Concurrencia de Procesos: Exclusión Mutua y Sincronización

Eloy Anguiano

Rosa Mª Carro

Ana González

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Introducción

Elementos a tener en cuenta

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

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 la 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 y se le deniega siempre el acceso a un recurso compartido (procesador y otros).

Introducción
Dificultades con la concurrencia

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

Ejemplo

Hora	Mi compañera	Yo
3:00	Mira en la nevera `No hay leche`	
3:05	Sale hacia la tienda	
3:10	Entra en la tienda	Miro en la nevera `No hay leche`
3:15	Compra leche	Salgo hacia la tienda
3:20	Sale de la tienda	Entro en la tienda (E/S Diferenciada)
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 `OH OH!!!`

Introducción
Dificultades con la concurrencia

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

- Supóngase que se lanzan dos procesos idénticos con la siguiente estructura y variables compartidas ent y sal:

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

- Posible ejecución concurrente:

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

```
8      ...
9      ...
10     ent = getchar();
11     sal = ent;
12     ...
13     ...
14     putchar(sal);
```

Introducción
Ejemplos de problemas

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

```
5  for(j=1 to 5) do {
6      x=x+1;
7  }
```

con las siguientes condiciones:

- Variable compartida x, con valor inicial x=0.
- 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.

Caso mínimo

i	inst	BCPa	Acc	X	BCPb	inst	j
1	LD Acc		0	0			
		Acc=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	**Acc=4**		
	SV Acc	0	1	**1**	4		

Introducción
Ejemplos de problemas

Caso mínimo

i	inst	BCPa	Acc	X	BCPb	inst	j
		Acc=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

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	

Introducción

Ejemplos de problemas

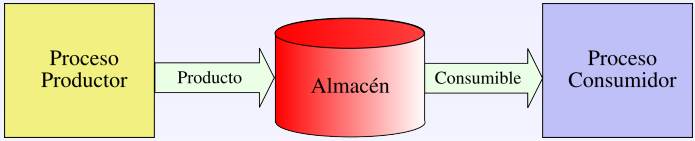
Caso máximo

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

Introducción
Ejemplos de problemas

El proveedor/productor 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:

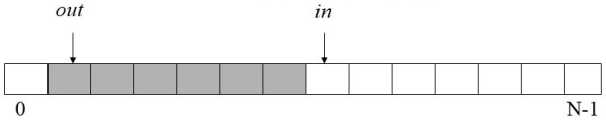
- Uno o más productores generan datos y los sitúan en un *buffer*.
- Un único consumidor saca elementos del *buffer* de uno en uno.
- Sólo un productor o consumidor puede acceder al *buffer* en un instante dado.

Introducción
Ejemplos de problemas

Solución utilizando **Memoria compartida**

Compartido

```
#define N 100                  /*máximo número de elementos */
int contador = 0;              /* contador del número de elementos disponibles */
typedef type item;             /* definición del producto/consumible */
item array[N] ;                /* array circular, con índice en módulo N (0..N-1) */
item *in = array;              /* puntero a la siguiente posición libre.
                                  Inicialmente apunta al primer elemento del array */
item *out = NULL;              /* puntero al primer elemento ocupado.
                                  Inicialmente no apunta a ningún sitio */
```



Cola vacía: *out==NULL*
Cola llena: *in==NULL*

Buffer de tamaño limitado

Introducción

Ejemplos de problemas

Compartido

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

Productor

```
8  item itemp;
9  while (1) {
10     produce_item (itemp);
11     while (contador==N); // no hace nada mientras la cola esté llena
12     contador=contador+1;
13     *in = itemp;
14     if(contador==1) out=in; // actualiza puntero de lectura de datos
15     if(contador==N) in=NULL; // actualiza puntero de entrada de datos
16     else (++in) % N;  // % es el operador módulo
17 }
```

