

## Tema 4- Comunicación y sincronización

**ISO-Facultat d'Informàtica de Barcelona**

## **1. Introducción**

## **2. Paso de mensajes**

1. Características del paso de mensajes
  1. Buzones
2. Dispositivos UNIX de paso de mensajes
  1. Pipes ordinarias
  2. Named pipes

## **3. Eventos**

1. Signal, kill, pause, alarm

## **4. Comunicación mediante memoria compartida**

1. Exclusión mutua
2. Posibles implementaciones
  1. Espera activa
  2. Bloqueo: semáforos
3. Abrazo mortal

## 4.1 Introducción

### ❑ Dos flujos (del mismo o distinto proceso) pueden interactuar de dos formas:

- Comunicación: Intercambio de información
- Sincronización: Ponerse de acuerdo para realizar una tarea

FLUJO 1

FLUJO 2

Si tarea\_1 NO acabada Entonces  
Esperar\_finalice\_tarea\_1  
FSI  
Ejecutar tarea\_2

Ejecutar tarea\_1  
Avisar tarea\_1 finalizada

SINCRONIZACIÓN

### ❑ Si la comunicación es entre flujos del mismo proceso la forma más sencilla es usando una variable global (**SOLO VÁLIDO PARA FLUJOS DEL MISMO PROCESO**)

```
int trabajo_acabado=FALSO;    /* Para sincronizar */  
int resultado_intermedio;     /* Para comunicar */
```

```
void flujo1()  
{  
    while (!trabajo_acabado);  
    acabar_trabajo(resultado_intermedio);  
}
```

```
void flujo2()  
{  
    resultado_intermedio=ini_trabajo();  
    trabajo_acabado=CIERTO;  
}
```

## 4.1 Introducción

- ❑ **Si los flujos no son del mismo proceso...**
  - Problema: No podemos usar variables globales
  - Solución: Usaremos el mecanismo de paso de mensajes
- ❑ **Un mensaje es un conjunto de datos que dos flujos intercambian mediante el S.O**

```
Void flujo1()
```

```
{  
  int dato;  
  dato=recibir_mensaje(proceso2);  
  acabar_trabajo(dato);  
}
```

Llamadas a sistema  
Para enviar/recibir  
Mensajes  
Podrían ser read/write

```
Void flujo2
```

```
{  
  int res;  
  res=init_trabajo();  
  enviar_mensaje(proceso1,res);  
}
```

- ❑ **Dos formas de comunicar**

- Memoria compartida: SOLO entre flujos del mismo proceso
- Paso de mensajes: entre cualquier tipo de flujos y OBLIGADO si no son del mismo proceso

## 4.2 Paso de mensajes

- ❑ Paso de mensajes es un mecanismo que proporciona el SO para sincronizar y comunicar dos flujos que **NO QUIEREN** o **NO PUEDEN** compartir memoria
- ❑ Dos llamadas: send/receive
- ❑ Características del paso de mensajes:
  - Usan dispositivos lógicos
- ❑ **Las características de estos dispositivos van a afectar a como se comunican los flujos**
  - Capacidad del dispositivo lógico
  - Comunicación directa/indirecta
  - Comunicación simétrica/asimétrica
  - Medida fija/variable de los mensajes

## 4.2.1 Características del paso de mensajes

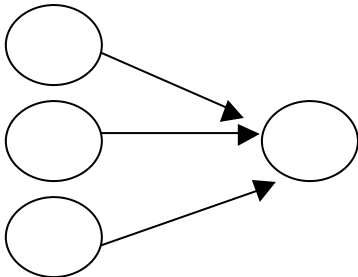
### □ Capacidad

- Cantidad de mensajes que se pueden almacenar en el dispositivo (sin leerlos)
- Si la capacidad es 0, los send's serán bloqueantes hasta que el mensaje sea leído (ej. Teléfono)
- Si la capacidad es  $>0$  sólo nos bloquearemos si: (ej. Correos)
  - Hacemos un send y está lleno
  - Hacemos un receive y está vacío

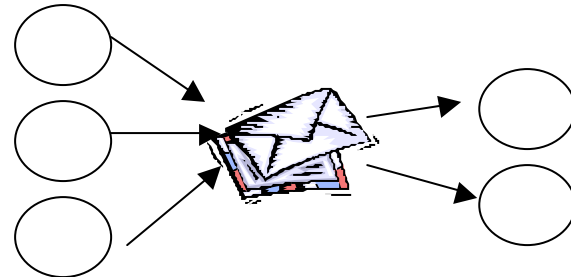
### □ Comunicación directa/indirecta (a quién enviamos el mensaje???)

