

POLITÉCNICA

Programación Avanzada

Claudio Rossi

Clase 4: Memoria compartida

INDUSTRIALES
ETSII | UPM



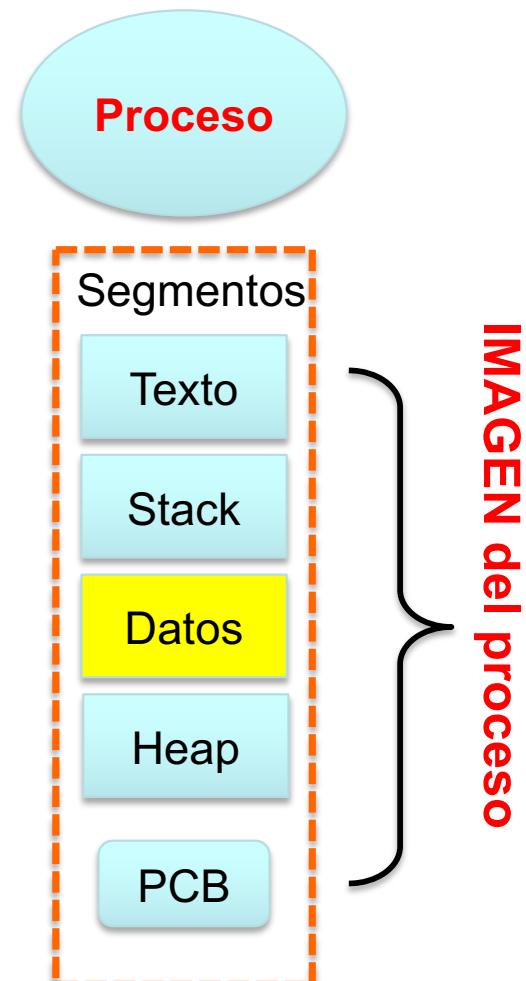
Objetivos:

- Comunicación interproceso a través de *memoria compartida*
- Sincronización de proceso a través de *busy waiting*
 - Memoria dinámica
 - Memoria compartida
 - Concurrencia
 - Busy waiting



Intro : memoria dinámica

- Memoria estática (variables locales)
 - Definida en el código
 - No puede cambiar
 - Ejemplo: `int v[100];`
`float z;`

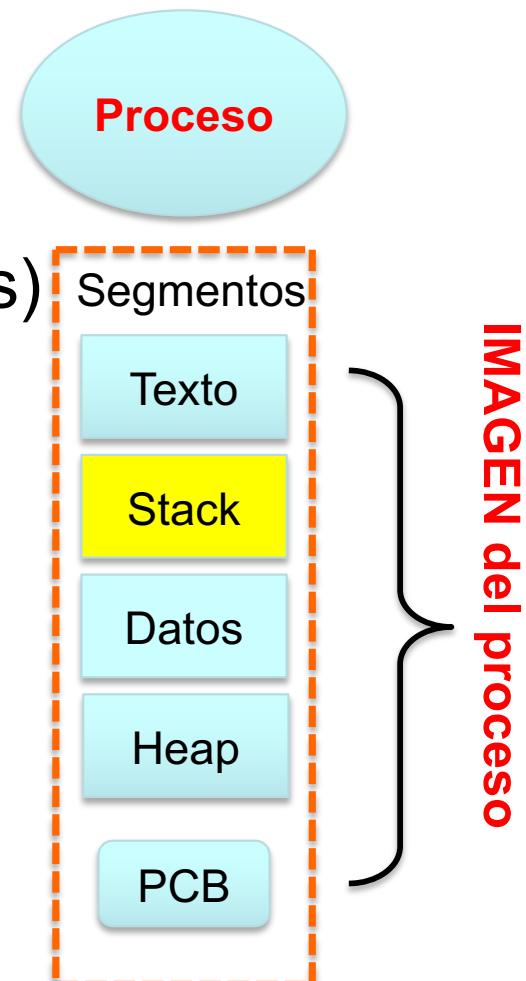




Intro : memoria dinámica

- Funciones (variables locales y parametros)
 - Ejemplo:

```
void f(int x)
{
    int a,b,c;
    ...
}
```

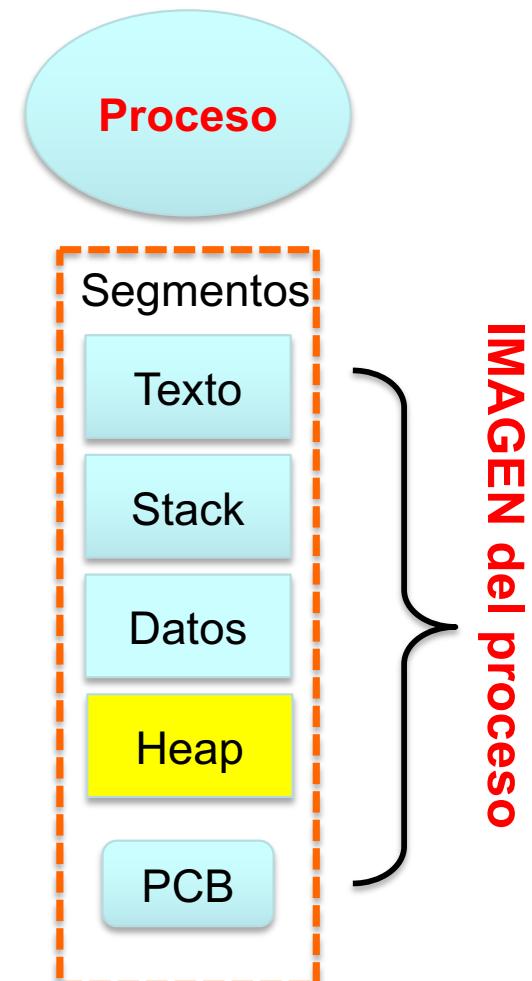




Memoria dinámica

Intro : memoria dinámica

- Memoria estática (variables locales)
 - Definida en el código
 - No puede cambiar
 - Ejemplo: `int v[100];`
- Necesidades:
 - No se sabe el tamaño a priori
 - Necesitamos cambiar el tamaño
- Solución:
 - Memoria dinámica





Intro : memoria dinámica

- **void *malloc(size_t size);**
- Ejemplo:

```
int *v; // recordad: ¡un vector es un puntero !
int n;

... // n tendrá algún valor según la logica del prog.

v = (int*)malloc(n*sizeof(int));
v[3] = 123; // se usa el vector "normalmente"
```



Intro : memoria dinámica

- `void *malloc(size_t size);`
- Ejemplo:

```
int *v;
int n;
... // n tendrá algún valor según la logica del prog.

v = (int*)malloc(n*sizeof(int));
v[3] = 123;
```

Cast al tipo que queremos

Uso de "sizeof"



- Ejemplo 2:

```
typedef struct {  
    int a;  
    float b;  
    ...  
} my_struct;
```

```
my_struct *v;  
int n;
```

```
v = (my_struct*)malloc(sizeof(my_struct)) ;
```

¡ojo con
“sizeof” !