Consumidor

```
19 item itemc;
20 while (1) {
21     while (contador ==0); // no hace nada mientras la cola este vacia
22       contador = contador -1;
23     itemc = *out;
24     if(contador ==N-1) in=out; // actualiza puntero de escritura de datos
25     if(contador ==0) out=NULL; // actualiza puntero de lectura de datos
26     else (++out) % N;
27     consume_item(itemc);
28 }
```

Introducción
Ejemplos de problemas

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

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

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

$T_0(productor) : registro_1 = contador (registro_1 = 5)$

$T_1(productor) : registro_1 = registro_1 + 1 (registro_1 = 6)$

Cambio de contexto

$T_2(consumidor) : registro_2 = contador (registro_2 = 5)$

$T_3(consumidor) : registro_2 = registro_2 - 1 (registro_2 = 4)$

$T_4(consumidor) : contador = registro_2 (contador = 4)$

Cambio de contexto

$T_5(productor) contador = registro_1 (contador = 6)$

Cuando se produce y consume un elemento 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 ( )

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

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=NULL, *s=NULL;
8      int shmid;
9      key_t key;
10
11     key=5678;
12     if((shmid = shmget(key,SHMSZ,IPC_CREAT|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     s = shm;
21     for(c='a';c<='z';c++) *s++ = c;
22     while (*shm != '*') sleep(1);
23     exit(0);
24 }
```

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      char c, *shm=NULL, *s=NULL;
8      int shmid;
9      key_t key;
10
11     key=5678;
12     if((shmid = shmget(key,SHMSZ,IPC_CREAT|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     s = shm;
21     for(c='a';c<='z';c++) *s++ = c;
22     while (*shm != '*') sleep(1);
23     exit(0);
24 }
```

Sincronización
Labores del Sistema Operativo

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

Existen tres formas diferentes de interacción entre procesos:

1. Los procesos no tienen conocimiento de los demás. **Competencia**
2. Los procesos tienen un conocimiento indirecto de los otros. **Cooperación por compartimiento**
3. Los procesos tienen un conocimiento directo de los otros (conocen el PID de los procesos). **Cooperación por comunicación**

Sincronización
Competencia

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

- **Exclusión mutua**:
  - Recurso en sección crítica:
    - 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**. Ej: 2 procesos necesitan 2 recursos; se asigna 1 a cada 1; ambos esperan conseguir el otro (tmb con señales).
- **Inanición**. Ej: 3 procesos necesitan 1 recurso; se va asignando al 1 y al 2 intermitentemente; el 3 sufre inanición

Sincronización
Compartición – Cooperación

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, aunque puede haber problemas de coherencia según el orden de las operaciones.

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

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

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
   - Monitores

Soluciones Software con Espera Activa

Primer intento (fallido)

Exclusión mediante el uso de turnos (variable compartida):

- 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); // Espera activa
5       ---SECCION CRITICA ---
6       turno= (i+1) %N; // Turno Rotatorio
7       ---RESTO DEL PROCESO ---
8  }
```

No cumple la condición 3 de entrada inmediata (un proceso debería poder entrar si no hay otro dentro de la sección crítica).

Soluciones Software con Espera Activa

Segundo intento (fallido)

- 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); //Espera Activa
2  interesado[i]=TRUE; // Expresa el interés de entrar en la Sección Critica
3  ---SECCION CRITICA ---
4  interesado[i]=FALSE; //Libera la Sección Crítica
5  ---RESTO DEL PROCESO ---
```

No cumple la condición 1 de exclusión mutua

Soluciones Software con Espera Activa

Tercer intento (fallido)