- Directa: a un proceso concreto, usando el PID
- Indirecta: Se envían a una zona intermedia (buzón) y quien quiera puede leerlos)

`send(pid1,mensaje)`



`send(buzon1,mensaje)`



## 4.2.1 Características del paso de mensajes

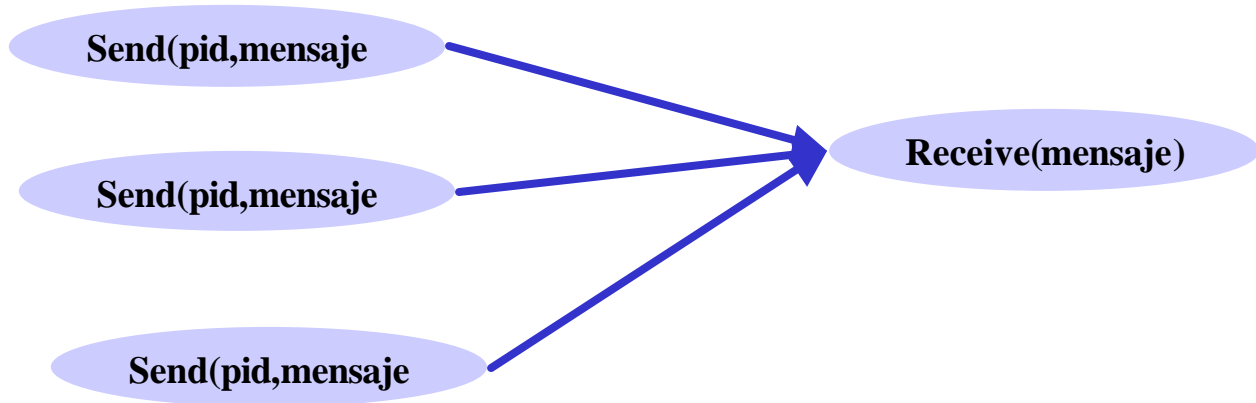
### □ Comunicación simétrica/asimétrica

- Simétrica : Mismo número de parámetros en el send y receive
- Asimétrica: Distinto número de parámetros

**Síncrona**



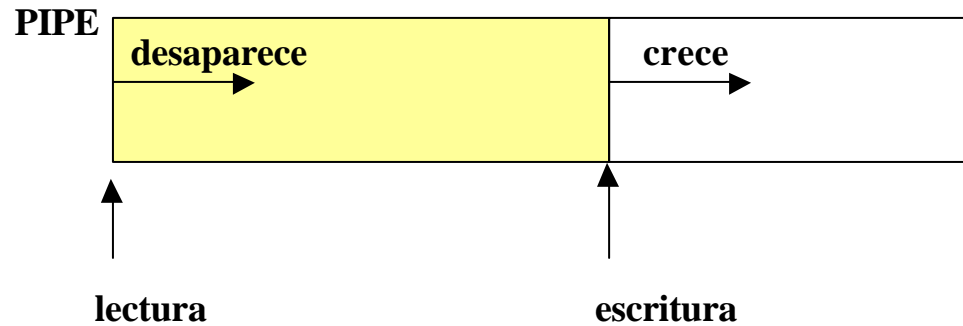
**Asíncrona**



## 4.2.2 Dispositivos UNIX de paso de mensajes: pipes

### □ Pipes

- Cola de bytes (SIN TIPO) que permite la comunicación entre dos procesos. Su contenido va desapareciendo a medida que lo leemos



