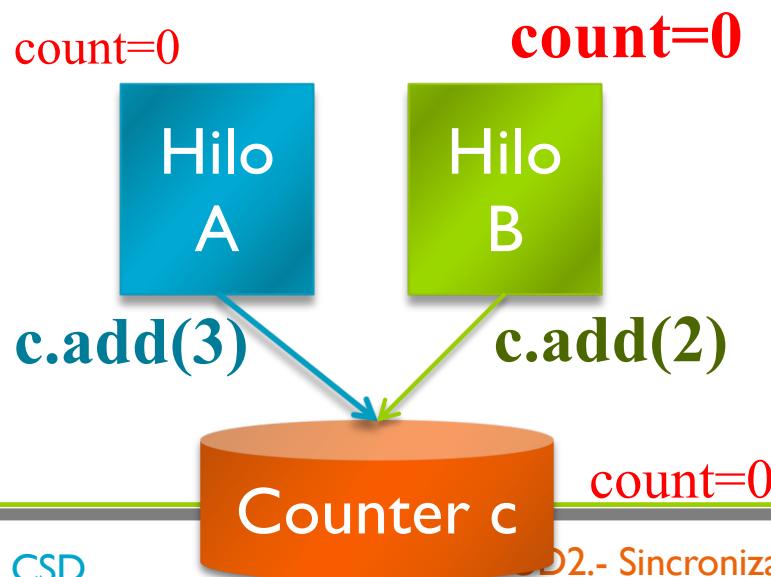
► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

► Ejemplo de intercalado:

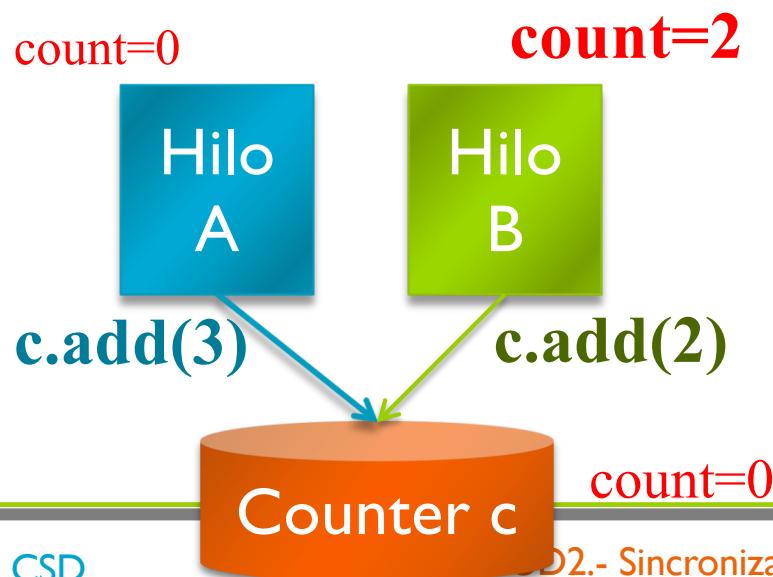
Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

B suma 2 (copia local=2)



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

► Ejemplo de intercalado:

Hilo A ejecuta c.add(3)

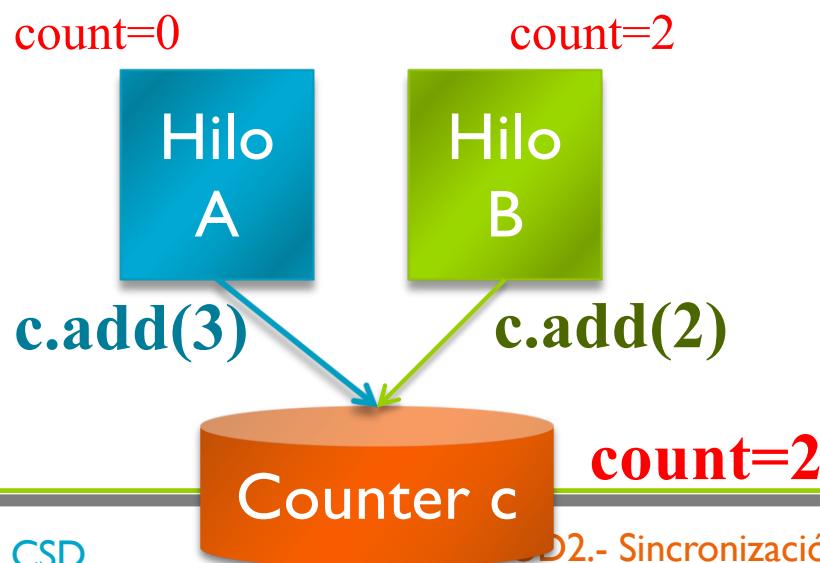
A carga c.count (copia local=0)

Hilo B ejecuta c.add(2)

B carga c.count (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (c.count=2)



```
public class Counter {
    protected long count = 0;
    public void add(long x) {
        count += x;
    }
    public long getCount() {
        return count;
    }
    ...
}
```

