



# Sistemas Operativos

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

## Concurrencia de Procesos: Exclusión Mutua y Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

Eloy Anguiano Rey  
eloy.anguiano@uam.es

Ana González  
ana.marcos@uam.es

Escuela Politécnica Superior  
Universidad Autónoma de Madrid

# Introducción

## Elementos a tener en cuenta

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

### Afecta a ..

- ... la comunicación entre procesos.
- ... la compartición y competencia por los recursos.
- ... la sincronización de la ejecución de varios procesos.
- ... la asignación del tiempo de procesador a los procesos.

### Presente en ...

- ... la ejecución de múltiples aplicaciones:
  - Multiprogramación
- ... las aplicaciones estructuradas:
  - Algunas aplicaciones pueden implementarse eficazmente como un conjunto de procesos concurrentes.
- ... la estructura del sistema operativo:
  - Algunos sistemas operativos están implementados como un conjunto de procesos o hilos.



# Introducción

## Términos clave

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

#### Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

**Sincronización:** Los procesos coordinan sus actividades

**Sección crítica:** Región de código que sólo puede ser accedida por un proceso simultáneamente (variables compartidas).

**Exclusión mutua:** Sólo un proceso puede estar en sección crítica accediendo a recursos compartidos

**Interbloqueo:** Varios procesos, todos tienen algo que otros esperan, y a su vez esperan algo de los otros.

**Círculo vicioso:** Procesos cambian continuamente de estado como respuesta a cambios en otros procesos, sin que sea útil (ej: liberar recurso)

**Condición de carrera:** Varios hilos/procesos leen y escriben dato compartido. El resultado final depende de coordinación.

**Inanición:** Proceso que está listo nunca se elige para ejecución

# Introducción

## Dificultades con la concurrencia

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

La ejecución intercalada de procesos mejora rendimiento, pero ... **la velocidad relativa de los procesos no puede predecirse** puesto que depende de:

- Actividades de otros procesos
- Forma de tratar interrupciones
- Políticas de planificación

Pero esto implica que surgen dificultades

### Ejemplo

Hora	Mi compañera	Yo
3:00	Mira en la nevera <b>No hay leche</b>	
3:05	Sale hacia la tienda	
3:10	Entra en la tienda	Miro en la nevera <b>No hay leche</b>
3:15	Compra leche	Salgo hacia la tienda
3:20	Sale de la tienda	Entro en la tienda
3:25	Llega a casa y guarda la leche	Compro leche
3:30		Salgo de la tienda
3:35		Llego a casa y guarda la leche <b>OH OH!!!</b>

# Introducción

## Dificultades con la concurrencia

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

### Dificultades

Ejemplos de  
problemas

### Memoria compartida en UNIX

### Sincronización

### Exclusión mutua

### Soluciones software

### Soluciones hardware

### Semáforos

### Monitores

La imprevisibilidad de la velocidad relativa de los procesos implica que es difícil ...

- ... compartir recursos. Ej: orden de lecturas y escrituras.
- ... gestionar la asignación óptima de recursos. Ej: recursos asignados a un proceso y éste se bloquea, ¿recurso bloqueado?  $\Rightarrow$  posible interbloqueo
- ... detectar errores de programación (resultados no deterministas, no reproducibles)

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

- Supóngase que se lanzan dos procesos idénticos con la siguiente estructura:

```
1 void echo()  
2 {  
3     ent = getchar();  
4     sal = ent;  
5     putchar(sal);  
6 }
```

- Cuando se ejecutan los dos simultáneamente la ejecución puede ser la siguiente:

```
1 ...  
2 ent = getchar();  
3 ...  
4 ...  
5 sal = ent;  
6 putchar(sal);
```

```
1 ...  
2 ...  
3 ent = getchar();  
4 sal = ent;  
5 ...  
6 ...  
7 putchar(sal);
```

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Supóngase la ejecución de estos dos procesos con variables compartidas

Proceso A

```
1  for(i=1 to 5) do {  
2      x=x+1;  
3  }
```

Proceso B

```
1  for(j=1 to 5) do {  
2      x=x+1;  
3  }
```

con las siguientes condiciones:

- Valor inicial  $x=0$ .
- Se comparten todas las variables.
- La operación de incremento se realiza en tres instrucciones atómicas:
  - ① **LD ACC, #** (Carga el contenido de una dirección en el ACC).
  - ② **ACC++** (Incrementa el acumulador).
  - ③ **SV ACC, #** (Almacena el valor del acumulador en una dirección).

**Ejercicio:** calcula todos los valores posibles de salida para la variable x.

# Introducción

## Ejemplos de problemas

### Caso mínimo

i	inst	BCPa	Acc	X	BCPb	inst	j
1	LD Acc		0	0			
		0	0	0		LD Acc	1
		0	1	0		Acc++	
		0	1	1		SV Acc	
		0	1	1		LD Acc	2
		0	2	1		Acc++	
		0	2	2		SV Acc	
		0	2	2		LD Acc	3
		0	3	2		Acc++	
		0	3	3		SV Acc	
		0	3	3		LD Acc	4
		0	4	3		Acc++	
		0	4	4		SV Acc	
	Acc++	0	1	4	4		
	SV Acc	0	1	1	4		

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

#### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores



# Introducción

## Ejemplos de problemas

### Caso mínimo

i	inst	BCPa	Acc	X	BCPb	inst	j
		1	1	1	4	LD Acc	5
2	LD Acc	1	1	1	1		
	Acc++	1	2	1	1		
	SV Acc	1	2	2	1		
3	LD Acc	1	2	2	1		
	Acc++	1	3	2	1		
	SV Acc	1	3	3	1		
4	LD Acc	1	3	3	1		
	Acc++	1	4	3	1		
	SV Acc	1	4	4	1		
5	LD Acc	1	4	4	1		
	Acc++	1	5	4	1		
	SV Acc	1	5	5	1		
			2	5		Acc++	
			2	2		SV Acc	

**X=2**

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

#### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

# Introducción

## Ejemplos de problemas

### Caso máximo

i	inst	BCPa	Acc	X	BCPb	inst	j
			0	0		LD Acc	1
			1	0		Acc++	
			1	1		SV Acc	
			1	1		LD Acc	2
			2	1		Acc++	
			2	2		SV Acc	
			2	2		LD Acc	3
			3	2		Acc++	
			3	3		SV Acc	
			3	3		LD Acc	4
			4	3		Acc++	
			4	4		SV Acc	
			4	4		LD Acc	5
			5	4		Acc++	
			5	5		SV Acc	

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

#### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

# Introducción

## Ejemplos de problemas

### Caso máximo

i	inst	BCPa	Acc	X	BCPb	inst	j
1	LD Acc		5	5			
	Acc++		6	5			
	SV Acc		6	6			
2	LD Acc		6	6			
	Acc++		7	6			
	SV Acc		7	7			
3	LD Acc		7	7			
	Acc++		8	7			
	SV Acc		8	8			
4	LD Acc		8	8			
	Acc++		9	8			
	SV Acc		9	9			
5	LD Acc		9	9			
	Acc++		10	9			
	SV Acc		10	10			

**X=10**

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

#### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave  
Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

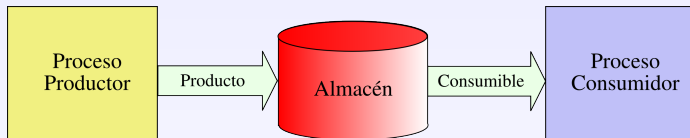
Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

El proveedor produce información para el consumidor. La concurrencia se produce mediante el uso de buffers y variables compartidas (compartición de memoria) o mediante compartición de ficheros.



