3

Sincronización y comunicación

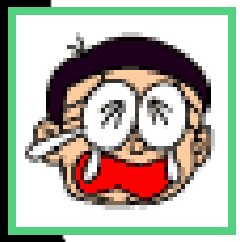
- El Problema de la Sección Crítica.
- Soluciones con:
  - Espera ocupada.
  - Soporte hardware a la sincronización
  - Soluciones con bloqueo: Semáforos.
- Paso de mensajes.

Procesos independientes

- Hasta ahora hemos hablado de **procesos** como entidades **independientes**:
  - No comparten estado (unos aislados de los otros)
  - Su ejecución es determinista = la ejecución del mismo programa con los mismos datos produce los mismos resultados.
  - Cada proceso puede avanzar a un ritmo arbitrario.

Procesos cooperantes

- En este tema hablaremos de procesos/hebras que **cooperan:**
  - Comparten estado, normalmente mediante variables compartidas entre los diferentes procesos/hebras.
  - Ejecución no determinista o difícil de reproducir, dado que esta sometida a *condiciones de carrera.*



Uso de procesos cooperantes

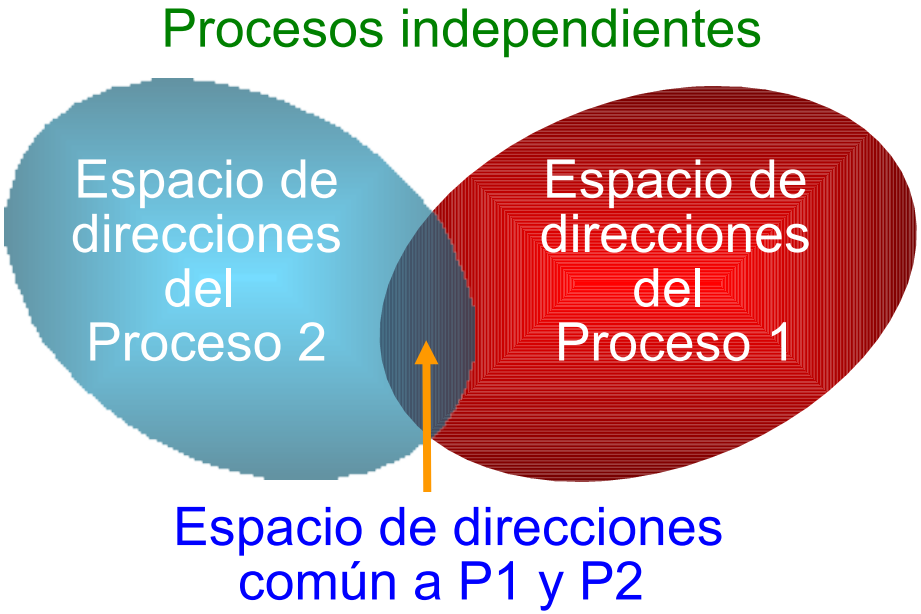
- Los procesos que cooperan se pueden utilizar para:
  - Ganar velocidad, solapando actividades o realizando trabajo en paralelo.
  - Compartir información entre trabajos.
  - Estructurar mejor una aplicación (recordar *Argumento de la simplicidad* visto en Tema 1).

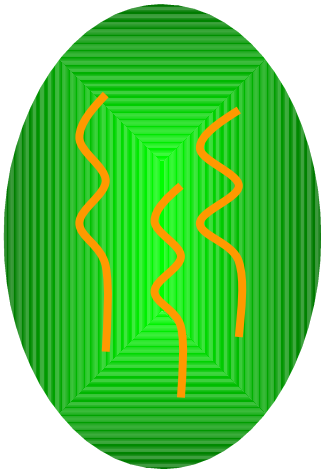
Concurrencia y paralelismo

- El paralelismo de una aplicación multi-procesador es el grado real de ejecución paralela alcanzado.
  - Está limitado por el nº de procesadores disponibles para la aplicación.
- La concurrencia de una aplicación es el máximo grado de paralelismo alcanzable con un "número ilimitado" de procesadores.
  - Depende de como está escrita la aplicación y del número de hebras de control que pueden ejecutarse simultáneamente.