Counter c= new Counter(); // inic a 0

Ejemplo 1

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

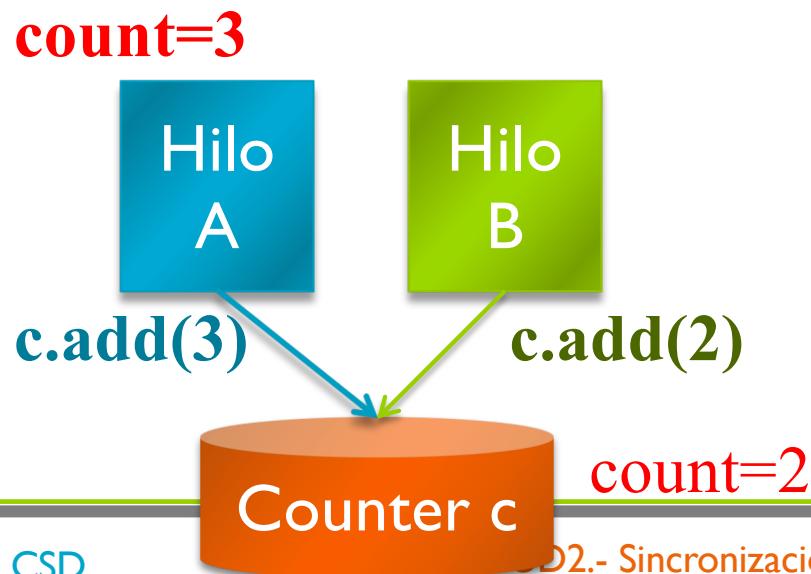
Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (`c.count=2`)

A suma 3 (copia local=3)



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

A vuelca al heap (`c.count=3`)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (`c.count=2`)