En el que se tienen las siguientes condiciones:

- Uno o más productores generan datos y los sitúan en un buffer.
- Un único consumidor saca elementos del buffer de uno en uno.
- Sólo un productor o consumidor puede acceder al buffer en un instante dado.

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Compartido

```

1  #define N 100 /*maximo numero de elementos */
2  int contador=0; /* contador del numero de elementos disponibles */
3  typedef type item; /* definicion del producto/consumible */
4  item array[N]; /* array circular, con indice en mdulo N (0..N-1) */
5  item *in=array; /* puntero a la siguiente posicion libre */
6  item *out=NULL; /* puntero primer elemento ocupado */

```

### Productor

```

1  item itemp;
2  while (1) {
3      produce_item (itemp);
4      while (contador==N); /* no hace nada mientras la cola est llena */
5      contador=contador+1;
6      *in = itemp;
7      if(contador==1) out=in; /* actualiza puntero de lectura de datos */
8      if(contador==N) in=NULL; /* actualiza puntero de entrada de datos */
9      else (++in) % N; /* % es el operador modulo */
10 }

```

### Consumidor

```

1  item itemc;
2  while (1) {
3      while (contador ==0); /* no hace nada mientras la cola este vacia */
4      contador = contador -1;
5      itemc = *out;
6      if(contador ==N-1) in=out; /* actualiza puntero de escritura de datos */
7      if(contador ==0) out=NULL; /* actualiza puntero de lectura de datos */
8      else (++out) % N;
9      consume_item(itemc);
10 }

```

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta  
Términos clave  
Dificultades  
Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

**Problema 1:** coordinar lecturas y escrituras, para evitar lecturas sobre elementos no dispensados

La siguiente traza puede dar este problema:

- 1 **Productor:** produce\_item (&itemp);
- 2 **Productor:** while (contador==N);
- 3 **Productor:** contador=contador+1;
- 4 **Consumidor:** while (contador ==0);
- 5 **Consumidor:** contador = contador -1;
- 6 **Consumidor:** itemc = \*out;
- 7 **Consumidor:** if(contador ==N-1) in=out;
- 8 **Productor:** \*in = itemp;
- 9 **Productor:** if(contador==1) out=in;
- 10 **Productor:** if(contador==N) in=NULL;
- 11 **Productor:** else (++in) % N;
- 12 **Consumidor:** if(contador ==0) out=NULL;
- 13 **Consumidor:** else (++out) % N;
- 14 **Consumidor:** consume\_item(itemc);



# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta  
Términos clave  
Dificultades  
Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

**Problema 2:** Si no hay coordinación de procesos, la ejecución concurrente de **`contador=contador+1`** y **`contador=contador-1`** puede dar resultados variados, dependiendo de la traza de ejecución de instrucciones en el procesado.

# Introducción

## Ejemplos de problemas

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Introducción

Elementos a tener en  
cuenta

Términos clave

Dificultades

Ejemplos de  
problemas

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

**contador=contador+1**

$registro_1 = contador$

$registro_1 = registro_1 + 1$

$contador = registro_1$

**contador=contador-1**

$registro_2 = contador$

$registro_2 = registro_2 - 1$

$contador = registro_2$

**Posible secuencia con contador=5**

$T_0(\text{productor}) : registro_1 = contador (registro_1 = 5)$

$T_1(\text{productor}) : registro_1 = registro_1 + 1 (registro_1 = 6)$

**Cambio de contexto**

$T_2(\text{consumidor}) : registro_2 = contador (registro_2 = 5)$

$T_3(\text{consumidor}) : registro_2 = registro_2 - 1 (registro_2 = 4)$

$T_4(\text{consumidor}) : contador = registro_2 (contador = 4)$

**Cambio de contexto**

$T_5(\text{productor}) : contador = registro_1 (contador = 6)$

Cuando se ha producido un elemento y consumido el contador debería de permanecer invariable, en este caso en **5**, sin embargo en esta secuencia el resultado es **6**.



# Memoria compartida en UNIX

## shmget

### shmget

**Crea un segmento de memoria compartida o solicita acceso a un segmento de memoria existente:**

```
int shmget(key_t key, int longitud, int shmflag)
```

### Salida y parámetros

- Devuelve el identificador de segmento para el programa llamante o -1 si hay error.
- **key** identifica el segmento unívocamente en la lista de segmentos compartidos mantenida por el SO.
- **longitud** es el tamaño de la región de memoria compartida.
- **shmflag** es un código octal que indica los permisos de la zona de memoria y se puede construir con un OR de los siguientes elementos.
  - **IPC\_CREAT** crea el segmento si no existe ya.
  - **IPC\_EXCL** si se usa en combinación con IPC\_CREAT, da un error si el segmento indicado por *key* ya existe.

# Memoria compartida en UNIX

## shmat

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

shmget  
**shmat**  
Implementación  
Productor-  
Consumidor

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

shmat

**Añade el segmento de memoria compartida a la memoria del proceso**

`char *shmat(int shmid, char *shmaddr, int shmflag)`

Salida y parámetros

- Devuelve un puntero al comienzo de la zona compartida o -1 si hay error.
- **shmaddr** si es 0, el SO trata de encontrar una zona donde “mapear” el segmento compartido.
- **shmflag** es un OR de los varios elementos, de entre los que podemos destacar:
  - *SHM\_RDONLY*, añade el segmento compartido como de sólo lectura.