Memoria compartida

- En el tema, nos moveremos en el paradigma de memoria compartida – dos ó más procesos comparten memoria.





Aplicación multihebrada

Cómo se puede compartir

- **Procesos independientes** – El SO suministra medios para compartir memoria:
  - Unix – *IPC shared memory* (**`shmget()`, ...**)
  - Windows y Unix – archivos proyectados en memoria.
- **Aplicaciones multihebradas** – las hebras de una aplicación comparten memoria de forma natural, pues comparten el mismo espacio de direcciones.

Condición de carrera

- Se produce una condición de carrera (*race condition*) cuando el resultado de la ejecución de dos o más procesos, que comparten variables comunes, depende de la velocidad relativa a la que cada proceso se ejecuta, es decir, del orden en el que se ejecutan las instrucciones.
- Para muestra, un botón ...

La carrera del 10

- Dos hebras ejecutan los código que se muestra abajo y comparten la variable `i=0` (inicialemente) ¿Cuál de ellas gana?

```
void incremento(int i) {
  while(i< 10) {
        i= i + 1;
  };
  printf("Hebra 1 ha
  ganado\n");
}
```

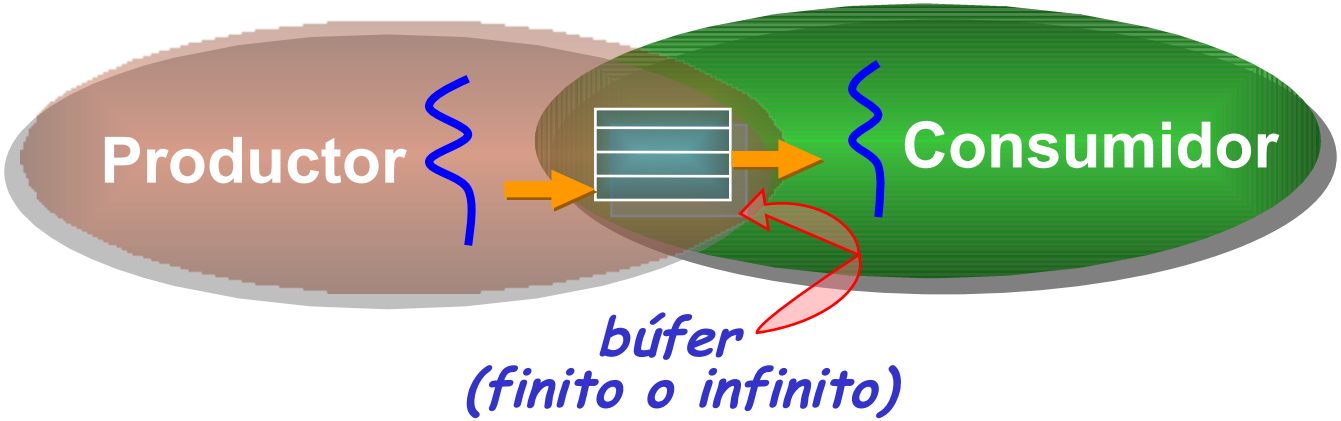
```
void decremento(int i) {
  while(i > -10) {
      i= i - 1;
  };
  printf("Hebra 2 ha
  ganado\n");
}
```

Código real del ejemplo

```
#include <windows.h>
#include <stdio.h>

volatile INT i=0; /* eliminar optimización compilador*/

void Incremento(void *) {

    while(i < 10) {
        i= i + 1;
         printf("Hebra 1 ha ganado\n");}

void Decremento(void *) {
        while (i > -10) {
                i= i- 1;
                printf("Hebra 2 ha gandado\n"); }

void main(void) {
  HANDLE Hebras[2];                    /*ejecución aquí*/

  Hebras[0]=CreateThread(0,0,Incremento,NULL,0, NULL);
  Hebras[1]=CreateThread(0,0,Decremento,NULL,0, NULL);
  WaitForMultipleObjects(2, handles, TRUE, INFINITE);
} /* El valor de la variable depende de la ejecución*/
```

