



Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue



Programación Concurrente



Exclusión mútua

Mecanismos

```
graph TD; A[Mecanismos] --> B[Semáforos]; A --> C[Métodos y Bloques sincronizados]; A --> D[Cerrojos];
```

Semáforos

Métodos y Bloques sincronizados

Cerrojos

Bloques sincronizados


- El objeto es el que se utiliza como llave para acceder a la sección crítica

```
synchronized (objeto) {  
    // zona de exclusión mutua  
}
```

el objeto es sobre el que
se debe
sincronizar

```
synchronized (cc) {  
    int valor = cc.getN(id);  
    valor++;  
    cc.setN(id, valor);  
}
```

Synchronized en Java



```
public synchronized void metodo() {  
    // codigo del metodo aca  
}
```



```
public void metodo() {  
    synchronized(this) {  
        // codigo del metodo aca  
    }  
}
```

```
public static synchronized void metodo() {  
    // codigo del metodo aca  
}
```



```
public static void metodo() {  
    synchronized(MiClase.class) {  
        // codigo del metodo aca  
    }  
}
```

Exclusión mútua

Mecanismos

```
graph TD; A[Mecanismos] --> B[Semáforos]; A --> C[Métodos y Bloques sincronizados]; A --> D[Cerrojos];
```

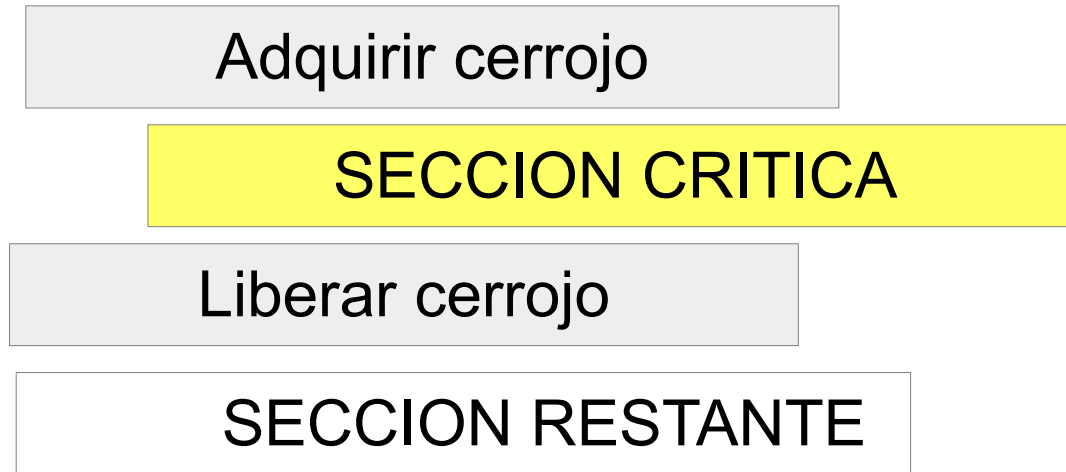
Semáforos

Métodos y Bloques sincronizados

Cerrojos

Mecanismo del cerrojo

- Mecanismo simple basado en cerrojos



- Las operaciones de adquisición y liberación del cerrojo han de ser atómicas (su ejecución ha de producirse en una unidad de trabajo indivisible).

Cerrojo – Interfaz Lock de Java

- Mecanismo de sincronización de hilos, es más flexible y sofisticado que los métodos sincronizados.
- La interfaz Lock define un conjunto de operaciones abstractas de toma y liberación de un lock.
- las operaciones de suspensión y liberación de un lock son explícitas.



A Investigar!!

Interfaz Lock

- **Lock** es una interfaz de Java dentro del package `java.util.concurrent.locks` con los siguientes métodos:
 - El método `lock()`
 - El método `tryLock()`
 - El método `unlock()`
- **ReentrantLock** es una implementación de la interfaz Lock Java dentro del package `java.util.concurrent.locks`

Comparamos...