- Señala el interés por entrar en la sección crítica antes de comprobar otros procesos.
- Si otro proceso ha mostrado el interes de entrar en la sección crítica, 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; // Muestra el interés por entrar en la Sección Crítica
2  while (interesado[j] == TRUE); // Cortesía: si el otro proceso tiene interes ==>
       Espera Activa
3  ---SECCION CRITICA ---
4  interesado[i]=FALSE; // Libera la Sección Crítica
5  ---RESTO DEL PROCESO ---
```

No cumple la condición 2 de interbloqueo (ambos esperan al otro)

Soluciones Software con Espera Activa
Cuarto intento (fallido)

- Un proceso activa su señal para indicar que desea entrar en la sección crítica (interesado[i]=TRUE), pero debe estar listo para desactivar la variable señal.
- Se comprueban los otros procesos. Si están en la sección crítica, la señal se desactiva (interesado[i]=FALSE) 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 con Espera Activa

Cuarto intento (fallido)

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; // Muestra el interes por entrar en la Sección Critica
2  while (interesado[j] == TRUE){
3        interesado[i] =FALSE; // Cortesía
4       ---ESPERA ---// Tiempo de espera
5        interesado[i]=TRUE; // Vuelvo a mostrar mi interés por entrar en la S. Crítica
6  }
7  ---SECCION CRITICA ---
8  interesado[i ]=FALSE; // Libera la Sección Crítica
9  ---RESTO DEL PROCESO ---
```

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

Soluciones Software con Espera Activa

Algoritmo de Dekker

- 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

```
7   while (1) {
8       interesado[i] =TRUE;
9       while (interesado [j] ==TRUE)
10          if (turno == j) {
11              interesado[i] =FALSE;
12              while (turno == j);
13              interesado[i] =TRUE;
14          }
15      ---SECCION CRITICA ---
16      turno = j; // cambia turno al otro proceso
17      interesado [i] =FALSE;
18      ---RESTO DEL PROCESO ---
19  }
```

Soluciones Software con Espera Activa

Solución de Peterson

Compartido

```
1  #define FALSE 0
2  #define TRUE 1
3  #define N 2 // Número 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

```
7  while (1) {
8      interesado[i] =TRUE;
9      turno = j; // cambia turno al otro proceso
10     while ((turno==j) && (interesado [j] ==TRUE));
11     ---SECCION CRITICA ---
12     interesado[i] =FALSE;
13     ---RESTO DEL PROCESO ---
14 }
```

Soluciones Software con Espera Activa

Solución de Peterson

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 con Espera Activa

Algoritmo de la Panadería de Lamport

Compartido

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

Proceso i

```
7  while (1) {
8      eligiendo[i] =TRUE; // Calcula el número de turno
9      numero[i]=max(numero[0],..., numero[n-1]) + 1;
10     eligiendo[i]=FALSE;
11     for(j=0;j<n;++j) { // Compara con todos los procesos
12         while (eligiendo[j] ==TRUE); // Si el proceso j está eligiendo ==> E. Activa
13         while ( (numero[j] !=0) && ((numero[j] < numero[i])
14                 || ((numero[j]== numero[i]) && (j<i)) ) );
15     }
16     ---SECCION CRITICA ---
17     numero[i] =0; //Libera la sección crítica
18     ---RESTO DEL PROCESO ---
19 }
```

Soluciones Hardware
Inhabilitación de interrupciones

- Para garantizar la exclusión mutua es suficiente con impedir que un proceso sea interrumpido.

```
1  while (1){
2      /* inhabilitar interrupciones */
3      ---SECCION CRITICA ---
4      /* habilitar interrupciones */
5      ---RESTO DEL PROCESO ---
6  }
```

- Un proceso continuará ejecutándose hasta que solicite un servicio del sistema operativo o hasta que 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.
- Ej 1: Leer y escribir.
- Ej 2: Intercambiar contenido de registro con posición de memoria cuyo acceso se bloquea para otras instrucciones.

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      int temp;
4      temp = memoria;
5      memoria = registro;
6      registro = temp;
7  }
```