El productor-consumidor

- Paradigma de los procesos cooperantes: el proceso *productor* genera información (bloque de disco, mensaje de red, caracteres de teclado, etc.) que es utilizada por el proceso *consumidor* (aplicación o SO)–o a la inversa.



Solución en memoria común

- Vamos a dar la solución al problema del productor-consumidor con búfer acotado.
- Datos compartidos:

```
int n;     /*tamaño del bufer */
struct item {...} ;
typedef bufer {0..n-1} item;
int ent, sal: 0..n-1;
ent = 0;   /*inicialmente*/
sal = 0;   /*inicialmente*/
```

Solución (cont.)

- Proceso productor

```
while (true) {
  ...
 produce itemp;
  ...
 while (ent+1 % n==
       sal) {/*nada*/};
 bufer[ent]=itemP;
 ent = ent+1 % n;
};
```

- Proceso consumidor

```
while (true) {
  while (ent == sal){/
  *nada*/};
  itemC=bufer[sal];
  sal = sal+1 % n;
  ...
  consume itemC;
  ...
};
```

Conclusión del ejemplo

- **La solución en memoria compartida del problema del búfer acotado es correcta pero sólo permite la existencia de `n-1` ítems en el búfer a la vez. ¿Por qué?**
- Para solventarlo, modificamos el código añadiendo una variable común, *contador*, inicializada a cero y que se incrementa cada vez que se añade un nuevo ítem al búfer, y se decrementa cada vez que retiramos un elemento.

2ª versión del productor-consumidor

- Datos compartidos:

```
int n;   /*tamaño del bufer */
struct item {...} ;
typedef bufer {0..n-1} item;
int ent, sal: 0..n-1;
ent = 0;        /*inicialmente*/
sal = 0;        /*inicialmente*/
cont: 0..n; /*lleva la cuenta del numero
                de items que hay en el
bufer*/
cont := 0;    /*inicialmente*/
```

Nuevo código

- Proceso productor

```
while (true) }
  ...
  produce itemP;
  ...
  while (cont == n)
       {/*nada*/;
  bufer[ent]=itemP;
  ent=ent+1 % n;
  cont= cont+1;
};
```

- Proceso consumidor

```
while (true) {
  while (cont = 0)
  {/nada*/};
  itemC=bufer[sal];
  sal=sal+1 % n;
  cont = cont-1;
  ...
  consume itemC;
  ...
};
```

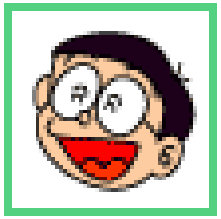
Intercalados y variables compartidas

- Operación n:=n+1 en seudoensamblador:
  - I_{11}: load M into R_1
  - I_{12}: add 1 to R_1
  - I_{13}: store R_1 in M
- Operación n:=n-1 en seudoensamblador:
  - I_{21}: load M into R_2
  - I_{22}: sub 1 to R_2
  - I_{23}: store R_2 in M

- Sea inicialmente n=5:
- El intercalado: $I_{11}, I_{12}, I_{13}, I_{21}, I_{22}, I_{23}$ daría el valor final **n=5**.
- Otro intercalado: $I_{11}, I_{21}, I_{12}, I_{13}, I_{22}, I_{23}$ daría finalmente **n=4**.
- Otro intercalado: $I_{21}, I_{21}, I_{22}, I_{23}, I_{12}, I_{13}$ daría finalmente **n=6**.
- **¡ Hay condición de carrera !**

Conclusión de la nueva versión

- **La corrección de la solución depende de que todas las declaraciones donde se manipulan variables compartidas se ejecuten de forma *atómica* (o indivisible). En este ejemplo, `cont==?, cont:=cont+1` y `cont:=cont-1`).**
- Conclusión: el acceso concurrente a los datos compartidos provoca inconsistencias, salvo que dispongamos de un mecanismo para asegurar la ejecución ordenada de los procesos cooperantes; esto es, que sólo permita los intercalados correctos.