`count=3`



```
public class Counter {  
    protected long count = 0;  
    public void add(long x) {  
        count += x;  
    }  
  
    public long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo 1

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

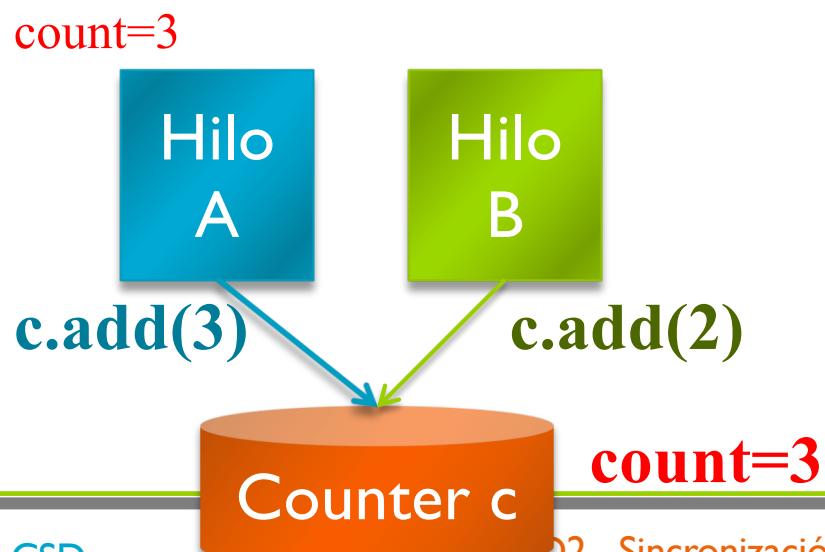
A vuelca al heap (`c.count=3`)

Hilo B ejecuta `c.add(2)`

B carga `c.count` (copia local=0)

B suma 2 (copia local=2)

B vuelca al heap (`c.count=2`)



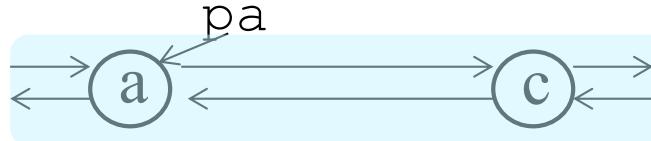
Valor final: 3
¡Esperábamos 5 !!!

Se ha producido una
condición de carrera

Ejemplo 2.- Inserta **b** entre **a** y **c** en lista doblemente enlazada

Como acción atómica

Nodo pa=... //inserta tras pa



Estado consistente

Nodo pb = new Nodo(b);



Estados intermedios

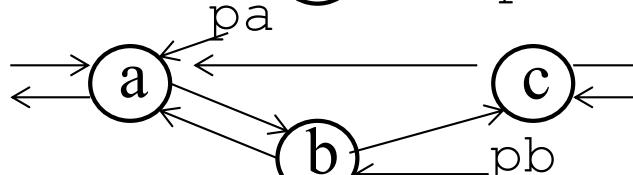
pb.sig = pa.sig;



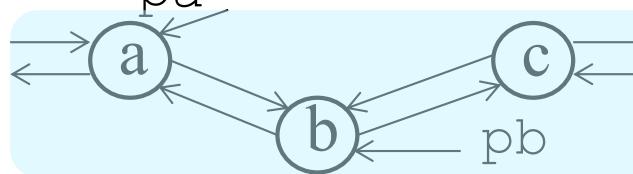
pb.prev = pa;



pa.sig = pb;



(pb.sig).prev = pb;



Estado consistente



Unidad 2 - Contenido

- ▶ **Sincronización de tareas**
 - ▶ Mecanismos de comunicación
 - ▶ Memoria compartida
 - ▶ Intercambio de mensajes
 - ▶ Tipos de sincronización
 - ▶ Exclusión mutua
 - Sección Crítica
 - Locks
 - ▶ Sincronización condicional



Objetivos de la sincronización

Objetivos de los Mecanismos de sincronización

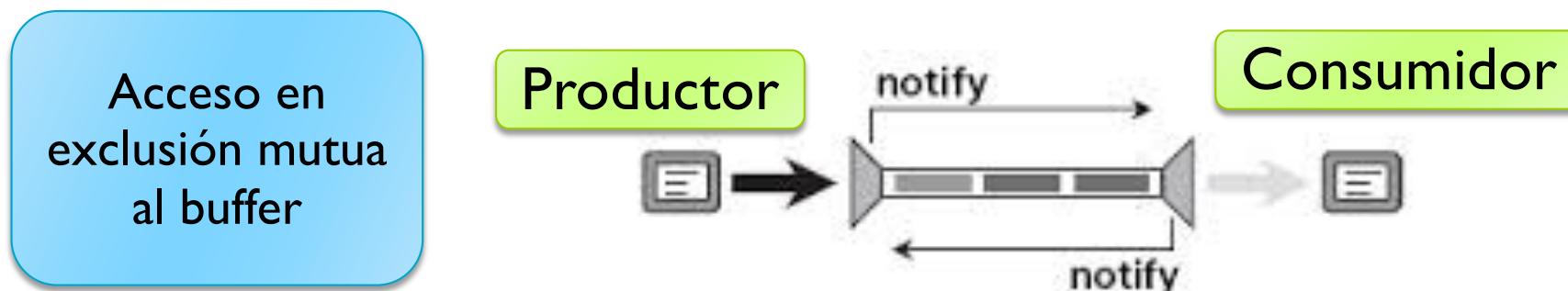
- garantizar orden de ejecución de sentencias
- respetar restricciones en la ejecución de código

► Tipos de sincronización:

- ▶ Exclusión mutua
- ▶ Sincronización condicional

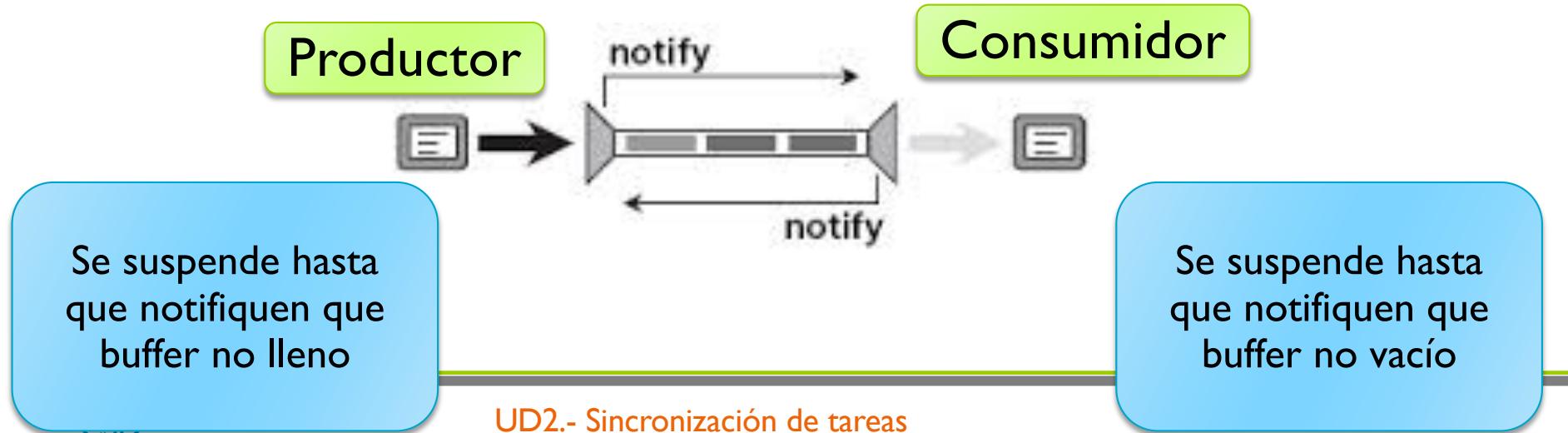
▶ Exclusión mutua

- ▶ SOLAMENTE UN HILO puede ejecutar la sección en cada momento
- ▶ Necesario para evitar interferencias entre hilos
- ▶ Aplicarlo para variables/objetos compartidos



▶ Sincronización condicional

- ▶ Un hilo debe suspenderse hasta que se cumpla determinada condición
- ▶ La condición depende del valor de alguna variable compartida
- ▶ Otros hilos, al modificar esas variables, conseguirán que se cumpla la condición, reactivando a los hilos suspendidos





Sección Crítica → ¿Qué es?

Sección crítica (SC): fragmento de código que puede provocar condiciones de carrera

- ▶ Acceso a *variables locales* → no es crítico
- ▶ Acceso a *objetos compartidos inmutables (constantes)* → no crítico
- ▶ Acceso a **variables u objetos compartidos** → **crítico**



Sección crítica (SC): fragmento que accede a variables u objetos compartidos por más de un hilo



Sección Crítica → ¿Cómo protegerla?

...

Resto de código

....

protocolo de entrada

Sección Crítica

protocolo de salida

...

Resto de código
(Sección no crítica)

....

Protocolos de entrada/salida
que garantizan (en común):

- **Exclusión mutua**
- **Progreso:** todo servicio solicitado se completa en algún momento