Soluciones Hardware

Instrucciones de máquina especiales

Instrucción TEST&SET

```
1  const int n = N;
2  int cerrojo; // Variable compartida
3  void P(int i ){ // Proceso i-esimo
4        while (TRUE){
5             while(!(TS(cerrojo)); // Espera Activa, TS ==> Fc. Atómica HW
6             ----> SECCION CRITICA
7             cerrojo =0;
8           ---->RESTO DEL PROCESO
9        }
10 }
11 void main( ){
12      cerrojo=0; // Inicializa cerrojo
13      parbegin(P(1), P(2),...,P(N));// Lanzamiento de N Procesos Concurrentes
14 }
```

Instrucción intercambiar

```
1   const int n = N;
2   int cerrojo; // Variable compartida
3   void P(int i ){
4       int clavei; // Variable local
5       while (TRUE){
6           clavei=1; //Inicializa clavei
7           while(clavei) intercambiar(clavei,cerrojo); // Espera Activa
8           ----> SECCION CRITICA
9           intercambiar(clavei,cerrojo); //Fc. atómica HW
10          ---->RESTO DEL PROCESO
11      }
12  }
13  void main( ){
14      cerrojo=0; // Inicializa cerrojo
15      parbegin(P(1), P(2),...,P(N)); // Lanzamiento de N Procesos Concurrentes
16  }
```

Soluciones Hardware
Instrucciones de máquina especiales

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 (cada una con su propia variable).

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 Software

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

- 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)

Semáforos
Funcionalidad

- 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, si fuera posible (cota inferior).
  - La operación *up( )* /*signal( )* incrementa el valor del semáforo, si fuera posible (cota superior).
- Si la variable sólo puede tomar valores 0 y 1 el semáforo se denomina binario. En caso contrario, semáforo general o N–ario, valores de $0, \ldots, N$.

Semáforos
Funcionalidad

- **DOWN** *down( )/wait( )*:
  - Comprueba el valor del semáforo antes de realizar la operación:
    - Si semáforo > 0: decrementa el semáforo. Proceso continua su ejecución.
    - Si semáforo $\equiv 0$, el proceso se echa a dormir (estado bloqueado) hasta que pueda decrementarlo.
- **UP** *up( ) /signal( )*:
  - Si semáforo $> 0 \rightarrow$ Incrementa el valor del semáforo (sin superar la cota superior, por ejemplo semáforos binarios).
  - Si semáforo $\equiv 0$ y no hay procesos en la cola del semáforo $\rightarrow$ Incrementa el valor del semáforo.
  - Si semáforo $\equiv 0$ y hay procesos en la cola del semáforo $\rightarrow$ Despierta uno de los procesos bloqueados en este semáforo y termina su *down( )/wait( )* $\rightarrow$ la variable no cambia de valor si había algún proceso durmiendo en ese semáforo.

Semáforos
Exclusión mutua

El uso eficiente de los semáforos garantiza 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

```
7  while (1) {
8      down(mutex);
9      ---SECCION CRITICA ---
10     up(mutex);
11     ---RESTO DEL PROCESO ---
12 }
```

Semáforos
Productor–Consumidor

Compartido

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

Semáforos
Productor–Consumidor

Solución correcta

Productor

```
12  while (1) {
13      produce_item (itemp);
14      down(vacio); // decrementa entradas vacías
15      down (mutex); // entra en la región critica
16      *in = itemp; // introduce el elemento en el almacén
17      update_input(in);
18      up (mutex); // sale de la región critica
19      up (lleno); // incrementa entradas ocupadas
20  }
```

Consumidor

```
23  while (1) {
24      down(lleno); // decrementa entradas ocupadas
25      down(mutex);  // entra en la región crítica
26      itemc = *out; // lee el elemento del almacén
27      update_output(out);
28      up (mutex); // sale de la región critica
29      up (vacio); // incrementa entradas vacías
30      consume_item (itemp);
31  }
```

Semáforos
Productor–Consumidor

¿Qué hay erróneo en esta solución?