# Memoria compartida en UNIX

## Implementación Productor-Consumidor

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

shmget  
shmat  
Implementación  
Productor-  
Consumidor

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

### Productor

```
1  #include <sys/types> #include <sys/ipc.h>
2  #include <sys/shm> #include <stdio.h>
3
4  #define SHMSZ 27
5
6  main{
7      char c, *shm, *s;
8      int shmid;
9      key_t key;
10
11     key=5678;
12     if((shmid = shmget(key,SHMSZ,IPC_CREAT|0666))
13         < 0){
14         perror(shmget);
15         exit(1);
16     }
17     if((shm = shmat(shmid,NULL,0))==(char *) -1){
18         perror(shmat);
19         exit(1);
20     }
21     s = shm;
22     for(c=a;c<=z;c++) *s++ = c;
23     while (*shm != *) sleep(1);
24     exit(0);
25 }
```

### Consumidor

```
1  #include <sys/types> #include <sys/ipc.h>
2  #include <sys/shm> #include <stdio.h>
3
4  #define SHMSZ 27
5
6  main(){
7      int shmid;
8      key_t key;
9      char *shm, *s;
10
11     key=5678;
12     if((shmid = shmget(key,SHMSZ,0666)) < 0){
13         perror(shmget);
14         exit(1);
15     }
16     if((shm = shmat(shmid,NULL,0))==(char *) -1){
17         perror(shmat);
18         exit(1);
19     }
20     for(s = shm; *s !=NULL; s++){
21         putchar(*s);
22         putchar('\n');
23         *shm = *s;
24         exit(0);
25     }
26 }
```



# Sincronización

## Labores del SO

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Labores del SO  
Interacción  
Competencia  
Cooperación

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

- ① Seguir la pista de los distintos procesos activos.
- ② Asignar y retirar los recursos:
  - Tiempo de procesador.
  - Memoria.
  - Archivos.
  - Dispositivos de E/S.
- ③ Proteger los datos y los recursos físicos.
- ④ Los resultados de un proceso deben ser independientes de la velocidad relativa a la que se realiza la ejecución de otros procesos concurrentes.



# Sincronización Interacción

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización  
Labores del SO  
**Interacción**  
Competencia  
Cooperación

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Existen tres formas diferentes de interacción entre procesos:

- 1 Los procesos no tienen conocimiento de los demás.
- 2 Los procesos tienen un conocimiento indirecto de los otros.
- 3 Los procesos tienen un conocimiento directo de los otros.



# Sincronización Competencia

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización  
Labores del SO  
Interacción  
**Competencia**  
Cooperación

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Cuando varios procesos entran en competencia se pueden producir las siguientes situaciones:

- **Exclusión mutua:**
  - Secciones críticas:
    - Sólo un programa puede acceder a su sección crítica en un momento dado.
    - Por ejemplo, sólo se permite que un proceso envíe una orden a la impresora en un momento dado.
- **Interbloqueo.**
- **Inanición.**

# Sincronización Cooperación

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización  
Labores del SO  
Interacción  
Competencia  
Cooperación

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

## Por compartición

- Para que los procesos puedan compartir recursos adecuadamente las operaciones de escritura deben ser mutuamente excluyentes.
- La existencia de secciones críticas garantizan la integridad de los datos.

## Por cooperación

La cooperación se puede realizar por paso de mensajes. En esta situación

- No es necesario el control de la exclusión mutua.
- Puede producirse un interbloqueo:
  - Cada proceso puede estar esperando una comunicación del otro.
- Puede producirse inanición:
  - Dos procesos se están mandando mensajes mientras que otro proceso está esperando recibir un mensaje.

# Exclusión mutua

## Requisitos

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

**Requisitos**

Soluciones para  
garantizarla

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

- 1 Sólo un proceso debe tener permiso para entrar en la sección crítica por un recurso en un instante dado.
- 2 No puede permitirse el interbloqueo o la inanición.
- 3 Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.
- 4 No se deben hacer suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.
- 5 Un proceso permanece en su sección crítica sólo por un tiempo finito.





# Exclusión mutua

## Soluciones para garantizarla

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Requisitos  
Soluciones para  
garantizarla

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

- ① Software con **Espera Activa**
- ② Hardware
  - Deshabilitar interrupciones
  - Instrucciones especiales de hardware
- ③ Con Soporte del SO o del lenguaje de programación (biblioteca):
  - Semáforos

# Soluciones software

## Primer intento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
Cuarto intento  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

### Exclusión mediante el uso de turnos (corrutinas):

- Un proceso está siempre en espera hasta que obtiene permiso (turno) para entrar en su sección crítica.

#### Proceso i

```
1  int turno; /* con valores de 1 a N, siendo N el numero de procesos concurrentes */  
2  
3  while (1) {  
4      while (turno!=i);  
5      ---- SECCION CRITICA ----  
6      turno= (i+1) %N;  
7      ---- RESTO DEL PROCESO ----  
8  }
```

No cumple la condición 3 de entrada inmediata.

# Soluciones software

## Segundo intento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
**Segundo intento**  
Tercer intento  
Cuarto intento  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

- Cada proceso puede examinar el estado del otro pero no lo puede alterar.
- Cuando un proceso desea entrar en su sección crítica comprueba en primer lugar el otro proceso.
- Si no hay otro proceso en su sección crítica fija su estado para la sección crítica.

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 /* Numero de procesos */
4  int interesado[N]; /* Todos los elementos iniciados a FALSE */
```

### Proceso i

```
1  while (interesado[j]== TRUE);
2  interesado[i]=TRUE;
3  ---- SECCION CRITICA ----
4  interesado[i]=FALSE;
5  ---- RESTO DEL PROCESO ----
```

No cumple la condición 1 de exclusión mutua

# Soluciones software

## Tercer intento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
**Tercer intento**  
Cuarto intento  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

- Dar valor a la señal para entrar en la sección crítica antes de comprobar otros procesos.
- Si hay otro proceso en la sección crítica cuando se ha dado valor a la señal, el proceso queda bloqueado hasta que el otro proceso abandona la sección crítica.

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 /* Numero de procesos */
4  int interesado[N]; /* Todos los elementos iniciados a FALSE */
```

### Proceso i

```
1  interesado[i]=TRUE;
2  while (interesado[j] == TRUE);
3  ---- SECCION CRITICA ----
4  interesado[i]=FALSE;
5  ---- RESTO DEL PROCESO ----
```

No cumple la condición 2 de interbloqueo



# Soluciones software

## Cuarto intento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
**Cuarto intento**  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

- Un proceso activa su señal para indicar que desea entrar en la sección crítica, pero debe estar listo para desactivar la variable señal.
- Se comprueban los otros procesos. Si están en la sección crítica, la señal se desactiva y luego se vuelve a activar para indicar que desea entrar en la sección crítica. Esto se repite hasta que el proceso puede entrar en la sección crítica.