- **Espera limitada**

Código thread-safe: puede ser ejecutado concurrentemente por distintos hilos de forma segura



Sección Crítica → ¿Cómo protegerla?

► Estrategias de solución según el nivel:

Hardware

- Inabilitación de interrupciones
- Sólo para codificar el núcleo del SO

Sistema Operativo

- Llamadas al sistema para utilizar semáforos, mutex, eventos, condiciones

Lenguaje de Programación

- *Locks, condiciones, monitores, regiones críticas, objetos protegidos*
- Ejemplos: Pascal Concurrente, Modula-2, Modula-3, Ada, Java, C#

Locks → ¿Qué es? ¿Cómo utilizarlos?

- ▶ Un **lock** es un objeto con dos estados (*abierto/cerrado*) y dos operaciones (*abrir/cerrar*).
- ▶ Al crear el lock, se encontrará inicialmente abierto.
- ▶ Uso del *lock*:

cerrar lock

SC

abrir lock

Sección no crítica



Locks → ¿Cómo utilizarlos?

► Cerrar lock:

- ▶ Si abierto → lo cierra
- ▶ Si cerrado por otro hilo → se suspende
- ▶ Si cerrado por mismo hilo → no tiene efecto

► Abrir lock:

- ▶ Si abierto → no tiene efecto
- ▶ Si cerrado por otro hilo → no tiene efecto
- ▶ Si cerrado por mismo hilo → se abre el lock
- ▶ Si alguien espera, quedará cerrado por uno de los que esperan
 - La elección depende de la implementación, aunque normalmente cumplirá equidad



Locks → ¿Qué es? ¿Cómo utilizarlos?

► Usando *locks* convertimos una sección crítica en una **acción atómica**

- **Atomicidad:** únicamente un hilo puede ejecutar el código protegido en un momento dado
- Garantiza actualización fiable de variables u objetos compartidos
 - Libre de condiciones de carrera o corrupción de estado
 - Sólo son visibles los estados consistentes
 - Asegura que todo hilo accede siempre al valor más reciente de cada variable u objeto compartido.

cerrar lock

SC

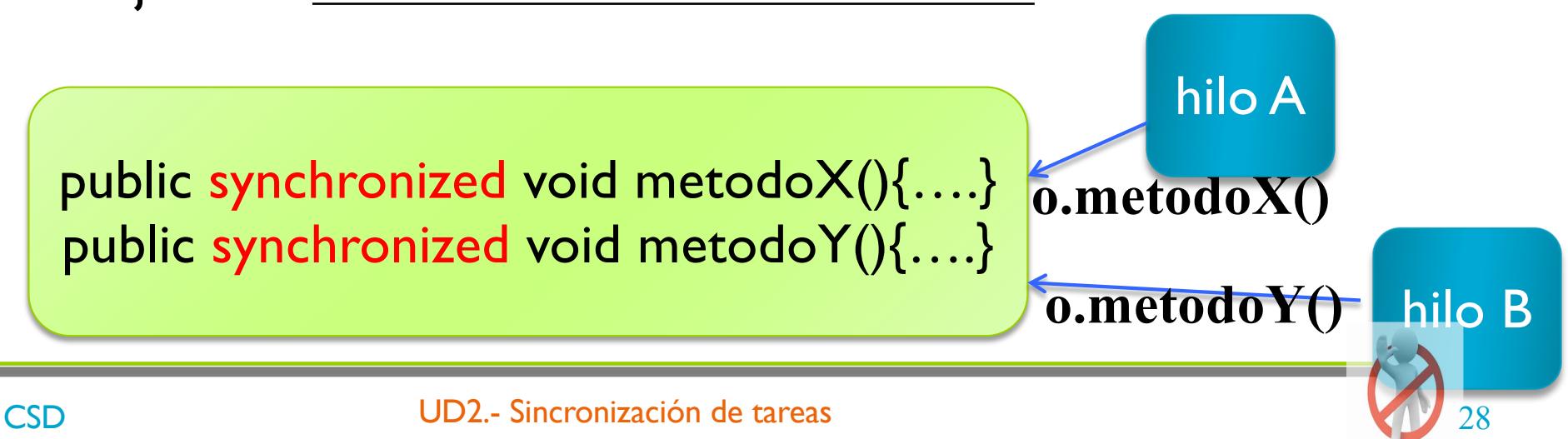
abrir lock

Sección no crítica



Locks en Java → ¿Cómo implementarlos?

- ▶ Todo objeto posee un lock asociado **implícito**
 - ▶ Etiquetar con **synchronized** los métodos que forman parte de la sección crítica
 - ▶ Al entrar en un método etiquetado como '**synchronized**' se cierra el lock, y al salir se abre.
 - ▶ Al salir, se establece una relación “ocurre-antes” con otras invocaciones posteriores de métodos sincronizados del mismo objeto.
 - ▶ Para un mismo objeto, todos sus métodos **synchronized** se ejecutan **en exclusión mutua entre sí**





Locks en Java → ¿Cómo implementarlos?

- ▶ **Implementación** → añadir etiqueta **synchronized** a todo método que:
 - ▶ Modifique un atributo que luego debe leer otro hilo
 - ▶ Lea un atributo actualizado por otro hilo



Locks en Java → ¿Cómo implementarlos?

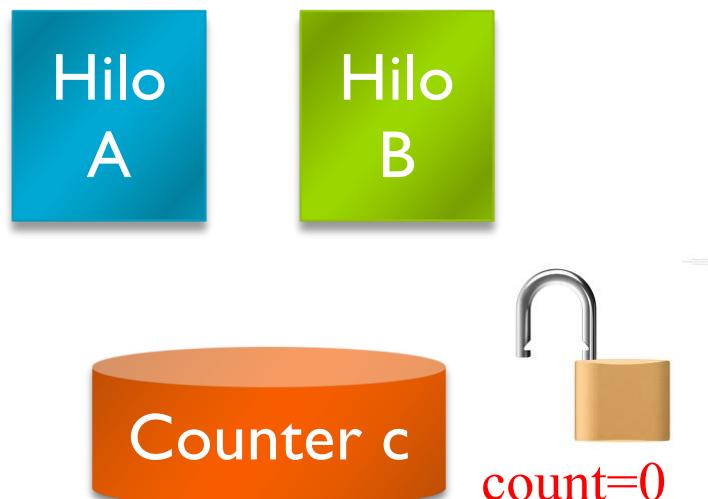
```
public class Counter {  
    private long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

