

Concurrencia de Procesos Exclusión Mutua y Sincronización

Secciones Stallings:

5.1 – A1 - 5.2



Contenido

- Concurrencia de Procesos
- Exclusión Mutua
 - Espera ocupada
 - Semáforos



Concurrencia afecta a:

- SSOO actuales: multiprogramación, multiprocesadores.
- **Concurrencia afecta a:**
 - **Comunicación** entre procesos
 - **Compartición y competencia** por los recursos
 - **Sincronización** de la ejecución de varios procesos
 - Asignación del **tiempo de procesador** a los procesos

Concurrencia presente en:

- Múltiples aplicaciones:
 - Multiprogramación.
- Aplicaciones estructuradas:
 - Algunas aplicaciones pueden implementarse eficazmente como conjunto de procesos concurrentes.
- Estructura del S.O.:
 - Algunos SS.OO. están implementados como conjunto de procesos o hilos.

Términos clave

- Sincronización: Los procesos coordinan sus actividades
- Sección crítica: Región de código q sólo puede ser accedida por 1 proceso simultánea/ (variables compartidas).
- Exclusión mutua: Sólo 1 proceso 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. Resultado final depende de coordinación.
- Inanición: Proceso que está listo al que nunca se le asigna recurso (procesador u otro).



Dificultades con la concurrencia (I)

- Ejecución intercalada de procesos mejora rendimiento, pero...la **velocidad relativa de los procesos no puede predecirse**. Depende de:
 - Actividades de otros procesos
 - Forma de tratar interrupciones
 - Políticas de planificación
- => Surgen dificultades



Dificultades con la concurrencia (II)

- Dificultades (dadas por la imprevisión de la velocidad relativa de los procesos):
 - Compartir recursos
Ej: orden lecturas-escrituras
 - Difícil gestionar la asignación óptima de recursos. Ej: recursos asignados a proceso, proceso se suspende, ¿recurso bloqueado? => posible interbloqueo
 - Detección de errores de programación
(resultados no deterministas, no reproducibles)⁷

Motivación

<i>tiempo</i>	Persona A	Persona B
3:00	Mirar en la nevera. No queda leche	
3:05	Salir hacia la tienda	
3:10	Entrar a la tienda	Mirar en la nevera. No queda leche
3:15	Comprar leche	Salir hacia la tienda
3:20	Dejar la tienda	Entrar a la tienda.
3:25	Llegar a casa, guardar la leche	Comprar leche
3:30		Dejar la tienda
3:35		Llegar a casa, OH! OH!

Deseable: que alguien compre leche, ¡pero no todos!

Un ejemplo sencillo

```
void echo()  
{  
    ent = getchar();  
    sal = ent;  
    putchar(sal);  
}
```

Procedimiento almacenado en memoria,
compartido para todas las aplicaciones
=> Varias aplicaciones lo usan
ent y sal son variables compartidas

Ejemplo 1:

Posible secuencia

Proceso P1	Proceso P2
·	·
ent = getchar();	·
·	ent = getchar();
·	sal = ent;
sal = ent;	·
putchar(sal);	·
·	putchar(sal);
·	·

Esto puede ocurrir en sistemas mono o multiprocesador

Ejemplo 2

Proceso a

```
For (i=0; i<5; i++)  
  x=x+1;
```

Proceso b

```
For (j=0; j<5; j++)  
  x=x+1;
```

- se comparten todas las variables
- valor inicial x=0
- operación de incremento = 3 instrucciones atómicas:
 - LD ACC, # (Carga el contenido de una dirección en el ACC)
 - ACC++ (Incrementa acumulador)
 - SV ACC, # (Almacena valor del acumulador en una dirección)

Calcula todos los valores posibles de salida para la variable x

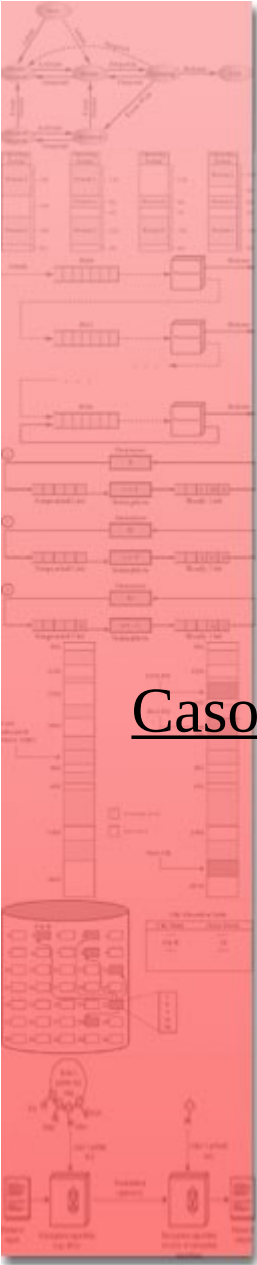
Soluciones

[illegible]

Solución: 2,3,4,5,
6,7,8,9,10

Caso mínimo:
2

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		
		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	



Caso máximo:
10

<div>máximo: 10</div>	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	
	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				

Condición de carrera: resultado final depende de orden de ejecución

Ejemplos (cont.)

Posible solución E1:

“Proteger” ejecución de *echo ()*: hasta que un proceso no termine, otro no puede ejecutar el mismo procedimiento => se bloquea y espera

Posible solución Ej2:

Ejecutar “atómicamente” las 3 instrucciones de ensamblador juntas


En general:

Controlar acceso a recursos compartidos



Labores del sistema operativo

- Seguir la **pista** de los distintos **procesos** activos
- **Asignar y retirar los recursos:**
 - Tiempo de procesador (planificación)
 - Memoria
 - Archivos
 - Dispositivos de E/S
- **Proteger** los **datos** y los **recursos** físicos de interacciones involuntarias de otros procesos
- Los **resultados** de un proceso deben ser **independientes** de la velocidad relativa a la que se ejecutan otros procesos concurrentes



Formas de interacción entre procesos

- Los procesos no tienen conocimiento de los demás
 - ⇒ Competencia
- Los procesos tienen un conocimiento indirecto de los otros (ej: no conocen PID pero sí comparten objetos) ⇒ Cooperación por compartimiento
- Los procesos tienen un conocimiento directo de los otros (hay primitivas de comunicación, conocen sus PIDs) ⇒ Cooperación por comunicación

Formas de interacción entre procesos


- **Los procesos no tienen conocimiento de los demás** \equiv **Competencia**
- Los procesos tienen un conocimiento indirecto de los otros (ej: comparten objetos)
 - Cooperación por compartimiento
- Los procesos tienen un conocimiento directo de los otros (hay primitivas de comunicación)
 - Cooperación por comunicación



Competencia entre procesos por los recursos

¿Por qué recursos compiten los procesos?

- Procesos en Tiempo Real: *CPU*
- Procesos que necesitan usar la pantalla o impresora: *E/S*
- Procesos que requieren mucha memoria: *RAM*
- Procesos que requieren utilizar la red: *Ancho de Banda*



Competencia por recursos - problemas de control

- Necesidad de exclusión mutua
- Interbloqueo (*deadlock*)
- Inanición

¿En qué consisten?

Competencia por recursos - problemas de control

- Necesidad de exclusión mutua
Recurso crítico con sección crítica
 - Sólo un programa puede acceder a su **sección crítica** en un momento dado.
 - Ej: sólo se permite que **un** proceso envíe **una** orden completa a la impresora en un momento dado.
- Interbloqueo (*deadlock*)
Ej: 2 procesos necesitan 2 recursos; se asigna 1 a cada 1; ambos esperan conseguir el otro (tmb con señales)
- Inanición (proceso no logra ejecutarse)
Ej: 3 procesos necesitan 1 recurso; se va asignando al 1 y al 2 intermitentemente; el 3 sufre inanición



Formas de interacción entre procesos

- Los procesos no tienen conocimiento de los demás \Rightarrow Competencia
- **Los procesos tienen un conocimiento indirecto de los otros (ej: comparten objetos)**
 - \Rightarrow **Cooperación por compartimiento**
- Los procesos tienen un conocimiento directo de los otros (hay primitivas de comunicación)
 - \Rightarrow Cooperación por comunicación

Cooperación entre procesos por compartimiento

- ¿En qué consiste?
- ¿Se puede garantizar la integridad de datos?
- ¿Posibles problemas?



Cooperación entre procesos por compartimiento

- Se comparten objetos (datos)
- Las operaciones de **escritura** deben ser **mutuamente excluyentes**.
- Las **secciones críticas** garantizan la **integridad** de los datos.
- Puede haber problemas con la **coherencia** de los datos
 - Aunque se respeten las secciones críticas

Incoherencia de los Datos

P1: P2:
 $a = a + 1;$ $b = b * 2;$
 $b = b + 1;$ $a = a * 2;$

$a = a + 1;$

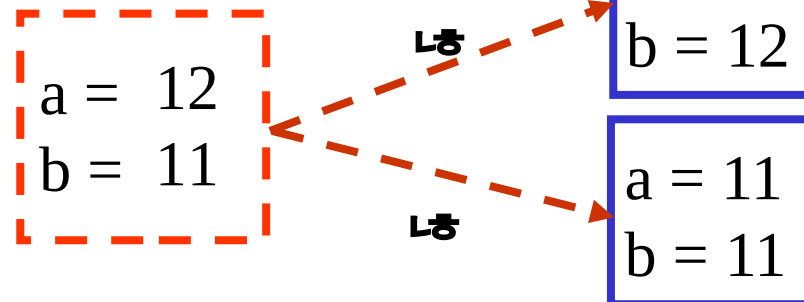
$b = b + 1;$

$b = b * 2;$

$a = a * 2;$

Los dos procesos trabajan con a y b sin intercalarse en medio de las 3 instrucciones de ensamblador