Intro : memoria dinámica

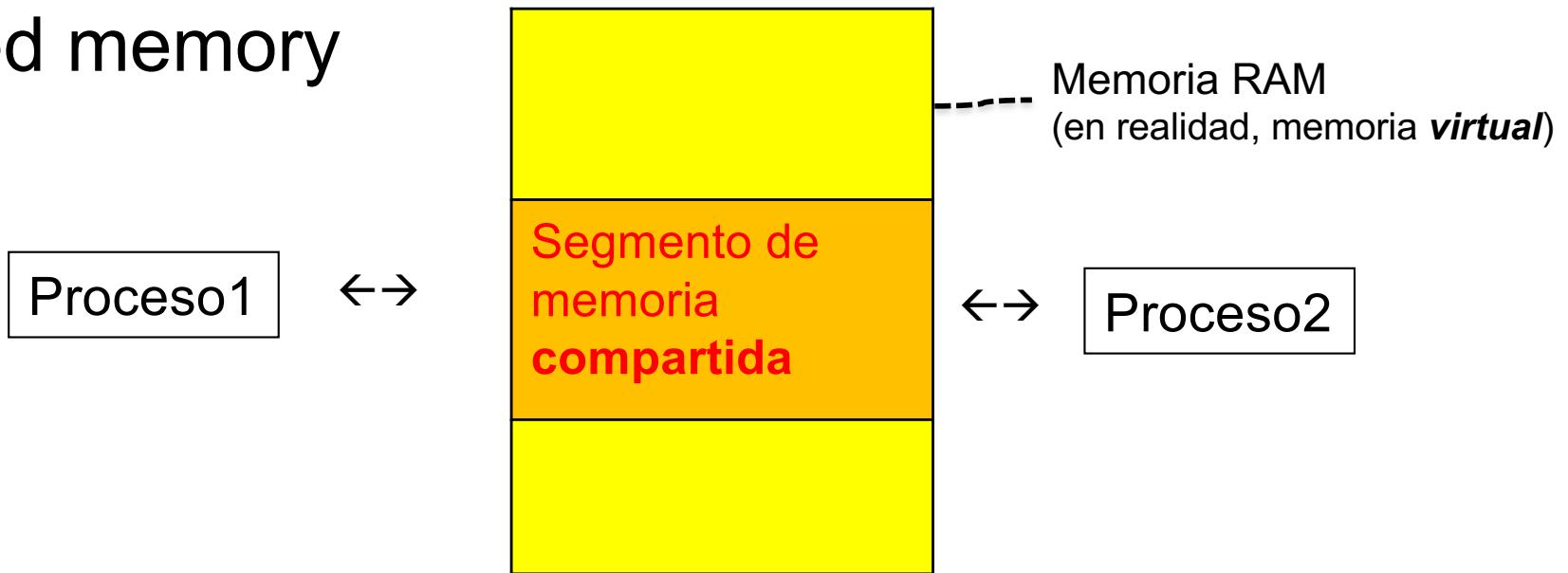
Otras funciones:

- **void free(void *ptr);**
 - Libera la memoria
- **void *calloc(size_t nmemb, size_t size);**
 - Como malloc pero pone a cero el contenido
 - Reserva **nmemb*size** bites
- **void *realloc(void *ptr, size_t size);**
 - Cambia el tamaño de un segmento (si se puede)



Memoria compartida

Shared memory



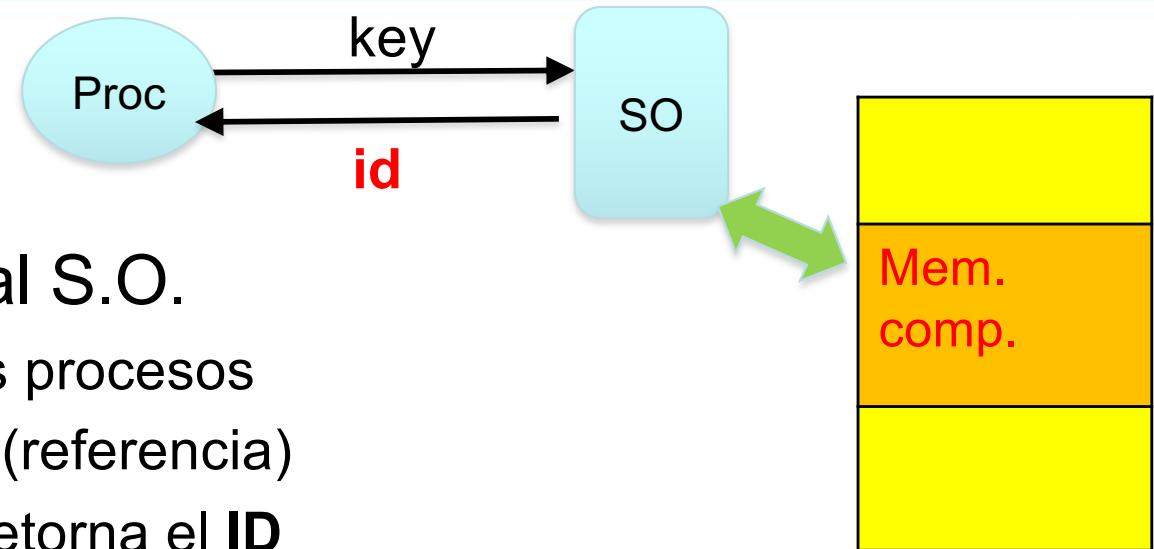
- No es necesario que Proceso1 y Proceso2 tengan relación de parentesco
- El segmento tiene tamaño predefinido
- Dos (**o más!**) procesos pueden leer y escribir en la zona de memoria



Memoria compartida

Procedimiento:

1. Petición de creación al S.O.
 - Clave común para los procesos
 - El S.O. retorna un **ID** (referencia)
 - Si ya existe, el S.O. retorna el **ID**