# Soluciones software

## Cuarto intento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
**Cuarto intento**  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 /* Numero de procesos */
4  int interesado[N]; /* Todos los elementos iniciados a FALSE */
```

### Proceso i

```
1  interesado[i] = TRUE;
2  while (interesado[j] == TRUE){
3      interesado[i] = FALSE;
4      ---- ESPERA ----
5      interesado[i] = TRUE;
6  }
7  ---- SECCION CRITICA ----
8  interesado[i] = FALSE;
9  ---- RESTO DEL PROCESO ----
```

No cumple la condición 2 de interbloqueo (live lock) ni la 4 de suposición indebida

# Soluciones software

## Algoritmo de Dekker

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

- Se impone un orden de actividad de los procesos.
- Si un proceso desea entrar en la sección crítica, debe activar su señal y puede que tenga que esperar a que llegue su turno.

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 /* Numero de procesos */
4  int turno=1; /* con valores de 0 o 1 */
5  int interesado[N]; /* inicializado a 0 para todos los elementos del array */
```

### Proceso i

```
1  while (1) {
2      interesado[i] =TRUE;
3      while (interesado [j] ==TRUE)
4          if (turno == j) {
5              interesado[i] =FALSE;
6              while (turno == j);
7              interesado[i] =TRUE;
8          }
9      --- SECCION CRITICA ---
10     turno = j; /* cambia turno al otro proceso */
11     interesado [i] =FALSE;
12     --- RESTO DEL PROCESO ---
13 }
```

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
Cuarto intento  
**Algoritmo de Dekker**  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware

# Soluciones software

## Solución de Peterson

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
Cuarto intento  
Algoritmo de Dekker  
**Solución de Peterson**  
Algoritmo de la  
panadería

Soluciones  
hardware

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 /* Nmero de procesos */
4  int turno; /* con valores de 0 o 1 */
5  int interesado[N]; /* inicializado a 0 para todos los elementos del array */
```

### Proceso i

```
1  while (1) {
2      interesado[i] =TRUE;
3      turno = j; /* cambia turno al otro proceso */
4      while ((turno==j) && (interesado [j] ==TRUE));
5      ---- SECCION CRITICA ----
6      interesado[i] =FALSE;
7      ---- RESTO DEL PROCESO ----
8  }
```



# Soluciones software

## Solución de Peterson

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
Cuarto intento  
Algoritmo de Dekker  
**Solución de Peterson**  
Algoritmo de la  
panadería

Soluciones  
hardware

### Desglosado

#### Proceso 1

```
1  while (1) {  
2      interesado[0] =TRUE;  
3      turno = 1; /* cambia turno al otro proceso */  
4      while ((turno==1) && (interesado [1] ==TRUE));  
5      ---- SECCION CRITICA ----  
6      interesado[0] =FALSE;  
7      ---- RESTO DEL PROCESO ----  
8  }
```

#### Proceso 2

```
1  while (1) {  
2      interesado[1] =TRUE;  
3      turno = 0; /* cambia turno al otro proceso */  
4      while ((turno==0) && (interesado [0] ==TRUE));  
5      ---- SECCION CRITICA ----  
6      interesado[1] =FALSE;  
7      ---- RESTO DEL PROCESO ----  
8  }
```

# Soluciones software

## Algoritmo de la panadería

### Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N n /* Nmero de procesos concurrentes */
4  int eligiendo [N]; /* con valores de 0 a n-1 */
5  int numero[N]; /* inicializado a 0 para todos los elementos del array */
```

### Proceso i

```
1  while (1) {
2      eligiendo[i] =TRUE;
3      numero[i]=max(numero[0],..., numero[n-1]) + 1;
4      eligiendo[i]=FALSE;
5      for(j=0;j<n;++j) {
6          while (eligiendo[j] ==TRUE);
7          while ( (numero[j] !=0) && ((numero[j] < numero[i])
8              || ((numero[j]== numero[i]) && (j<i)) ) );
9      }
10     ---- SECCION CRITICA ----
11     numero[i] =0;
12     ---- RESTO DEL PROCESO ----
13 }
```

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Primer intento  
Segundo intento  
Tercer intento  
Cuarto intento  
Algoritmo de Dekker  
Solución de Peterson  
Algoritmo de la  
panadería

Soluciones  
hardware



# Soluciones hardware

## Inhabilitación de interrupciones

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Inhabilitación de  
interrupciones  
Instrucciones de  
máquina especiales

Semáforos

Monitores

Mensajes

- Un proceso continuará ejecutándose hasta que solicite un servicio del sistema operativo o hasta que sea interrumpido.
- Para garantizar la exclusión mutua es suficiente con impedir que un proceso sea interrumpido.
- Se limita la capacidad del procesador para intercalar programas.
- Multiprocesador:
  - Inhabilitar las interrupciones de un procesador no garantiza la exclusión mutua.

# Soluciones hardware

## Instrucciones de máquina especiales

### Instrucciones especiales de máquina

- Se realizan en un único ciclo de instrucción.
- No están sujetas a injerencias por parte de otras instrucciones.
- Leer y escribir.
- Leer y examinar.

### Instrucción TEST&SET

```
1  booleano TS (int i){  
2      if (i == 0) {  
3          i = 1;  
4          return cierto;  
5      } else return falso;  
6  }
```

### Instrucción intercambiar

```
1  void intercambiar(int registro,  
2      int memoria)  
3  {  
4      int temp;  
5      temp = memoria;  
6      memoria = registro;  
7      registro = temp;  
8  }
```

# Soluciones hardware

## Instrucciones de máquina especiales

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Inhabilitación de  
interrupciones  
Instrucciones de  
máquina especiales

Semáforos

Monitores

Mensajes

### Ventajas

- Es aplicable a cualquier número de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador.
- Es simple y fácil de verificar.
- Puede usarse para disponer de varias secciones críticas.

### Desventajas

- Interbloqueo: si un proceso con baja prioridad entra en su sección crítica y existe otro proceso con mayor prioridad, entonces el proceso cuya prioridad es mayor obtendrá el procesador para esperar a poder entrar en la sección crítica.
- La espera activa consume tiempo del procesador.