Interferencia



Condición de
competencia o carrera

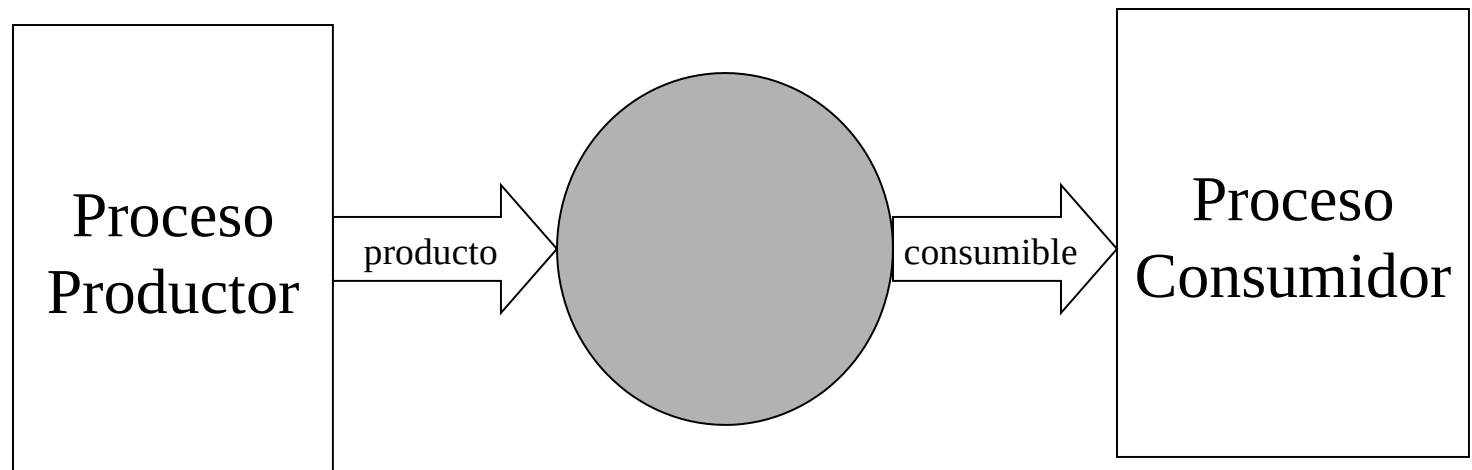


Cooperación entre procesos por compartimiento

```
while (cierto) {  
  
    /* código anterior*/  
  
    entrada_sección_crítica ( );  
  
    /* código sección crítica */  
  
    salida_sección_crítica ( );  
  
    /* resto proceso*/  
}
```

Ejemplo 3: Problema del Productor/Consumidor

- Productor: produce información para el consumidor.
- Concurrencia: uso de buffers y variables compartidas (**compartición de memoria**) ó **compartición de ficheros**



Problema del productor/consumidor

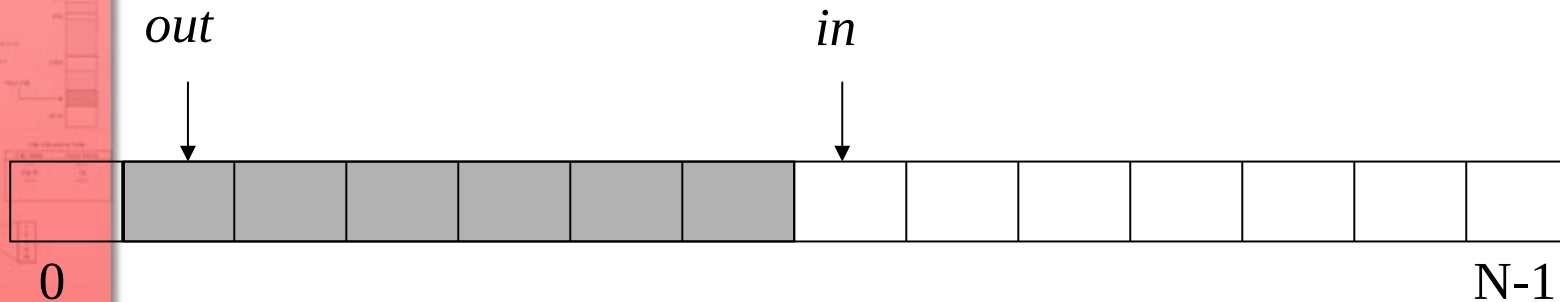
Definición del problema:

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

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 */
```



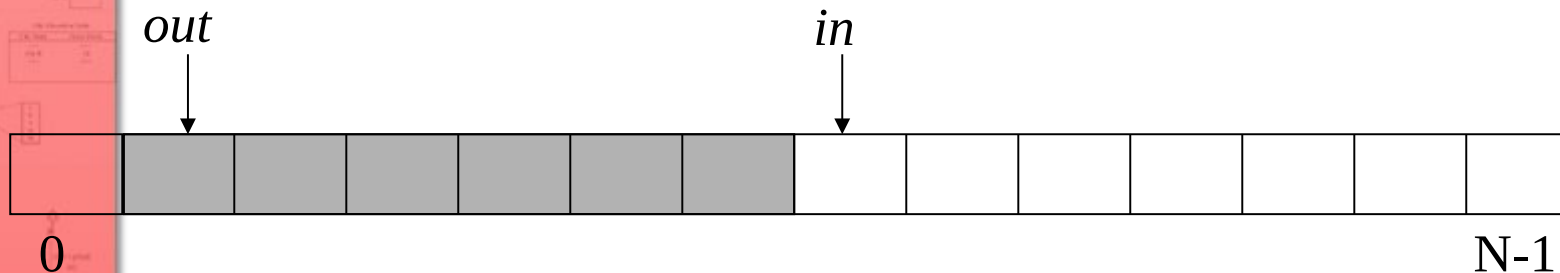
Cola vacía: $out == NULL$

Cola llena: $in == NULL$

Buffer de tamaño limitado

Productor

```
item itemp;  
...  
while (1) {  
    produce_item (itemp);  
    while (contador == N); /* no hace nada mientras la cola esté llena */  
    contador ++;  
    *in = itemp;  
    if(contador == 1) out = in; /* actualiza puntero de lectura de datos */  
    if(contador == N) in = NULL; /* actualiza puntero de entrada de datos */  
    else (++in) % N; /* % es el operador módulo */  
}
```



Cola vacía (contador==0): $out == NULL$

Cola llena (contador==N): $in == NULL$

Consumidor

```
item itemc;
```

```
...
```

```
while (1) {
```

```
    while (contador == 0);    /* no hace nada mientras la cola esté vacía */
```

```
    contador --;
```

```
    itemc = *out;
```

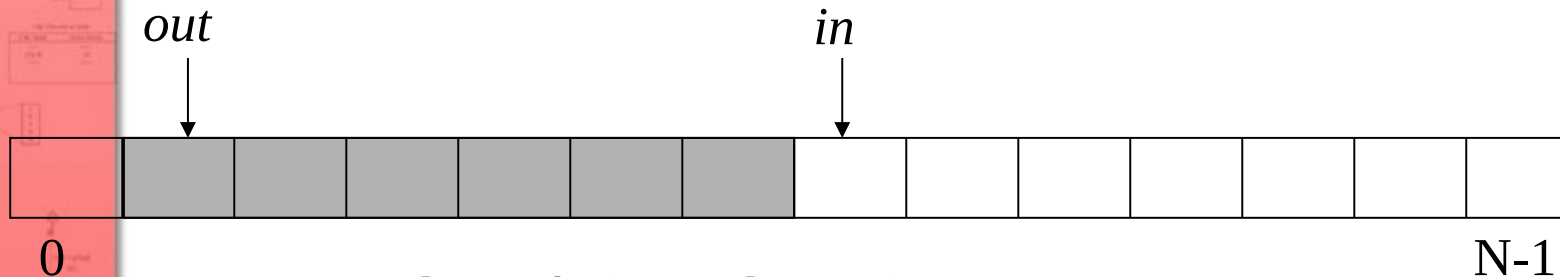
```
    if (contador == N-1) in = out; /* actualiza puntero de escritura de datos */
```

```
    if (contador == 0) out = NULL; /* actualiza puntero de lectura de datos */
```

```
    else (++out) % N;
```

```
    consume_item(itemc);
```

```
}
```



Cola vacía (contador==0): *out*==NULL

Cola llena (counter==N): *in*==NULL



Problemas

1. Coordinar lecturas y escrituras, para evitar lecturas sobre elementos no dispensados
2. La ejecución concurrente de contador++ y contador-- puede dar resultados variados

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

Productor

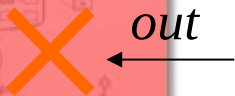
→ **produce_item** (&itemp);
→ **while** (contador==N);
→ *contador*++;
→ **in* = *itemp*;
→ **if**(*contador*==1) *out*=*in*;
→ **if**(*contador*==N) *in*=NULL;
→ **else** (++*in*) % N;

Consumidor

→ **while** (*contador* ==0);
→ *contador* - -;
→ *itemc* = **out*;
→ **if**(*contador* ==N-1) *in*=*out*;
→ **if**(*contador* ==0) *out*=NULL;
→ **else** (++*out*) % N;
→ **consume_item**(*itemc*);

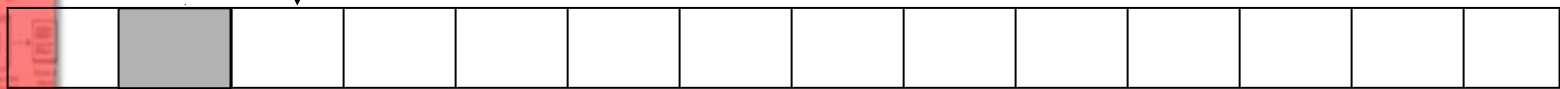


Contador = 0



out

in



0

N-1

Problema 2: la ejecución concurrente de **contador++** y **contador--** puede dar resultados variados

contador=contador+1

