

Esquemas con semáforos

Programación concurrente

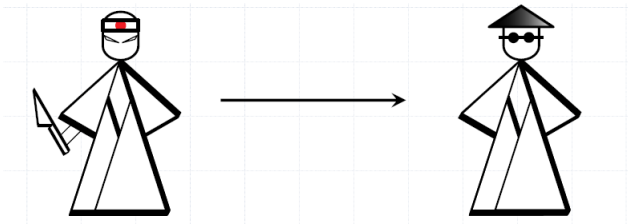
- ▶ **Semáforo:** Un tipo abstracto de datos con dos operaciones
 - ▶ acquire
 - ▶ release
- ▶ Con semáforos podemos:
 - ▶ Resolver el problema de la exclusión mutua.
 - ▶ Sincronizar threads cooperativos.

Hoy veremos

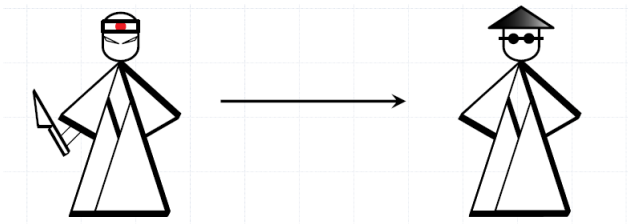
- ▶ Problemas recurrentes en el área
- ▶ Soluciones esquemáticas probadas

Productor consumidor

Productor consumidor

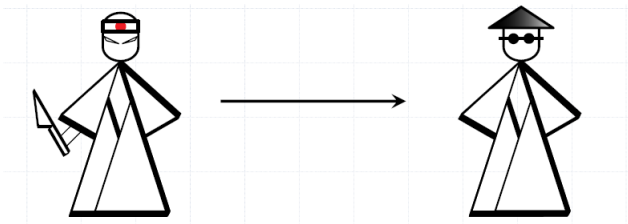


Productor consumidor



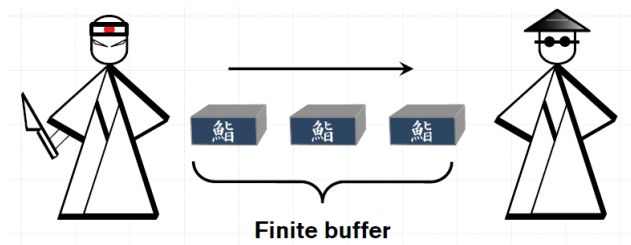
- Un patrón de interacción frecuente.

Productor consumidor

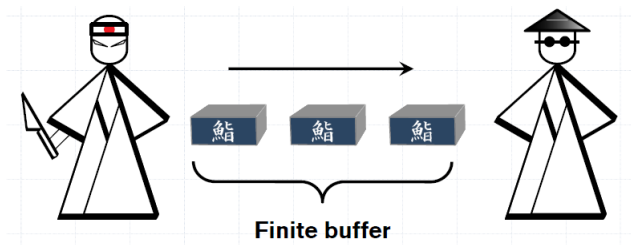


- ▶ Un patrón de interacción frecuente.
- ▶ Debe contemplar la diferencia de velocidad entre las partes.

Buffer acotado

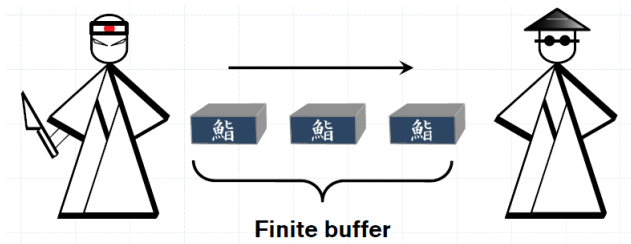


Buffer acotado



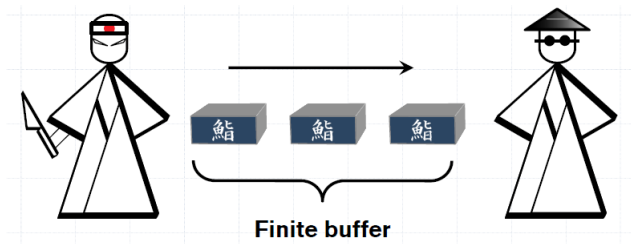
- ▶ ¿Cuándo puede producir el productor?