- Hay dos tipos
  - **Named pipes** (pipes con nombre), tienen nombre en el sistema de fichero. Permite que se comuniquen dos procesos no emparentados que conozcan el nombre de la pipe.
  - **Pipes sin nombre**. No tienen nombre en el sistema de ficheros, sólo sirve para comunicar procesos emparentados.
- Se gestionan con las mismas llamadas a sistema que los ficheros normales pero tienen características especiales
- Tamaño típico 4K

# Pipes

## □ Named pipes:

- Es necesario crear el dispositivo antes de poder abrirlo (con la llamada open)
- `int mknod(char *nombre_pipe, int mode, dev_t device)`
- Crea un dispositivo (genérico), lo usaremos sólo para pipes
  - Nombre\_pipe, es el nombre que tendrá el fichero tipo pipe
  - Mode, indica que es una pipe y las protecciones del fichero.
  - Device, no es necesario ponerlo en el caso de una pipe
  - Devuelve 0 si OK y -1 si ERROR
- Ej: `mknod("mi_pipe", S_I F I F O | 0666)`

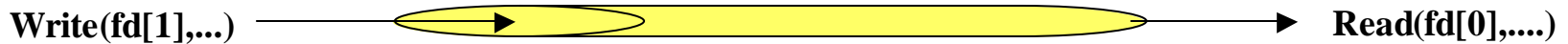
□ Una vez creada la pipe para poder usarla es necesario abrir el dispositivo haciendo open.

□ El mknod sólo es necesario hacerlo una vez en la vida de la pipe.

# Pipes

## □ Pipes ordinarias

- Usaremos una **ÚNICA** llamada a sistema que nos devolverá dos canales ya abiertos.
- `int pipe(int fd[2])`
- Abre dos canales de acceso a una pipe.
  - `fd[0]` canal de sólo lectura,
  - `fd[1]`, canal de sólo escritura
- Lo que se escribe en `fd[1]` se lee por `fd[0]`



## □ Después de abrir los canales .....

- con la llamada `pipe` si es una pipe sin nombre
- Creándola (`mknod`) + abriéndola (`open`) si `named pipe`

... ya podemos usarlos (`read`, `write`,...)

# Utilización de las llamadas a sistema con Pipes

## ❑ Open

- **Bloquea** si no existe su pareja (read  $\leftrightarrow$  write)
- **SÓLO se utiliza open con pipes con nombre**

## ❑ Read de una pipe vacia

- Si ningún proceso la tiene abierta para escritura devuelve 0
- Si existe **ESCRITOR** nos bloqueamos hasta que
  1. Alguien escriba, o
  2. Los **ESCRITORES** desaparecen.
  3. En el caso 1 nos desbloqueamos y leemos lo escrito, en el caso 2 la llamada devuelve 0.
- ¿Qué pasa si el flag **O\_NDELAY** o **O\_NONBLOCK** están activados?

- ❑ Por eso es **MUY IMPORTANTE** cerrar los canales de escritura de las pipes cuando no vayamos a enviar más datos

# Pipes

## □ Write

- Si escribimos en una pipe que no tiene lectores devuelve -1 y errno=EPIPE y el proceso recibe un evento (signal) SIGPIPE
- Si la pipe está llena el proceso se bloquea