```
registro1=contador;  
registro1= registro1 +1;  
contador=registro1;
```

contador=contador-1

```
registro2=contador;  
registro2=registro2-1;  
contador=registro2;
```

Posible secuencia (contador=5)

T ₀ (productor):	registro ₁ = contador;	(registro ₁ =5)
T ₁ (productor):	registro ₁ =registro ₁ +1;	(registro ₁ =6)
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)
T ₅ (productor):	contador = registro ₁ ;	(contador =6)

Cambio
contexto

Cambio
contexto

Resultado: Condición de competencia: contador dice que hay 6 elementos en el array cuando sólo hay 5

Implementación del Productor/Consumidor (Memoria Compartida): Productor

```
#include <sys/types>
#include <sys/ipc.h>
#include <sys/shm>
#include <stdio.h>

#define SHMSZ 27

main{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key=5678;
    if((shmid = shmget(key, SHMSZ,
        IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

```
...

if((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat");
    exit(1);
}

s = shm;
// Escribe todo el buffer
for(c='a';c<='z';c++)
    *s++ = c;
//Espera que el consumidor termine de leer
while (*shm != '*')
    sleep(1);

exit(0);
}
```

Implementación del Productor/Consumidor (Memoria Compartida): Consumidor

```
#include <sys/types>
#include <sys/ipc.h>
#include <sys/shm>
#include <stdio.h>

#define SHMSZ 27

main{
    int shmid;
    key_t key;
    char *shm, *s;

    key=5678;
    if((shmid = shmget(key, SHMSZ,
        0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    ...
```

```
...

if((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat");
    exit(1);
}
// Va escribiendo caracteres del buffer
for(s = shm; *s !=NULL; s++)
    putchar(*s);
    putchar('\n');

*shm = '*';

    exit(0);
}
```



Formas de interacción entre procesos

- Los procesos no tienen conocimiento de los demás \Rightarrow Competencia
- Los procesos tienen un conocimiento indirecto de los otros (ej: comparten objetos)
 - \Rightarrow Cooperación por compartimiento
- Los procesos tienen un conocimiento directo de los otros (hay primitivas de comunicación)
 - \Rightarrow Cooperación por comunicación



Cooperación entre procesos por comunicación

- Paso de mensajes:
 - No es necesario control de la exclusión mutua (procesos no comparten datos).
- Puede producirse un interbloqueo:
 - Cada proceso puede estar esperando una comunicación del otro.
- Puede producirse inanición:
 - Dos procesos se están enviando mensajes mientras que un 3^{er} proceso está esperando recibir un mensaje.



Requisitos para la exclusión mutua (en general)

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.



Requisitos para la exclusión mutua (en general)

4. **No** se deben hacer **suposiciones** sobre la **velocidad relativa** de los procesos o el n° de procesadores.
5. Un proceso permanece en su sección crítica sólo por un **tiempo finito**.

Exclusión Mutua

- Soluciones para garantizarla:
 - Software con **Espera Activa**
 - Hardware:
 - Deshabilitar interrupciones
 - Instrucciones especiales de Hw
 - Con Soporte del SO o del lenguaje de programación (biblioteca):
 - Semáforos



Primer intento (Corrutinas)

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

Primer intento: Código

Compartido

```
int turno; /* con valores de 1 a N, siendo N  
           el número de procesos concurrentes */
```

Proceso i

```
while (turno != i); /* No hacer nada */  
--->SECCION CRÍTICA  
turno = (i + 1) % N;  
---->RESTO DEL PROCESO
```

Primer Intento: Ejemplo

Proceso 1

1

while (*turno*!=1); /* No hacer nada */

2

--->SECCION CRÍTICA

3

turno = (*i*+1) % *N*;

4

---->**RESTO DEL PROCESO**
(largo)

Proceso 2

5, 9

while (*turno*!=2); /* No hacer nada */

6

--->SECCION CRÍTICA

7

turno = (*i*+1) % *N*;

8

---->RESTO DEL PROCESO (corto)

turno = 1

- P2 quiere entrar en sección crítica, P1 no está => ¡debería poder!

No cumple: 4

Segundo intento

- 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

Segundo intento: Código

Compartido

```
#define FALSE 0
#define TRUE 1
#define N 2 /* Número de procesos */
int interesado[N] = {FALSE}; /* Todos los elementos inicializados a FALSE */
```

Proceso i

```
while (interesado[j] == TRUE); /* No hacer nada */
interesado[i] = TRUE;
--->SECCION CRÍTICA
interesado[i] = FALSE;
---->RESTO DEL PROCESO
```

Segundo Intento: Ejemplo

Proceso 1

```
while (interesado[2] ==  
TRUE);
```

```
/* No hacer nada */
```

```
interesado[1] == TRUE;
```

--->SECCION CRÍTICA

```
interesado[1] == FALSE;
```

---->RESTO DEL PROCESO

Proceso 2

```
while (interesado[1] ==  
TRUE);
```

```
/* No hacer nada */
```

```
interesado[2] == TRUE;
```

--->SECCION CRÍTICA

```
interesado[2] == FALSE;
```

---->RESTO DEL PROCESO

interesado[1] = TRUE
interesado[2] = TRUE

*Ambos en
Sección Crítica*

No cumple: 1, Exclusión Mutua

Tercer intento

- **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.

Tercer intento: Código

Compartido

```
#define FALSE 0
#define TRUE 1
#define N 2      /* Número de procesos */
int interesado[N] = {0}; /* Todos los elementos inicializados a FALSE */
```

Proceso i

```
interesado[i] = TRUE;
while (interesado[j] == TRUE); /* No hacer nada */
--->SECCION CRÍTICA
interesado[i] = FALSE;
---->RESTO DEL PROCESO
```

Tercer intento: Ejemplo

Proceso 1

`interesado[1] == TRUE;`

while (`interesado[2] == TRUE`);

`/* No hacer nada */`

--->SECCION CRÍTICA

`interesado[1] == FALSE;`

---->RESTO DEL PROCESO

Proceso 2

`interesado[2] == TRUE;`

while (`interesado[1] == TRUE`);

`/* No hacer nada */`

--->SECCION CRÍTICA

`interesado[2] == FALSE;`

---->RESTO DEL PROCESO

`interesado[1] = TRUE`
`interesado[2] = TRUE`

*Ambos esperan
por siempre*

No cumple: 3, Interbloqueo

Cuarto intento

- **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.

Cuarto intento: Código

Compartido

```
#define FALSE 0
#define TRUE 1
#define N 2 /* Número de procesos */
int interesado [N] = 0; /* Todos los elementos inicializados a FALSE */
```

Proceso *i*

```
interesado[i] = TRUE;
while (interesado[j] == TRUE) {
    interesado[i] = FALSE;
    --> ESPERA
    interesado[i] = TRUE;
}
---> SECCION CRÍTICA
interesado[i] = FALSE;
----> RESTO DEL PROCESO
```

Cuarto intento: Ejemplo

Proceso 1

```
interesado[1] == TRUE;  
while (interesado[2] == TRUE) {  
    interesado[1] == FALSE;  
    --> ESPERA  
    interesado[1] == TRUE;  
}
```

--->SECCION CRÍTICA

interesado[1] == FALSE;

---->RESTO DEL PROCESO

Proceso 2

```
interesado[2] == TRUE;  
while (interesado[1] == TRUE) {  
    interesado[2] == FALSE;  
    --> ESPERA  
    interesado[2] == TRUE;  
}
```

--->SECCION CRÍTICA

interesado[1] == FALSE;

---->RESTO DEL PROCESO

[interesado[1] = TRUE
 interesado[2] = TRUE]

*Ambos esperan
por siempre*

No cumple: 3 Bloqueo Vital (live lock)

Primera solución correcta

- 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**.
- Variables globales:
 - Interesado \Rightarrow indica la posición de cada proceso respecto a la exclusión mutua.

Código Algoritmo de Dekker

Compartido

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

```
while (1) {
    interesado[i] = TRUE;
    while (interesado[j] == TRUE) {
        if (turno == j) {
            interesado[i] = FALSE;
            while (turno == j); /* No hacer nada */
            interesado[i] = TRUE;
        }
    }
}
```

--->SECCION CRÍTICA

turno = j; /* cambia turno al otro proceso */

interesado[i] = FALSE;

---->RESTO DEL PROCESO

Proceso i entra en sección crítica cuando interesado[i]=TRUE e interesado[j] = FALSE => EXCLUSIÓN MUTUA

Proceso que quiere entrar en sección crítica no se retrasa indefinidamente:
turno=i, interesado[i] = FALSE
turno=j, interesado[j] = FALSE

}

Segunda solución correcta

- Es más **simple** y elegante
- Variables globales (igual que en Dekker):
 - interesado [i] \Rightarrow indica la posición de cada proceso respecto a la exclusión mutua.
 - turno \Rightarrow define quién tiene acceso a la sección crítica
- La inanición se previene porque **al turno se accede a través del proceso perdedor**

Código Algoritmo de Peterson

Compartido

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

Proceso *i*

```
while (1) {
    interesado[i] = TRUE;
    turno = j; /* cambia turno al otro proceso */
    while ((turno == j) && (interesado[j] == TRUE)); /* No hacer nada */
    ---->SECCION CRÍTICA
    interesado[i] = FALSE;
    ---->RESTO DEL PROCESO
}
```

Código Algoritmo de Peterson: Desglose por Procesos

Proceso 0

```
while (1) {  
    interesado[0] == TRUE;  
    turno = 1; /* cambia turno al otro proceso */  
    while ((turno == 1) && (interesado [1] == TRUE));  
    ---->SECCION CRÍTICA  
    interesado[0] == FALSE;  
    ---->RESTO DEL PROCESO  
}
```