Buffer acotado



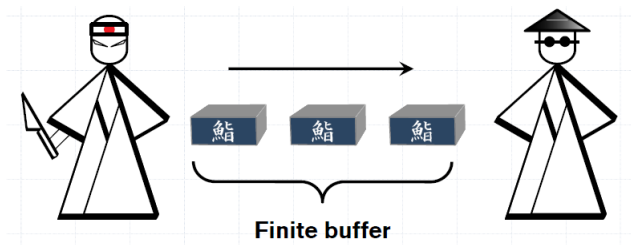
- ▶ ¿Cuándo puede producir el productor?
- ▶ ¿Cuándo puede consumir el consumidor?

Buffer acotado



- ▶ ¿Cuándo puede producir el productor? **Cuando hay espacio.**
- ▶ ¿Cuándo puede consumir el consumidor?

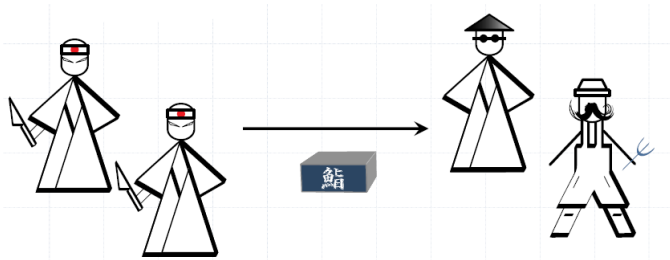
Buffer acotado



- ▶ ¿Cuándo puede producir el productor? **Cuando hay espacio.**
- ▶ ¿Cuándo puede consumir el consumidor? **Cuando hay platos.**

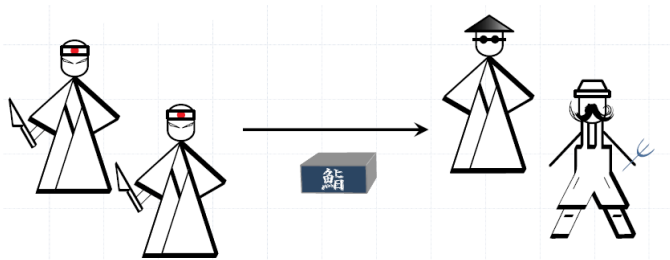
Buffer usando semáforos

► Capacidad 1



Buffer usando semáforos

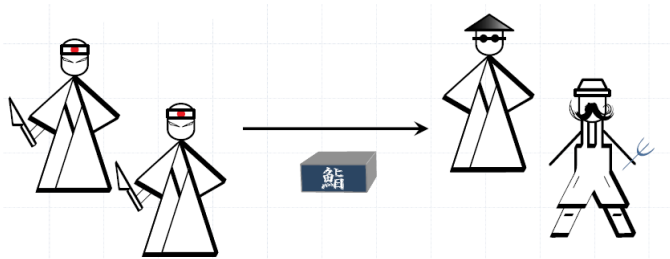
► Capacidad 1



► Varios productores

Buffer usando semáforos

- ▶ Capacidad 1

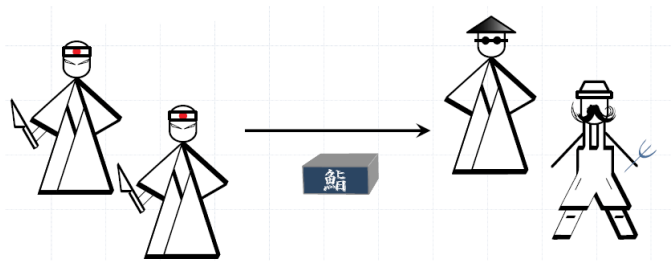


- ▶ Varios productores

- ▶ Varios consumidores

Buffer usando semáforos

- ▶ Capacidad 1



- ▶ Varios productores
- ▶ Varios consumidores
- ▶ ¿Qué semáforos necesitamos?