## □ Lseek

- No se aplica a las pipes

## 4.3 Eventos (signals)

### ❑ Eventos: signals

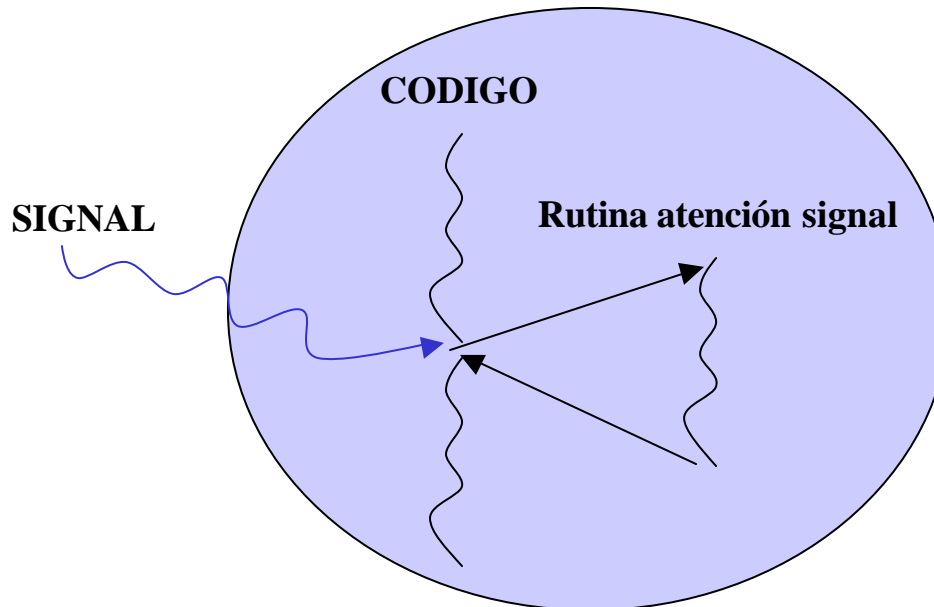
- Facilidad que da el sistema para tratar eventos asíncronos (pueden llegar en cualquier momento)

### ❑ Se suelen usar para notificar eventos

- Es un tipo de comunicación donde no hay intercambio de datos, sólo notificación de sucesos

### ❑ Son muy similares a las interrupciones hardware (EC)

- Interrupción software



# Eventos

## ❑ Hay 32 signals (UNIX estándar)

- En la mayoría el significado viene dado por el SO, pero algunos pueden ser definidos por el usuario

## ❑ Quien los provoco:

- El S.O
- El propio proceso
- Otro proceso

## ❑ Que hacer cuando recibimos un signal???

- Tratamiento por defecto (SIG\_DFL)
  - Exit: Acabar el proceso
  - Exit+Core: Acabar el proceso y volcar la memoria sobre un fichero
  - Stop: para el proceso hasta que reciba el signal SIG\_CONT
  - Ignore: No hacer nada
- No hacer nada (SIG\_IGN)
- Ejecutar una rutina de usuario (especificada por el usuario y programada por el)

# Eventos

## □ Tipos de signals

- SIGINT, ^c durante la ejecución , Def=Exit
- SIGSEGV, dirección ilegal, Def=exit+Core, No ignorar
- SIGALARM, expiración timer, Def=Exit
- SIGSTOP, parar proceso (^z), Def=Stop, No ignorar, No reprogramar
- SIGCONT, continuar proceso parado, Def=continuar
- SIGUSR1, definida por el usuario, Def=Exit
- SIGUSR2, definida por el usuario, Def=Exit
- SIGPIPE, escritura sobre pipe sin lector, Def=Exit
- SIGCHLD, ha muerto un hijo, Def=ignorar
- SIGKILL, killed, Def=Exit , No reprogramar, No ignorar
- SIGTERM, terminated, Def=Exit
- ...

# Eventos: Envío de un signal

## ❑ **Cómo se envían signals????**

- Llamada a sistema: `int kill(int pid, int sig)`
- Parámetros:
  - Pid: identificador de proceso al que se le envía el signal
  - Sig: tipo de signal que enviamos
- Que devuelve:
  - 0 si enviado OK
  - -1 si error ( no existe el proceso, etc)

## ❑ **Desde el shell .....**

- `Kill -sig pid`

# Eventos: Programación de un signal

## ❑ Como se programa un signal?????

- Llamada a sistema: `void (*signal (int sig, void (function)(int)))(int)`
- Parámetros:
  - Sig: signal que estamos programando
  - Function: puede ser (SIG\_DFL/SIG\_IGN/función de usuario)
- Que devuelve?
  - (void \*)-1 en caso de error
  - La dirección de la anterior rutina de tratamiento

## ❑ No todos los signals se pueden reprogramar

## ❑ UNIX estándar:

- La programación del signal sólo sirve para una vez, luego hay que reprogramarla

## ❑ La programación de signals se hereda al hacer un fork, pero se pierde al hacer un exec (se pierde el código anterior)

## Eventos: Recepción de un signal

- ❑ **¿Qué hace el SO cuando se envía un signal a un proceso?**
  - Salva el contexto del proceso
  - Ejecuta la rutina asociada al signal
  