Productor

```
1   item itemp;
2   while (1) {
3       produce_item (itemp);
4       down (mutex); // entra en la región crítica
5       down(vacio); // decrementa entradas vacías
6       *in = itemp; // introduce el elemento en el almacén
7       update_input(in);
8       up (lleno); // incrementa entradas ocupadas
9       up (mutex); // sale de la región critica
10  }
```

Consumidor

```
12  item itemc;
13  while (1) {
14      down(mutex);  // entra en la región crítica
15      down(lleno); // decrementa entradas ocupadas
16      itemc = *out; // lee el elemento del almacén
17      update_output(out);
18      up (vacio); // incrementa entradas vacías
19      up (mutex); // sale de la región crítica
20      consume_item (itemp);
21  }
```

Semáforos
Productor–Consumidor

¿Qué hay erróneo en esta solución?

Productor

```
1   item itemp;
2   while (1) {
3       produce_item (itemp);
4       down (mutex); // entra en la región crítica
5       down(vacio); // decrementa entradas vacías
6       *in = itemp; // introduce el elemento en el almacén
7       update_input(in);
8       up (lleno); // incrementa entradas ocupadas
9       up (mutex); // sale de la región crítica
10  }
```

Consumidor

```
12  item itemc;
13  while (1) {
14      down(lleno); // decrementa entradas ocupadas
15      down(mutex);  // entra en la región crítica
16      itemc = *out; // lee el elemento del almacén
17      update_output(out);
18      up (vacio); // incrementa entradas vacías
19      up (mutex); // sale de la región crítica
20      consume_item (itemp);
21  }
```

Semáforos

Problema del baile de salón

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 sólos ni quedarse en la pista sólos. Sin embargo no tienen por qué salir con la pareja con la que entraron.

Una posible solución:

```
1  semaf mutex1=0, mutex2=0;
2  Hombre(){
3      while (TRUE){
4          up(mutex1);
5          down(mutex2);
6          Baila();
7      }
8  }
9  Mujer(){
10     while (TRUE){
11         down(mutex1);
12         up(mutex2);
13         Baila();
14     }
15 }
```

Semáforos

Problema del baile de salón

Otra posible solución:

Rendezvouz

```
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);
```

Semáforos

Problema de los Babuinos

En el parque nacional Kruger en Sudáfrica hay un cañón muy profundo con una simple cuerda para cruzarlo. Los babuinos cruzan ese cañón constantemente en ambos sentidos utilizando la cuerda. Sin embargo:

- Como los babuinos son muy agresivos, si dos de ellos se encuentran en cualquier punto de la cuerda yendo en sentido opuestos, se pelearán y terminarán cayendo por el cañón con un desenlace fatal.
- La cuerda no es muy resistente y aguanta 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

Solución 1: todo junto

Variables globales:

int cVa = 0, cVi = 0; // num de babuinos que van y num babuinos que vienen

semaf mutVa = 1, mutVi = 1; //mútex protegen contadores anteriores

semaf babVa = 5, babVi = 5; // hasta 5 pueden pasar una vez que entra el 1°

sem cuerda = 1; // o van hacia un lado o hacia el otro (solo hacia un lado)

Semáforos

Problema de los Babuinos

Solución 1: todo junto (cont)

Babuinos que van

```
1  BabuinoVa() {
2    down(babVa); //si hay 5, espero
3    down (mutVa); //Que voy!
4    ++ cVa;  //Incremento contador
5    if ( cVa == 1) //Si soy el primero
6      down (cuerda); //espero cuerda
           libre, para pasar
7    up (mutVa);
8    CruzaBabuino();
9    down (mutVa); //Ya he pasado!
10   --cVa;  //Decremento contador
11   if ( cVa == 0) //Si era el último...
12     up (cuerda); //libero cuerda
13   up (mutVa);
14   up(babVa);
```

Babuinos que vienen

```
1  BabuinoViene() {
2      down(babVi);
3      down (mutVi);
4      ++ cVi;
5      if ( cVi == 1)
6          down (cuerda);
7      up (mutVi);
8      CruzaBabuino();
9      down (mutVi);
10     --cVi;
11     if ( cVi == 0)
12         up (cuerda);
13     up (mutVi);
14     up(babVi);
15 }
```

Semáforos
Problema de los Babuinos

Solución 2 equivalente, utilizando 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

Solución 2 equivalente, utilizando Light Switch (cont.)