# Soluciones hardware

## Instrucciones de máquina especiales

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Inhabilitación de  
interrupciones  
Instrucciones de  
máquina especiales

Semáforos

Monitores

Mensajes

Con el fin de solucionar estos problemas en la sincronización se crean múltiples soluciones que veremos en las siguientes secciones y entre las que cabe destacar:

- 1 Semáforos
- 2 Mensajes
- 3 Monitores



# Semáforos

## Definición y Propiedades

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades

Funcionalidad

Exclusión mutua

Productor-  
Consumidor

Problema del baile de

- Los procesos se pueden coordinar mediante el traspaso de señales.
- La señalización se tramita mediante variable especial llamada semáforo.
- Una señal se transmite mediante una operación atómica *up* (signal).
- Una señal se recibe mediante una operación atómica *down* (wait).
- Un proceso en espera de recibir una señal es bloqueado hasta que tenga lugar la transmisión de la señal.
- Los procesos en espera se organizan en una cola de procesos.
- Dependiendo de la política de ordenamiento de procesos en espera:
  - Semáforos robustos: FIFO. Garantizan la no inanición y fuerzan un orden. (Linux)
  - Semáforos débiles: otra política. No garantizan la no inanición (Mac OS X)

- Un semáforo se puede ver como una variable que tiene un valor entero:
  - Puede iniciarse con un valor no negativo.
  - La operación *down* (wait) disminuye el valor del semáforo.
  - La operación *up* (signal) incrementa el valor del semáforo.
- Si la variable sólo puede tomar valores 0 y 1 el semáforo se denomina binario.
- **DOWN** (wait):
  - Comprueba el valor del semáforo:
    - Si semáforo  $> 0$ : decrementa el semáforo.
    - Si semáforo  $\equiv 0$ , el proceso se echa a dormir hasta que pueda decrementarlo.
- **UP** (signal):
  - Incrementa el valor del semáforo.
  - Despierta uno de los procesos durmiendo en este semáforo, que termina su DOWN (signal)  $\rightarrow$  la variable no cambia de valor si había algún proceso durmiendo en ese semáforo.



# Semáforos

## Exclusión mutua

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades

Funcionalidad

Exclusión mutua

Productor-  
Consumidor

Problema del baile de  
mujeres

Los semáforos garantizan la exclusión mutua en el acceso a secciones críticas.

Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N /* Numero de procesos */
4  typedef int semaforo;
5  semaforo mutex=1; /* control de accesos a region critica */
```

Proceso i

```
1  while (1) {
2      down(mutex);
3      ---- SECCION CRITICA ----
4      up(mutex);
5      ---- RESTO DEL PROCESO ----
6  }
```

# Semáforos

## Productor–Consumidor

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua

**Productor–  
Consumidor**

Problema del baile de

### Compartido

```
1  #define N 100
2  typedef type item;
3  item array[N]; /* array circular */
4  item *in=array; /* puntero a la siguiente posicin libre */
5  item *out=NULL; /* puntero primer elemento ocupado */
6
7  typedef int semaforo;
8
9  semaforo mutex=1; /* control de accesos a regin critica */
10 semaforo vacio=N; /* cuenta entradas vacas en el almacen, se
11     inicializa a N, que es el tamao del array */
12 semaforo lleno=0; /* cuenta espacios ocupados en el almacen */
```

# Semáforos

## Productor–Consumidor

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua

Productor–  
Consumidor

Problema del baile de

### Solución correcta

#### Productor

```
1  item itemp;
2  while (1) {
3      produce_item (itemp);
4      down(vacio); /* decrementa entradas vacias */
5      down (mutex); /* entra en la region critica */
6      *in = itemp; /* introduce el elemento en el almacen */
7      up (mutex); /* sale de la region critica */
8      up (lleno); /* incrementa entradas ocupadas */
9  }
```

#### Consumidor

```
1  item itemc;
2  while (1) {
3      down(lleno); /* decrementa entradas ocupadas */
4      down(mutex); /* entra en la region critica */
5      itemc = *out; /* lee el elemento del almacen */
6      up (mutex); /* sale de la region critica */
7      up (vacio); /* incrementa entradas vacias */
8      consume_item (itemc);
9  }
```

# Semáforos

## Productor-Consumidor

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua

Productor-  
Consumidor

Problema del baile de

¿Qué hay erróneo en esta solución?

**Productor**

```
1  item itemp;
2  while (1) {
3      produce_item (itemp);
4      down (mutex); /* entra en la region critica */
5      down(vacio); /* decrementa entradas vacias */
6      *in = itemp; /* introduce el elemento en el almacen */
7      up (lleno); /* incrementa entradas ocupadas */
8      up (mutex); /* sale de la region critica */
9  }
```

**Consumidor**

```
1  item itemc;
2  while (1) {
3      down(mutex); /* entra en la region critica */
4      down(lleno); /* decrementa entradas ocupadas */
5      itemc = *out; /* lee el elemento del almacen */
6      up (vacio); /* incrementa entradas vacias */
7      up (mutex); /* sale de la region critica */
8      consume_item (itemc);
9  }
```

# Semáforos

## Productor–Consumidor

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua

Productor–  
Consumidor

Problema del baile de

¿Qué hay erróneo en esta solución?

**Productor**

```
1  item itemp;
2  while (1) {
3      produce_item (itemp);
4      down(vacio); /* decrementa entradas vacias */
5      down (mutex); /* entra en la region critica */
6      *in = itemp; /* introduce el elemento en el almacen */
7      up (lleno); /* incrementa entradas ocupadas */
8      up (mutex); /* sale de la region critica */
9  }
```

**Consumidor**

```
1  item itemc;
2  while (1) {
3      down(lleno); /* decrementa entradas ocupadas */
4      down(mutex); /* entra en la region critica */
5      itemc = *out; /* lee el elemento del almacen */
6      up (vacio); /* incrementa entradas vacias */
7      up (mutex); /* sale de la region critica */
8      consume_item (itemc);
9  }
```



# Semáforos

## Problema del baile de salón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor

Problema del baile de

En el hotel Hastor de New York existe una sala de baile en la que los hombres se ponen en una fila y las mujeres en otra de tal forma que salen a bailar por parejas en el orden en el que están en la fila. Por supuesto ni un hombre ni una mujer pueden salir a bailar solos ni quedarse en la pista solos. Sin embargo no tienen por qué salir con la pareja con la que entraron.

# Semáforos

## Problema del baile de salón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor

Problema del baile de

### Rendezvous

```
1  semaf mutex1 = 0, mutex2 = 0
2
3  Lider(mutex1,mutex2)
4  {
5      up(mutex1);
6      down(mutex2);
7  }
8
9  Seguidor(mutex1,mutex2)
10 {
11     down(mutex1);
12     up(mutex2);
13 }
```

### Solución

```
1  semaf mutex1 = 0, mutex2 = 0
2  semaf mutex3 = 0, mutex4 = 0
3
4  Hombre()
5  {
6      Lider(mutex1,mutex2);
7      Baila();
8      Seguidor(mutex3,mutex4);
9  }
10
11 Mujer()
12 {
13     Seguidor(mutex1,mutex2);
14     Baila();
15     Lider(mutex3,mutex4);
16 }
```



# Semáforos

## Problema de los Babuinos

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de

En el parque nacional Kruger en Sudáfrica hay un cañón muy profundo con una simple cuerda para cruzarlo. Los babuinos necesitan cruzar ese cañón constantemente en ambas direcciones gracias a una cuerda. Sin embargo:

- Como los babuinos son muy agresivos, si dos de ellos se encuentran en cualquier punto de la cuerda yendo en direcciones opuestas, estos se pelearán y terminarán cayendo por el cañón y muriendo.
- La cuerda no es muy resistente y aguanta a un máximo de cinco babuinos simultáneamente. Si en cualquier instante hay más de cinco babuinos en la cuerda, ésta se romperá y los babuinos caerán también al vacío.



# Semáforos

## Problema de los Babuinos

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de

### Light Switch

Estructura de semáforos que permite controlar el acceso a un determinado recurso a procesos de un sólo tipo.

#### Apropiación del recurso