Otro ejemplo

- Tendemos a pensar que solo las operaciones de ++ ó -- producen problemas, pero éstos se dan en cualquier manipulación de variables compartidas.
- Un fragmento de código erróneo que solemos usar en los primeros ejercicios es el siguiente:

```
var: variable compartida;
```

```
if (var > 0)
…
else
…;
```

Hebra_1

```
x= sin(5);
y = cos(3)
var = 3*x + 2*y;
…
```

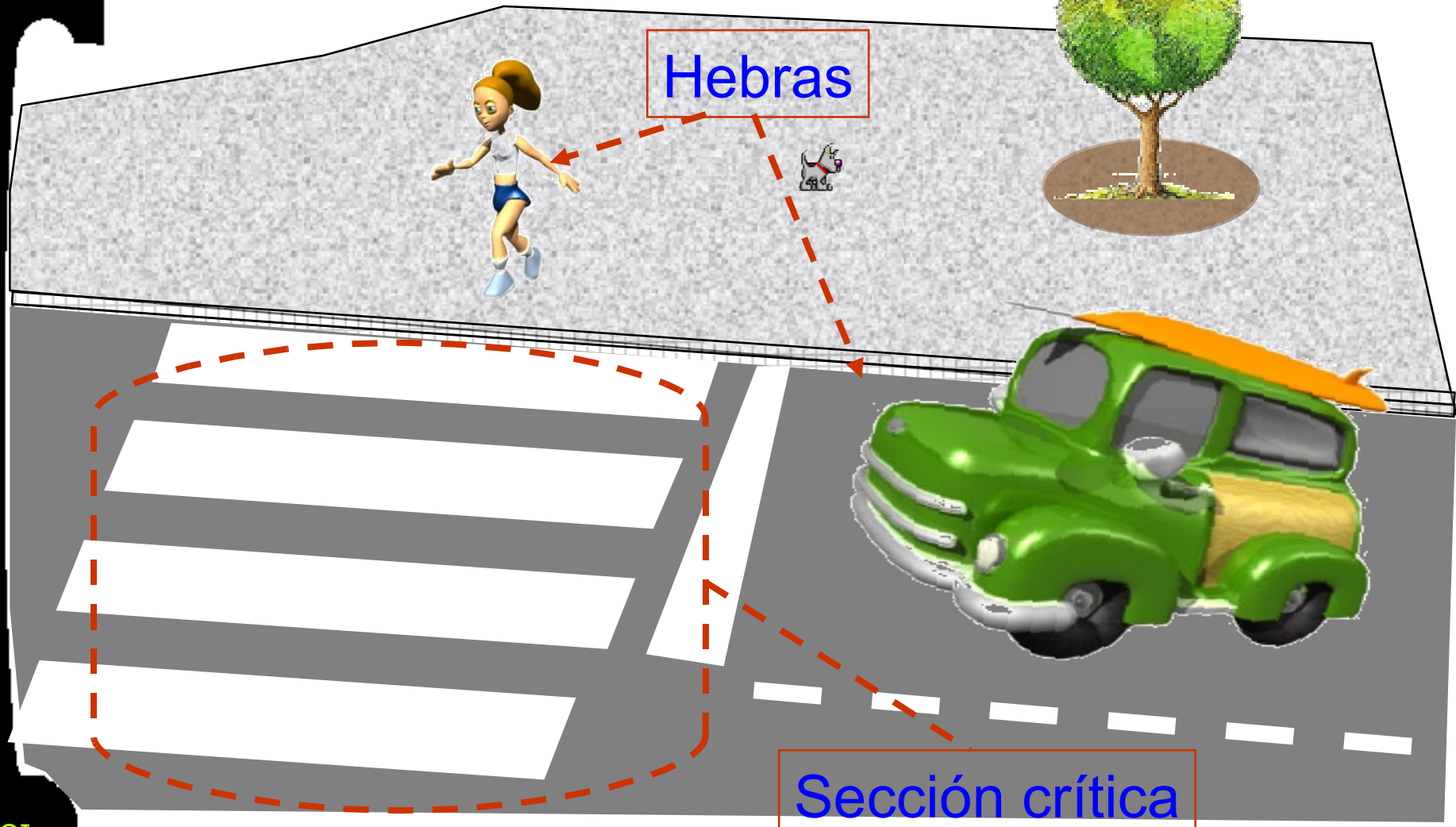
Hebra_2

- ¿Qué intercalados se pueden producir?

El problema de la Sección Crítica

- Sean *n* procesos compitiendo todos para usar algunos datos compartidos.
- Cada proceso tiene un segmento de código, denominado *Sección Crítica (SC)*, en el que accede a los datos compartidos.