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 }
```

Semáforos
Problema del Barbero Dormilón

Semáforos
Problema del Barbero Dormilón

Aspectos a tener en cuenta

Control de acceso a los recursos compartidos

- Sala de espera, con un aforo limitado: 20.
- Sofá (o sillas de espera), con un número de plazas limitado: 4.
- Sillas de barbero, con un número de sillas limitado: 3.

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

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

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

Cliente en la silla asociada al barbero: cliente_listo

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

Corte acabado: terminado

- El barbero indica que ha terminado
- El cliente se puede levantar de la silla

Semáforos
Problema del Barbero Dormilón

Semáforos (continuación)

Silla de barbero libre: dejar_silla_b

- El cliente indica que ha dejado la silla
- El barbero avisa a otro cliente, que puede dejar su sofá y ocupar la silla

Aviso de pago: pago

- El cliente indica que paga
- El barbero cobra

Aviso de pago terminado: recibo

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

Semáforos
Problema del Barbero Dormilón

Solución 1
Compartido

```
1   typedef int semaforo;
2
3   semaforo max_capacidad=20; // capacidad del local
4   semaforo sofa=4; // sofá de espera
5   semaforo silla_barbero=3; // sillas de la barbería
6
7
8   semaforo cliente_listo=0; // clientes en espera de servicio
9   semaforo terminado=0;//barbero termina de cortar pelo
10  semaforo dejar_silla_b=0; // espera a que se levante el cliente antes de cobrarle
11
12  semaforo caja =1; // barberos usan caja de 1 en 1
13  semaforo pago=0; // coordina el pago
```

Solución 1 (cont)

Barbero

```
16  void barbero(void)
17  {
18      down (cliente_listo);
19      cortar_pelo();
20      up (terminado);
21      down (dejar_silla_b);
22      up (silla_barbero);
23      down (pago);
24      down (caja);
25      aceptar_pago();
26      up (caja);
27      up (recibo);
28    }
29  }
```

Cliente

```
31  void cliente(void)
32  {
33      down (max_capacidad);
34      entrar_tienda();
35      down (sofa);
36      sentarse_sofa();
37      down (silla_barbero);
38      levantarse_sofa();
39      up (sofa);
40      sentarse_silla_barbero();
41      up (cliente_listo);
42      down (terminado);
43      levantarse_silla_barbero();
44      up (dejar_silla_b);
45      pagar();
46      up (pago);
47      down (recibo);
48      salir tienda();
49      up (max_capacidad);
50  }
```

Problema: ¿Cómo se sabe a qué cliente avisa un barbero para que se levante? Todos esperan en el mismo semáforo ... ¡Identificar clientes!

Semáforos
Problema del Barbero Dormilón

Solución 2
Compartido

```
1  typedef int semaforo;
2  int contador = 0; // número de clientes en la tienda
3  Cola c; // cola en la que se insertan los ids de clientes
4  semaforo max_capacidad=20; // capacidad del local
5  semaforo sofa=4; // sofá de espera
6  semaforo silla_barbero=3; // sillas de la barbería
7  semaforo cobrando=1; // limita a 1 el acceso a la caja
8  semaforo mutex1=1; // controla el acceso a contador
9  semaforo mutex2=1; // controla el acceso al identificador 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 que se levante el cliente
       antes de cobrarle
12 semaforo pago=0; // coordina el pago
13 semaforo recibo=0; // coordina el recibo
```

Semáforos

Problema del Barbero Dormilón

Barbero