```
1  semaf mutex = 1 /* mutex */
2  semaf recurso = 1 /* recurso */
3  int cuentaRec = 0 /* cuenta */
4
5  lightSwitchOn(mutex,recurso,cuentaRec)
6  {
7      down (mutex);
8      ++ cuentaRec;
9      if ( cuentaRec == 1)
10         down (recurso);
11         up (mutex);
12 }
```

#### Liberación del recurso

```
1  lightSwitchOff(mutex,recurso,cuentaRec)
2  {
3      down (mutex);
4      -- cuentaRec;
5      if ( cuentaRec == 0)
6          up (recurso);
7      up (mutex);
8  }
```

# Semáforos

## Problema de los Babuinos

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

### Solución

#### Babuino genérico

```
1 Babuino(bab,mut,rec,cRec)
2 {
3     down(bab);
4     lightSwitchOn(mut,rec,cRec);
5     CruzaBabuino();
6     lightSwitchOff(mut,rec,cRec);
7     up(bab);
8 }
```

#### Babuinos

```
1 semaf mutVa = 1, mutVi = 1
2 semaf babVa = 5, babVi = 5
3 sem rec = 1
4 int cVa = 0, cVi = 0
5
6 BabuinoVa()
7 {
8     Babuino(babVa,mutVa,rec,cVa);
9 }
10
11 BabuinoViene()
12 {
13     Babuino(babVi,mutVi,rec,cVi);
14 }
```

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades

Funcionalidad

Exclusión mutua

Productor-  
Consumidor

Problema del baile de  
máscaras

# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de  
módulos



# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de

### Aspectos a tener en cuenta

#### Control de acceso a los recursos compartidos

- Sala de espera
- Sofá
- Sillas

#### Sincronización de acciones entre cliente, barbero y cajero

- El **barbero** espera al cliente en la silla de barbero → el **cliente** se sienta
- El **cliente** espera el corte → el **barbero** indica que ha terminado
- El **barbero** espera que cliente se levante → el **cliente** indica que se ha levantado
- El **cajero** espera a que cliente le pague → el **cliente** paga
- El **cliente** espera el recibo → el **cajero** se lo da

# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de  
mujeres

### Semáforos

#### Capacidad de la tienda: max\_capacidad

- Si entra cliente, se decrementa
- Si sale cliente, se incrementa
- Si la tienda está llena, espera

#### Capacidad del sofá: sofa

- Si se sienta un cliente, se decrementa
- Si se levanta un cliente, se incrementa
- Si están ocupados los sofás (lleno), el cliente espera de pie

# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de

### Semáforos (continuación)

#### Capacidad de sillas: silla\_barbero

- Si se sienta cliente, se decrementa
- Si se levanta cliente, se incrementa
- Si las sillas están llenas, el cliente espera en los sofás

#### Ciente en la silla: cliente\_listo

- Si el barbero está dormido lo despierta
- Si no, impide que se duerma al acabar con otro cliente

#### Corte acabado: terminado

- El barbero indica que ha terminado
- El cliente se puede levantar

# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de  
módulo

### Semáforos (continuación)

#### Silla de barbero libre: dejar\_silla\_b

- El cliente indica que ha dejado la silla
- Otro cliente puede dejar su sofá y ocupar la silla

#### Aviso de pago: pago

- El cliente indica que paga
- El cajero despierta y cobra

#### Aviso de pago terminado: recibo

- El cajero da el recibo y se duerme
- El cliente puede irse

# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades

Funcionalidad

Exclusión mutua

Productor-  
Consumidor

Problema del baile de

### Compartido

```
1  typedef int semaforo;
2  int contador = 0; /* nmero de clientes en la tienda */
3  semaforo max_capacidad=20; /* capacidad del local */
4  semaforo sofa=4; /* sofa de espera */
5  semaforo silla_barbero=3; /* sillas de la barberia */
6  semaforo cobrando=1; /* Limita a 1 el acceso a la caja */
7  semaforo mutex1=1; /* controla el acceso a contador */
8  semaforo mutex2=1; /* controla el acceso al identificador
9      de clientes */
10 semaforo cliente_listo=0; /* clientes en espera de servicio */
11 semaforo dejar_silla_b=0; /* evita colisiones en la caja, espera a
12     que se levante el cliente antes de
13     cobrarle */
14 semaforo pago=0; /* coordina el pago */
15 semaforo recibo=0; /* coordina el recibo */
16 semaforo terminado[50]={0}; /* identifica el usuario servido */
```



# Semáforos

## Problema del Barbero Dormilón

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor-  
Consumidor  
Problema del baile de  
mujeres

### Barbero

```
1 void barbero(void)
2 {
3     int cliente_b;
4     while (1) {
5         down (cliente_listo);
6         down (mutex2);
7         sacar_cola1(cliente_b);
8         up (mutex2);
9         cortar_pelo();
10        up (terminado[cliente_b]);
11        down (dejar_silla_b);
12        up (silla_barbero);
13        down (pago);
14        down (cobrando);
15        aceptar_pago();
16        up (cobrando);
17        up (recibo);
18    }
19 }
```

### Cliente

```
1 void cliente(void)
2 {
3     int num_cliente;
4     down (max_capacidad);
5     entrar_tienda();
6     down (mutex1);
7     contador++;
8     num_cliente=contador;
9     up (mutex1);
10    down (sofa);
11    sentarse_sofa();
12    down (silla_barbero);
13    levantarse_sofa();
14    up (sofa);
15    sentarse_silla_barbero();
16    down (mutex2);
17    poner_cola1(num_cliente);
18    up (cliente_listo);
19    up (mutex2);
20    down (terminado[num_cliente]);
21    levantarse_silla_barbero();
22    up (dejar_silla_b);
23    pagar();
24    up (pago);
25    down (recibo);
26    salir_tienda();
27    up (max_capacidad);
28 }
```



# Semáforos

## Lectores–Escritores

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor–  
Consumidor  
Problema del baile de

### Problema

Acceso a recursos compartidos de lectura/escritura

### Propiedades

- Cualquier número de lectores puede leer un archivo simultáneamente.
- Sólo puede escribir en el archivo un escritor en cada instante.
- Si un escritor está accediendo al archivo, ningún lector puede leerlo.

# Semáforos

## Lectores–Escritores

### Prioridad a los lectores Compartido