- ❑ **¿Que hace el proceso que recibe un signal????**
  - Si esta ejecutándose:
    - Se ejecuta la rutina de atención al signal
    - Si ha de continuar lo hace en el punto donde estaba antes de recibir el signal
  - Si estaba bloqueado en una llamada a sistema
    - Se ejecuta la rutina de atención al signal
    - La llamada a sistema devuelve -1 y errno=EI NTR (finalización incorrecta)

# Eventos: Esperar un evento

- ❑ Un evento puede llegar en cualquier momento (asíncrono), pero es posible que queramos esperar explícitamente la llegada de un evento
- ❑ **Cómo espera un proceso una notificación???**
  - Llamada a sistema: `int pause()`
    - **Bloquea** al proceso hasta que llega un evento que esté programado por el usuario (cualquier evento de los programables por el usuario)
  - Que devuelve???
    - Si el proceso acaba no devuelve nada (lógico), sino, devuelve -1 y `errno=EINVAL`
  - Cuando el proceso está bloqueado no consume CPU

# Eventos

## ❑ Programar una alarma de tiempo

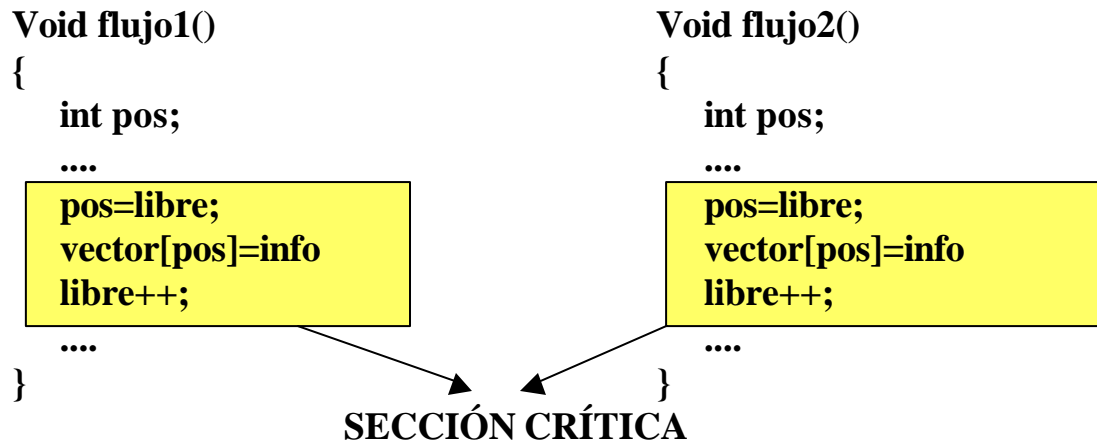
- Llamada a sistema: `unsigned int alarm(unsigned int sec)`
  - Solicitud de aviso mediante signal SIGALRM al cabo de `sec` segundos de tiempo real
- Qué devuelve???
  - Número de segundos restantes de la última petición
- La solicitud de aviso se hace al proceso
  - Si el proceso hace un `fork` el proceso hijo no lo recibe
  - Si el proceso hace un `exec` el aviso sigue pendiente. RECORDAR: la programación del SIGALRM si se perdería
- Las peticiones no se acumulan, si hacemos una y antes de que acabe el tiempo otra la primera se pierde. Si hacemos `alarm(0)` se desactiva la que esté pendiente

## ❑ SIGALRM es un evento y hay que programarlo específicamente (llamada a sistema signal)

## ❑ La llamada a sistema `alarm` no bloquea al proceso

## 4.4 Comunicación mediante memoria compartida

- ❑ Dos flujos de un mismo proceso pueden usar la memoria física que comparten para comunicarse: memoria compartida
- ❑ Hay que asegurar que el resultado no dependa del orden de ejecución ya que no lo conocemos. Esta situación se llama "race condition"
  - Asegurar que dos flujos no puedan leer/escribir en la misma posición de memoria a la vez
- ❑ El conjunto de instrucciones que acceden a variables compartidas es una "sección crítica"



## 4.4.1 Exclusión mutua (mutual exclusion)

### ❑ **Exclusión mutua**

- Mecanismo para garantizar que dos flujos no acceden simultáneamente a una sección crítica