```
16  void barbero(void)
17  {
18    int cliente_b;
19    while (1) {
20      down (cliente_listo);
21      down (mutex2);
22      extraer_de_cola (c, cliente_b);
23      up (mutex2);
24      cortar_pelo();
25      up (terminado[cliente_b]);
26      down (dejar_silla_b);
27      up (silla_barbero);
28      down (pago);
29      down (cobrando);
30      aceptar_pago();
31      up (cobrando);
32      up (recibo);
33    }
34  }
```

Cliente

```
36  void cliente(void)
37  {
38      int num_cliente;
39      down (max_capacidad);
40      entrar_tienda();
41      down (mutex1);
42      contador++;
43      num_cliente=contador;
44      up (mutex1);
45      down (sofa);
46      sentarse_sofa();
47      down (silla_barbero);
48      levantarse_sofa();
49      up (sofa);
50      sentarse_silla_barbero();
51      down (mutex2);
52      insertar_en_cola (c, num_cliente);
53      up (cliente_listo);
54      up (mutex2);
55      down (terminado[numcliente]);
56      levantarse_silla_barbero();
57      up (dejar_silla_b);
58      pagar();
59      up (pago);
60      down (recibo);
61      salir tienda();
62      up (max_capacidad);
63  }extraer_de_cola
```

Semáforos
Lectores–Escritores

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.

Condiciones y semáforos

- Mientras escritor escribe, nadie lee. Exclusión mutua. Semáforo esem.
- Varios lectores pueden leeer a la vez. Cuando no hay ninguno leyendo, el 1º espera (en esem). Cuando hay al menos uno, los demás no esperan. Necesario contar cuántos hay. Contador + semáforo mutex para protegerlo.

Semáforos

Lectores–Escritores

Prioridad a los lectores (en cuanto uno lee, los demás también pasan y leen - los escritores esperan...)

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

```
6  void lector(void)
7  {
8    while (1) {
9      down (mutex);
10     contlect = contlect + 1;
11     if(contlect ==1) down (esem);
12     up (mutex);
13     lee_recurso();
14     down (mutex);
15     contlect = contlect -1;
16     if(contlect ==0) up (esem);
17     up (mutex);
18   }
19 }
```

Escritor

```
21 void escritor(void)
22 {
23   while (1) {
24     down (esem);
25     escribe_en_recurso();
26     up (esem);
27   }
28 }
```

Semáforos
Lectores–Escritores

Prioridad a los escritores

No se permite acceder a nuevo lector si un escritor declara su deseo de escribir.

- Inhibir lecturas mientras haya algún escritor que desee escribir. Semáforo lsem (cola de lectores != cola de escritores)
- Necesario contar cuántos escritores desean escribir: variable contesc. Necesario proteger acceso a dicha variable: semáforo mutex2.
- Si hay lectores esperando y escritor desea escribir, debe "colarse". Separar lect/escr. Cola adicional de lectores en espera.

Semáforos
Lectores–Escritores

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 semáforo lsem
7  semaforo esem=1; // controla el acceso de escritura
8  semaforo lsem=1; // controla el acceso de lectura
```

Semáforos
Lectores–Escritores

Prioridad a los escritores

Lector

```
10  void lector(void)
11  {
12    while (1) {
13      down (mutex3);
14      down (lsem);
15      down (mutex1);
16      contlect = contlect + 1;
17      if(contlect ==1) down (esem);
18      up (mutex1);
19      up (lsem);
20      up (mutex3);
21      lee_recurso();
22      down (mutex1);
23      contlect = contlect -1;
24      if(contlect ==0) up (esem);
25      up (mutex1);
26    }
27  }
```

