FACULTAD DE INGENIERÍA

Sistemas Operativos

Sincronización

Lic. R. Alejandro Mansilla

Licenciatura en Ciencias de la Computación
Facultad de Ingeniería
Universidad Nacional de Cuyo

Sincronización

El término se refiere a las relaciones entre fenómenos o eventos – cualquier número de eventos y cualquier clase de relación, tal como antes, durante o después.

- El evento A debe ocurrir antes que el evento B o, los eventos A y B no deben ocurrir a la vez.
- Normalmente se necesita de un reloj para forzar la sincronización pero no siempre es posible. Por eso, nos valemos de técnicas en software para lograrlo.
- Si tuviéramos una computadora capaz de cargar un único programa y ejecutarlo de principio a fin, no tendríamos necesidad de sincronización. Por eso, estas técnicas le darán a nuestros procesos esa ilusión

Sincronización

Por ejemplo, consideremos un programa en el que **dos** hilos se ejecutan concurrentemente para sumar 1 a la variable compartida x.

$$x:=x+1$$

Esto da lugar a tres instrucciones de un lenguaje de ensamblador cualquiera:

	H1	H2
1	LOAD X R1	LOAD X R1
2	ADD R1 1	ADD R1 1
3	STORE R1 X	STORE R1 X

Sincronización

Tendríamos los siguientes pasos:

1. Cargar desde memoria el valor de x en un registro.
2. Incrementar el valor del registro.
3. Almacenar el contenido del registro en la posición de memoria de x

	H1	H2
1	LOAD X R1	LOAD X R1
2	ADD R1 1	ADD R1 1
3	STORE R1 X	STORE R1 X

En programación concurrente existe un orden parcial, cualquier intercalado entre estas instrucciones es válido. Pero no siempre obtendremos el resultado esperado.

x	0	0	0	0	1	1	1
H1	1	2			3		
H2			1	2		3	

Se nos perdió un incremento!

Sincronización - Condición de carrera

El valor final de x no es el esperado, se ha perdido un incremento.

El problema estriba en que dos hilos distintos están accediendo al mismo tiempo a una variable compartida entre los dos para actualizarla.

Una situación como ésta, en la que varios procesos o hilos acceden a, y manipulan, los mismos datos de forma concurrente, y el resultado de la ejecución depende del orden en que haya ocurrido el acceso, se denomina **condición de carrera** *(o competencia) (a.k.a race condition)*.

Sincronización - Sección Crítica

Si estos dos hilos necesitan incrementar la misma variable global, cada uno de ellos deberá asegurar que tiene acceso exclusivo a esta variable durante algún período de tiempo.

Denominaremos sección crítica al segmento de código en el cual un hilo está accediendo en exclusividad a un recurso compartido (una variable, una estructura de datos, un dispositivo) y que no debe ser accedido concurrentemente por otro.

La ejecución de secciones críticas de los procesos o hilos es mutuamente excluyente en el tiempo.

Exclusión mutua - Soluciones de hardware

- Inhabilitación de las interrupciones. *(evitar la ejecución de rutinas de servicio)*
- Instrucciones especiales de máquina.
  - TSL *(test and set lock) Cerrojo exclusivo al hilo invocador, ningún otro hilo tendrá acceso*
  - CAS *(compare and swap) La operación falla si el valor fue alterado por otro hilo)*
  - Fetch-and-add *(trae y suma en una única operación)*
  - Read-modify-write *(idem anteriores, operación atómica)*
  - Load-link/store-conditional

Exclusión mutua - Soluciones de software

- Algoritmo de Dekker.
- Algoritmo de Peterson.

Requisitos de una solución:

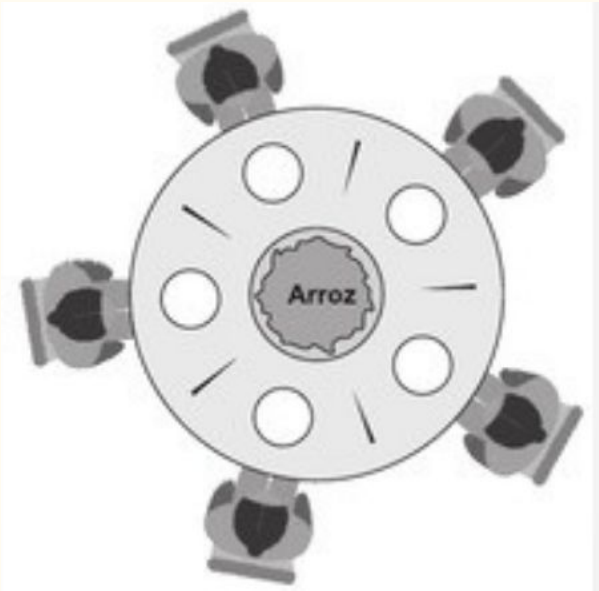
- Exclusión mutua. *Solo un hilo podrá estar ejecutando en la sección crítica*
- Progreso *Se elige entre los hilos que están interesados en acceder a la sección crítica.*
- Espera limitada. *Límite en la cantidad de intentos que hace un hilo para entrar en su sección crítica hasta que finalmente logra hacerlo.*

Mutex/Var de condición/ Spinlock

- Una **variable mutex** es un objeto en un programa que sirve para trabar y se usa para lograr exclusión mutua (y de ahí su nombre: de la contracción en inglés de mutual exclusión o exclusión mutua). Está siempre en uno de dos estados: “trabado” o “destrabado” (locked/unlocked). Valores binarios “1” o “0”.
- Una **variable de condición** está asociada con una variable mutex y refleja un estado lógico o boolean, en inglés (es decir, verdadero o falso). Las variables de condición indican eventos. Normalmente las dos operaciones que se proveen para una variable de condición son wait() y signal().
- Un **spinlock** es una traba (lock) que se puede adquirir para acceso exclusivo y que si no está disponible se espera en un lazo comprobando repetidamente (spin) hasta que lo esté. Interesante, si el tiempo de espera es menor que la sobrecarga de un cambio de contexto.

Problemas clásicos: La cena de los filósofos

Cinco filósofos chinos se pasan la vida pensando y comiendo, comparten una mesa circular. La mesa está puesta con cinco palillos y cinco platos.
Cuando un filósofo piensa no interactúa con sus colegas. Cuando quiere comer trata de tomar dos palillos cercanos, a su izquierda y a su derecha (si es que no están ocupados). Si lo consigue, come sin soltarlos y cuando termina los deja en su lugar, y sigue pensando. La solución al problema, por lo tanto, consiste en inventar un ritual (algoritmo) que permita comer a los filósofos

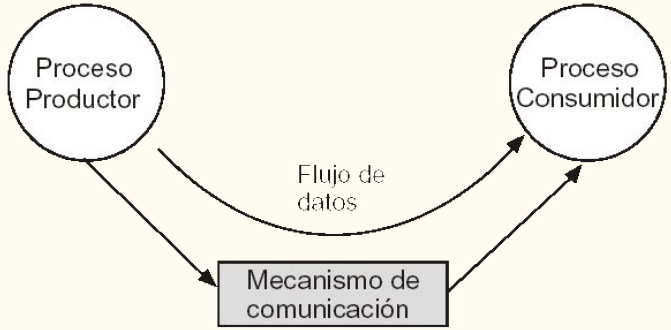


Problemas clásicos: El peluquero dormilón

Hay una peluquería que tiene una sala de espera separada del salón de peinados. La sala de espera tiene una entrada y cerca de ella una puerta que conduce a la habitación donde se corta y peina; ambas comparten la misma puerta deslizante que siempre cierra una de ellas. Cuando el peluquero ha finalizado un corte de cabello, abre la puerta a la sala de espera y si no está vacía invita al próximo cliente sino se va a dormir en una de las sillas. Si llega un cliente lo despierta.

Problema del productor-consumidor

En este tipo de problemas, uno más procesos (o hilos), que se denominan **productores**, generan cierto tipo de datos que son utilizados o consumidos por otros procesos, que se denominan **consumidores**.