### ❑ **El S.O nos proporciona la herramientas pero es el programador el que ha de usarlas adecuadamente**

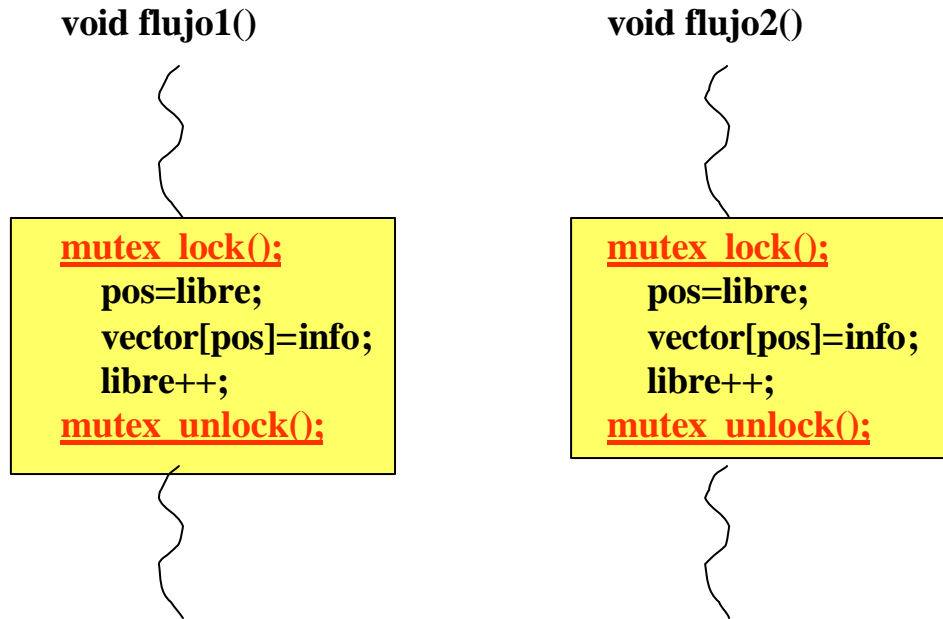
### ❑ **Condiciones que garantizan un acceso correcto (Dijkstra)**

- Si un flujo está ejecutando una sección crítica, ningún otro flujo podrá entrar en la misma sección crítica
- Un flujo no puede esperar indefinidamente para entrar en una sección crítica
- Un flujo que está ejecutando fuera de una sección crítica no puede impedir que los otros flujos entren en ella
- No se pueden hacer hipótesis sobre el número de procesadores, ni la velocidad de ejecución

### ❑ **Las secciones críticas tienen un punto de entrada y un punto de salida**

## 4.4.1 Exclusión mutua (mutex)

- ❑ La exclusión mutua se ha de cumplir aunque haya cambios de contexto dentro de la sección crítica
- ❑ Se secuencializa el acceso a las variables compartidas



## 4.4.2.1 Implementaciones: Espera activa

### ❑ Espera activa (busy waiting)

- Necesitaremos soporte de la arquitectura: instrucción atómica, es decir, ininterrumpible (instrucción de lenguaje máquina)
  - Consulta y modificación de una variable de forma atómica
- El equivalente en alto nivel sería...

```
int test_and_set(int *a)
{
    int tmp=*a;
    *a=1;
    return(tmp);
}
```

Es necesario hacerlo  
Por hardware para que  
Sea atómico

- Como se usa????

(variable global)

```
int hay_alguien=0;
```

Sólo se inicializa una vez

Flujo1

```
while(test_and_Set(hay_alguien));
```

SECCIÓN CRITICA

```
hay_alguien=0;
```

Flujo2

```
while(test_and_Set(hay_alguien));
```

SECCIÓN CRITICA

```
hay_alguien=0;
```

lock

unlock

## 4.4.2.1 Espera activa

### ❑ Inconvenientes:

- Ocupamos la cpu mientras esperamos poder entrar en una sección crítica, (trabajo no útil)
  - Podría resultar que no dejamos avanzar al flujo que ha de liberar la sección crítica
- Podríamos colapsar el bus de memoria (siempre accediendo a la misma instrucción y la misma variable)
- El resto de los usuarios no pueden aprovechar la cpu, el proceso no está bloqueado