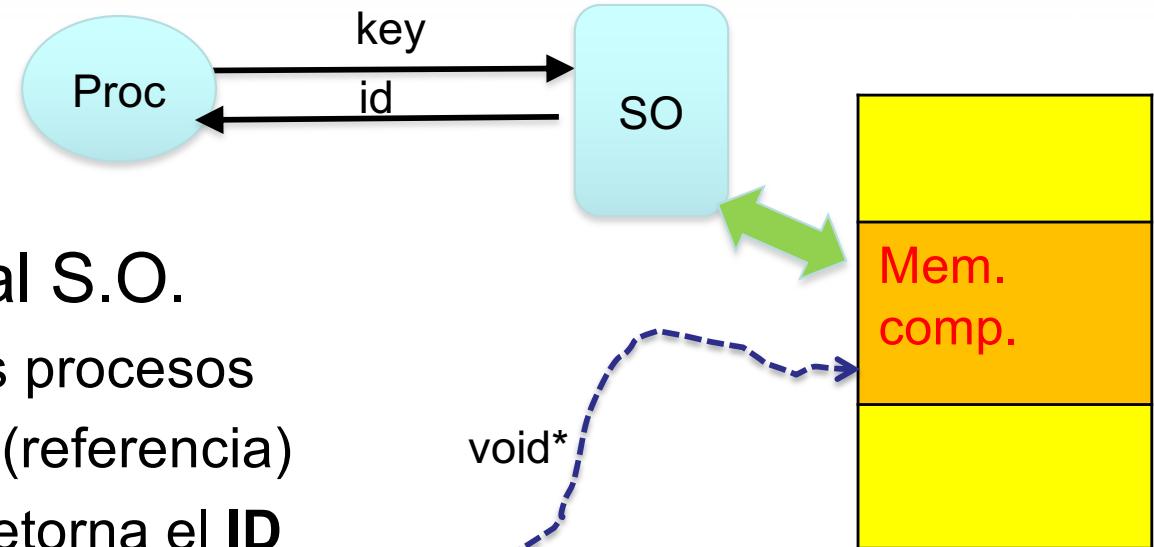


Memoria compartida

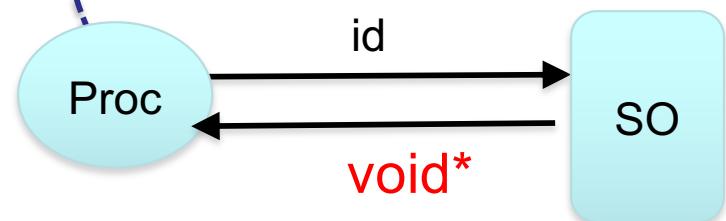
Procedimiento:

1. Petición de creación al S.O.

- Clave común para los procesos
- El S.O. retorna un **ID** (referencia)
- Si ya existe, el S.O. retorna el **ID**



2. Inicialización de un puntero para acceder a la memoria (“attach”)



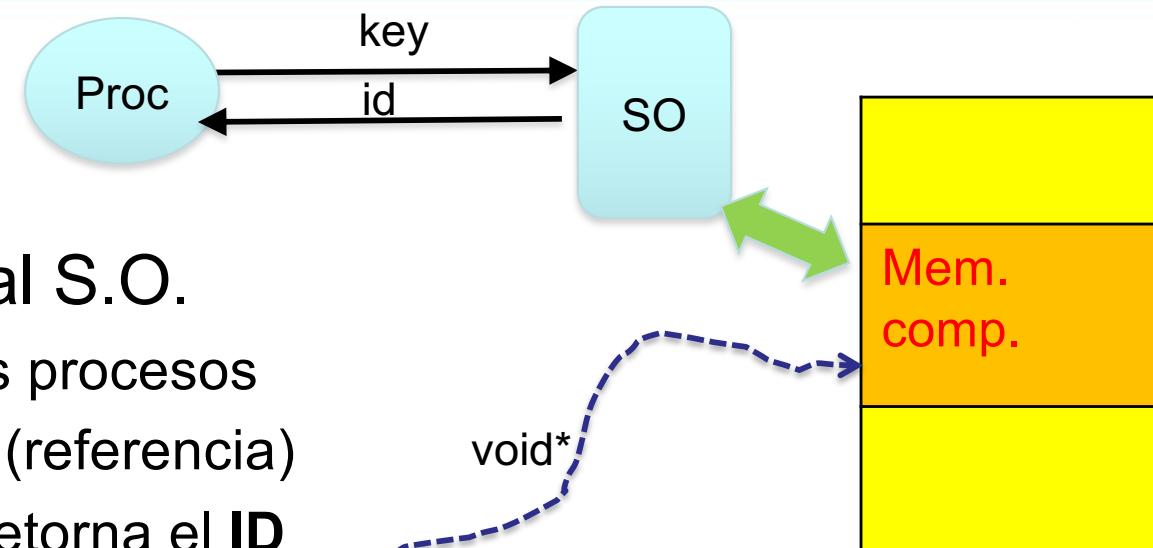


Memoria compartida

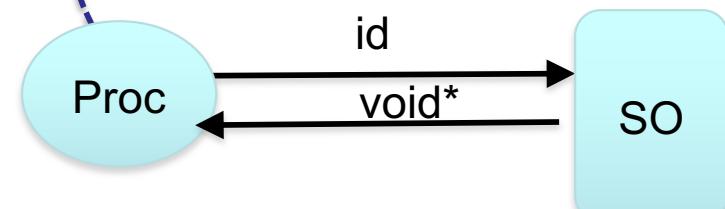
Procedimiento:

1. Petición de creación al S.O.

- Clave común para los procesos
- El S.O. retorna un **ID** (referencia)
- Si ya existe, el S.O. retorna el **ID**



2. Inicialización de un puntero para acceder a la memoria (“attach”)



3. Uso: a través del puntero (no funciones específicas)

- Cast al tipo deseado si es preciso
- Los procesos usan la memoria de forma “normal”



Funciones principales usadas:

- int ***shmget(int key, int size, int flag)***: creación de una zona de memoria, retornando un “**id**” del segmento
 - “key” es una clave común para que los procesos puedan conectarse.
 - Size es el tamaño de la zona
 - Flag es una máscara de bits con instrucciones para la creación (ej: IPC_CREAT) y permisos de acceso con formato rwxrwxrwx



Funciones principales usadas:

- int **shmget(int key, int size, int flag)**: creación de una zona de memoria, retornando un “id” del segmento
 - “key” es una clave común para que los procesos puedan conectarse.
 - Size es el tamaño de la zona
 - Flag es una máscara de bits con instrucciones para la creación (ej: IPC_CREAT) y permisos de acceso con formato rwxrwxrwx
- void***shmat(int id, char *shmaddr, int flag)**: retorna un puntero a la zona especificada por el descriptor *id*
 - shmaddr es la dirección deseada (si=0, el SO decide)
 - Flag establece el modo de acceso (p.e. SHM_RDONLY: solo lectura)

1

2



Funciones principales usadas/2:

- int ***shmctl(int id, int cmd, struct msqid_ds * buf)*** proporciona diversos modos de control sobre la cola de mensaje *id*. La operación queda definida por *cmd*, por ejemplo:
 - IPC_RMID, borra la memoria
 - SHM_LOCK: bloquea la zona de memoria (previene *swapping*) (linux)
 - SHM_UNLOCK: desbloquea la zona de memoria (permite *swapping*) (linux)
- El tercer parámetro, *buf*, es un puntero a una estructura con campos para almacenar el usuario, grupo de usuarios, etc. (no usados aquí).



Comando del S.O.:

- **ipcs**: imprime en pantalla información del estado de las primitivas de IPC (colas, semáforos, memoria compartida)
 - Ejemplo:

```
[claudio:~] ipcs
IPC status from <> as of Thu Mar 29 15:19:51 CEST 2012
T      ID      KEY          MODE          OWNER          GROUP
Message Queues:

T      ID      KEY          MODE          OWNER          GROUP
Shared Memory:
m 393216 0x000003e9 --rw-rw-rw-  claudio        504
T      ID      KEY          MODE          OWNER          GROUP
Semaphores:
```



Comando del S.O./2:

- **ipcrm**: elimina del sistema las estructuras de IPC (colas, semáforos, memoria compartida) especificadas

```
usage: ipcrm [-q msqid] [-m shmid] [-s semid]
              [-Q msgkey] [-M shmkey] [-S semkey] ...
```

Ejemplo: ipcrm -m 393216



Uso: 3

- La memoria creada no tiene formato, son “size” bytes seguidos
- shmat retorna un puntero sin tipo (“void”) al primer byte
- El formato depende de la aplicación, y se aplica mapeando los bytes a estructuras con operaciones de “cast” (cambio de tipo).
 - Ejemplo:

```
float *zona_comun;  
shmid = shmget(...);  
zona_comun = (float*) shmat(shmid, 0, 0);
```



Memoria compartida

```
#define KEY 123
struct X
{
    int a;
    char b[10];
};
```

```
struct X *zona_comun;
```

```
shmid=shmget(int KEY, sizeof(struct X), IPC_CREAT|0600);
```

```
zona_comun = (struct X *) shmat(shmid,0,0);
```

```
zona_comun->a = 10;
(*zona_comun).a = 10;
```

Estas operaciones son ejecutadas por 2 o más procesos, que de esta forma compartirán el segmento de memoria

(Es lo mismo)



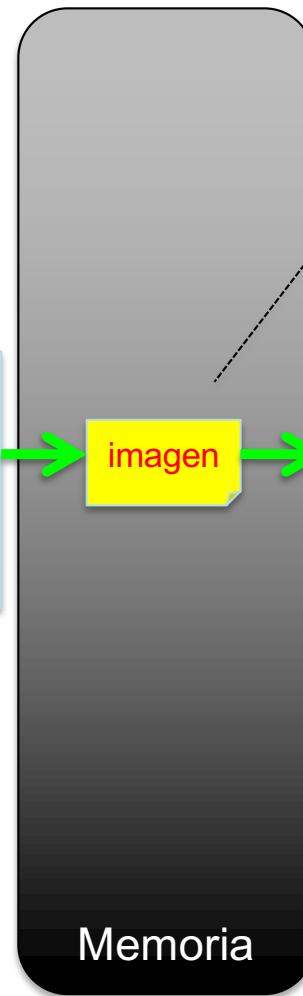
Memoria compartida

Ejemplo



E/S

Captura
imagen



Procesa
imagen



Memoria
compartida



Concurrencia



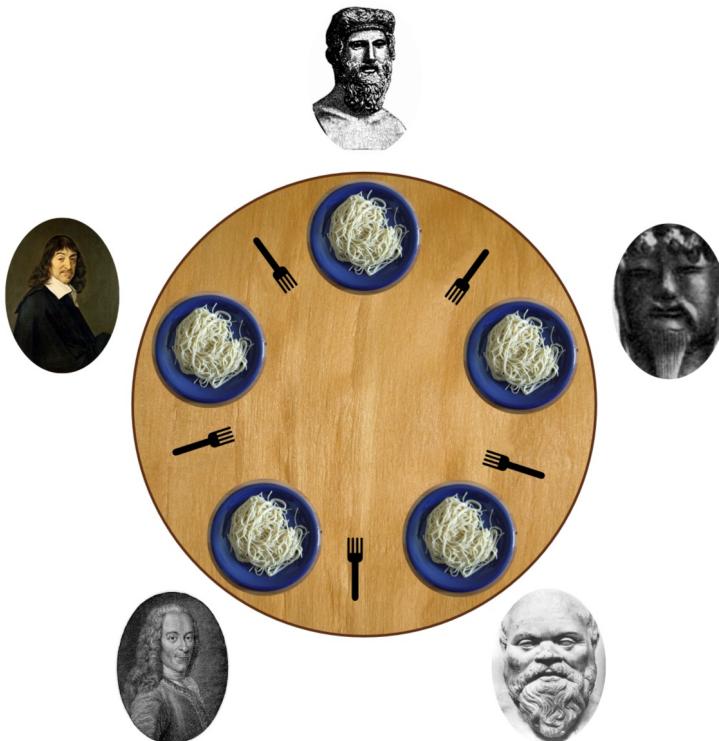
- Más proceso tienen acceso simultaneo a la memoria (o a cualquier otro recurso compartido)
→ Concurrencia

Problema:

- Sincronización en el acceso



- Problemas de los 5 filósofos y del barbero durmiente





Concurrencia

- Problema: el barbero durmiente



- En la tienda hay:
 - Un barbero
 - Un sillón de barbero
 - Una sala de espera con n sillas

Fuente: A. Tanenbaum "modern operating systems"



Concurrencia

- Problema: el barbero durmiente



- En la tienda hay:
 - Un barbero
 - Un sillón de barbero
 - Una sala de espera con n sillas

- Cuando ha terminado con un cliente, mira las sillas por si hay otro
- Si no hay clientes, el barbero se sienta en el sillón y se duerme
- Cuando llega un cliente:
 - Si el barbero está durmiendo, lo despierta
 - Si el barbero está despierto, y hay una silla libre, se sienta y espera, si no se marcha

Donde están los problemas ?