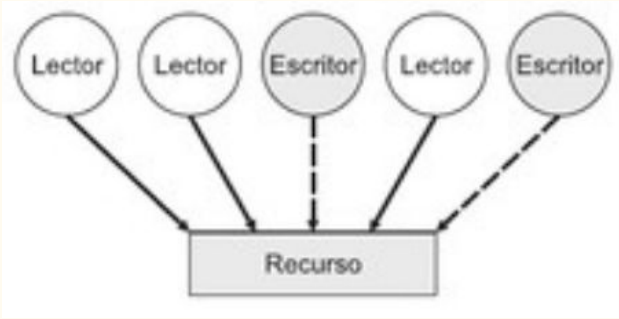


En esta clase de problemas, es necesario disponer de algún mecanismo de **comunicación** que permita a los procesos productor y consumidor intercambiar información. Ambos procesos, además, deben **sincronizar** su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática.

Este tipo de mecanismos requiere de servicios que asistan a la sincronización

Problema de los lectores-escritores

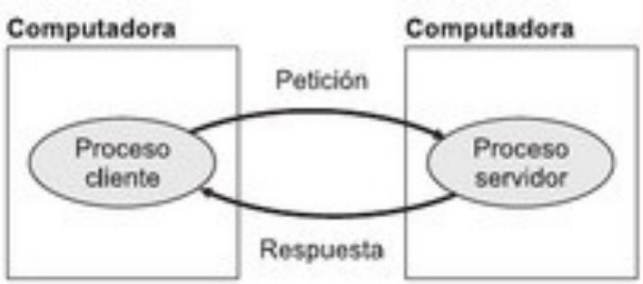
En este problema, existe un determinado objeto, que puede ser un archivo, un registro dentro de un archivo, etc., que va a ser utilizado y compartido por una serie de procesos concurrentes. Algunos de estos procesos sólo van a acceder al objeto sin modificarlo, mientras que otros van a acceder al objeto para modificar su contenido. A los primeros procesos se les denomina **lectores** y a los segundos se les denomina **escritores**.



- Solo se permite que un escritor tenga acceso al objeto al mismo tiempo y durante ese tiempo, nadie más accede.
- Sin embargo, sí se permiten que múltiples lectores accedan al objeto en simultáneo

Cliente - Servidor

En este modelo tenemos un proceso llamado **servidor**, por una parte, que ofrece un **servicio** a otros procesos, por otra parte, que se denominan **clientes**. El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión.



Semáforos

Para transmitir una señal por el semáforo ***s*** los procesos ejecutan la operación *signal(**s**)* mientras que para recibir una señal del semáforo s, los procesos ejecutan la operación *wait(**s**)*; si la señal correspondiente aún no se ha transmitido, el proceso se suspende hasta que tenga lugar la transmisión.

- **wait(s).** Decrementa el valor de **s** si éste es mayor que cero. Si **s** es igual a 0, el proceso se bloqueará en el semáforo.
- **signal(s)**. Desbloquea algún proceso bloqueado en **s**, y en el caso de que no haya ningún proceso incrementa el valor de **s**.

Semáforos *(cont.)*

Los semáforos admiten por lo general valores enteros positivos. Como caso particular existen los **semáforos binarios** que admiten solo 0 o 1

wait(s):	signal(s):
if s>0 then	if (hay procesos bloqueados) then
s:=s-1	desbloquear un proceso
else bloquear proceso;	else s:=s+1

Tanto wait() como signal() deben ejecutarse en forma indivisible, y con eso se garantiza la exclusión mutua.

Monitores

Un monitor es un mecanismo de abstracción de datos, que permite representar el recurso compartido.

Es similar a un objeto en la programación orientada a objetos ya que consta de variables y procedimientos.

Las variables pueden accederse sólo a través de los procedimientos del monitor pero a diferencia de los objetos, solo 1 proceso a la vez podrá acceder a estos procedimientos.

Cuando un proceso ejecuta un procedimiento del monitor se dice que el proceso ha “entrado en el monitor”. Cualquier otro proceso que quiera acceder se bloqueará hasta que el proceso anterior “salga del monitor”.

