

Sincronización con Semáforos

PROGRAMACIÓN CONCURRENTE – TEMA 3.1



Universidad
Rey Juan Carlos

Sincronización con Semáforos

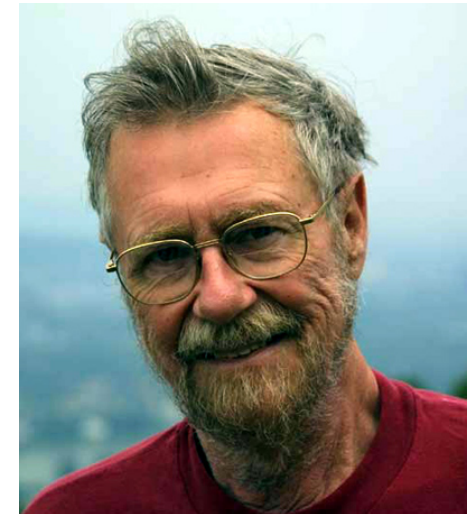
PROGRAMACIÓN CONCURRENTES

- **¿Qué es un Semáforo?**
- Sincronización Condicional
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
- Conclusiones

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Dijkstra (1968), introdujo la primera primitiva de sincronización de procesos y la denominó **Semáforo**
- Son herramientas de bajo nivel que permiten la sincronización condicional entre procesos y la exclusión mutua en el acceso a las secciones críticas



Dijkstra
(1930-2002)

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Un semáforo es una **Clase** (Programación Orientada a Objetos) o **Tipo Abstracto de Datos** (Programación Estructurada)
- En la parte privada tiene un **contador** de permisos y un **conjunto de procesos bloqueados**
- En la parte pública tiene métodos (o procedimientos) para **bloquear** y **desbloquear** procesos dependiendo del número de **permisos** del semáforo

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Los semáforos en Java se representan como objetos de la clase **java.util.concurrent.Semaphore**
- Se comportan igual que los semáforos diseñados por Dijkstra en el 68
 - Internamente tienen un contador de permisos (*permits*)
 - El método **acquire()** se bloquea hasta que adquiere un permiso
 - El método **release()** incrementa el número de permisos del semáforo

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- `void acquire()`

- Un proceso invoca este método para intentar “adquirir” un permiso del semáforo. Si lo consigue, continúa la ejecución. Si no, queda bloqueado
- Si el número de permisos del semáforo es mayor que 0 (**permits > 0**), se **decrementa** una unidad el número de permisos y el proceso **continúa** su ejecución

$$\text{permits} = \text{permits} - 1$$

- Si el número de permisos del semáforo es cero (**permits = 0**), el proceso suspende su ejecución, pasa al estado **bloqueado** y se añade al conjunto de procesos bloqueados en el semáforo

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- **void release() ;**

- Un proceso invoca este método para “liberar” un permiso previamente adquirido. Si otro proceso estaba esperando un permiso, lo consigue en la misma operación.
- Si no existen procesos bloqueados en el semáforo, este método **incrementa** el número de permisos y **continúa la ejecución**

$$\text{permits} = \text{permits} + 1$$

- Si existen procesos bloqueados en el semáforo, **se desbloquea** aleatoriamente a uno cualquiera y **continúa su ejecución**

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Se puede pensar en un semáforo como una caja de bolas
 - El número de permisos corresponde con el número de bolas de la caja
 - Cada bola corresponde a un permiso
 - `acquire()`
 - El proceso que ejecuta `acquire()` necesita **adquirir** una bola de la caja
 - Si no hay bolas, se bloquea hasta que estén disponibles
 - `release()`
 - Siempre **libera** una bola en la caja
 - Cuando una bola se libera, se incluye en la caja de nuevo
 - Si algún proceso estaba esperando bola, la coge y se desbloquea

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

Operación que ejecuta un proceso	Antes		Después	
	Permisos	Procesos Bloqueados	Permisos	Procesos Bloqueados
acquire()	3	Ninguno	2	Ninguno
acquire()	0	P ₁	0	P ₁ , P ₂
release()	1	Ninguno	2	Ninguno
release()	0	Ninguno	1	Ninguno
release()	0	P ₁ , P ₃	0	P ₁ *

* Podría haberse desbloqueado cualquiera de los procesos

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Diferentes nombres para las operaciones acquire y release
 - Las operaciones de gestión del semáforo reciben diferentes nombres dependiendo del sistema operativo, lenguaje de programación y/o librería.