Semáforos binarios *split*

```
global Object buffer;

thread Productor:                thread Consumidor:

    while (true) {                while (true) {

        buffer = producir();        consumir(buffer);

    }                               }
```

Semáforos binarios *split*

```
global Object buffer;
global Semaphore vacio = new Semaphore(1);
global Semaphore lleno = new Semaphore(0);

thread Productor:                                thread Consumidor:

    while (true) {                                while (true) {
        vacio.acquire();                          lleno.acquire();
        buffer = producir();                      consumir(buffer);
        lleno.release();                          vacio.release();
    }                                              }
```

Semáforos binarios *split*

- ▶ Dos semáforos
- ▶ Inicialización
- ▶ Invariante

Semáforos binarios *split*

- ▶ Dos semáforos
 - ▶ 1 para indicar cuando está vacío
- ▶ Inicialización
- ▶ Invariante

Semáforos binarios *split*

- ▶ Dos semáforos
 - ▶ 1 para indicar cuando está vacío
 - ▶ 1 para indicar cuando está lleno
- ▶ Inicialización
- ▶ Invariante

Semáforos binarios *split*

- ▶ Dos semáforos
 - ▶ 1 para indicar cuando está vacío
 - ▶ 1 para indicar cuando está lleno
- ▶ Inicialización
 - ▶ `vacio = 1`
- ▶ Invariante

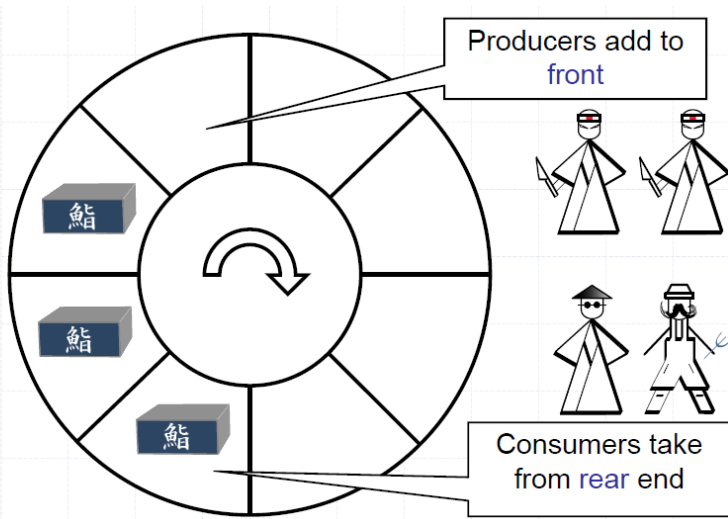
Semáforos binarios *split*

- ▶ Dos semáforos
 - ▶ 1 para indicar cuando está vacío
 - ▶ 1 para indicar cuando está lleno
- ▶ Inicialización
 - ▶ `vacio = 1`
 - ▶ `lleno = 0`
- ▶ Invariante

Semáforos binarios *split*

- ▶ Dos semáforos
 - ▶ 1 para indicar cuando está vacío
 - ▶ 1 para indicar cuando está lleno
- ▶ Inicialización
 - ▶ `vacío = 1`
 - ▶ `lleno = 0`
- ▶ Invariante
 - ▶ `vacío + lleno ≤ 1`