Ejemplo:

- Hay un solo cliente, en el sillón
- No hay ninguna silla ocupada
- El barbero termina de cortar el pelo al cliente, mira las sillas, están vacías, y decide dormirse
- Justo un instante después (el barbero todavía está despierto) entra un cliente
- El cliente toma asiento





Ejemplo 2:

- Queda **una silla libre** en la tienda del barbero
- Dos clientes entran en la tienda a la vez
- Ambos ven la silla libre
- Ambos intentan sentarse





Concurrencia

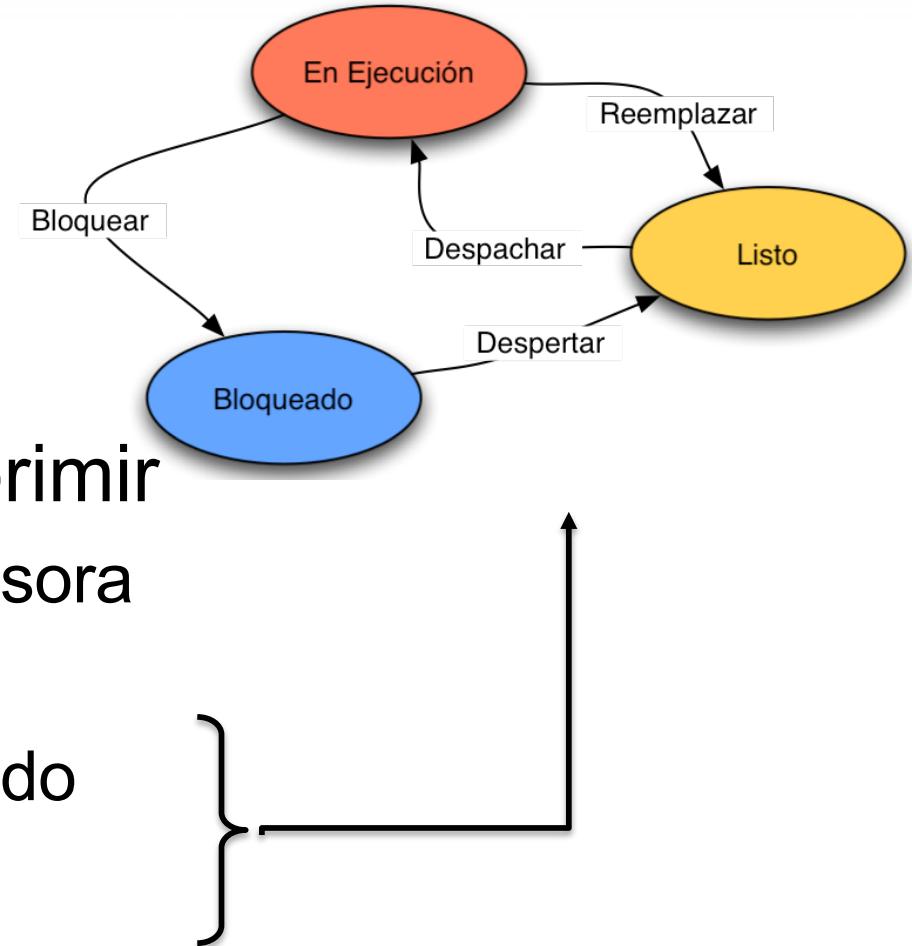
- Traducción:
- Hay un recurso que está libre (ej.: impresora)
- Dos **procesos** van a imprimir
- Ambos ven la impresora libre
- Ambos intentan imprimir
- ¿Es posible?





Concurrencia

- El recurso está libre
- Dos **procesos** van a imprimir
 - El proceso 1 mira la impresora
 - Ve la impresora libre
 - El proceso 1 es interrumpido
 - El proceso 2 es activado
 - El proceso 2 mira la impresora
 - “ “ ve la impresora libre → imprime
 - El proceso 1 es activado → imprime





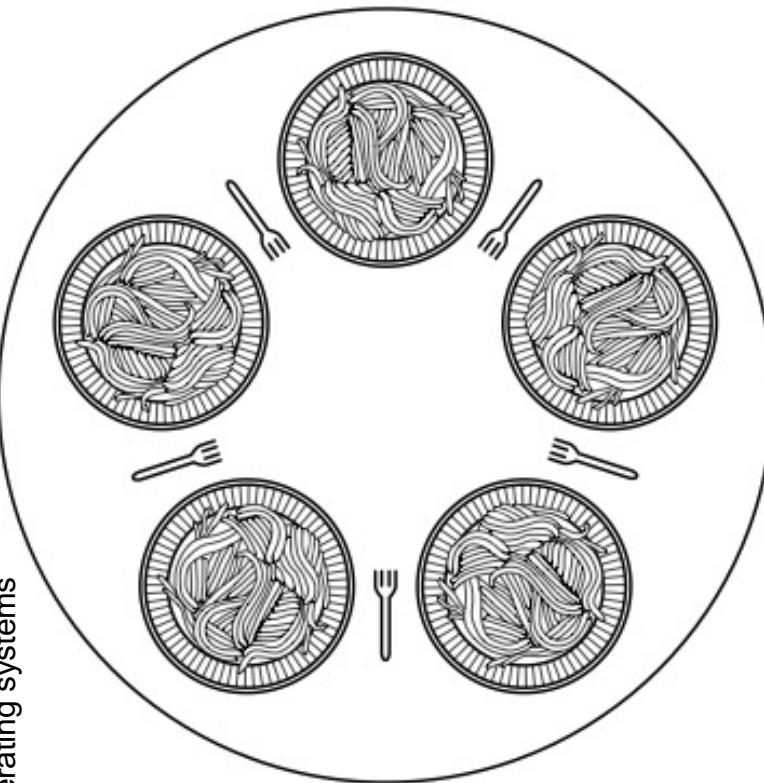
Concurrencia

- Hay dos (o más) operaciones para el uso del recurso, y los procesos pueden ser interrumpidos en cualquier momento



- Problema 1: los 5 filósofos

Fuente: A. Tanenbaum "modern operating systems"



Cada filosofo aplica el
mismo algoritmo

- Cada filósofo tiene 2 actividades: pensar y comer
- Para comer necesita dos tenedores (izquierda y derecha)
 - Cuando necesita comer, coge primero un tenedor y luego el otro
- Terminado de comer, el filosofo devuelve los tenedores
- Los periodos de comer/pensar no son predecibles, así como el orden en el que coge los tenedores



- Problema 1: los 5 filósofos - Una NO-solución

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();  
        take_fork(i);  
        take_fork((i+1) % N);  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
    }  
}
```

/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */

“take_fork” coge el tenedor si está libre, si no espera a que se libere...

Fig. 2-32. A nonsolution to the dining philosophers problem.



- Problema 1: los 5 filósofos
 - Una NO-solución
 - ¿Por que?
 - ¿Cómo se puede mejorar?



