

Semáforos

Programación concurrente

Hasta ahora vimos

Hasta ahora vimos

- ▶ Características de la concurrencia

Hasta ahora vimos

- ▶ Características de la concurrencia
 - ▶ Ventajas

Hasta ahora vimos

- ▶ Características de la concurrencia
 - ▶ Ventajas
 - ▶ Problemas

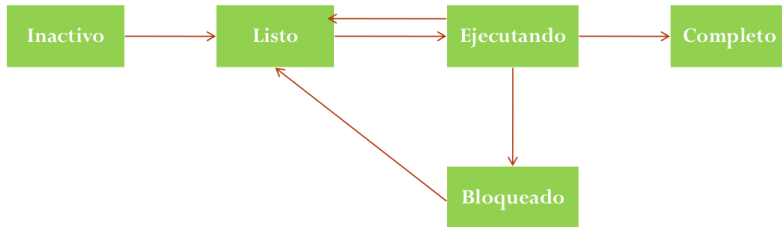
Hasta ahora vimos

- ▶ Características de la concurrencia
 - ▶ Ventajas
 - ▶ Problemas
 - ▶ Exclusión mutua

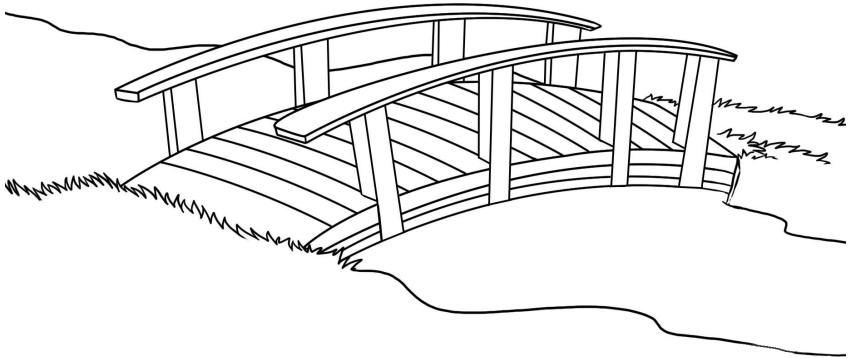
Hasta ahora vimos

- ▶ Características de la concurrencia
 - ▶ Ventajas
 - ▶ Problemas
 - ▶ Exclusión mutua
 - ▶ Operaciones atómicas

Estados de un proceso



Ejemplo: Puente



Un semáforo es un tipo abstracto de datos con las operaciones:

- ▶ `acquire`
- ▶ `release`

Un semáforo es un tipo abstracto de datos con las operaciones:

- ▶ `acquire`
- ▶ `release`

Una posible representación interna consta de:

- ▶ `entero: permisos`
- ▶ `conjunto: procesos`

Acquire

Acquire consume un permiso o espera si no hay uno disponible.

```
atomic acquire() {  
    currentThread = Thread.currentThread();  
    if (permisos > 0) {  
        permisos--;  
    } else {  
        procesos.add(currentThread);  
        currentThread.state = BLOCKED;  
    }  
}
```

Release libera un permiso (despierta a un thread bloqueado)

```
atomic release() {  
    if (procesos.empty()) {  
        permisos++;  
    } else {  
        wakingThread = procesos.removeAny();  
        wakingThread.state = READY;  
    }  
}
```

Mutex

Llamaremos mutex a un semáforo que sólo admite 0 o 1 permisos.

```
atomic release() {  
    if (permisos == 1) {  
        // indefinido  
    } else if (procesos.empty()) {  
        permisos++;  
    } else {  
        wakingThread = procesos.removeAny();  
        wakingThread.state = READY;  
    }  
}
```

Mutex

Llamaremos mutex a un semáforo que sólo admite 0 o 1 permisos.

```
atomic release() {  
    if (permisos == 1) {  
        // indefinido  
    } else if (procesos.empty()) {  
        permisos++;  
    } else {  
        wakingThread = procesos.removeAny();  
        wakingThread.state = READY;  
    }  
}
```

Usaremos un Semaphore y nos aseguraremos de nunca liberar permisos inválidos.

Exclusión mutua usando mutex

Exclusión mutua usando mutex

```
global Semaphore mutex = new Semaphore(1);

thread {                                thread {
    // seccion no critica                // seccion no critica
    mutex.acquire();                     mutex.acquire();
    // seccion critica                   // seccion critica
    mutex.release();                     mutex.release();
    // seccion no critica                 // seccion no critica
}
```

Exclusión mutua usando mutex

```
global Semaphore mutex = new Semaphore(1);

thread {                                thread {
    // seccion no critica                // seccion no critica
    mutex.acquire();                     mutex.acquire();
    // seccion critica                   // seccion critica
    mutex.release();                     mutex.release();
    // seccion no critica                 // seccion no critica
}
```

Esta solución no usa busy waiting ya que un proceso bloqueado en el acquire pasa al estado bloqueado y no vuelve al estado listo hasta tanto se le de permiso.