Ejemplo: Versión del contador sin condiciones de carrera

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Inicialmente: lock implícito de
Counter está abierto



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

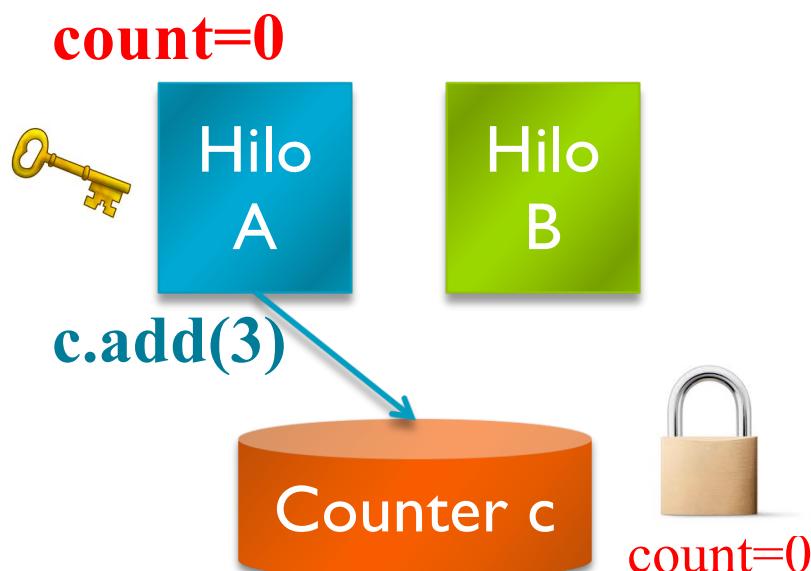
Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A cargo `c.count` (copia local=0)

Y se cierra
lock implícito



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

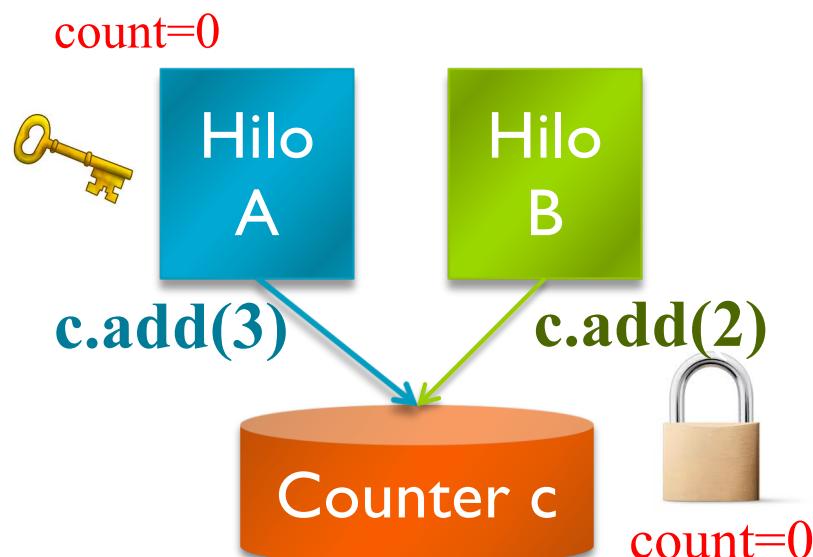
► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