```
void philosopher(int i)
{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

“take_fork” coge el tenedor si está libre, si no espera a que se libere...
¿Y si en lugar de esperar suelta la que tiene y sigue?

DEBERES: pensar en una solución.

PARA LOS VAGOS: buscar una solución en internet y entenderla



Concurrencia

- Definiciones:
 - “Hay ocasiones en las que dos (o más) procesos reclaman un recurso”
- **RACE CONDITION** (condición de carrera): condición en las que dos o más procesos intentan acceder a un recurso compartido a la vez



- Definiciones:

- “Hay momentos “delicados”, en los que los procesos acceden a los recurso, que hay que proteger”

→ **SECCIÓN CRITICA:** Segmento de código que accede a recursos compartidos

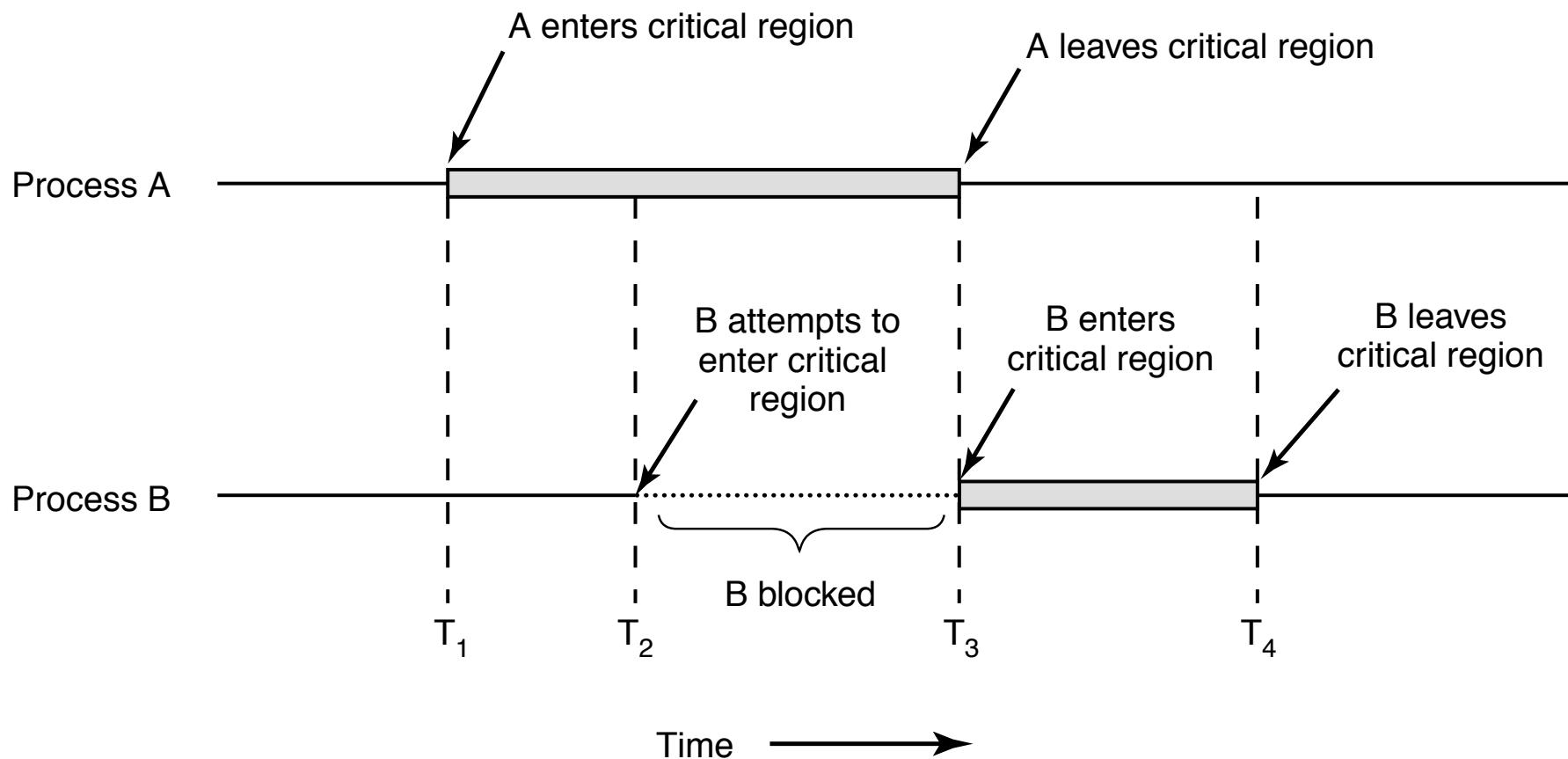
- “Cuando un proceso accede a un recurso, otros no puedan hacerlo”

→ **EXCLUSIÓN MUTUA**



Concurrencia

-

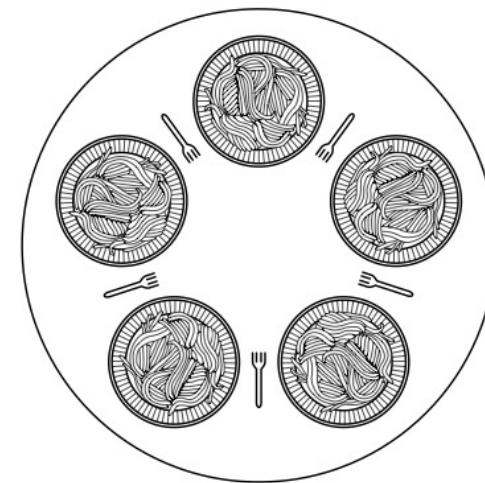
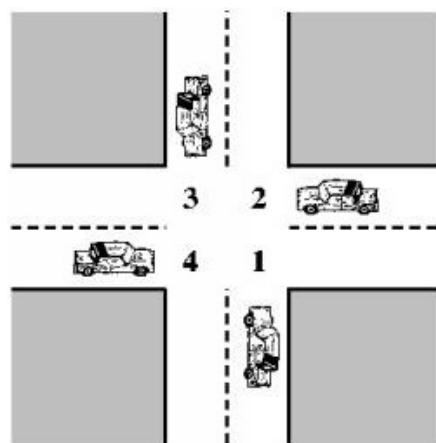