acquire	release	Descripción
P	V	Los nombres que Dijkstra puso originalmente a las operaciones en holandés. V proviene de <i>verhogen</i> (incrementar) y P proviene de <i>portmanteau prolaag</i> (intentar reducir)
Down	Up	ALGOL 68, el kernel de Linux kernel y algunos libros de texto
Wait	Signal	PascalFC y algunos libros de texto
Pend	Post	
Procure	Vacate	Procure significa obtener y vacate desocupar

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Tipos de Semáforos

- Según el número de permisos

- Semáforos Binarios: Como máximo sólo pueden gestionar un permiso
- Semáforos Generales: Pueden gestionar cualquier número de permisos

- Según la política de desbloqueo de procesos

- FIFO (*First In, First Out*): Los procesos se desbloquean en orden de llegada
- Aleatorio: Los procesos se desbloquean aleatoriamente

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Justicia

- En el constructor se puede indicar la justicia (*fairness*) del semáforo
- Si es justo (**fair = true**) entonces los hilos se desbloquean en orden de llegada (**cola FIFO**)
- Si no es justo (**fair = false**) entonces los hilos se desbloquean de forma **aleatoria**
- Por defecto los semáforos en Java no son justos porque es más eficiente y en los usos normales no plantea problemas de **inanición** (*starvation*)

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Versiones del método **acquire()**
 - **acquire()**: Eleva la excepción **InterruptedException**
 - **acquireUninterruptibly()**: El hilo no se desbloquea en caso de que se interrumpa al hilo. No se lanza la **InterruptedException**.
 - **tryAcquire()**
 - Si hay permisos devuelve **true** y decrementa los permisos del semáforo
 - Si no hay permisos, devuelve **false** inmediatamente
 - **tryAcquire(long timeout, TimeUnit unit)**: Versión de **tryAcquire()** con tiempo de bloque máximo antes de devolver **false**

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Operaciones en bloque
 - Permiten adquirir o liberar varios permisos en una única llamada
 - **release(int permits)**
 - **acquire(int permits)**
 - **tryAcquire(int permits)**
 - **tryAcquire(int permits, long timeout, TimeUnit unit)**

¿Qué es un Semáforo?

SINCRONIZACIÓN CON SEMÁFOROS

- Métodos avanzados
 - Gestión de permisos
 - **int drainPermits()**: si hay permisos, ejecuta de forma atómica un `acquire()` por cada permiso. Si no, no hace nada
 - Gestión de hilos bloqueados
 - **getQueueLength()**: Devuelve el número de hilos bloqueados
 - **hasQueuedThreads()**: Indica si hay hilos bloqueados

Sincronización con Semáforos

PROGRAMACIÓN CONCURRENTE

- ¿Qué es un Semáforo?
- **Sincronización Condicional**
- Exclusión Mutua
- Metodología de Desarrollo
- Sincronización Avanzada
- Conclusiones

Sincronización Condicional

SINCRONIZACIÓN CON SEMÁFOROS

- Se produce cuando un proceso debe esperar a que se cumpla una cierta condición para proseguir su ejecución
- Esta condición sólo puede ser activada por otro proceso