Ejemplo de monitor

```
monitor Counter {
private:
    int count=0;
public:
    int value()  {return count;}
    void incr()  {count = count + 1;}
    void decr()  {count = count - 1;}
}
```

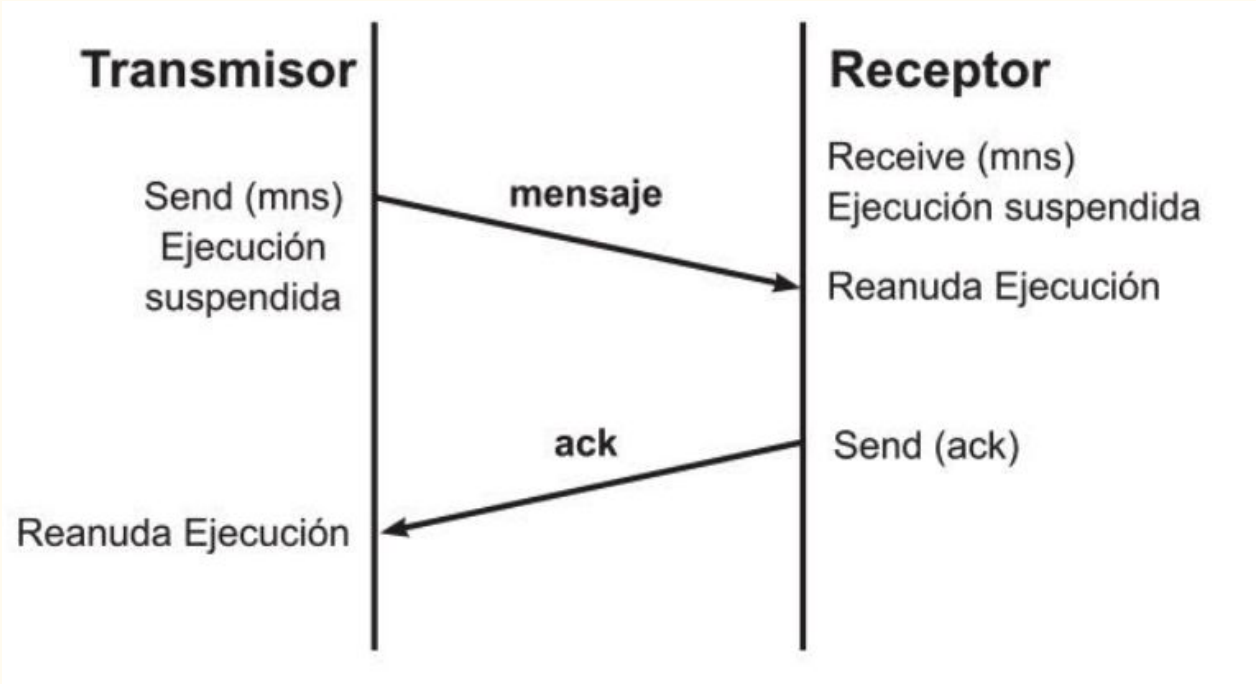
Sincronización en Pase de mensajes

- Cuando queremos comunicar o sincronizar dos procesos y no contamos con una memoria compartida como los casos que hemos visto hasta ahora, se debe recurrir al pase de mensajes.
- Los procesos pueden ser remotos (máquinas distintas) o locales.
- Normalmente en este tipo de comunicaciones, un proceso suele ser más lento que el otro, en estos casos se debe recurrir al uso de memorias intermedias o Buffers.

Características de la sincronización

- Semántica de la comunicación: bloqueante o no bloqueante
- La sincronización entre dos procesos que se comunican básicamente depende de dos primitivas: **send()** y **receive()**
- Ambas pueden ser tanto bloqueantes como no bloqueantes
- En el caso de *receive()* no bloqueante, existen dos formas de que el proceso se entere que llegó el mensaje:
  - Consulta: (polling)
  - Interrupción
- Cuando tanto *send()* como *receive()* usan una semántica bloqueante decimos que los procesos se comunican de forma Síncrona.
- Un sistema de pasajes de mensajes flexible provee ambas primitivas send y receive tanto bloqueantes como no bloqueantes.