Fuente: A. Tanenbaum "modern operating systems"



DEADLOCK (interbloqueo):

- *Una situación tal que dos o mas acciones competitadoras están esperando a que acaben las otras y, en consecuencia, no termina ninguna*
- Estado tal que dos o más procesos quedan indefinidamente a la espera de que se libere algún recurso, bloqueado por el otro





Concurrencia

- Condiciones para garantizar una concurrencia correcta:
 1. No tiene que haber 2 (o más) procesos contemporáneamente en su sección critica
 2. Un proceso que no esté en su sección critica no debe bloquear a otros procesos
 3. Ningún proceso debe esperar un tiempo ∞ para acceder a su sección critica
 4. Los procesos son imprevisibles: no se puede asumir nada sobre tiempos



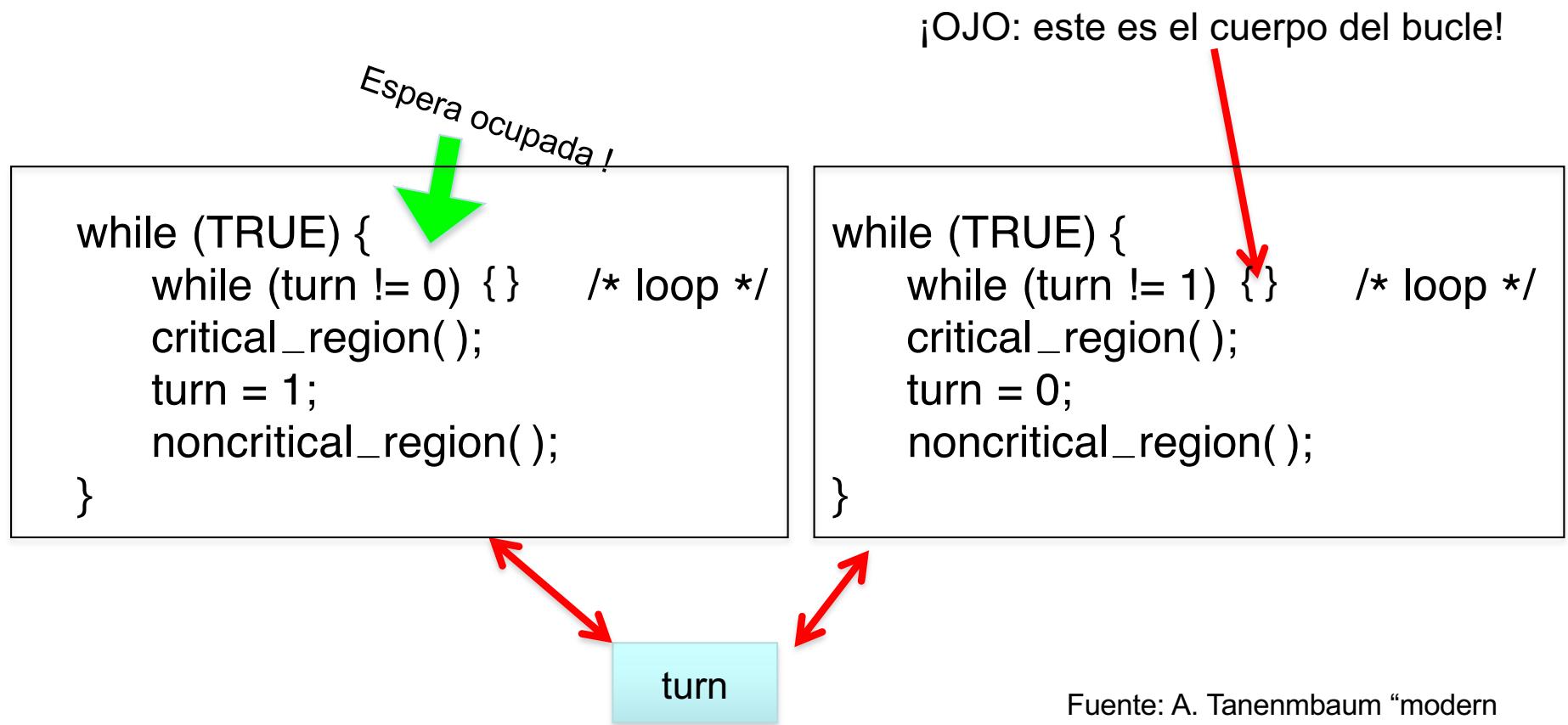
Soluciones

- Hardware (deshabilitar interrupciones)
- “Busy waiting” a través de:
 - Variables compartidas (“lock”)
 - Instrucción “Test and Set Lock”
- Semaforos



Concurrencia

- Ejemplo (busy waiting y turnos a través de variable compartida)





- Soluciones menos malas: TSL
 - Necesita HW especializado
 - Se supone que existe una instrucción “Test and Set Lock”
 - **“Atómica”**: comprueba el valor y lo cambia

– $\text{TSL}(x) \equiv \{ x == 0 ? ; \quad x = 1 ; \}$

Retorna TRUE si era cero, y pone la var. a 1
(en realidad cualquier valor $\neq 0$ vale)

– Uso:

```
while (TLS(lock) == 1) /*espera ocupada*/  
// entra ...
```



- Problemas Busy Waiting:
 - Viola condición 2
 - Si, por ejemplo, el proceso 1 es más rápido y vuelve a intentar entrar en la sección critica, no puede porque es el turno de 0
→ un proc NO en SC está bloqueando otro proceso
 - Malgasta tiempo de CPU
 - Bucle infinito, usa CPU sin I/O



- Más problemas:

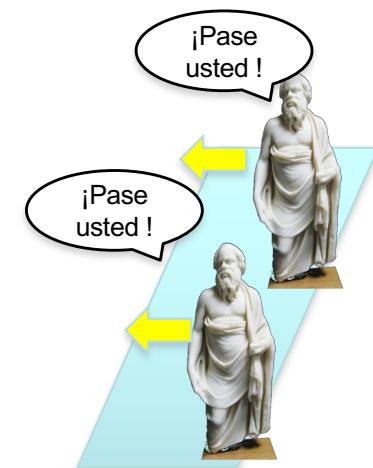
- Inversión de prioridades: un proceso con A prioridad baja esté en su sección critica, y uno B con prioridad alta en busy waiting: el proceso B podría no ser activado nunca ! (según las politica de schedulig)



- Más problemas:

2. Interbloqueo activo (livelock): dos procesos se encuentran contemporáneamente en espera activa, cada uno esperando a que el otro libre el recurso

- Puede darse si se usan variables compartidas, y sucede una interrupción entre el momento de la lectura y el momento de la escritura
- O cuando dos procesos se intentan ceder el paso mutuamente, al infinito



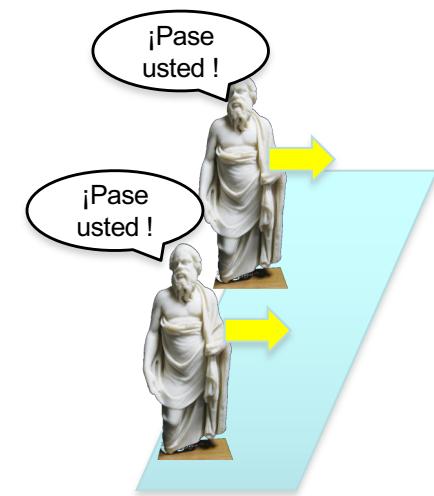


Concurrencia

- Más problemas:

2. Interbloqueo activo (livelock): dos procesos se encuentran contemporáneamente en espera activa, cada uno esperando a que el otro libre el recurso

- Puede darse si se usan variables compartidas, y sucede una interrupción entre el momento de la lectura y el momento de la escritura
- O cuando dos procesos se intentan ceder el paso mutuamente, al infinito

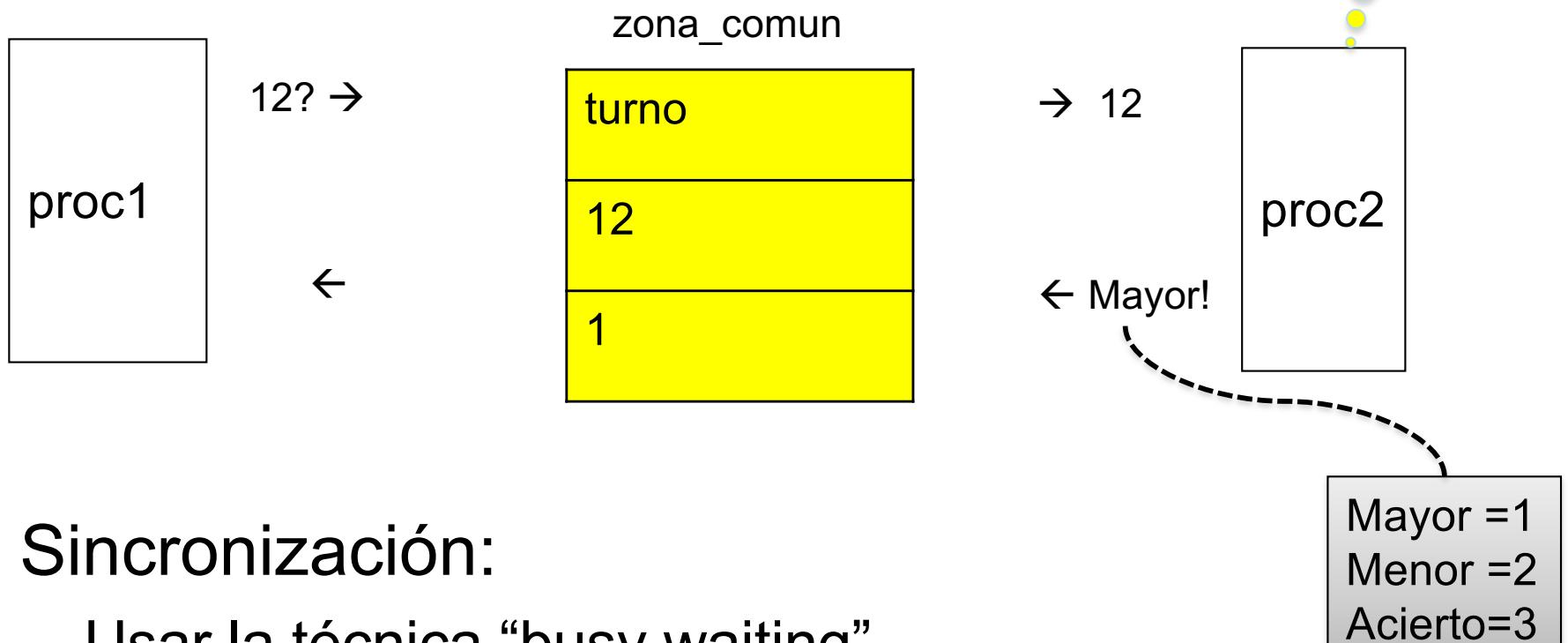




Practica 4



Vuestro problema: *adivina el numero!*



- Sincronización:
 - Usar la técnica “busy waiting”
 - `zona_comun->turno = 1` → turno del proceso1
 - `zona_comun->turno = 2` → turno del proceso2



- Solución (mala) con busy waiting
 - Establecer turnos
 - Una variable compartida, pero la cambia el proceso mismo (pasa el turno)
 - Cruzar los dedos

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)



Practica 4: proceso 1

```
// 1. crea memoria compartida
// 2. se otorga el derecho da acceso zona_comun->turno==1
// 3. escribe su intento
// 4. pasa el turno: zona_comun->turno==2
// 5. bucle principal
while("no he acertado")
{
    while( zona_comun->turno==2) /*busy waiting*/
        // es mi turno
        // accedo a la memoria (leo respuesta y actualizo
        // intento)
        // cedo el turno: zona_comun->turno=2;
}
//6. borra memoria compartida
```

Sección
crítica !



Practica 4: proceso 2

```
// 1. se conecta a la memoria compartida
// 2. asume que el primer turno es del proceso 1
zona_comun->turno==1
// 3. bucle principal
while("no ha acertado")
{
    while( zona_comun->turno==1) /*busy waiting*/
        // es mi turno
        // accedo a la memoria (leo intento y actualizo
        respuesta)
        // cedo el turno:     zona_comun->turno=1;
}
```

Sección
crítica !