Escritor

```
29  void escritura(void)
30  {
31    while (1) {
32      down (mutex2);
33      contesc = contesc + 1;
34      if(contesc ==1) down (lsem);
35      up (mutex2);
36      down(esem);
37      escribe_en_recurso();
38      up (esem);
39      down (mutex2);
40      contesc = contesc -1;
41      if(contesc ==0) up (lsem);
42      up (mutex2);
43    }
44  }
```

Semáforos
UNIX

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

Semáforos
UNIX

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

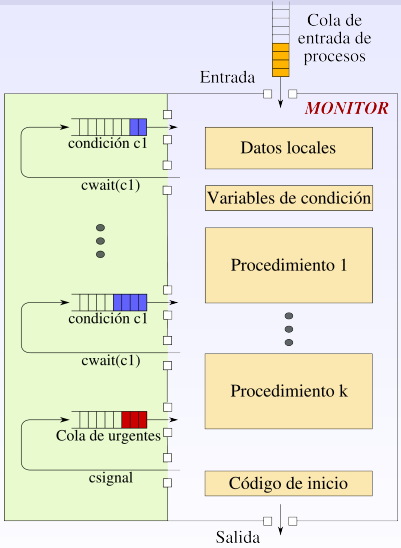
- Un monitor es un módulo de software que consta de uno o más procedimientos, una secuencia de inicialización y unos datos locales.
- Características básicas:
  - Las variables de datos locales están sólo accesibles para el monitor.
  - Un proceso entra en el monitor invocando a uno de sus procedimientos.
  - 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



Mensajes
Definición

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

Mensajes
Sincronización

- El emisor y el receptor pueden ser bloqueantes o no bloqueantes (esperando 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

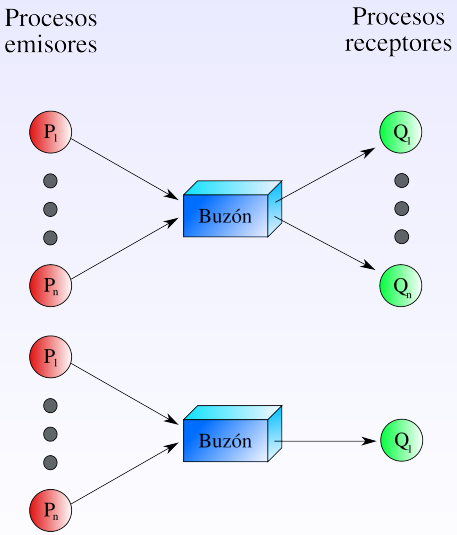
Direccionamiento directo

- 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.

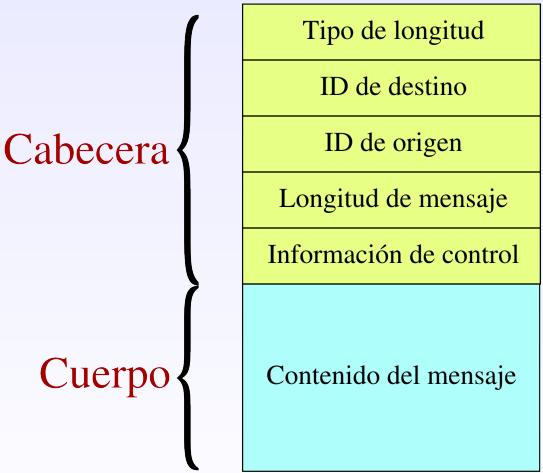
Direccionamiento indirecto

- 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.

Mensajes
Comunicación



Mensajes
Formato



Mensajes
Lectores–Escritores

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, msjl);
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 }
```

Mensajes
UNIX

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

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.

Mensajes
UNIX

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 del 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 recive una señal que debe capturar, por lo que el programa continúa como se le indique en la rutina correspondiente.

Mensajes
UNIX

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 */
}
```

Mensajes
UNIX

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.

Mensajes
UNIX

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 del 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.