Proceso 0 entra en sección crítica cuando interesado[0]=TRUE e interesado [1] = FALSE => EXCLUSIÓN MUTUA

```
while (1) {  
    interesado[1] == TRUE;  
    turno = 0; /* cambia turno al otro proceso */  
    while ((turno == 0) && (interesado [0] == TRUE));  
    ---->SECCION CRÍTICA  
    interesado[1] == FALSE;  
    ---->RESTO DEL PROCESO  
}
```

0 bloqueado =>
turno=1, interesado[1] = TRUE
=> 1 no bloqueado
=> NO INTERBLOQUEO


Código 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 }
```




Exclusión mutua: **soluciones por hardware**

- Inhabilitación de interrupciones
- Instrucciones especiales de máquina

Exclusión mutua: soluciones por hardware

- Un proceso se ejecuta hasta que solicita un servicio del SO ó es interrumpido.
- Inhabilitación de interrupciones para que proceso ejecute sin problemas su sección crítica:

```
while (1){  
    /* inhabilitar interrupciones */  
    /* sección crítica */  
    /* habilitar interrupciones */  
    /* resto */  
}
```



Exclusión mutua: soluciones por hardware

- Inhabilitación de interrupciones:
 - Ventaja: impedir que un proceso sea interrumpido => garantizar la exclusión mutua.
 - Desventaja: se limita la capacidad del procesador para intercalar programas.
- Multiprocesador:
 - Procesadores comparten acceso a la memoria
 - ≡ inhabilitar interrupciones en 1 procesador no garantiza la exclusión mutua.



Exclusión mutua: soluciones por hardware

- Instrucciones especiales de máquina:
 - Se ejecutan en un único ciclo de instrucción.
 - No están sujetas a intrusiones por parte de otras instrucciones:
 - El acceso a una posición de memoria excluye cualquier otro acceso a la misma posición
 - Ejs de instrucciones que realizan 2 acciones en 1 ciclo de instrucción:
 - Leer y escribir.
 - Leer y examinar.

Exclusión mutua: soluciones por hardware

- La instrucción Comparar y Fijar (*Test and Set*)

```
booleano TS (int i)
{
    if (i == 0) {
        i = 1;
        return cierto;
    }
    else {
        return falso;
    }
}
```


Exclusión mutua: soluciones por hardware

- Ej. de uso (*Test and Set*)

```
int cerrojo; /*var compartida*/ Proceso i  
while (1){  
    while (!TS (cerrojo)); /*espera activa*/  
    /* sección crítica*/  
    cerrojo = 0;  
    /* resto*/  
}
```

```
void main( ){  
    cerrojo=0; /*Inicializa cerrojo*/  
    parbegin(P(1), P(2),...,P(N));  
}
```

El único proceso que puede entrar en la sección crítica es el que encuentre *cerrojo* igual a cero

Exclusión mutua: soluciones por hardware

- La instrucción Intercambiar:
 - intercambia contenido de un registro con el de una posición de memoria
=> bloquea el acceso a la posición de memoria de cualquier otra instrucción que haga referencia a la misma posición

Exclusión mutua: soluciones por hardware

```
void intercambiar(int registro,  
                 int memoria)  
{  
    int temp;  
    temp = memoria;  
    memoria = registro;  
    registro = temp;  
}
```

Exclusión mutua: soluciones por hardware

- Ej. de uso (*intercambiar*)

```
int cerrojo; /*var compartida*/ Proceso i
while (1){
    clavei=1;
    while (clavei)
        intercambiar(cclavei,cerrojo);/*espera*/
    /*sección crítica*/
    intercambiar(cclavei,cerrojo);/*Fc atómica*/
    /*resto*/
}
```

```
void main( ){
    cerrojo=0; /*Inicializa cerrojo*/
    parbegin(P(1), P(2),...,P(N));
}
```

El proceso que puede entrar en la sección crítica es el que logra poner a 1 el cerrojo (y su clave=0)

Instrucciones de máquina y exclusión mutua

- Ventajas instrucciones especiales máquina:
 - Aplicable a cualquier número de procesos en sistemas con memoria compartida, tanto monoprocesador como multiprocesador
 - Memoria compartida, pero a nivel Hw se excluye acceso a posiciones de memoria involucradas en instrucciones especiales d Hw
 - Algoritmo simple y fácil de verificar.
 - Puede usarse para disponer de varias secciones críticas
 - Cada sección crítica con su propia variable

Soluciones por Hw y por Sw vistas

- Desventajas:
 - Interbloqueo:
 - Si P1, de baja prioridad, entra en sección crítica y llega otro proceso P2 con mayor prioridad, se da paso a P2, que no puede entrar en la sección crítica hasta que P1 salga! ⇨ interbloqueo
 - Espera activa ⇨ consume tiempo del procesador ejecutando bucles de espera

TEMA: Concurrency

- Concurrency of processes
 - Cómo afecta al diseño
 - Requisitos: labores del SSOO
 - Interacción entre procesos
 - Exclusión mutua, interbloqueo, espera vital, inanición
- Excl. mutua: soluciones por Sw (ej. Productor/consumidor)
 - Varios intentos fallidos
 - Algoritmo de Dekker
 - Algoritmo de Peterson
- Excl. mutua: soluciones por HW
 - Inhabilitación de interrupciones
 - Instrucciones especiales (programadas en Hw)
- Semáforos
 - Propiedades
 - Problemas

Semáforos

- ¿Qué es un semáforo?
 - A alto nivel (para qué sirve)
 - A bajo nivel (tipo de dato, primitivas)
- ¿Organización de procesos en espera?
 - Semáforos robustos vs. no robustos
- ¿Ventaja principal frente a algoritmos de Dekker o Peterson?
- ¿Qué garantizan (exclusión mutua, no inanición, no interbloqueo,...)?

Semáforos

- ¿Qué es un semáforo?
 - A alto nivel:
 - Coordinación de procesos
 - Detención
 - A bajo nivel
 - Qué es (tipo de dato)
 - Operaciones (primitivas de semáforo - atómicas)
- ¿Organización de procesos en espera?
 - Semáforos robustos (FIFO) vs. no robustos
- ¿Ventaja principal frente a algoritmos de Dekker o Peterson?
 - Evitan espera activa
- ¿Qué garantizan (exclusión mutua, no inanición, no interbloqueo)?
 - Exclusión mutua

Semáforos

- 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 *signal/up*
- Una señal se recibe mediante una operación atómica *wait/down*

Semáforos

- Un proceso en espera de recibir una señal (*wait*) es detenido hasta que tenga lugar la transmisión de la señal (*signal*).
- Los procesos en espera se organizan en una cola de procesos
- Dependiendo de la política de ordenación de procesos en espera:
 - Semáforos robustos: FIFO (garantizan la no inanición, fuerzan un orden – Ej: Linux)
 - Semáforos débiles: otra política (no garantizan la no inanición – ej: Mac OS X)

Semáforos

- Semáforo: variable con valor entero:
 - Puede iniciarse con un valor no negativo.
 - La operación *wait/down* disminuye el valor del semáforo, si se puede.
 - La operación *signal/up* incrementa el valor del semáforo.
- Si la variable sólo puede tomar valores 0 y 1 el semáforo se denomina **binario**
- Si no, se llama general o N-ario (0...N)

Semáforos

- **DOWN (*wait*):**

Comprueba valor del semáforo:

- Si > 0 : decrementa semáforo y proceso sigue ejecución
- Si $= 0$: el proceso se bloquea

- **UP (*signal*):**

- Si semáforo > 0 : incrementa el valor del semáforo
- Si semáforo $= 0$ y no hay procesos en la cola del semáforo: incrementa el valor del semáforo
- Si semáforo $= 0$ y había procesos bloqueados: despierta a uno de los bloqueados

Semáforos binarios

- DOWN:

- Si semáforo = 0, el proceso se bloquea
- Si semáforo = 1, pone semáforo = 0 y sigue ejecutando

- UP:

- Si hay proceso bloqueado en cola: desbloquea
- Si no, pone semáforo = 1

Semáforos - Propiedades

- Los semáforos garantizan la **exclusión mutua** en el acceso a secciones críticas:

```
#define FALSE 0
#define TRUE 1
#define N      /* Número de procesos */
typedef int  semaforo;
semaforo  mutex=Z; /* control de accesos a región crítica */
```

Compartido

```
while (1) {
    X (mutex);
    --->SECCION CRÍTICA
    Y (mutex);
    ---->RESTO DEL PROCESO
}
```

Proceso *i*

Semáforos - Propiedades

- Los semáforos garantizan la **exclusión mutua** en el acceso a secciones críticas:

```
#define FALSE 0
#define TRUE 1
#define N      /* Número de procesos */
typedef int  semaforo;
semaforo  mutex=1; /* control de accesos a región crítica */
```

Compartido

```
while (1) {
    wait (mutex);
    --->SECCION CRÍTICA
    signal (mutex);
    ---->RESTO DEL PROCESO
}
```

Proceso *i*

Solución del problema Consumidor/Proveedor con Semáforos – 1er intento

```
#define N 100
```

Compartido

```
typedef type item;
```

```
item array[N] = 0; /* array circular */
```

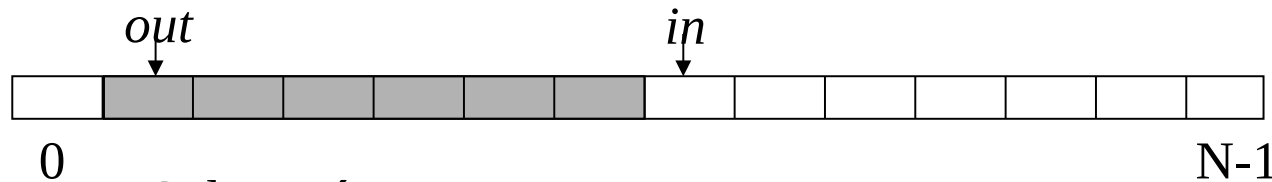
```
item *in = array; /* puntero a la siguiente posición libre */
```