- **Problema**: asegurar que cuando un proceso está ejecutando su SC, ningún otro proceso puede ejecutarse en la suya, es decir, asegurar que múltiples instrucciones, las SCs, parezcan una única instrucción.

Sección crítica



Estructura de la solución

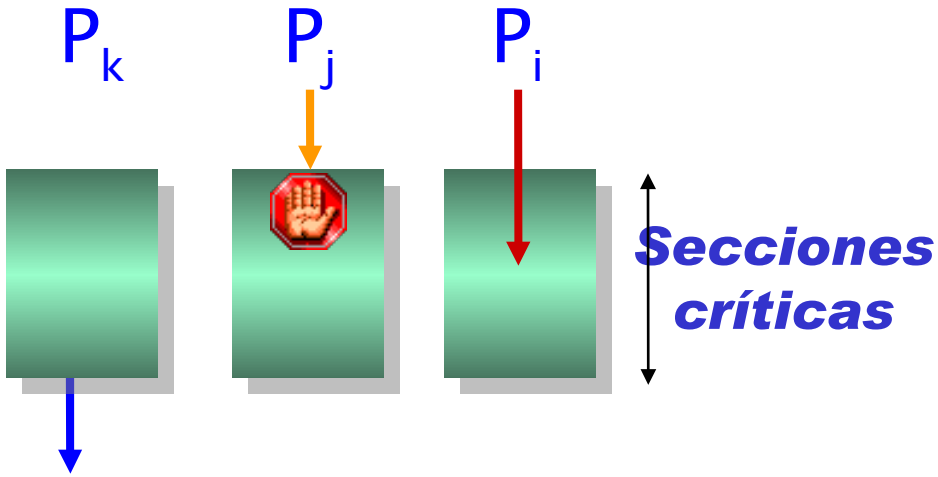
- Estructura de P_i:

```
while (true){
   entrada a sección

   sección crítica

   salida de sección

   resto de código
};
```

- Construiremos una solución que de la forma :
- Protocolo de entrada a SC – controla el acceso de los procesos a las SCs.
- Protocolo de salida de la SC – notifica salida de un proceso de su SC.
- Y cumpla las propiedades …