Invariantes de un semáforo

Invariantes de un semáforo

▶ `permisos` ≥ 0

Invariantes de un semáforo

- ▶ $\text{permisos} \geq 0$
- ▶ $\text{permisos} = k + \text{\#releases} - \text{\#acquires}$

Invariantes de un semáforo

- ▶ $\text{permisos} \geq 0$
- ▶ $\text{permisos} = k + \text{\#releases} - \text{\#acquires}$

Nota: Se considera que un proceso bloqueado no realizó la operación acquire.

Invariantes de la exclusión mutua con semáforos

Invariantes de la exclusión mutua con semáforos

▶ $\#criticalSections + \textit{permisos} = 1$

Invariantes de la exclusión mutua con semáforos

- ▶ $\#criticalSections + \textit{permisos} = 1$
- ▶ $\#criticalSections = \#acquires - \#releases$

Invariantes de la exclusión mutua con semáforos

- ▶ $\#criticalSections + \textit{permisos} = 1$
- ▶ $\#criticalSections = \#acquires - \#releases$

Esto garantiza:

Invariantes de la exclusión mutua con semáforos

- ▶ $\#criticalSections + \textit{permisos} = 1$
- ▶ $\#criticalSections = \#acquires - \#releases$

Esto garantiza:

- ▶ Exclusión mutua ($\#criticalSections \leq 1$)

Invariantes de la exclusión mutua con semáforos

- ▶ $\#criticalSections + permisos = 1$
- ▶ $\#criticalSections = \#acquires - \#releases$

Esto garantiza:

- ▶ Exclusión mutua ($\#criticalSections \leq 1$)
- ▶ Ausencia de *deadlock* (no sucede que $permisos = 0$ y $\#criticalSections = 0$)

Invariantes de la exclusión mutua con semáforos

- ▶ $\#criticalSections + permisos = 1$
- ▶ $\#criticalSections = \#acquires - \#releases$

Esto garantiza:

- ▶ Exclusión mutua ($\#criticalSections \leq 1$)
- ▶ Ausencia de *deadlock* (no sucede que $permisos = 0$ y $\#criticalSections = 0$)
- ▶ No hay *starvation* entre dos procesos

El problema del comensal

```
global Semaphore mutex = new Semaphore(1);

comensal() {
    while (true) {
        mutex.acquire();
        comer();
        mutex.release();
    }
}

repeat (N)
    thread comensal();
```

El problema del comensal

```
global Semaphore mutex = new Semaphore(1);

comensal() {
    while (true) {
        mutex.acquire();
        comer();
        mutex.release();
    }
}

repeat (N)
    thread comensal();
```

- ▶ Si el semáforo usa un conjunto (débil) este programa tiene starvation.

El problema del comensal

```
global Semaphore mutex = new Semaphore(1);

comensal() {
    while (true) {
        mutex.acquire();
        comer();
        mutex.release();
    }
}

repeat (N)
    thread comensal();
```

- ▶ Si el semáforo usa un conjunto (débil) este programa tiene starvation.
- ▶ Si el semáforo usa una cola (fuerte) este programa no tiene starvation.

Semáforos fuertes

Cuando tenemos más de dos procesos existe posibilidad de *starvation*.

Esto puede solucionarse cambiando el conjunto por una cola.

Semáforos fuertes

Cuando tenemos más de dos procesos existe posibilidad de *starvation*.

Esto puede solucionarse cambiando el conjunto por una cola.

En java se indica en la construcción

```
/** Creates a Semaphore with the given number of permits  
    and the given fairness setting. */  
Semaphore(int permits, boolean fair)
```

El problema del molinete

```
global int contador = 0;
global Semaphore mutex = new Semaphore(1);

molinete() {
    repeat (100) {
        mutex.acquire();
        contador++;
        mutex.release();
    }
}

repeat (N)
    thread molinete();
```

Sincronización de procesos

¿Cómo hago para imprimir el total del contador de N molinetes?

Sincronización de procesos

¿Cómo hago para imprimir el total del contador de N molinetes?

```
repeat (N)
  thread molinete();
print("Total = " + contador);
```

Sincronización de procesos

¿Cómo hago para imprimir el total del contador de N molinetes?

```
repeat (N)
    thread molinete();
print("Total = " + contador);
```

¿Qué sucede al ejecutar este código?