```
item *out = NULL; /* puntero primer elemento ocupado */
```

```
typedef int semaforo;
```

```
semaforo mutex = 1; /* control de accesos a región crítica */
```

```
semaforo retraso = 0; /* para esperar si el buffer está vacío */
```



Cola vacía: $out == NULL$

Cola llena: $in == NULL$

Buffer de tamaño limitado

Solución del problema **Productor/Consumidor** con **Semáforos** – 1er intento

item *itemp*;

while (1) {

produce_item (*itemp*);

wait (*mutex*); /* entra en la región crítica */

**in* = *itemp*; /* introduce el elemento en el almacén */

 contador ++;

if (contador == 1)

signal (*retraso*); /*incrementa contador de entradas ocupadas*/

signal (*mutex*); /* sale de la región crítica */

}

Productor

item *itemp*;

wait(*retraso*); /* decrementa el contador de entradas ocupadas */

while (1) {

wait(*mutex*); /* entra en la región crítica */

itemc = **out*; /* lee el elemento del almacén */

 contador --;

signal (*mutex*); /* sale de la región crítica */

consume_item (*itemp*);

if (contador == 0)

wait (*retraso*); /* incrementa el contador de entradas vacías */

}

Consumidor

Solución del problema **Productor/Consumidor** con **Semáforos** – 1er intento - Problemas

```
item itemp;
```

```
while (1) {
```

```
    produce_item (itemp);
```

```
    wait (mutex); /* entra en la región crítica */
```

```
    *in = itemp; /* introduce el elemento en el almacén */
```

```
    contador ++;
```

```
    if (contador == 1)
```

```
        signal (retraso); /*incrementa contador de entradas ocupadas*/
```

```
    signal (mutex); /* sale de la región crítica */
```

```
}
```

Productor

```
item itemp;
```

```
wait (retraso); /* decrementa el contador de entradas ocupadas */
```

```
while (1) {
```

```
    wait(mutex); /* entra en la región crítica */
```

```
    itemc = *out; /* lee el elemento */
```

```
    contador --;
```

```
    signal (mutex); /* sale de la región crítica */
```

```
    consume_item (itemp);
```

```
    if (contador == 0)
```

```
        wait (retraso); /* incrementa el contador de entradas vacías */
```

```
}
```

Consumidor

Después de comprobar,
productor puede
incrementar contador.
¡Ya no es necesario que
consumidor espere!

Solución del problema **Productor/Consumidor** con **Semáforos** – 2º intento

```
item itemp;
```

Productor

```
while (1) {
```

```
    produce_item (itemp);
```

```
    wait (mutex); /* entra en la región crítica */
```

```
    *in = itemp; /* introduce el elemento en el almacén */
```

```
    contador ++;
```

```
    if (contador == 1)
```

```
        signal (retraso); /*incrementa contador de entradas ocupadas*/
```

```
    signal (mutex); /* sale de la región crítica */
```

```
}
```

```
item itemp;
```

Consumidor

```
wait (retraso); /* decreenta el contador de entradas ocupadas */
```

```
while (1) {
```

```
    wait(mutex); /* entra en la región crítica */
```

```
    itemc = *out; /* lee el elemento del almacén */
```

```
    contador --;
```

```
    if (contador ==0) /* Metemos contador dentro de sección crítica */
```

```
        wait (retraso);/* incrementa el contador de entradas vacías */
```

```
    signal (mutex); /* sale de la región crítica */
```

```
    consume_item (itemp);
```

```
}
```

Solución del problema **Productor/Consumidor** con **Semáforos** – 2º intento - Problemas

```
item itemp;
```

```
while (1) {
```

```
    produce_item (itemp);
```

```
    wait(mutex); /* entra en la región crítica */
```

```
    *in = itemp; /* introduce el elemento en el almacén */
```

```
    contador ++;
```

```
    if (contador == 1)
```

```
        signal (retraso); /* incrementa contador de entradas ocupadas */
```

```
    signal (mutex); /* sale de la región crítica */
```

```
}
```

Productor

```
item itemp;
```

```
wait(retraso); /* decrementa el contador de entradas ocupadas */
```

```
while (1) {
```

```
    wait(mutex); /* entra en la región crítica */
```

```
    itemc = *out; /* lee el elemento */
```

```
    contador --;
```

```
    if (contador == 0) // Lo mete
```

```
        wait(retraso); /* incrementa contador de salidas ocupadas */
```

```
    signal (mutex); /* sale de la región crítica */
```

```
    consume_item (itemp);
```

```
}
```

Consumidor

¡Interbloqueo!

**Productor espera a entrar en sección crítica
Consumidor está dentro, esperando
que haya elementos a consumir**

Solución del problema **Consumidor/Proveedor** con **Semáforos**

Compartido

```
#define N 100

typedef type item;
item array[N] = {0}; /* array circular */
item *in = array; /* puntero a la siguiente posición libre */
item *out = NULL; /* puntero primer elemento ocupado */

typedef int semaforo;

semaforo mutex = 1; /* control de accesos a región crítica */
semaforo vacio = N; /* cuenta entradas vacías en el almacén, se
                     inicializa a N, que es el tamaño del array */
semaforo lleno = 0; /* cuenta espacios ocupados en el almacén */
```

Solución del problema Consumidor/Proveedor con Semáforos

Productor

item *itemp*;

...

while (1) {

produce_item (*itemp*);

wait(*vacio*); /* decrementa el contador de entradas vacías */

wait (*mutex*); /* entra en la región crítica */

 añade_item (*itemp*);/* introduce el elemento en el almacén */

signal (*mutex*); /* sale de la región crítica */

signal (*lleno*); /* incrementa el contador de entradas
 ocupadas */

}

/* añade_item incluye operaciones sobre el array, *in, *out, contador, etc.*/

Solución del problema Consumidor/Proveedor con Semáforos

Consumidor

```
item itemc;
```

```
...
```

```
while (1) {
```

```
    wait(lleno); /* decrementa el contador de  
                entradas ocupadas */
```

```
    wait(mutex); /* entra en la región crítica */
```

```
    extrae_item (itemp); /* lee el elemento del almacén */
```

```
    signal (mutex); /* sale de la región crítica */
```

```
    signal (vacio); /* incrementa el contador de  
                   entradas vacías */
```

```
    consume_item (itemp);
```

```
}
```

```
/* extrae_item incluye operaciones sobre el array, *in, *out, contador,  
etc.*/
```


¿Solución correcta?

item *itemp*;

Productor

while (1) {

produce_item (*itemp*);

wait (*mutex*); /* entra en la región crítica */

wait(*vacío*); /* decrementa el contador de entradas vacías */

 añade_item (*itemp*);/* introduce el elemento en el almacén */

signal (*mutex*); /* sale de la región crítica */

signal (*lleno*); /* incrementa el contador de entradas ocupadas */

}

item *itemc*;

Consumidor

while (1) {

wait(*mutex*); /* entra en la región crítica */

wait(*lleno*); /* decrementa el contador de entradas ocupadas */

 extrae_item (*itemp*);/* lee el elemento del almacén */

signal (*mutex*); /* sale de la región crítica */

signal (*vacío*); /* incrementa el contador de entradas vacías */

consume_item (*itemp*);

}

¿Solución correcta?

Productor

```
item itemp;  
while (1) {  
    produce_item (itemp);  
    wait (mutex); /* entra en la región crítica */  
    wait(vacio); /* decrementa el contador de entradas ocupadas */  
    añade_item (itemp); /* introduce el elemento en el buffer */  
    signal (mutex); /* sale de la región crítica */  
    signal (lleno); /* incrementa el contador de entradas ocupadas */  
}
```

Espera entrar a región crítica

Consumidor

```
item itemc;  
while (1) {  
    wait(mutex); /* entra en la región crítica */  
    wait(lleno); /* decrementa el contador de entradas ocupadas */  
    extrae_item (itemp); /* lee el elemento del almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (vacio); /* incrementa el contador de entradas vacías */  
    consume_item (itemp);  
}
```

Entra con buffer vacío,
Espera a que haya algo

¿Queda algo erróneo?

Productor

```
item itemp;  
while (1) {  
    produce_item (itemp);  
    wait (mutex); /* entra en la región crítica */  
    wait(vacio); /* decrementa el contador de entradas vacías */  
    añade_item (itemp); /* introduce el elemento en el almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (lleno); /* incrementa el contador de entradas ocupadas */  
}
```

Consumidor

```
item itemc;  
while (1) {  
    wait(lleno); /* decrementa el contador de entradas ocupadas */  
    wait(mutex); /* entra en la región crítica */  
    extrae_item (itemp); /* lee el elemento del almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (vacio); /* incrementa el contador de e */  
    consume_item (itemp);  
}
```

**CAMBIO
DE ORDEN**

¿Queda algo erróneo?

Productor

```
item itemp;  
while (1) {  
    produce_item (itemp);  
    wait (mutex); /* entra en la región crítica */  
    wait(vacio); /* decrementa el contador de entradas vacías */  
    añade_item (itemp); /* introduce el elemento en el almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (lleno); /* incrementa el contador de entradas ocupadas */  
}
```

Consumidor

```
item itemc;  
while (1) {  
    wait(lleno); /* decrementa el contador de entradas ocupadas */  
    wait(mutex); /* entra en la región crítica */  
    extrae_item (itemp); /* lee el elemento del almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (vacio); /* incrementa el contador de e  
    consume_item (itemp);  
}
```

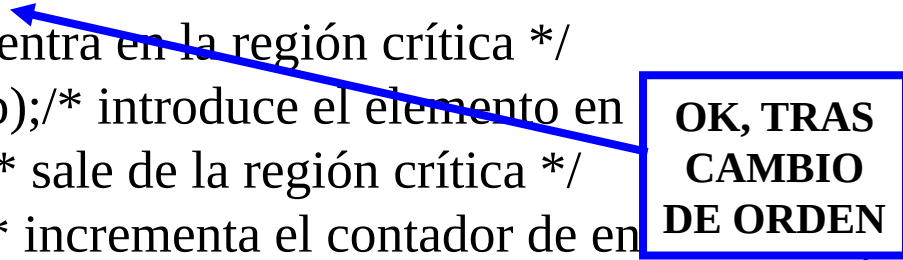
**Productor en región crítica,
espera hueco.**

Consumidor espera a entrar

¿Solución final?

Productor