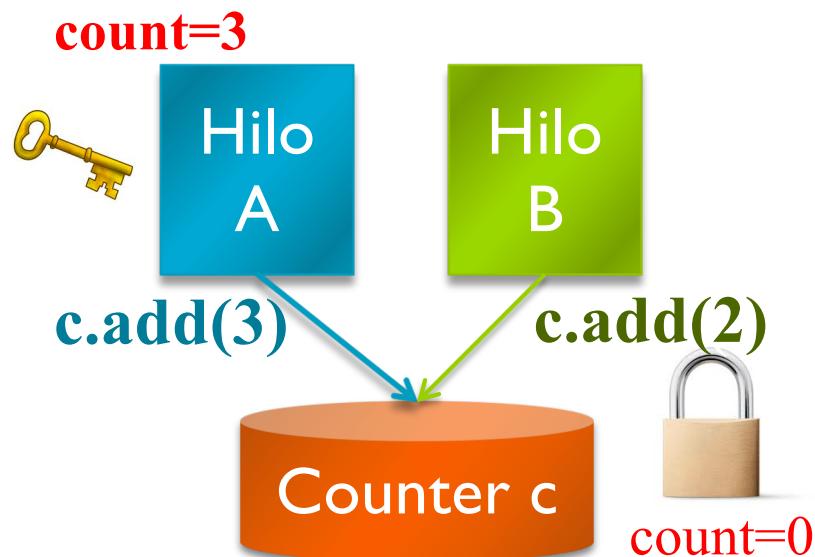
Hilo A ejecuta `c.add(3)`

A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

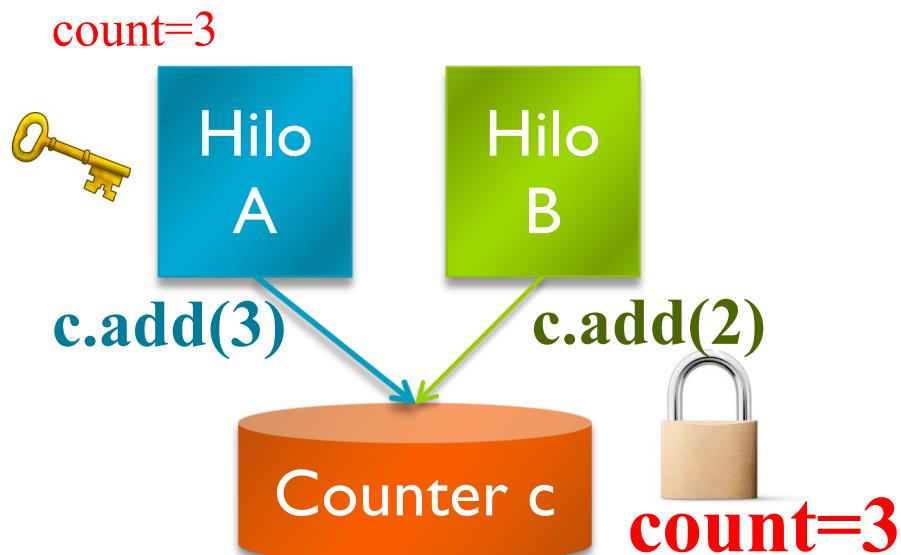
A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

A vuelve al heap (`c.count=3`)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)



```
public class Counter {
    protected long count = 0;
    public synchronized void add(long x) {
        count += x;
    }
    public synchronized long getCount() {
        return count;
    }
    ...
}
```

Counter c= new Counter(); // inic a 0

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

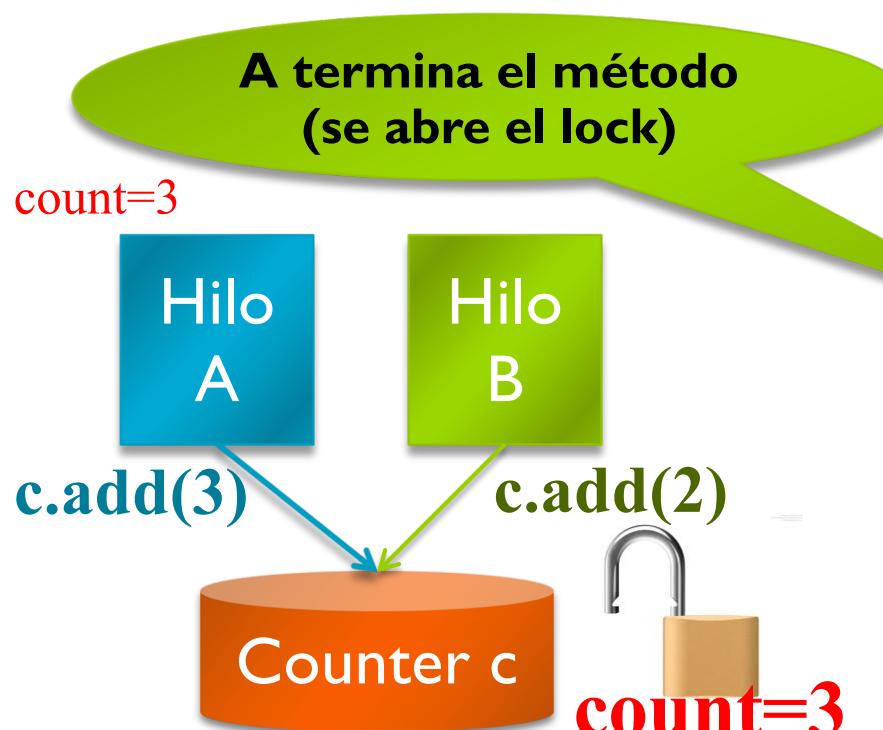
A carga `c.count` (copia local=0)