- **Bloques sincronizados**
- **Métodos sincronizados**
- **Locks explícitos/cerrojos**
- **Semáforos**





Temario



- Correctitud
- Propiedades de seguridad (safety)
- Propiedades de viveza (liveness)
- Problemas clasicos de Prog Conc

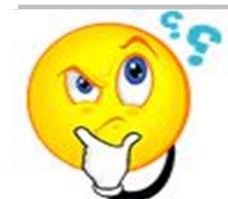


Programas correctos

- (PS) Programa secuencial:
 - **Parcialmente correcto:** Dadas unas precondiciones correctas, **si** el programa termina **entonces** se cumplen las postcondiciones
 - **Totalmente correcto:** Dadas unas precondiciones correctas **el programa termina y** se cumplen las postcondiciones
- (PC) Programación concurrente:
 - Pueden **no terminar** nunca y ser **correctos**.
 - Puede tener **múltiples secuencias de ejecución**
 - Cuando es correcto es porque se refiere a **todas** sus posibles **secuencias de ejecución**.

Programas correctos en PC

Entonces....Cuando los programas concurrentes son correctos?



- Pueden **no terminar** nunca y ser **correctos**.
- Puede tener **múltiples secuencias de ejecución**
- Cuando es correcto es porque se refiere a **todas** sus posibles **secuencias de ejecución**.



POO concurrente

correctitud

- Sistema orientado a objetos: en qué puede estar centrado?

- Centrado en objetos:

- colección de objetos interconectados

- Centrado en actividades

- Colección de actividades posiblemente concurrentes

- Cada actividad puede envolver varios hilos

- Un objeto puede estar envuelto en múltiples actividades

- Una actividad puede abarcar múltiples objetos

seguridad

reusabilidad

viveza

performance



POO concurrente - Seguridad

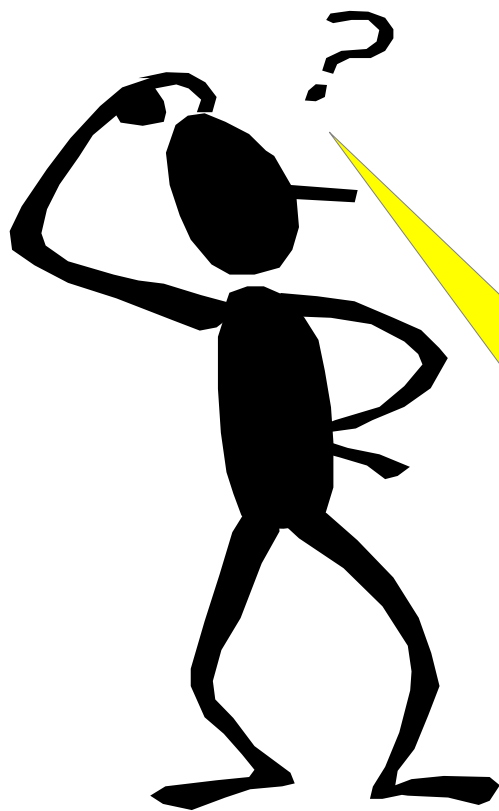
Asegurar consistencia



Emplear técnicas de exclusión




Garantizar la atomicidad de acciones públicas



Cada acción se ejecuta hasta que se completa sin interferencia de otras

Un balance de cuenta bancaria es incorrecto después de un intento de retirar dinero en medio de una transferencia automática

POO concurrente - Seguridad

- Inconsistencias  **condiciones de carrera**
- Conflictos de lectura/escritura: *un hilo lee un valor de una variable mientras otro hilo escribe en ella. El valor visto por el hilo que lee es difícil de predecir – depende de que hilo ganó la “carrera” para acceder a la variable primero*
- Conflictos de escritura/escritura: *dos hilos tratan de escribir la misma variable. El valor visto en la próxima lectura es difícil de predecir*

Ejemplos de condición de carrera

- Cuando el resultado depende de la ejecución de los hilos.
- Problema: varios hilos hacen **lectura + modificación + escritura** de una variable y el ciclo completo se interrumpe
- Muy difícil de detectar, pues el programa puede funcionar bien muchas veces y fallar de repente.