```
item itemp;  
while (1) {  
    produce_item (itemp);  
    wait(vacio); /* decrementa el contador de entradas vacías */  
    wait (mutex); /* entra en la región crítica */  
    añade_item (itemp); /* introduce el elemento en el almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (lleno); /* incrementa el contador de entradas ocupadas */  
}
```



OK, TRAS CAMBIO DE ORDEN

Consumidor

```
item itemc;  
while (1) {  
    wait(lleno); /* decrementa el contador de entradas ocupadas */  
    wait(mutex); /* entra en la región crítica */  
    extrae_item (itemp); /* lee el elemento del almacén */  
    signal (mutex); /* sale de la región crítica */  
    signal (vacio); /* incrementa el contador de entradas vacías */  
    consume_item (itemp);  
}
```

Problema de la barbería

3 sillas

3 barberos

Sofá para 4 personas

Máximo nº personas: 20

Si no caben más, no entra cliente

Orden: sillas, sofá, de pie (FIFO)

Solo cobrar a 1 cliente cada vez

Barberos:

Cortar el pelo

Aceptar pago

Dormir esperando clientes

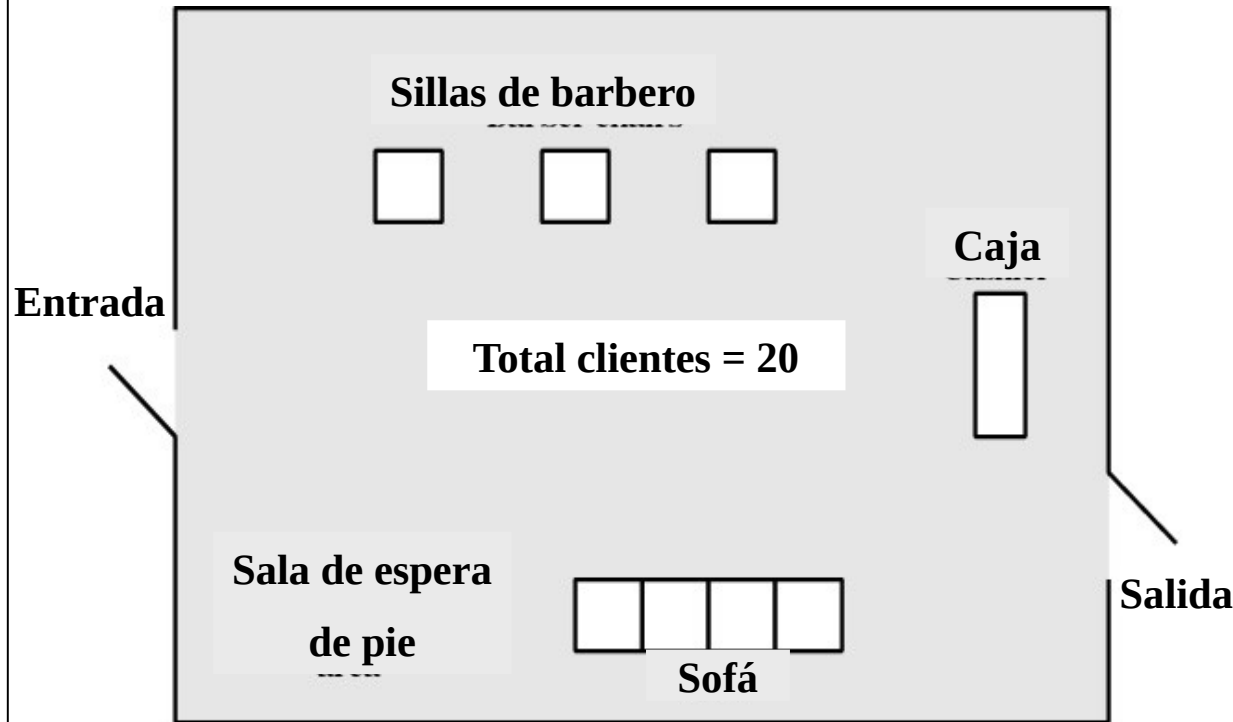


Figura 5.18. La barbería.

Problema de la barbería

¿Cuál es el problema?





Problema de la barbería

¿Cuál es el problema?

Procesos concurrentes

Manejar la ***concurrency*** de los procesos:

- Acceso a ***recursos compartidos***
- ***Sincronización*** de procesos



Problema de la barbería

Aspectos a tener en cuenta para manejar la concurrencia/sincronización de procesos en la barbería:

- Controlar acceso a **recursos compartidos**:
- **Sincronizar acciones** entre cliente y barbero:

Problema de la barbería

Aspectos a tener en cuenta para manejar la concurrencia/sincronización de procesos en la barbería:

- Controlar acceso a **recursos compartidos**:
 - Sala de espera
 - Sofá
 - Sillas
 - Caja
- **Sincronizar acciones** entre cliente y barbero:
 - Barbero espera cliente en silla - cliente se sienta
 - Cliente espera corte – barbero indica que ha terminado
 - Barbero espera q. cliente se levante – cliente indica q. se ha levantado – barbero da paso a otro cliente a la silla y mientras...
 - Barbero espera que cliente le pague – cliente paga
 - Cliente espera recibo – barbero se lo da

Funcionamiento

- Capacidad de la tienda max_capacidad
 - Si entra cliente, se decrementa
 - Si sale cliente, se incrementa
 - Si tienda llena, espera
- Capacidad del sofá sofa
 - Si se sienta cliente, se decrementa
 - Si se levanta cliente, se incrementa
 - Si sofá lleno, cliente espera de pie
- Capacidad de sillas silla_barbero
 - Si se sienta cliente, se decrementa
 - Si se levanta cliente, se incrementa
 - Si sillas llenas, cliente espera de pie
- Caja caja
 - Si barbero la usa, se decrementa
 - Si barbero la deja, se incrementa

Funcionamiento

- Cliente del sofá, espera silla silla_barbero
- Barbero indica silla vacía
- Cliente de pie, espera sofá sofa
- Cliente que se levanta del sofá, indica hueco en sofá
- Cliente que se sienta en silla, despierta al barbero cliente_listo
(si no, barbero siempre cortando el pelo)
- Barbero indica que ha terminado => que cliente se terminado
puede levantar
- Si no se levanta (pasa a ejecutarse otro proceso), no dejar_silla_b
puede sentarse otro => necesario otro semáforo para
parar al barbero (evitar que éste dé permiso a otro
para sentarse) hasta que el cliente anuncie que se ha
levantado (signal al barbero).

Funcionamiento

- Cliente paga pago
- Y espera recibo recibo
- Barbero espera poder usar la caja caja
- Barbero espera pago y envía recibo pago, recibo



Semáforos

Semáforo	Wait	Signal

Max_capacidad	Cliente espera entrar	Cliente al salir avisa a otro
Sofa	Cliente espera sentarse	Cliente que pasa a silla avisa
Silla_barbero	Cliente espera silla	Barbero avisa cuando hay libre
Caja	Barbero espera caja libre	Barbero avisa que deja caja libre
Cliente_listo	Barbero espera a que esté	Cliente avisa de que se ha sentado
Terminado	Cliente espera	Barbero indica que ha terminado
Dejar_silla_b	Barbero espera levante	Cliente avisa cuando se levanta
Pago	Barbero espera pago	Cliente avisa que ya ha pagado
Recibo	Cliente espera recibo	Barbero avisa que pago aceptado

Posible solución

3 barberos

Compartido

```
typedef int semaforo;
```

```
semaforo max_capacidad= ? ; /* capacidad del local */
```

```
semaforo sofa= ? ; /* sofá de espera */
```

```
semaforo silla_barbero= ? ; /* sillas de la barbería */
```

```
semaforo cliente_listo= ? ; /* clientes en espera de servicio */
```

```
semaforo terminado= ? ; /* barbero termina de cortar el pelo */
```

```
semaforo dejar_silla_b= ? ; /* evita colisiones en la caja, espera a que  
se levante el cliente antes de cobrarle */
```

```
semaforo caja= ? ; /* barberos usan caja de uno en uno */
```

```
semaforo pago= ? ; /* coordina el pago */
```

```
semaforo recibo= ? ; /* coordina el recibo */
```

Posible solución

3 barberos

Compartido

```
typedef int semaforo;
```

```
semaforo max_capacidad=20; /* capacidad del local */
```

```
semaforo sofa=4;          /* sofá de espera */
```

```
semaforo silla_barbero=3; /* sillas de la barbería */
```

```
semaforo cliente_listo=0; /* clientes en espera de servicio */
```

```
semaforo terminado=0;     /* barbero termina de cortar el pelo */
```

```
semaforo dejar_silla_b=0; /* espera a que se levante el cliente antes  
de cobrarle */
```

```
semaforo caja=1;          /* barberos usan caja de uno en uno */
```

```
semaforo pago=0;          /* coordina el pago */
```

```
semaforo recibo=0;        /* coordina el recibo */
```


Posible solución 3 barberos

```
void cliente( ) {
```

```
void barbero( ) {
```

```
}
```

Posible solución

3 barberos

```
void cliente( ) {
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);
```

```
}
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);
```

```
}
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);
```

```
}
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);
```

```
}
```

Posible solución

3 barberos


```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
}
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);
```

```
void barbero( ) {  
    while (1) {  
        t (cliente_listo);
```

```
}
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);
```


```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();
```

```
}
```


Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);
```



```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);
```