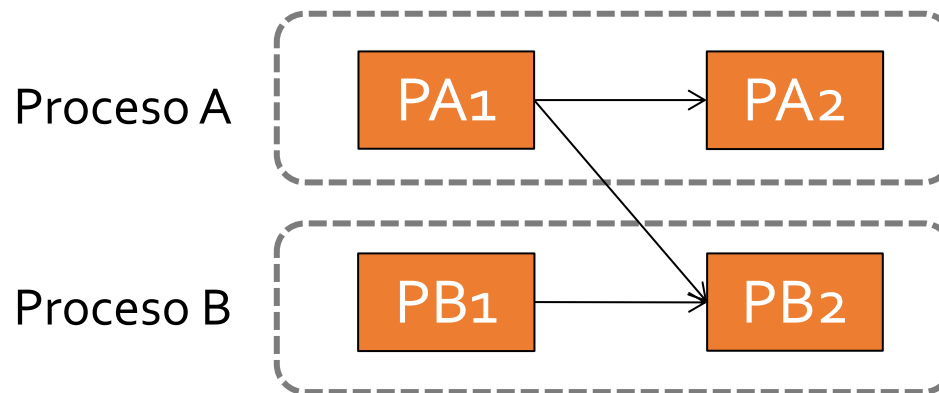


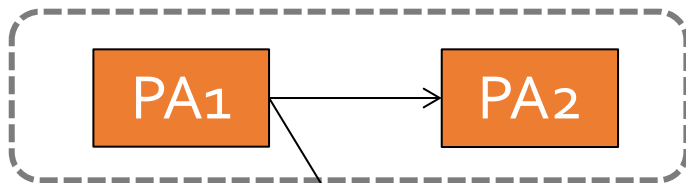
Diagrama de Precedencia

Sincronización Condicional

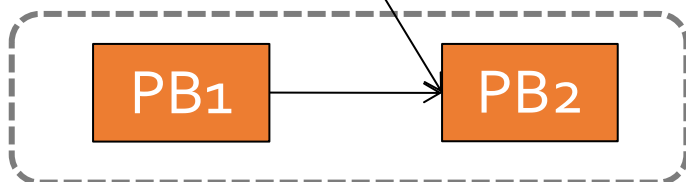
SINCRONIZACIÓN CON SEMÁFOROS

- Implementación con Espera Activa
- Ineficiente

Proceso A



Proceso B



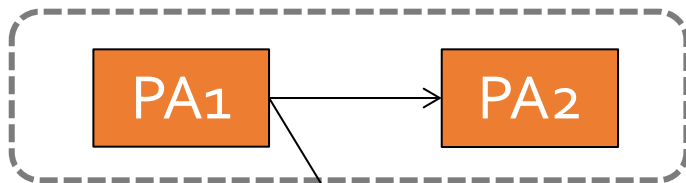
```
public class SincCondicionalEjemplo1 {  
  
    static volatile boolean continuar; ←  
    public static void procA() throws  
        InterruptedException{  
        new Thread(() -> {  
            System.out.println("PA1 ");  
            continuar = true; ←  
            System.out.println("PA2 ");  
        }).start();  
    }  
  
    public static void procB() throws  
        InterruptedException{  
        new Thread(() -> {  
            System.out.println("PB1 ");  
            while (!continuar) {} ←  
            System.out.println("PB2 ");  
        }).start();  
    }  
  
    public static void main(String[] args) throws  
        InterruptedException {  
        continuar = false; ←  
        procA();  
        procB();  
    }  
}
```

Sincronización Condicional

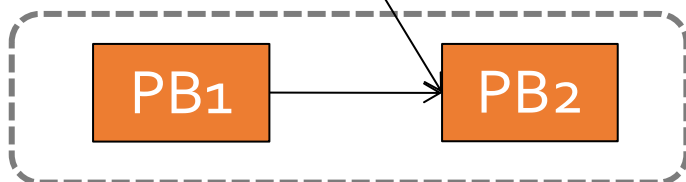
SINCRONIZACIÓN CON SEMÁFOROS

- Implementación con Semáforos
- Eficiente

Proceso A



Proceso B



```
public class SincronizacionCondicionalSemaforo {  
    static private Semaphore continuar; ←  
  
    public static void procesoA() {  
        new Thread() -> {  
            System.out.println("PA1 ");  
            continuar.release(); ←  
            System.out.println("PA2 ");  
        }, "Proceso A").start();  
    }  
  
    public static void procesoB() {  
        new Thread() -> {  
            System.out.println("PB1 ");  
            continuar.acquire(); ←  
            System.out.println("PB2 ");  
        }, "Proceso B").start();  
    }  
  
    public static void main(String[] args) {  
        continuar = new Semaphore(0); ←  
        procesoA();  
        procesoB();  
    }  
}
```

Sincronización Condicional

SINCRONIZACIÓN CON SEMÁFOROS

- Comportamiento de los procesos con herramientas bloqueantes
 - Bloquearse (Detener su ejecución)
 - Un proceso **se bloquea a sí mismo** si no puede proseguir su ejecución
 - Un proceso **nunca bloquea a otro** proceso
 - Desbloquearse (Reanudar su ejecución)
 - Un proceso **nunca se desbloquea a sí mismo**
 - Un proceso **desbloquea a otro** proceso si ese otro proceso puede proseguir su ejecución

Sincronización con Semáforos

PROGRAMACIÓN CONCURRENTE

- ¿Qué es un Semáforo?
- Sincronización Condicional
- **Exclusión Mutua**
- Metodología de Desarrollo
- Sincronización Avanzada
- Conclusiones

Exclusión Mutua

SINCRONIZACIÓN CON SEMÁFOROS

- **El problema de la Exclusión Mutua**

- Se tienen dos o más procesos (N) concurrentes, que ejecutan indefinidamente una secuencia de instrucciones dividida en dos secciones
- **Sección Crítica**
 - Representa una secuencia de instrucciones que debe ejecutar cada proceso sin que haya interferencia con la ejecución de instrucciones de las secciones críticas de los demás procesos.
 - Normalmente se corresponde con la utilización de un cierto recurso común de acceso exclusivo.
- **Sección No Crítica**
 - Representa las sentencias que pueden ser ejecutadas concurrentemente por todos los procesos