	thread 1 { $x = x + 5;$ }	thread 2 { $x = x + 100;$ }
t0	x vale 100	
t1	lee 100	
t2	incrementa a 105	
t3		lee 100
t4	escribe 105	
t5		incrementa a 200
t6		escribe 200
t7	x vale 200	



Seguridad (safety)

- Un programa es seguro cuando no genera nunca resultados incorrectos.
- No tiene inconsistencias

Vivacidad (liveness)

- Cuando el programa responde ágilmente a las solicitudes del usuario.
 - cada actividad eventualmente progresa hacia su finalización.
 - cada método invocado eventualmente se ejecuta

Propiedades de la PC

Seguridad (safety)

Nada malo va a pasar

- Exclusión mutua
- Evitar deadlock

Viveza (liveness)

Algo bueno va a pasar

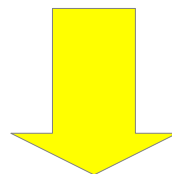
- Evitar inanición
- Livelock

Propiedades de la PC

Seguridad (**safety**)

- Se debe asegurar que los objetos esten en estados estables y consistentes, para ello ...

- **Exclusión mutua**



Condición de sincronización

- Interbloqueo (pasivo) – **deadlock**:
dependencias circulares entre locks



Propiedades de la PC

Viveza (**liveness**)

- Se debe asegurar que todos los procesos *progresen* durante la ejecución del programa
 - Interbloqueo (activo) – **livelock**: una acción de reintentar continuamente, continuamente falla
 - **Inanición** – **starvation**: la JVM/SO falla en asignar tiempo de CPU a un hilo



Estados consistentes - sistemas seguros

- En un **sistema seguro** cada objeto se protege de **violaciones de integridad**.
- Las técnicas de exclusión mutua preservan los **invariantes de los objetos** y evitan efectos indeseados
- Las técnicas de programación logran la exclusión mutua previniendo que múltiples hilos modifiquen o actúen de forma concurrente sobre la representación de los objetos.



Estados consistentes - sistemas seguros

Exclusión mutua: mantener estados consistentes de los objetos evitando interferencias NO deseadas entre actividades concurrentes

Clases “Thread safe” , es decir clases seguras para un ambiente concurrente

¿La clase “pila” implementada en materias anteriores es Thread safe?

Estrategias para asegurar la Exclusión mutua (seguridad)

- Eliminar la necesidad de control de exclusión asegurando que los métodos nunca modifican la representación de un objeto



inmutabilidad

- Asegurar dinámicamente que solamente 1 hilo por vez puede acceder el estado del objeto usando cerrojos y construcciones relacionadas



sincronización

- Asegurar estructuralmente que solamente 1 hilo por vez, puede utilizar un objeto dado ocultándolo o restringiendo el acceso a él.



confinamiento

Inmutabilidad - Seguridad

- Si un objeto no puede cambiar su estado, nunca puede encontrar conflictos o inconsistencias cuando múltiples actividades intentan cambiar su estado.
- Los objetos inmutables mas simples no tienen campos internos, sus métodos son sin estado (**Stateless**)
- **Aplicaciones**
 - **TDA: Integer, Date, Fraction, etc.** - Instancias de estas clases nunca alteran los valores de sus atributos. Proveen métodos para crear objetos que representan nuevos valores
 - **Contenedores de valores:** es conveniente establecer un estado consistente una vez y mantenerlo por siempre.
 - Representaciones de estado compartidas: en general se trata de clases de utilidad.

Confinamiento - Seguridad

- Se emplean técnicas de encapsulación para garantizar estructuralmente que una actividad por vez pueda acceder a un objeto dado.
- ...