Buffer de tamaño N



Semáforos generales

Semáforos generales

- ▶ Los semáforos cuentan la cantidad de espacios en el buffer

Semáforos generales

- ▶ Los semáforos cuentan la cantidad de espacios en el buffer
- ▶ Inicialización
- ▶ Invariante

Semáforos generales

- ▶ Los semáforos cuentan la cantidad de espacios en el buffer
- ▶ Inicialización
 - ▶ Hay N espacios vacíos
- ▶ Invariante

Semáforos generales

- ▶ Los semáforos cuentan la cantidad de espacios en el buffer
- ▶ Inicialización
 - ▶ Hay N espacios vacíos
 - ▶ Hay 0 espacios llenos
- ▶ Invariante

Semáforos generales

- ▶ Los semáforos cuentan la cantidad de espacios en el buffer
- ▶ Inicialización
 - ▶ Hay N espacios vacíos
 - ▶ Hay 0 espacios llenos
- ▶ Invariante
 - ▶ `vacio + lleno <= N`

Prodctor-Consumidor únicos

```
global Object[]  buffer = new Object[N];
global Semaphore vacio  = new Semaphore(N);
global Semaphore lleno  = new Sempahore(0);
global int inicio = 0;
global int fin     = 0;

thread Productor:
    while (true) {
        vacio.acquire();
        buffer[inicio] = producir();
        inicio = (inicio+1) % N;
        lleno.release();
    }

thread Consumidor:
    while (true) {
        lleno.acquire();
        consumir(buffer[fin]);
        fin = (fin+1) % N;
        vacio.release();
    }
```


Múltiples productores

Hay que garantizar exclusión mutua entre los productores.

Múltiples productores

Hay que garantizar exclusión mutua entre los productores.

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

Productor() {
    while (true) {
        vacio.acquire();
        mutexP.acquire();
        buffer[inicio] = producir();
        inicio = (inicio+1) % N;
        mutexP.release();
        lleno.release();
    }
}
```

Múltiples consumidores

Hay que garantizar exclusión mutua entre los consumidores.

Múltiples consumidores

Hay que garantizar exclusión mutua entre los consumidores.

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

Consumidor() {
    while (true) {
        lleno.acquire();
        mutexC.acquire();
        consumir(buffer[fin]);
        fin = (fin+1) % N;
        mutexC.release();
        vacio.release();
    }
}
```

Lectores escritores

Lectores escritores

- ▶ Existen recursos compartidos por dos tipos de threads

Lectores escritores

- ▶ Existen recursos compartidos por dos tipos de threads
 - ▶ *Lectores*: acceden al recurso sin modificarlo
 - ▶ *Escritores*: acceden al recurso y pueden modificarlo

Lectores escritores

- ▶ Existen recursos compartidos por dos tipos de threads
 - ▶ *Lectores*: acceden al recurso sin modificarlo
 - ▶ *Escritores*: acceden al recurso y pueden modificarlo
- ▶ La exclusión mutua resulta demasiado restrictiva

- ▶ Existen recursos compartidos por dos tipos de threads
 - ▶ *Lectores*: acceden al recurso sin modificarlo
 - ▶ *Escritores*: acceden al recurso y pueden modificarlo
- ▶ La exclusión mutua resulta demasiado restrictiva
 - ▶ *Lectores*: pueden acceder al mismo tiempo
 - ▶ *Escritores*: como máximo uno en cualquier momento

Características de una solución

Características de una solución

- ▶ Cada operación de leer o escribir debe ocurrir dentro de una sección crítica

Características de una solución

- ▶ Cada operación de leer o escribir debe ocurrir dentro de una sección crítica
- ▶ Debe garantizar exclusión mutua entre los escritores

Características de una solución

- ▶ Cada operación de leer o escribir debe ocurrir dentro de una sección crítica
- ▶ Debe garantizar exclusión mutua entre los escritores
- ▶ Debe permitir que múltiples lectores ejecuten su sección crítica simultáneamente