```
1  typedef int semaforo;  
2  int contlect = 0; /* contador de lectores */  
3  semaforo mutex=1; /* controla el acceso a contlec */  
4  semaforo esem=1; /* controla el acceso de escritura */
```

### Lector

```
1  void lector(void)  
2  {  
3      while (1) {  
4          down (mutex);  
5          contlect = contlect + 1;  
6          if(contlect ==1) down (esem);  
7          up (mutex);  
8          lee_recurso();  
9          down (mutex);  
10         contlect = contlect - 1;  
11         if(contlect ==0) up (esem);  
12         up (mutex);  
13     }  
14 }
```

### Escritor

```
1  void escritor(void)  
2  {  
3      while (1) {  
4          down (esem);  
5          escribe_en_recurso();  
6          up (esem);  
7      }  
8  }
```

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor–  
Consumidor  
Problema del baile de  
mujeres

# Semáforos

## Lectores–Escritores

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades  
Funcionalidad  
Exclusión mutua  
Productor–  
Consumidor  
Problema del baile de

## Prioridad a los escritores

### Compartido

```
1  typedef int semaforo;  
2  int contlect = 0; /* contador de lectores */  
3  int contesc = 0; /* contador de escritores */  
4  semaforo mutex1=1; /* controla el acceso a contlec */  
5  semaforo mutex2=1; /* controla el acceso a contesc */  
6  semaforo mutex3=1; /* controla el acceso al semaforo lsem */  
7  semaforo esem=1; /* controla el acceso de escritura */  
8  semaforo lsem=1; /* controla el acceso de lectura */
```

# Semáforos

## Lectores–Escritores

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Definición y  
Propiedades

Funcionalidad

Exclusión mutua

Productor–  
Consumidor

Problema del baile de

### Prioridad a los escritores

#### Lector

```
1 void lector(void)
2 {
3     while (1) {
4         down (mutex3);
5         down (lsem);
6         down (mutex1);
7         contlect = contlect + 1;
8         if(contlect ==1) down (esem);
9         up (mutex1);
10        up (lsem);
11        up (mutex3);
12        lee_recurso();
13        down (mutex1);
14        contlect = contlect - 1;
15        if(contlect ==0) up (esem);
16        up (mutex1);
17    }
18 }
```

#### Escritor

```
1 void escritura(void)
2 {
3     while (1) {
4         down (mutex2);
5         contesc = contesc + 1;
6         if(contesc ==1) down (lsem);
7         up (mutex2);
8         down(esem);
9         escribe_en_recurso();
10        up (esem);
11        down (mutex2);
12        contesc = contesc - 1;
13        if(contesc ==0) up (lsem);
14        up (mutex2);
15    }
16 }
```

## semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(int key, const void nsems, size_t semflg);
```

**Accede o crea los semáforos identificados por key**

## Significado de los parámetros

- **key**: Puede ser IPC\_PRIVATE o un identificador.
- **nsems**: Número de semáforos que se definen en el array de semáforos.
- **semflg**: Es un código octal que indica los permisos de acceso a los semáforos y se puede construir con un OR de los siguientes elementos:
  - IPC\_CREAT: crea el conjunto de semáforos si no existe.
  - IPC\_EXCL: si se usa en combinación con IPC\_CREAT, da un error si el conjunto de semáforos indicado por key ya existe.
- Devuelve un identificador

## semop

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
```

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

**Realiza distintas operaciones sobre el array de semáforos**

## Significado de los parámetros

- **semid**: Identificador del conjunto de semáforos.
- **sops**: Conjunto de operaciones.
- **nsops**: Número de operaciones.

# Monitores

## Definición

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores  
Definición  
Esquema

Mensajes

- Un monitor es un módulo de software.
- Características básicas:
  - Las variables de datos locales están sólo accesibles para el monitor.
  - Un proceso entra en el monitor invocando a uno de sus procedimientos.
  - Sólo un proceso se puede estar ejecutando en el monitor en un instante dado.

## Sincronización

- La sincronización se consigue mediante variables de condición accesibles sólo desde el interior del monitor.
- Funciones básicas:
  - **cwait(c)**: bloquea la ejecución del proceso invocante. Libera el monitor.
  - **csignal(c)**: reanuda la ejecución de algún proceso bloqueado con un cwait sobre la misma condición.



# Monitores Esquema

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

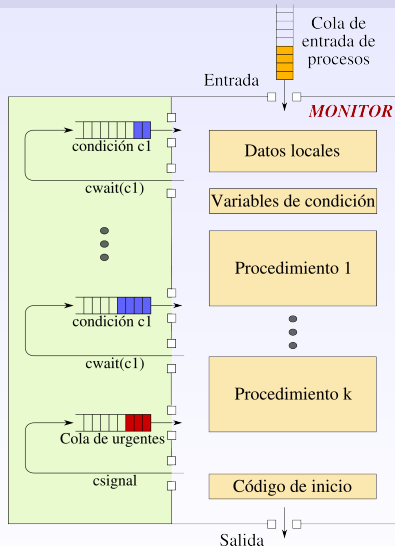
Soluciones  
hardware

Semáforos

Monitores

Definición  
Esquema

Mensajes





# Mensajes

## Definición

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

**Definición**

Sincronización  
Direccionamiento

- Se utilizan como:
  - Refuerzo de la exclusión mutua, para sincronizar los procesos.
  - Medio de intercambio de información.
- La funcionalidad de paso de mensajes se implementa mediante dos primitivas:
  - send (destino, mensaje).
  - receive (origen, mensaje).

- El emisor y el receptor pueden ser bloqueantes o no bloqueantes (esperando un a que se lea un mensaje o a que se escriba un nuevo mensaje).

Hay varias combinaciones posibles:

- Envío bloqueante, recepción bloqueante:
  - Tanto el emisor como el receptor se bloquean hasta que se entrega el mensaje.
  - Esta técnica se conoce como *rendezvous*.
- Envío no bloqueante, recepción bloqueante:
  - Permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sera posible.
  - El receptor se bloquea hasta que llega el mensaje solicitado.
- Envío no bloqueante, recepción no bloqueante:
  - Nadie debe esperar.

# Mensajes

## Direccionamiento

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

Definición  
Sincronización  
Direccionamiento

### Direccionamiento directo

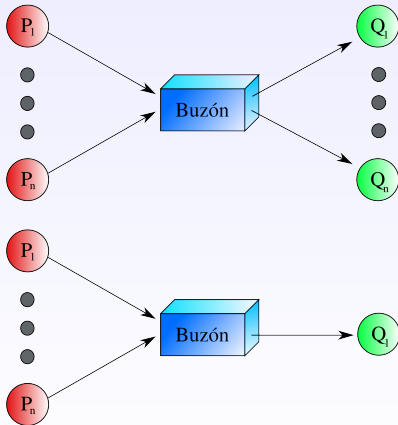
- La primitiva `send` incluye una identificación específica del proceso de destino.
- La primitiva `receive` puede conocer de antemano de qué proceso espera un mensaje.
- La primitiva `receive` puede utilizar el parámetro origen para devolver un valor cuando se haya realizado la operación de recepción.

### Direccionamiento indirecto

- Los mensajes se envían a una estructura de datos compartida formada por colas.
- Estas colas se denominan buzones (*mailboxes*).
- Un proceso envía mensajes al buzón apropiado y el otro los coge del buzón.

Procesos  
emisores

Procesos  
receptores



Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

Definición  
Sincronización  
Direccionamiento

Cabecera

Cuerpo

Tipo de longitud

ID de destino

ID de origen

Longitud de mensaje

Información de control

Contenido del mensaje

# Mensajes

## Lectores–Escritores

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

Definición  
Sincronización  
Direccionamiento

### Controlador