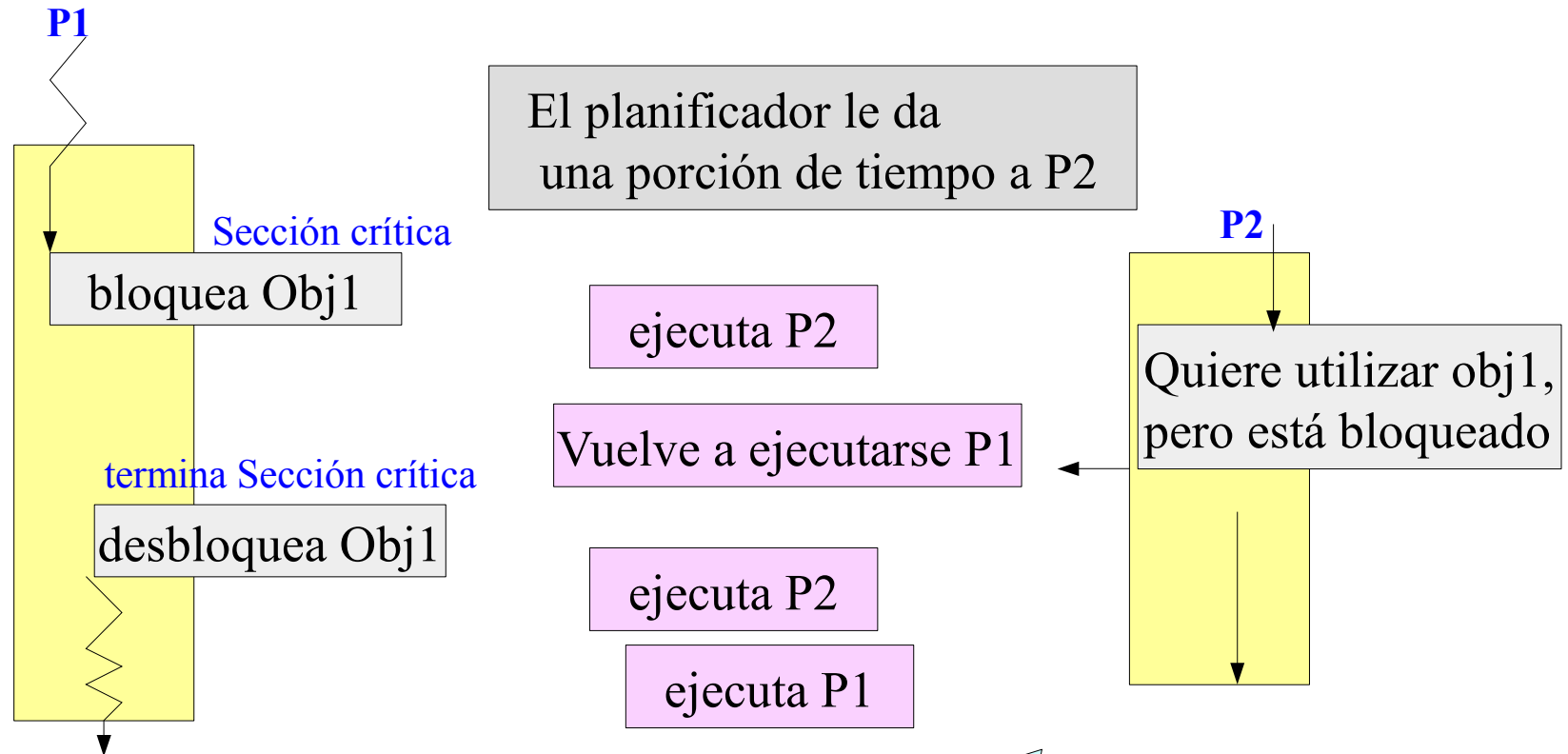


Prog concurrente – Seguridad y Viveza

Problema serio: falta de progreso permanente

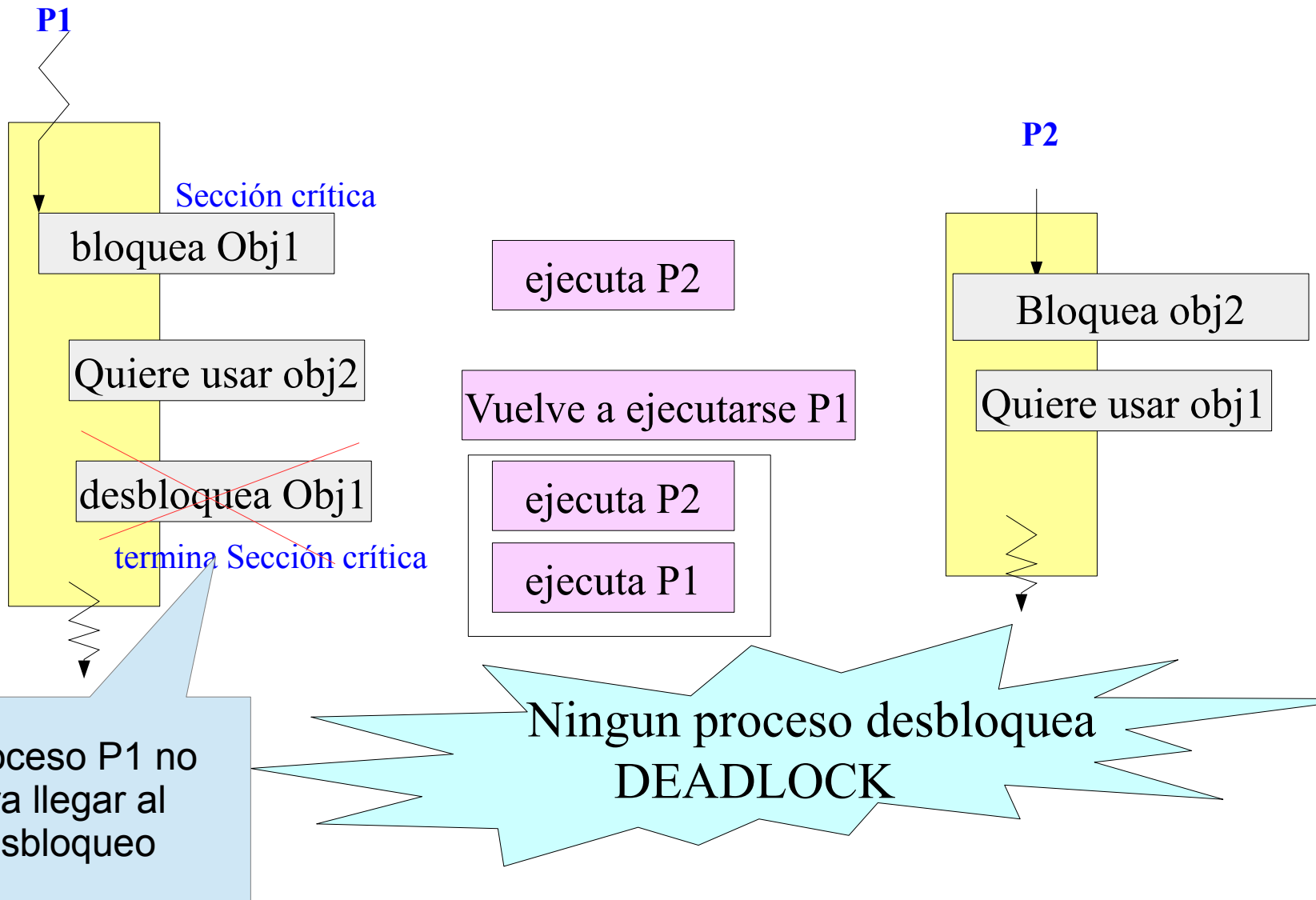
- **Deadlock:** dependencias circulares
 - “deadlock” (interbloqueo por inactividad)
- **Livelock:** una acción falla continuamente
 - “livelock” (interbloqueo por hiperactividad)
- **Starvation/inanición:** un hilo espera por siempre, la maquina virtual falla siempre en asignarle tiempo de CPU
- **Falta de recursos:** un grupo de hilos tienen todos los recursos, un hilo necesita recursos adicionales pero no puede obtenerlos
- Otros (investigar)