}

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
}
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();  
        signal (terminado);  
        → it (dejar_silla_b);  
    }
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);  
}
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);  
        → signal (silla_barbero);  
        wait (pago);  
    }
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);  
}
```

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);  
        signal (silla_barbero);  
        wait (pago);  
        → wait (caja);  
    }
```

Posible solución


3 barberos



```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);
```



}

```
void barbero( ) {  
    while (1) {  
        wait (cliente_listo);  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);  
        signal (silla_barbero);  
        wait (pago);  
        wait (caja);  
        ptar_pago();  
        signal (caja);  
        signal (recibo);
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);  
    salir_tienda();  
    signal (max_capacidad);  
}
```

```
void barbero( ) {  
    while (1) {  
        cliente_listo;  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);  
        signal (silla_barbero);  
        wait (pago);  
        wait (caja);  
        aceptar_pago();  
        signal (caja);  
        signal (recibo);  
    }  
}
```

Posible solución

3 barberos

```
void cliente( ) {  
    wait (max_capacidad);  
    entrar_tienda();  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();  
    signal (cliente_listo);  
    wait (terminado);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);  
    salir_tienda();  
    signal (max_capacidad);  
}
```

```
void barbero( ) {  
    while (1) {  
        cliente_listo;  
        cortar_pelo();  
        signal (terminado);  
        wait (dejar_silla_b);  
        signal (silla_barbero);  
        wait (pago);  
        wait (caja);  
        aceptar_pago();  
        signal (caja);  
        signal (recibo);  
    }  
}
```

Barbería equitativa

Problema:

- Clientes, mientras les cortan el pelo: wait (*terminado*)
- Barberos, al terminar de cortar: signal (*terminado*)
- Pero... ¿a cuál de los clientes que esperan les llega la señal?

Al primero que se sentó, que puede no corresponder con el barbero que ha terminado (con el cliente con el que se ha terminado más rápido)

- ¿Solución?

Barbería equitativa

Problema:

- Clientes, mientras les cortan el pelo: wait (*terminado*)
- Barberos, al terminar de cortar: signal (*terminado*)
- Pero... ¿a cuál de los clientes que esperan les llega la señal?

Al primero que se sentó, que puede no corresponder con el barbero que ha terminado (con el cliente con el que se ha terminado más rápido)

Solución:

- **Array de semáforos, 1/cliente**
- **Identificar a cada cliente mediante n° que coge al entrar (*contador*)**
- **Proteger acceso concurrente a *contador***
- **Asignación de clientes a barberos: *cola* de clientes en sillas (cliente pone su id, barbero lo toma)**

Barbería equitativa (v1)

```
typedef int semaforo;
```

Compartido

```
semaforo max_capacidad=20  /* capacidad del local */
```

```
semaforo sofa=4           /* sofá de espera */
```

```
semaforo silla_barbero=3   /* sillas de la barbería */
```

```
semaforo caja=1;          /* Limita a 1 el acceso a la caja */
```

```
int contador = 0;         /* número de clientes – turno de entrada */
```

```
Cola c;                   /* cola donde los clientes ponen su n°*/
```

```
semaforo mutex1=1;        /* controla el acceso a contador */
```

```
semaforo mutex2=1;        /* controla el acceso a cola de clientes */
```

```
semaforo cliente_listo=0; /* clientes en espera de servicio */
```

```
semaforo dejar_silla_b=0; /* evita colisiones en la caja, espera a que  
se levante el cliente antes de cobrarle */
```

```
semaforo terminado [50]= {0}; /* identifica el usuario servido */
```

```
semaforo pago=0;           /* coordina el pago */
```

```
semaforo recibo=0;         /* coordina el recibo */
```

Barbería equitativa (v1)

```
void cliente(void) {  
    int num_cliente;  
    wait (max_capacidad);  
    entrar_tienda();  
  
    wait (mutex1);  
    contador++;  
    num_cliente=contador;  
    signal (mutex1);  
  
    wait (sofa);  
    sentarse_sofa();  
    wait (silla_barbero);  
    levantarse_sofa();  
    signal (sofa);  
    sentarse_silla_barbero();
```

```
    wait (mutex2);  
    insertar_enCola (c, num_cliente);  
    signal (cliente_listo);  
    signal (mutex2);  
  
    wait (terminado[numcliente]);  
    levantarse_silla_barbero();  
    signal (dejar_silla_b);  
    pagar();  
    signal (pago);  
    wait (recibo);  
    salir_tienda();  
    signal (max_capacidad);  
}
```

Barbería equitativa (v1)

```
void barbero (void) {  
    int cliente_b;  
    while (1) {  
        wait (cliente_listo);  
        wait (mutex2);  
        extraer_de_cola (c, cliente_b);  
        signal (mutex2);  
        cortar_pelo();  
        signal (terminado[cliente_b]);  
        wait (dejar_silla_b);  
        signal (silla_barbero);  
        wait (pago);  
        wait (caja);  
        aceptar_pago();  
        signal (caja);  
        signal (recibo);  
    }  
}
```

Barbería equitativa (v2)

Otras consideraciones:

- Cliente deja silla: **signal** (*dejar_silla_b*)
Barbero estaba: **wait** (*dejar_silla_b*)

Si varios clientes se han levantado y hay varios barberos esperando, ¿se garantiza la correspondencia cliente-barbero?

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 ambas direcciones 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 mas de cinco babuinos en la cuerda, esta se romperá y los babuinos caerán también al vacío.



Babuinos (v1)

semef **babVa** = 5, **babVi** = 5 // *Hasta 5 pueden pasar una vez que entra el primero*
semef **cuerda** = 1 // *O van hacia un lado o hacia el otro (solo hacia un lado)*
int **cVa** = 0, **cVi** = 0 // *Nº de babuinos que van y nº babuinos que vienen*
semef **mutVa** = 1, **mutVi** = 1 // *Mútex protege contadores de los que van y los que vienen*

```
BabuinoVa ( ) {  
    down(babVa);  
    down (mutVa);  
        ++ cVa;  
    if ( cVa == 1)  
        down (cuerda);  
    up (mutVa);  
        CruzaBabuino();  
    down (mutVa);  
        --cVa;  
    if ( cVa == 0)  
        up (cuerda);  
    up (mutVa);  
    up (babVa);  
}
```

```
BabuinoViene ( ) {  
    down(babVi);  
    down (mutVi);  
        ++ cVi;  
    if ( cVi == 1)  
        down (cuerda);  
    up (mutVi);  
        CruzaBabuino();  
    down (mutVi);  
        --cVi;  
    if ( cVi == 0)  
        up (cuerda);  
    up (mutVi);  
    up(babVi);  
}
```

Babuinos (v2: funciones)

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 }
```

Variables compartidas

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  }
```


Babuinos (v2: funciones)

Variables compartidas

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 }
```

Bailes 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 solos ni quedarse en la pista solos.

Sin embargo, no tienen por qué salir con la pareja con la que entraron.



Bailes de salón

```
semaph mutex1=0, mutex2=0;
```

```
Hombre(){  
    while (TRUE){  
        up(mutex1);  
        down(mutex2);  
        Baila();  
    }  
}
```

```
Mujer(){  
    while (TRUE){  
        down(mutex1);  
        up(mutex2);  
        Baila();  
    }  
}
```

Bailes de salón

Otra posible solución:

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);
```



Problema de los 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.
- Nota: Los lectores no escriben y los escritores no leen nada
- Semejanza con catálogo de biblioteca

Problema de los lectores/escritores

- Elaboramos la solución:
 - Cond. 1: Mientras escritor escribe, nadie lee
=> ¿qué necesitamos? (semáforo de escribir)

Problema de los lectores/escritores

- Elaboramos la solución:
 - Cond. 1: Mientras escritor escribe, nadie lee
=> necesario respetar exclusión mutua
=> semáforo *esem* (semáforo de escribir)

Problema de los lectores/escritores

- Elaboramos la solución:
 - Cond. 1: Mientras escritor escribe, nadie lee
=> necesario respetar exclusión mutua
=> semáforo *esem* (semáforo de escribir)
 - Cond.2: Varios lectores pueden leer a la vez
 - cuando no hay ninguno leyendo ¿qué debe hacer el que desea leer?
 - cuando hay al menos uno, ¿necesitan esperar los demás?
- => ¿qué necesitamos?

Problema de los lectores/escritores

• Elaboramos la solución:

- Cond. 1: Mientras escritor escribe, nadie lee
=> necesario respetar exclusión mutua
=> semáforo *esem*
 - Cond.2: Varios lectores pueden leer a la vez:
 - cuando no hay ninguno leyendo, el **1º espera** en *esem*.
 - cuando hay al menos uno, los demás no necesitan esperar
- => necesario **contar cuántos lectores** hay
=> variable *contlect*
- => necesario **proteger acceso a dicha variable**
=> semáforo q. controla acceso simultáneo a *contlect*

Solución 1

```
typedef int semaforo;
```

```
int contlect = 0;      /* contador de lectores */
```

```
semaforo esem=1;      /* controla el acceso de escritura */
```

```
semaforo mutex=1;     /* controla el acceso a contlec */
```

Compartido

```
void lector () {  
    while (1) {  
        // Incrementar contador  
  
        // Acciones dependientes  
        // de si es el 1º o no =>  
        // comprobar si lo es  
        // Sí => espera en esem  
        // No=> lee directamente  
  
        // Decrementar contador  
        // Si no quedan lectores,  
        // dar paso a los escritores  
    }  
}
```

```
void escritor () {  
    while (1) {  
        // Esperar en esem  
        // Escribir  
        // Actualizar esem  
    }  
}
```

Solución 1

```
typedef int semaforo;
```

```
int contlect = 0;      /* contador de lectores */
```

```
semaforo esem=1;      /* controla el acceso de escritura */
```

```
semaforo mutex=1;     /* controla el acceso a contlec */
```

Compartido

```
void lector () {  
    while (1) {  
        wait (mutex);  
        contlect ++;  
        if(contlect ==1)  
            wait (esem);  
        signal (mutex);  
        lee_recurso();  
        wait (mutex);  
        contlect --;  
        if(contlect ==0)  
            signal (esem);  
        signal (mutex);  
    }  
}
```

```
void escritor () {  
    while (1) {  
        wait (esem);  
        escribe_en_recurso();  
        signal (esem);  
    }  
}
```

En este caso no importa que haya un wait dentro de una sección crítica, porque si se bloquea el 1er lector no es necesario que ningún otro lector entre en esa sección crítica (debe esperar a que el 1º

Solución 1

```
typedef int semaforo;
```

```
int contlect = 0;      /* contador de lectores */
```

```
semaforo esem=1;      /* controla el acceso de escritura */
```

```
semaforo mutex=1;     /* controla el acceso a contlec */
```

Compartido

```
void lector () {  
    while (1) {  
        wait (mutex);  
        contlect ++;  
        if(contlect ==1)  
            wait (esem);  
        signal (mutex);  
        lee_recurso();  
        wait (mutex);  
        contlect --;  
        if(contlect ==0)  
            signal (esem);  
        signal (mutex);  
    }  
}
```

```
void escritor () {  
    while (1) {  
        wait (esem);  
        escribe_en_recurso();  
        signal (esem);  
    }  
}
```

¿Alguno de los dos
procesos tiene
prioridad?