Primera solución: Prioridad lectores

Primera solución: Prioridad lectores

- ▶ Se usa un semáforo para controlar el permiso de escritura

Primera solución: Prioridad lectores

- ▶ Se usa un semáforo para controlar el permiso de escritura
- ▶ Antes de escribir se debe obtener el permiso y liberarlo al terminar

Primera solución: Prioridad lectores

- ▶ Se usa un semáforo para controlar el permiso de escritura
- ▶ Antes de escribir se debe obtener el permiso y liberarlo al terminar
- ▶ El primer lector debe “robar” el permiso de escritura y el último devolverlo

Primera solución: Prioridad lectores

- ▶ Se usa un semáforo para controlar el permiso de escritura
- ▶ Antes de escribir se debe obtener el permiso y liberarlo al terminar
- ▶ El primer lector debe “robar” el permiso de escritura y el último devolverlo
 - ▶ Es necesario contar la cantidad de lectores dentro

Primera solución: Prioridad lectores

- ▶ Se usa un semáforo para controlar el permiso de escritura
- ▶ Antes de escribir se debe obtener el permiso y liberarlo al terminar
- ▶ El primer lector debe “robar” el permiso de escritura y el último devolverlo
 - ▶ Es necesario contar la cantidad de lectores dentro
 - ▶ A su vez es necesario acceder al contador en exclusión mutua

Primera solución: Prioridad lectores

```
global Semaphore permisoE = new Semaphore(1);
global Semaphore mutexL   = new Semaphore(1);
global Semaphore mutexP   = new Semaphore(1);
global int lectores = 0;
```

```
Escritor() {
    mutexP.acquire();
    permisoE.acquire();
    escribir();
    permisoE.release();
    mutexP.release();
}
```

```
Lector() {
    mutexL.acquire();
    lectores++;
    if (lectores == 1)
        permisoE.acquire();
    mutexL.release();

    leer();

    mutexL.acquire();
    lectores--;
    if (lectores == 0)
        permisoE.release();
    mutexL.release();
}
```

Primera solución: Prioridad lectores

```
global Semaphore permisoE = new Semaphore(1);
global Semaphore mutexL   = new Semaphore(1);
global Semaphore mutexP   = new Semaphore(1);
global int lectores = 0;
```

```
Escritor() {
    mutexP.acquire();
    permisoE.acquire();
    escribir();
    permisoE.release();
    mutexP.release();
}
```

```
Lector() {
    mutexL.acquire();
    lectores++;
    if (lectores == 1)
        permisoE.acquire();
    mutexL.release();

    leer();

    mutexL.acquire();
    lectores--;
    if (lectores == 0)
        permisoE.release();
    mutexL.release();
}
```

Observación: Los escritores pueden sufrir starvation.

Primera solución: Prioridad lectores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Lector1
 2. Llega Escritor1 (espera)
 3. Llega Escritor2 (espera)
 4. Sale Lector1
 5. Entra Escritor1
 6. Llega Lector2 (espera)
 7. Sale Escritor1
- ▶ ¿Quién debería entrar a continuación?
 - ▶ ¿Quién entra si no está el mutexP?

Primera solución: Prioridad lectores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Lector1
 2. Llega Escritor1 (espera)
 3. Llega Escritor2 (espera)
 4. Sale Lector1
 5. Entra Escritor1
 6. Llega Lector2 (espera)
 7. Sale Escritor1
- ▶ ¿Quién debería entrar a continuación?
El Lector2, ya que tiene prioridad.
 - ▶ ¿Quién entra si no está el mutexP?

Primera solución: Prioridad lectores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Lector1
2. Llega Escritor1 (espera)
3. Llega Escritor2 (espera)
4. Sale Lector1
5. Entra Escritor1
6. Llega Lector2 (espera)
7. Sale Escritor1

► ¿Quién debería entrar a continuación?

El Lector2, ya que tiene prioridad.

► ¿Quién entra si no está el mutexP?