Deadlock: Situación 1

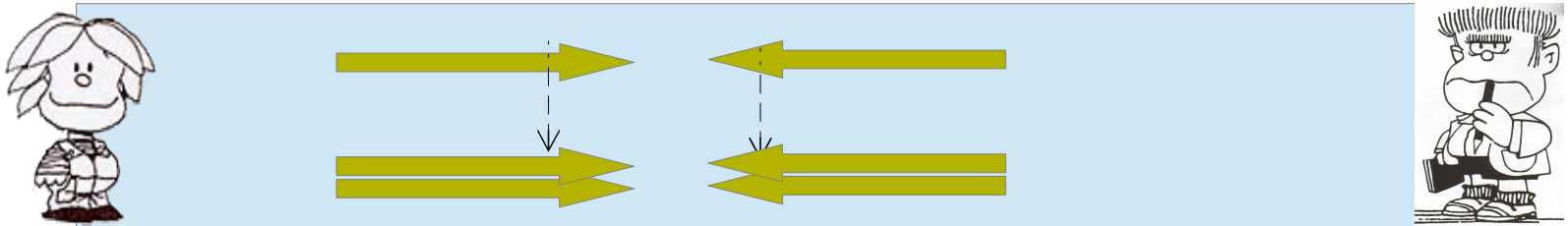


Se ejecutan ambos procesos
NO HAY DEADLOCK

Deadlock: Situación 2



Livelock



- Un livelock es similar a un deadlock,
- Los dos procesos envueltos con respecto al otro.
- Livelock es una forma de inanición
- Ejemplo: dos personas, al encontrarse en un pasillo angosto avanzando en sentidos opuestos, y cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar
- Ninguna puede pasar pues ambos se mueven hacia el mismo lado, al mismo tiempo.
- Livelock es un riesgo que se puede detectar y recuperar, asegurando que sólo un proceso (escogido al azar o por prioridad) tome acción.



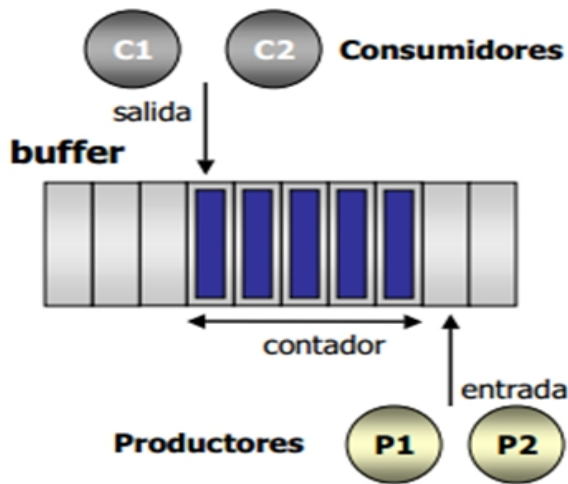
Problemas Clásicos

- 1.- Productor / Consumidor
- 2.- Lectores / Escritores
- 3.- Barbero Dormilón
- 4.- Filósofos cenando
- 5.- Otros: Taxista - ...

Problema de sincronización:

Productor/Consumidor

Una parte produce algún producto (datos en nuestro caso) que se coloca en algún lugar (una cola, por ejemplo) para que sea consumido por otra parte.



- El productor genera sus datos en cualquier momento
- El consumidor puede tomar un dato sólo cuando hay
- Para el intercambio de datos se usa una cola a la cual ambos tienen acceso, así se garantiza el orden correcto (Esto puede variar)
- Todo lo que se produce debe ser consumido

Tenemos que garantizar que el consumidor no consuma más rápido de lo que produce el productor