### ❑ Posible solución

- Consultar si podemos acceder a la sección crítica, y si no podemos, bloquear el proceso hasta que nos avisen que podemos acceder
  - En lugar de estar consultando continuamente como en espera activa

## 4.4.2.2 Bloqueo

### ❑ Idea:

- Evitar el consumo inútil de tiempo de cpu.
- Reducir el número de accesos a memoria

### ❑ Propuesta:

- Bloquear a los flujos que no pueden entrar en ese momento
- Desbloquearlos cuando quede libre la sección crítica

### ❑ Utilizaremos SEMAFOROS

- `sem_init(sem,n)`: crea un semáforo
- `sem_wait(sem)`: entrada en exclusión mutua (equivale al lock)
- `sem_signal(sem)`: salida de exclusión mutua (equivale al unlock)

### ❑ Qué es un semáforo??

- Es una estructura de datos del SO que tendrá asociado un contador y una cola de procesos bloqueados. Sirve para proteger el acceso a recursos.
- El contador indica la cantidad de accesos simultáneos que permitimos al recurso que protege el semáforo
  - Si usamos el semáforo para hacer exclusión mútua: Recurso=sección crítica,  $n=1$ .

# Semáforos

□ `sem_init(sem,n)`

```
sem->count=n;  
Ini_queue(sem->queue);
```

□ `sem_wait(sem)`

```
Sem->count--;  
If (sem->count<0{  
    bloquear_flujo(sem->queue);  
}
```

*/\* bloquea al flujo que hace la llamada\*/*

□ `sem_signal(sem)`

```
sem->count++;  
If (sem->count<=0){  
    despertar_flujo(sem->queue);  
}
```

*/\* despierta un flujo de la cola \*/*

# Semáforos

## □ En función del valor inicial del contador usaremos el semáforo para distintos fines

- `sem_init(sem,1)`: MUTEX (permitimos que 1 flujo acceda de forma simultánea a la sección crítica)
- `sem_init(sem,0)`: SINCRONIZACIÓN
- `sem_init(sem,N)`: RESTRICCIÓN DE RECURSOS, genérico

## □ Habitualmente usaremos:

- Espera activa si los tiempos de espera se prevén cortos
- Bloqueo si se prevén largos
  - Bloquear un flujo es costoso (entrar a sistema)

## □ Ejemplo lectores/escritores

## 4.4.3 Deadlock (Abrazo mortal)

- Se produce un abrazo mortal entre un conjunto de flujos si cada flujo del conjunto está bloqueado esperando un acontecimiento que solamente puede estar provocado por otro flujo del conjunto

```
Flujo 1
Conseguir(impresora)
Conseguir(cinta)
    imprimir_datos_cinta()
Liberar(cinta)
Liberar(impresora)
```

```
Flujo2
Conseguir(cinta)
Conseguir(impresora)
    imprimir_datos_cinta()
Liberar(cinta)
Liberar(impresora)
```

- Se han de cumplir 4 condiciones a la vez para que haya abrazo mortal
  1. Exclusion mutua (mutual exclusion)
  2. Hold&Wait
  3. No preempción (no preemption)
  4. Espera circular (circular wait)

## Abrazo mortal

- ❑ **Mutual exclusion:** mínimo de 2 recursos no compartibles
- ❑ **Hold&Wait:** un flujo consigue un recurso y espera por otro
- ❑ **No preempción:** si un flujo consigue un recurso sólo él puede liberarlo y nadie se lo puede quitar
- ❑ **Circular wait:** ha de haber una cadena circular de 2 o más flujos donde cada uno necesita un recurso que esta siendo usado por otro de la cadena

# Abrazo mortal

- ❑ **Como evitarlos????, evitar que se cumpla alguna de las condiciones anteriores**
  - Tener recursos compartidos
  - Poder quitarle un recurso a un flujo
  - Poder conseguir todos los recursos que necesitas de forma atómica
  - Ordenar las peticiones de recursos (tener que conseguirlos en el mismo orden)