Sincronización de procesos (molinete)

```
global int contador = 0;
global Semaphore mutex = new Semaphore(1);
global Semaphore finish = new Semaphore(0);

molinete() {
    repeat (100) {
        mutex.acquire();
        contador++;
        mutex.release();
    }
    finish.release();
}

repeat (N)
    thread molinete();
```

Sincronización de procesos (molinete)

```
global int contador = 0;
global Semaphore mutex = new Semaphore(1);
global Semaphore finish = new Semaphore(0);

molinete() {
    repeat (100) {
        mutex.acquire();
        contador++;
        mutex.release();
    }
    finish.release();
}

repeat (N)
    thread molinete();

repeat (N)
    finish.acquire();
print("Total = " + contador);
```


Filósofos comensales

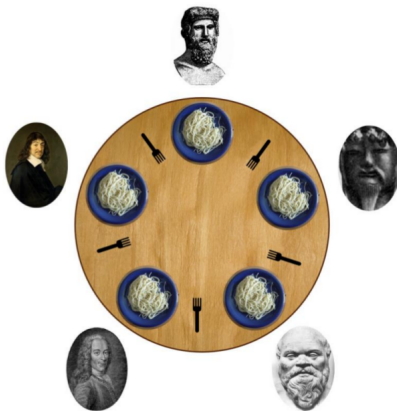


Filósofos comensales



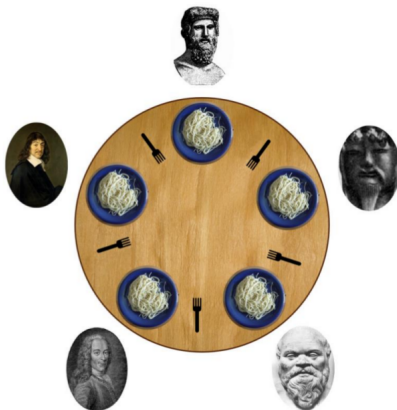
- Los filósofos piensan y comen alternadamente

Filósofos comensales



- ▶ Los filósofos piensan y comen alternadamente
- ▶ Sólo puede comer cuando tiene dos tenedores

Filósofos comensales



- ▶ Los filósofos piensan y comen alternadamente
- ▶ Sólo puede comer cuando tiene dos tenedores
- ▶ Sólo se pueden tomar los tenedores a izquierda y derecha

Filósofos comensales (características)

```
Filosofo(id) {  
    while (true)  
        // pensar  
        // tomar tenedores  
        // comer  
        // dejar tenedores  
}
```

Filósofos comensales (características)

```
Filosofo(id) {  
    while (true)  
        // pensar  
        // tomar tenedores  
        // comer  
        // dejar tenedores  
}
```

- ▶ Mutex: en un momento dado sólo un filósofo puede tener un tenedor dado
- ▶ Sincronización: un filósofo sólo puede comer cuando tiene dos tenedores
- ▶ Deadlock
- ▶ Livelock
- ▶ Starvation

Filósofos comensales (intento naive)

```
global Semaphore[] tenedores = [1,...,1]; // N

Filosofo(id) {
    izq = id;
    der = (id+1) % N;

    while (true) {
        // pensar
        tenedores[izq].acquire();
        tenedores[der].acquire();
        // comer
        tenedores[izq].release();
        tenedores[der].release();
    }
}
```

Filósofos comensales (intento naive)

```
global Semaphore[] tenedores = [1,...,1]; // N

Filosofo(id) {
    izq = id;
    der = (id+1) % N;

    while (true) {
        // pensar
        tenedores[izq].acquire();
        tenedores[der].acquire();
        // comer
        tenedores[izq].release();
        tenedores[der].release();
    }
}
```

Deadlock: Si todos toman el tenedor izquierdo se produce una espera circular.

Filósofos comensales (semáforo general)

```
global Semaphore[] tenedores = [1,...,1]; // N
global Semaphore sillas = new Semaphore(N-1);

Filosofo(id) {
    izq = id;
    der = (id+1) % N;

    while (true) {
        // pensar
        sillas.acquire();
        tenedores[izq].acquire();
        tenedores[der].acquire();
        // comer
        tenedores[izq].release();
        tenedores[der].release();
        sillas.release();
    }
}
```

Filósofos comensales (ruptura de simetría)

```
global Semaphore[] tenedores = [1,...,1]; // N

Filosofo(id) {
    if (id == 0) {
        izq = 1;
        der = 0;
    } else {
        izq = id;
        der = (id+1) % N;
    }

    while (true) {
        // pensar
        tenedores[izq].acquire();
        tenedores[der].acquire();
        // comer
        tenedores[izq].release();
        tenedores[der].release();
    }
}
```