El Escritor2, ya que habría sido encolado primero en permisoE.

Segunda solución: Prioridad escritores

Segunda solución: Prioridad escritores

- ▶ Los lectores no pueden entrar si hay escritores esperando

Segunda solución: Prioridad escritores

- ▶ Los lectores no pueden entrar si hay escritores esperando
 - ▶ Es necesario contar la cantidad de escritores esperando

Segunda solución: Prioridad escritores

- ▶ Los lectores no pueden entrar si hay escritores esperando
 - ▶ Es necesario contar la cantidad de escritores esperando
 - ▶ A su vez es necesario acceder al contador en exclusión mutua

Segunda solución: Prioridad escritores

- ▶ Los lectores no pueden entrar si hay escritores esperando
 - ▶ Es necesario contar la cantidad de escritores esperando
 - ▶ A su vez es necesario acceder al contador en exclusión mutua
- ▶ Antes de leer los lectores deben obtener el permiso de lectura

Segunda solución: Prioridad escritores


```
Escritor() {  
  
    mutexE.acquire();  
    escritores++;  
    if (escritores == 1)  
        permisoL.acquire();  
    mutexE.release();  
  
    permisoE.acquire();  
    escribir();  
    permisoE.release();  
  
    mutexE.acquire();  
    escritores--;  
    if (escritores == 0)  
        permisoL.release();  
    mutexE.release();  
}
```

```
Lector() {  
    mutexP.acquire();  
    permisoL.acquire();  
    mutexL.acquire();  
    lectores++;  
    if (lectores == 1)  
        permisoE.acquire();  
    mutexL.release();  
    permisoL.release();  
    mutexP.release();  
  
    leer();  
  
    mutexL.acquire();  
    lectores--;  
    if (lectores == 0)  
        permisoE.release();  
    mutexL.release();  
}
```

Segunda solución: Prioridad escritores

```
Escritor() {  
  
    mutexE.acquire();  
    escritores++;  
    if (escritores == 1)  
        permisoL.acquire();  
    mutexE.release();  
  
    permisoE.acquire();  
    escribir();  
    permisoE.release();  
  
    mutexE.acquire();  
    escritores--;  
    if (escritores == 0)  
        permisoL.release();  
    mutexE.release();  
}
```

```
Lector() {  
    mutexP.acquire();  
    permisoL.acquire();  
    mutexL.acquire();  
    lectores++;  
    if (lectores == 1)  
        permisoE.acquire();  
    mutexL.release();  
    permisoL.release();  
    mutexP.release();  
  
    leer();  
  
    mutexL.acquire();  
    lectores--;  
    if (lectores == 0)  
        permisoE.release();  
    mutexL.release();  
}
```

Observación: Los lectores pueden sufrir starvation. 

Segunda solución: Prioridad escritores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Escritor1
 2. Llega Lector1 (espera)
 3. Llega Lector2 (espera)
 4. Sale Escritor1
 5. El Lector1 adquiere permisoE
 6. Llega Escritor2 (espera)
 7. Sale Lector1
- ▶ ¿Quién debería entrar a continuación?
 - ▶ ¿Quién entra si no está el mutexP?

Segunda solución: Prioridad escritores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Escritor1
 2. Llega Lector1 (espera)
 3. Llega Lector2 (espera)
 4. Sale Escritor1
 5. El Lector1 adquiere permisoE
 6. Llega Escritor2 (espera)
 7. Sale Lector1
- ▶ ¿Quién debería entrar a continuación?
El Escritor2, ya que tiene prioridad.
 - ▶ ¿Quién entra si no está el mutexP?