Exclusión Mutua

SINCRONIZACIÓN CON SEMÁFOROS

```
while(true) {  
  
    //Preprotocolo  
  
    // Sección Crítica  
    printlnI("P1_SC1");  
    printlnI("P1_SC2");  
  
    //Postprotocolo  
  
    // Sección No Crítica  
    printlnI("P1_SNC1");  
    printlnI("P1_SNC2");  
}
```

- El **preprotocolo** y **postprotocolo** son las secuencias de instrucciones que deben ejecutar los procesos para garantizar que las instrucciones de la **sección crítica** se ejecutan cumpliendo los requisitos
- Se asume que los procesos finalizan su ejecución fuera de la sección crítica, de forma que siempre hayan ejecutado el **postprotocolo**

Exclusión mutua

ALGORITMO DE DECKER





- La exclusión mutua con espera activa se implementa usando el **algoritmo de Dekker**
- A partir de ahora, usaremos los **semáforos** para implementar la exclusión mutua (y cualquier otra sincronización de procesos)

```
for (int i = 0; i < 5; i++) {  
  
    // Preprotocolo  
    p1p = true;  
    while (p2p) {  
        if (turno != 1) {  
            p1p = false;  
            while (turno != 1);  
            p1p = true;  
        }  
    }  
  
    // Sección Crítica  
    printlnI("P1_SC1 ");  
    printlnI("P1_SC2 ");  
  
    // Postprotocolo  
    p1p = false;  
    turno = 2;  
  
    // Sección No Crítica  
    printlnI("P1_SNC1 ");  
    printlnI("P1_SNC2 ");  
}
```

```
static volatile boolean p1p;  
static volatile boolean p2p;  
static volatile int turno;
```


Exclusion Mutua

- Cuando el semáforo tiene 1 permiso (*permits=1*), la sección crítica está libre
- Cuando el semáforo no tiene permisos (*permits=0*), la sección crítica está ocupada por otro proceso
- Para entrar en la sección crítica hay que coger una bola de la caja, y dejarla al salir para que la pueda coger otro proceso

```
public class ExclusionMutuaSemaforo {  
  
    private static Semaphore semExcMut;   
  
    private static void p() throws InterruptedException {  
        for (int i = 0; i < 5; i++) {  
            // Preprotocolo  
            semExcMut.acquire();   
  
            // Sección Crítica  
            System.out.println("SC1 ");  
            System.out.println("SC2 ");  
  
            // Postprotocolo  
            semExcMut.release();   
  
            // Sección No Crítica  
            System.out.println("SNC1 ");  
            System.out.println("SNC2 ");  
        }  
    }  
  
    public static void proceso(String name) {  
        new Thread(() -> {  
            //En un try/catch  
            p();  
        }, name).start();  
    }  
  
    public static void main(String[] args) {  
        semExcMut = new Semaphore(1);   
        proceso("hilo 1");  
        proceso("hilo 2");  
    }  
}
```

Sincronización con Semáforos

PROGRAMACIÓN CONCURRENTES

- ¿Qué es un Semáforo?
- Sincronización Condicional
- Exclusión Mutua
- **Metodología de Desarrollo**
- Sincronización Avanzada
- Conclusiones

Metodología de Desarrollo

SINCRONIZACIÓN CON SEMÁFOROS

- A medida que los programas concurrentes se hacen más **complejos**, se hace necesario seguir una **metodología** que permita guiar el desarrollo de los mismos
- A continuación se definen una serie de **pasos** que se pueden seguir para crear un **programa concurrente** partiendo de unos **requisitos**

Metodología de Desarrollo

SINCRONIZACIÓN CON SEMÁFOROS

- **1)** Definir la arquitectura de los procesos
 - Número de Procesos
 - Tipo de Procesos
- **2)** Implementar lo que tiene que hacer cada proceso de forma secuencial
- **3)** Determinar los puntos de sincronización en el código
 - ¿Sincronización Condicional o Exclusión Mutua?
 - Número de Semáforos Necesarios: ¿Se pueden bloquear todos los procesos juntos?
¿Se puede desbloquear cualquiera de ellos?
- **4)** Programación de `acquire()` y `release()` definiendo las variables necesarias para controlar la sincronización
- **5)** Gestión de variables
 - Inicialización de booleanas y contadores
 - Bajo Exclusión Mutua si son compartidas