Modo síncrono



Memoria intermedia o Buffer

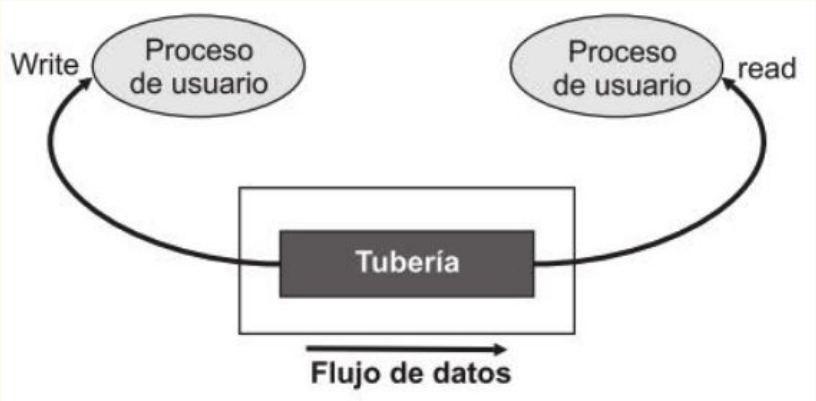
- Los mensajes son copiados desde el espacio de memoria del emisor hacia el del receptor.
- En algunos casos se recurre al núcleo del sistema operativo para que almacene el mensaje hasta que el receptor esté listo. El proceso receptor debe contar con un buffer para poder almacenar el mensaje.
- Si tenemos comunicación síncrona, es imposible usar buffer, (buffer nulo) en el caso de la comunicación asíncrona, deberíamos tener un buffer de capacidad ilimitada.

Estrategias de buffering

- Buffer nulo. Sin almacenamiento temporal del mensaje
  - Usado en comunicaciones síncronas
  - Cita o *randez-vous*: Send() se bloquea hasta recibir una señal de receptor que está listo.
  - Descarte: send() espera un timeout, si no recibe acuse de recibo, retransmite.
- Buffer de mensaje único.
  - Ubicado en el receptor, con capacidad para un solo mensaje
  - Usado en comunicaciones síncronas, el buffer almacena el mensaje hasta que el receptor esté listo
- Buffer de capacidad ilimitada. Prácticamente imposible (teórico)
  - Usado para comunicación asíncrona
  - pueden haber varios mensaje pendientes de recepción
  - Debe brindar garantía de entrega
- Buffer de capacidad limitada. Realista, con cantidad finita de mensajes.
  - 2 estrategias:
    - Comunic. no exitosa: Fracasa cuando el buffer está lleno. Es poco confiable.
    - Comunic de flujo controlado. Implementa control de flujo entre emisor y receptor, por este motivo opera en modo síncrono

Sincronización en tuberías

- Pseudo archivo mantenido por el sistema operativo
- Cada proceso participante ve a la tubería como un conducto con dos extremos
- Por un extremo se puede escribir datos y por el otro leerlo
- La comunicación es unidireccional
- Son dispositivos serial, se respeta el orden
- Pueden o no tener nombre
- FIFO es una tubería con nombre y disponible para cualquier proceso, no solo para quien la creó



Señales

- Las señales pueden utilizarse para sincronizar procesos
- Un proceso o hilo puede bloquearse en el servicios *pause()* esperando la recepción de una señal proviniente de otro proceso
- Otros procesos pueden enviar la señal *kill()* o *sigusr1()* para desbloquear al otro proceso
- Desventajas:
  - Las señales tienen un comportamiento asíncrono. Los procesos pueden recibir la señal en cualquier momento.
  - Las señales no se encolan, esto puede provocar la pérdida de eventos de sincronización importante

Bloqueo Mutuo

En un entorno de multi-programación varios procesos pueden competir por un número finito de recursos. Si estos no están libres el proceso pasa a un estado de espera.

Bloqueo mutuo, Interbloqueo o deadlock

Puede suceder que los procesos en espera nunca vuelvan a cambiar de estado, porque los recursos que solicitaron están tomados por otros procesos que también están esperando.

Condiciones para interbloqueo

- **Exclusión mutua:** al menos un recurso debe adquirirse de manera exclusiva. Los otros procesos deberán esperar.
- **Retención y espera:** mientras retiene un recurso, solicita otros y debe esperar por ellos. Los procesos no liberan los recursos asignados previamente mientras espera por otros.
- **No expropiación:** Los recursos no se pueden arrebatar. La liberación de un recurso siempre es voluntaria por parte del proceso.
- **Espera circular:** Debe existir un conjunto de procesos en espera por recursos adquiridos por otro proceso anterior que a su vez está esperando por otro en las mismas condiciones y así sucesivamente hasta que hay uno que espera por el primero formando una cadena cerrada de bloqueos.