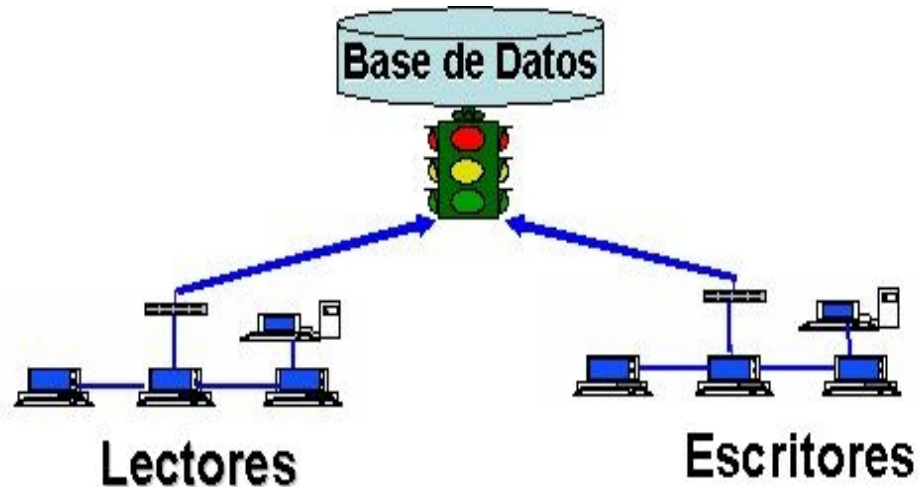
Acceso de los productores y consumidores de forma de asegurar consistencia de la información almacenada en la cola.

La cola puede tener capacidad limitada: a) un productor no puede producir un elemento en una cola llena, b) un consumidor no puede extraer un elemento de una cola vacía

Problema de sincronización:

Lectores/escritores

Dos grupos de procesos (Lectores/escritores) que utilizan los mismos recursos (documento)



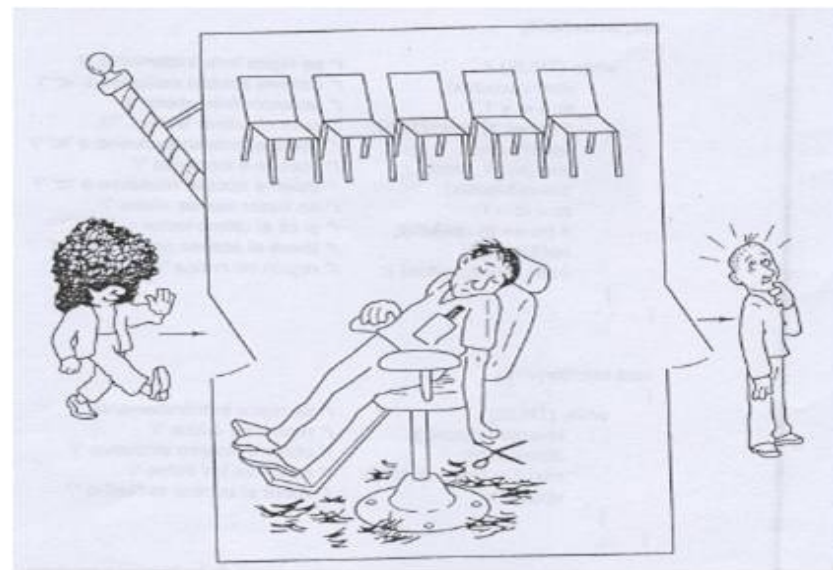
- Un grupo de lectores, sólo leen los datos.
- Los escritores, leen y escriben.
- Varios lectores pueden acceder simultáneamente a un proceso compartido
- Se debe evitar que accedan simultáneamente un proceso escritor y cualquier otro proceso.

Problema de sincronización:

Barbero dormilón

En una barbería trabaja un barbero que tiene un único sillón de barbero y varias sillas para esperar.

- Cuando no hay clientes, el barbero se sienta en una silla y se duerme.
- Cuando llega un nuevo cliente,
 - si el barbero duerme: despierta al barbero o
 - si el barbero está afeitando a otro cliente: se sienta en una silla



(si todas las sillas están ocupadas por clientes esperando, se va).

Problema de sincronización:

Cena de filósofos

Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha.



Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.

Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente (deadlock), entonces los filósofos se morirán de hambre