A suma 3 (copia local=3)

A vuelve al heap (`c.count=3`)

Hilo B ejecuta `c.add(2)`

B queda bloqueado (lock está cerrado, por otro hilo)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

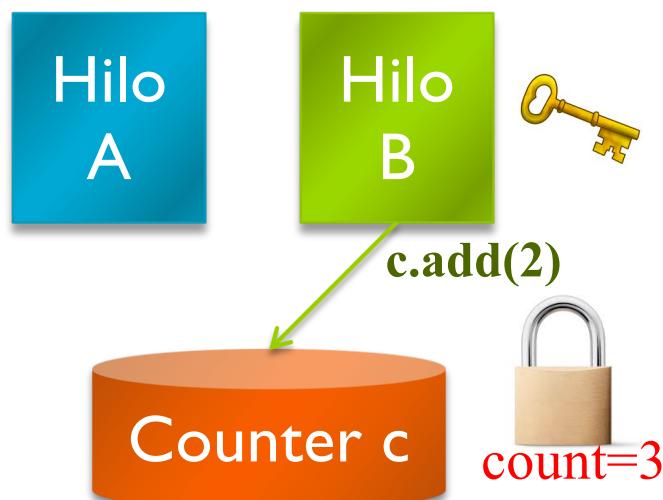
Locks en Java → Veamos un ejemplo

- ▶ Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

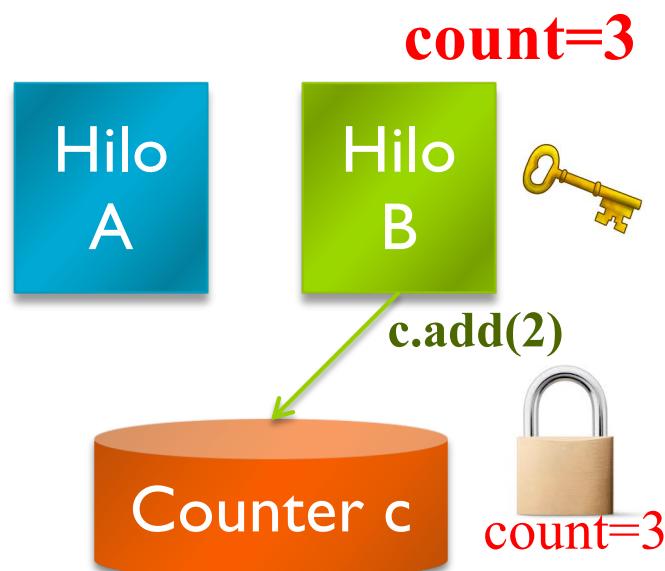
► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga c.count (copia local=3)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

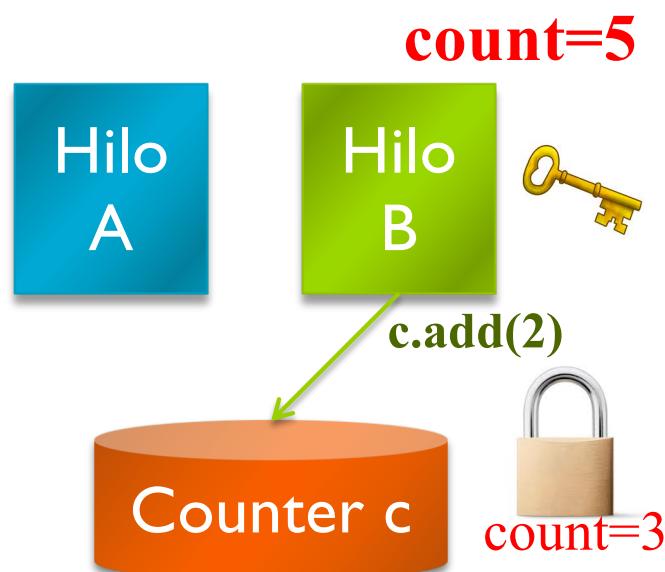
Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga `c.count` (copia local=3)