```
1 void controlador(void)
2 {
3     while (1) {
4         if (cont>0) {
5             if (!vacio(terminado)) {
6                 receive (terminado, msj);
7                 cont++;
8             } else if (!vacio(pedir_escritura)) {
9                 receive (pedir_escritura, msj);
10                escritor_id = msj.id;
11                cont = cont-100;
12            } else if (!vacio(pedir_lectura)) {
13                receive (pedir_lectura, msj);
14                cont--;
15                send(buzon[msj.id], "OK");
16            }
17        }
18        if (cont==0) {
19            send (buzon[escritor_id] , "OK");
20            receive (terminado, msj);
21            cont==100;
22        }
23        while(cont<0) {
24            receive (terminado, msj);
25            cont++;
26        }
27    } /* while(1) */
28 } /* fin función controlador */
```

### Lector

```
1 void lector(int i)
2 {
3     mensaje msjl;
4     while (1) {
5         msjl=i;
6         send (pedir_lectura, msjl);
7         receive (buzon[i], msjl);
8         lee_unidad();
9         msjl=i;
10        send (terminado, msjl);
11    }
12 }
```

### Escritor

```
1 void escritura(int j)
2 {
3     mensaje msje;
4     while (1) {
5         msje=i;
6         send (pedir_escritura, msjl);
7         receive (buzon[j], msje);
8         escribe_unidad();
9         msje=j;
10        send (terminado, msje);
11    }
12 }
```

## msgsnd

```
#include <sys/msg.h>
```

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Envía un mensaje a la cola asociada con el identificador msqid

## Significado de los parámetros

- *msgp* apunta a un buffer definido por el usuario, encabezado por un valor de tipo `long int` que indica el tipo del mensaje, seguido de los datos a enviar. Se puede implementar como una estructura:

```
struct mymsg {  
    long mtype; /* tipo de mensaje */  
    char mtext[1]; /* texto del mensaje */  
}
```





# Mensajes UNIX

Concurrencia de  
Procesos:  
Exclusión Mutua y  
Sincronización

Introducción

Memoria  
compartida en  
UNIX

Sincronización

Exclusión mutua

Soluciones  
software

Soluciones  
hardware

Semáforos

Monitores

Mensajes

Definición  
Sincronización  
Direccionamiento

## Significado de los parámetros (continuación)

- *msgsz* indica el tamaño del mensaje, que puede ser hasta el máximo permitido por el sistema.
- *msgflg* indica la acción que se debe llevar a cabo si ocurre alguno de las siguientes circunstancias:
  - El número de bytes en la cola es ya igual a *msg\_qbytes*.
  - El número total de mensajes en en colas del sistema es igual al máximo permitido.
- Si la función se ejecuta correctamente, la función devuelve un 0. En caso contrario -1.

## Significado de los parámetros (continuación)

- Las acciones posibles en función de *msgflg* son:
  - Si (*msgflg*&*IPC\_NOWAIT*) es distinto de 0 el mensaje no se envía y el proceso no se bloquea en el envío.
  - Si (*msgflg*&*IPC\_NOWAIT*) es 0, se bloquea la ejecución del proceso hasta que ocurre uno de estos eventos:
    - Se solventa la condición que ha provocado el bloqueo, en cuyo caso el mensaje se envía.
    - *msqid* se elimina del sistema. Esto provoca el retorno de la función con un *errno* igual a *EIDRM*.
    - El proceso que está intentando enviar el mensaje recibe una señal que debe capturar, por lo que el programa continúa como se le indique en la rutina correspondiente.

## msgrcv

```
#include <sys/msg.h>
```

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

Lee un mensaje a la cola asociada con el identificador *msqid* y lo guarda en el buffer apuntado por *msgp*

## Significado de los parámetros

- *msgp* apunta a un buffer definido por el usuario, encabezado por un valor de tipo `long int` que indica el tipo del mensaje, seguido de los datos a enviar. Se puede implementar como una estructura:

```
struct mymsg {  
    long mtype; /* tipo de mensaje */  
    char mtext[1]; /* texto del mensaje */  
}
```

## Significado de los parámetros (continuación)

- *msgtyp* indica el tipo del mensaje a recibir. Si es 0 se recibe el primero de la cola; si es  $>0$  se recibe el primer mensaje de tipo *msgtyp*; si es  $<0$  se recibe el primer mensaje del tipo menor que el valor absoluto de *msgtyp*.
- *msgsz* indica el tamaño en bytes del mensaje a recibir. El mensaje se trunca a ese tamaño si es mayor de *msgsz* bytes y (*msgflg*&MSG\_NOERROR) es distinto de 0. La parte truncada se pierde.
- Si la función se ejecuta correctamente, la función devuelve un 0. En caso contrario -1.

## Significado de los parámetros (continuación)

- *msgflg* indica la acción que se debe llevar a cabo no se encuentra un mensaje del tipo esperado en la cola. Las acciones posibles son:
  - Si (*msgflg*&IPC\_NOWAIT) es distinto de 0 el proceso vuelve inmediatamente y la función devuelve -1.
  - Si (*msgflg*&IPC\_NOWAIT) es 0, se bloquea la ejecución del proceso hasta que ocurre uno de estos eventos:
    - Se coloca un mensaje del tipo esperado en la cola.
    - *msqid* se elimina del sistema. Esto provoca el retorno de la función con un *errno* igual a EIDRM.
    - El proceso que está intentando enviar el mensaje recibe una señal que debe capturar, por lo que el programa continúa como se le indique en la rutina correspondiente.