Segunda solución: Prioridad escritores (mutexP)

El mutexP es importante para garantizar la prioridad.
Consideremos el siguiente orden de llegada:

1. Entra Escritor1
2. Llega Lector1 (espera)
3. Llega Lector2 (espera)
4. Sale Escritor1
5. El Lector1 adquiere permisoE
6. Llega Escritor2 (espera)
7. Sale Lector1

- ▶ ¿Quién debería entrar a continuación?
El Escritor2, ya que tiene prioridad.
- ▶ ¿Quién entra si no está el mutexP?
El Lector2, ya que habría sido encolado primero en permisoL.

Orden de los semáforos

Considerar la siguiente variante del esquema:

```
Escritor() {  
  
    mutexE.acquire();  
    escritores++;  
    if (escritores == 1)  
        permisoL.acquire();  
    mutexE.release();  
  
    permisoE.acquire();  
    escribir();  
    permisoE.release();  
  
    mutexE.acquire();  
    escritores--;  
    if (escritores == 0)  
        permisoL.release();  
    mutexE.release();  
}
```

```
Lector() {  
    mutexP.acquire();  
    mutexL.acquire();  
    permisoL.acquire();  
    lectores++;  
    if (lectores == 1)  
        permisoE.acquire();  
    permisoL.release();  
    mutexL.release();  
    mutexP.release();  
  
    leer();  
  
    mutexL.acquire();  
    lectores--;  
    if (lectores == 0)  
        permisoE.release();  
    mutexL.release();  
}
```

Orden de los semáforos

El orden en que se piden y liberan los semáforos es importante!
Consideremos el siguiente escenario:

1. Llega Lector1
 - ▶ Está solo, entra, se pone a leer
2. Llega Escritor1
 - ▶ Toma permisoL
 - ▶ Se queda esperando por permisoE
3. Llega Lector2
 - ▶ Toma mutexL
 - ▶ Se queda esperando por permisoL
4. Lector1 quiere salir y....
 - ▶ ¿Qué sucede a continuación?

Orden de los semáforos

El orden en que se piden y liberan los semáforos es importante!
Consideremos el siguiente escenario:

1. Llega Lector1
 - ▶ Está solo, entra, se pone a leer
2. Llega Escritor1
 - ▶ Toma permisoL
 - ▶ Se queda esperando por permisoE
3. Llega Lector2
 - ▶ Toma mutexL
 - ▶ Se queda esperando por permisoL
4. Lector1 quiere salir y....
 - ▶ ¿Qué sucede a continuación?
A Lector1 le falta mutexL, a Escritor1 le falta permisoE, y a Lector2 le falta permisoL.

Orden de los semáforos

El orden en que se piden y liberan los semáforos es importante!
Consideremos el siguiente escenario:

1. Llega Lector1
 - ▶ Está solo, entra, se pone a leer
2. Llega Escritor1
 - ▶ Toma permisoL
 - ▶ Se queda esperando por permisoE
3. Llega Lector2
 - ▶ Toma mutexL
 - ▶ Se queda esperando por permisoL
4. Lector1 quiere salir y....
 - ▶ ¿Qué sucede a continuación?
A Lector1 le falta mutexL, a Escritor1 le falta permisoE, y a Lector2 le falta permisoL. Estamos en **Deadlock**.

1. Basta con semáforos para resolver problemas de sincronización

1. Basta con semáforos para resolver problemas de sincronización
2. A medida que los problemas requieren grados de concurrencia más diversos las soluciones se complican

1. Basta con semáforos para resolver problemas de sincronización
2. A medida que los problemas requieren grados de concurrencia más diversos las soluciones se complican
3. No es necesario reinventar la rueda cada vez, ya que estos esquemas pueden ser aplicados en una gran cantidad de casos

1. Basta con semáforos para resolver problemas de sincronización
2. A medida que los problemas requieren grados de concurrencia más diversos las soluciones se complican
3. No es necesario reinventar la rueda cada vez, ya que estos esquemas pueden ser aplicados en una gran cantidad de casos
4. Los esquemas son flexibles, en el sentido que con una pequeña modificación pueden adaptarse a otros problemas