B suma 2 (copia local=5)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`

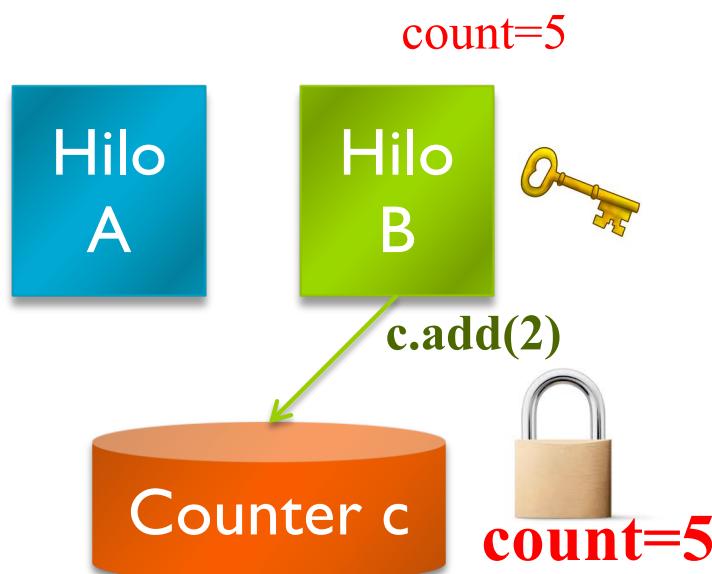
Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga `c.count` (copia local=3)

B suma 2 (copia local=5)

B vuelca al heap (`c.count=5`)



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
...  
Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

► Ejemplo de intercalado:

Hilo A ejecuta `c.add(3)`



Hilo B ejecuta `c.add(2)`

B se reactiva y cierra el lock

B carga `c.count` (copia local=3)

B suma 2 (copia local=5)

B vuelca al heap (`c.count=5`)

```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
  
    public synchronized long getCount() {  
        return count;  
    }  
}  
...  
Counter c= new Counter(); // inic a 0
```

Locks en Java → Veamos un ejemplo

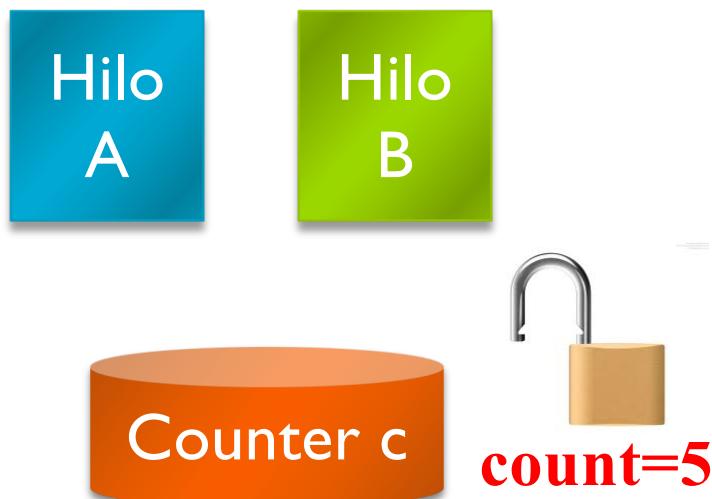
- ▶ Independientemente del intercalado:

Hilo A ejecuta `c.add(3)`

Hilo B ejecuta `c.add(2)`

Valor final: SIEMPRE 5!!

Código libre de
condiciones de carrera



```
public class Counter {  
    protected long count = 0;  
    public synchronized void add(long x) {  
        count += x;  
    }  
    public synchronized long getCount() {  
        return count;  
    }  
    ...  
    Counter c= new Counter(); // inic a 0
```



Sincronización en Java

- ▶ Java proporciona dos expresiones básicas de sincronización:
 - ▶ **Métodos sincronizados:** utilizando la etiqueta **synchronized** en la declaración del método
 - ▶ **Sentencias sincronizadas (Synchronized Statements):** indicando el objeto que proporciona el lock implícito
 - ▶ Permite utilizar más de un lock dentro de un mismo método.

- Los métodos *inc1* e *inc2* pueden intercalarse
- Pero la actualización de *c1* es en exclusión mutua
- Idem para *c2*

```
public class MsLunch{  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1){ c1++; }  
    }  
  
    public void inc2() {  
        synchronized(lock2){ c2++; }  
    }  
}
```



Sincronización Condicional

- ▶ El concepto de **sección crítica** evita interferencias en el acceso a las variables compartidas
- ▶ Pero también debemos resolver el problema de la *sincronización condicional*
 - ▶ Hacer esperar a un hilo hasta que se cumpla una determinada condición
- ▶ En el tema siguiente estudiamos el concepto de *monitor*
 - ▶ Resuelve ambos problemas (exclusión mutua y sincronización condicional) mediante una única construcción



Resultados de aprendizaje de la Unidad Didáctica

- ▶ Al finalizar esta unidad, el alumno deberá ser capaz de:
 - ▶ Identificar las secciones de una aplicación que deban o puedan ser ejecutadas concurrentemente por diferentes actividades.
 - ▶ Identificar los problemas de la comunicación con memoria compartida.
 - ▶ Identificar los tipos de sincronización.
 - ▶ Distinguir las secciones críticas de una aplicación y protegerlas mediante los mecanismos de sincronización adecuados.