Solución 1

```
typedef int semaforo;
```

```
int contlect = 0; /* contador de lectores */
```

```
semaforo esem=1; /* controla el acceso de escritura */
```

```
semaforo mutex=1; /* controla el acceso a contlec */
```

Compartido

```
void lector () {  
    while (1) {  
        wait (mutex);  
        contlect ++;  
        if(contlect ==1)  
            → wait (esem);  
        signal (mutex);  
        lee_recurso();  
        wait (mutex);  
        contlect --;  
        if(contlect ==0)  
            → signal (esem);  
        signal (mutex);  
    }  
}
```

```
void escritor () {  
    while (1) {  
        → wait (esem);  
        escribe_en_recurso();  
        → signal (esem);  
    }  
}
```

¿Alguno de los dos
procesos tiene
prioridad?

Solución 2: prioridad para los escritores

- Modificación: no se permite acceder a nuevo lector si un escritor declara su deseo de escribir
 - ⇒ Inhibir lecturas mientras haya algún escritor que desee escribir
 - ⇒ ¿qué necesitamos?
 - ⇒ Necesario contar cuántos escritores que desean escribir:
 - ⇒ ¿qué necesitamos?
 - ⇒ Si hay lectores esperando y un escritor desea escribir, debe poder “colarse”
 - ⇒ ¿qué necesitamos?

Solución 2: prioridad para los escritores

- Modificación: 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**

Solución 2: prioridad para los escritores

Funcionamiento:

Lector:

- Se pone en cola de lectores (*wait (lsem)*)
- Incrementa contador de lectores
- Si es el primer lector, comprueba si hay escritores: espera, en cola para leer tras ellos (*wait (esem)*)
- (Si no es el primero o si no hay escritores) Envía señal al resto de lectores para que pasen a leer (*signal (lsem)*)
- Lee datos
- Decrementa contador de lectores
- Si es el último lector (de los que ya estaban leyendo): señal para que puedan entrar escritores (*signal (esem)*)

Solución 2: prioridad para los escritores

Funcionamiento:

Escritor:

- Incrementa contador de escritores
- Si es el primer escritor, comprueba si hay lectores leyendo (se pone en cola para leer tras ellos):
wait (lsem)
- (Si no quedaban lectores leyendo) Se pone en cola de escritores (*wait (esem)*)
- Escribe datos
- *signal (esem)*
- Decrementa contador de escritores
- Si es el último escritor: señal para que puedan entrar lectores (*signal (lsem)*)

Solución 2: prioridad para los escritores

- Problema:
 - Si van llegando lectores, se encolan en *lsem*.
 - Cuando llega el primer escritor, se pone detrás de todos los que están en *lsem*.
 - Cuando un lector consigue acceso a sección crítica, envía *signal (lsem)*
 - ⇒ entran primero los lectores que llegaron antes que el escritor
 - ⇒ ¡escritores no tienen prioridad!
- ¿Solución?

Solución 2: prioridad para los escritores

- Problema:
 - Si van llegando lectores, se encolan en *lsem*.
 - Cuando llega el primer escritor, se pone detrás de todos los que están en *lsem*.
 - Cuando un lector consigue acceso a sección crítica, envía *signal (lsem)*
 - ⇒ entran primero los lectores que llegaron antes que el escritor
 - ⇒ ¡escritores no tienen prioridad!
- ¿Solución?
 - Encolar lectores en cola anterior a *lsem* (*mutex3*), que esperen allí y pasen de 1 en 1 a *lsem*.
 - Cuando llega el primer escritor, se encola en *lsem*. (adelanta a los lectores que esperan en *mutex3*)

Solución 2: prioridad para los escritores

Compartido

```
typedef int semaforo;
```

```
int contlect = 0;      /* contador de lectores */
```

```
int contesc = 0;      /* contador de escritores */
```

```
semaforo mutex1=1; /* controla el acceso a contlec */
```

```
semaforo mutex2=1; /* controla el acceso a contesc */
```

```
semaforo mutex3=1; /* controla el acceso al semáforo lsem */
```

```
semaforo esem=1;      /* controla el acceso de escritura */
```

```
semaforo lsem=1;      /* controla el acceso de lectura */
```

Solución 2: prioridad para los escritores

```
void lector(void)
{
    while (1) {
        wait (mutex3); //Acceso a lsem
        wait (lsem);
        wait (mutex1);
        contlect ++;
        if(contlect ==1)
            wait (esem);
        signal (mutex1);
        signal (lsem);
        signal (mutex3);
        lee_recurso();
        wait (mutex1);
        contlect --;
        if(contlect ==0)
            signal (esem);
        signal (mutex1);
    }
}
```

```
void escritor(void)
{
    while (1) {
        wait (mutex2);
        contesc ++;
        if(contesc ==1)
            wait (lsem);
        signal (mutex2);
        wait(esem);
        escribe_en_recurso();
        signal (esem);
        wait (mutex2);
        contesc --;
        if(contesc ==0)
            signal (lsem);
        signal (mutex2);
    }
}
```

Solución 2: prioridad para los escritores

```
void lector(void)
{
    while (1) {
        wait (mutex3); //Acceso a lsem
        wait (lsem);
        wait (mutex1);
        contlect ++;
        if(contlect ==1)
            wait (esem);
        signal (mutex1);
        signal (lsem);
        signal (mutex3);
        lee_recurso();
        wait (mutex1);
        contlect --;
        if(contlect ==0)
            signal (esem);
        signal (mutex1);
    }
}
```

```
void escritor(void)
{
    while (1) {
        wait (mutex2);
        contesc ++;
        if(contesc ==1)
            wait (lsem);
        signal (mutex2);
        wait(esem);
        escribe_en_recurso();
        signal (esem);
        wait (mutex2);
        contesc --;
        if(contesc ==0)
            signal (lsem);
        signal (mutex2);
    }
}
```




Estado de colas

Sólo lectores	<ul style="list-style-type: none">- Activar <i>esem</i> (1º <i>wait</i>, último <i>signal</i>)- Sin colas (van pasando todos lectores, lecturas simultáneas)
Sólo escritores	<ul style="list-style-type: none">- Activar <i>esem</i> (para sección crítica) y <i>lsem</i> (1º <i>wait</i>, último <i>signal</i>)- Escritores en cola en <i>esem</i> (sólo escribe 1 cada vez)
Lectores y escritores con un lector antes	<ul style="list-style-type: none">- Lectores activan <i>esem</i> (1º <i>wait</i>, último <i>signal</i>)- Escritores activan <i>lsem</i> (1º <i>wait</i>, último <i>signal</i>)- Escritores en cola en <i>esem</i>- 1 lector en cola en <i>lsem</i>- Resto de lectores en cola en <i>mutex3</i>
Lectores y escritores con un escritor antes	<ul style="list-style-type: none">- Escritores activan <i>esem</i> (sección crítica)- Escritores activan <i>lsem</i> (1º <i>wait</i>, último <i>signal</i>)- Escritores en cola en <i>esem</i>- 1 lector en cola en <i>lsem</i>


Repaso tema: concurrencia

- ¿Requisitos para la exclusión mutua (en general)?



Requisitos para la exclusión mutua (en general)

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.



Requisitos para la exclusión mutua (en general)

4. **No** se deben hacer **suposiciones** sobre la **velocidad relativa** de los procesos o el n° de procesadores.
5. Un proceso permanece en su sección crítica sólo por un **tiempo finito**.

Repaso tema: concurrencia

- Formas de satisfacer los requisitos de la exclusión mutua:

Repaso tema: concurrencia

- Formas de satisfacer los requisitos de la exclusión mutua:
 - Dejar responsabilidad a los procesos, quienes se coordinan sin ayuda del S.O. => soluciones por Sw
 - Ej: algoritmo de Dekker, algoritmo de Peterson, Panadería de Lamport
 - Desventajas:
 - Propensas a errores
 - Fuerte carga de proceso
 - Espera activa
 - Soluciones por HW:
 - uso de instrucciones especiales (programadas en Hw)
 - Ventaja: reducen sobrecarga. Instrucciones atómicas.
 - Desventaja: hay espera activa
 - inhabilitación de interrupciones,
 - Ventaja: garantizan exclusión mutua (monoprocesador) y evitan espera activa
 - Desventajas: no aprovechan al máx. multiprogramación y no sirven para multiprocesadores
 - Dar soporte mediante semáforos
 - Ventajas: evitan espera activa, garantizan exclusión mutua (si robustos; si no, no garantizan la no inanición)