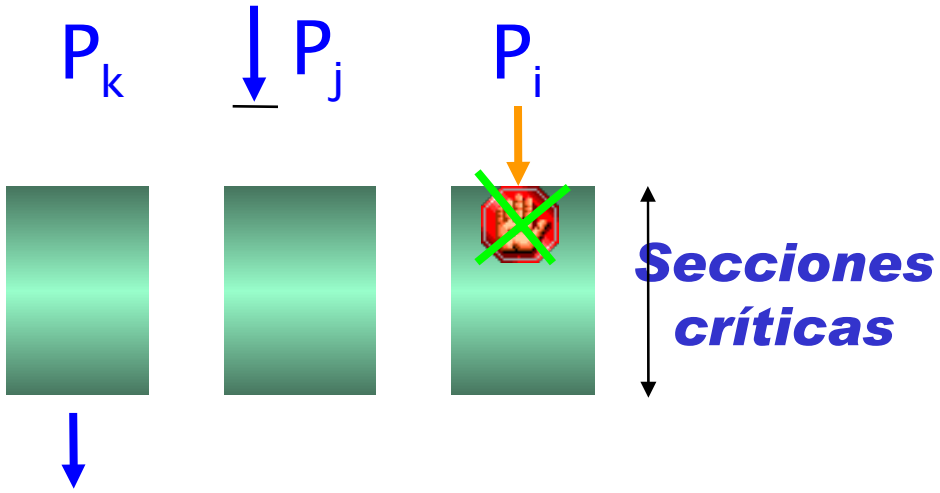
1º requisito de una solución al p.s.c.

- **Exclusión mutua** – Si P_i se está ejecutando en su SC, entonces ningún P_j (con $j \neq i$) puede ejecutarse en su SC.



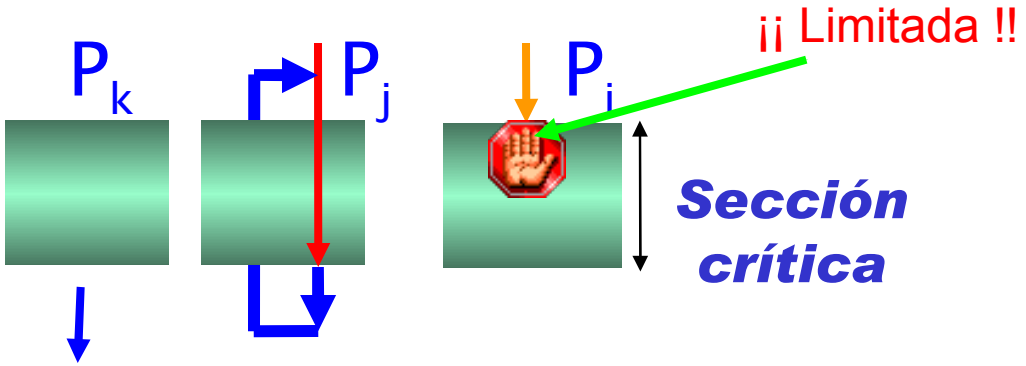
2º requisito de una solución al p.s.c.

- **Progreso** (o libre de interbloqueo) – Si no hay P_js en sus SCs y existe P_i que desea entrar en su SC, entonces la selección de P_i no debe posponerse indefinidamente.



3º requisito de una solución al p.s.c.

- **Espera limitada** (o libre inanición) – Debe existir un límite al número de veces que se les permite a los P_j's entrar en sus SC's después de que un P_i haya realizado un petición de entrar en la suya y antes de que petición se satisfaga.



Condiciones de entorno y propiedades

- Una solución correcta al problema de la SC:
  - No debe suponer nada a cerca de las velocidades relativas de los *n* procesos.
  - Supone que ningún proceso se ejecuta a velocidad cero.
- +Propiedades deseables de la solución:
  - Imparcialidad: no hacer esperar más a unos que a otros.
  - Eficiencia: no gastar más recursos de los necesarios.
  - ¡Ah sí!, y que sea lo más simple posible.

Soluciones para la Exclusión Mutua

- **Exclusión frente al hardware** – Deshabilitar las interrupciones para proteger nuestra SC de la rutina de servicio de interrupción.
- **Exclusión frente a otros procesos:**
  - **Espera ocupada**–los procesos comprueban constantemente si es seguro entrar a la SC
  - **Bloqueo** –los procesos se bloquean al intentar entrar en su SC si ya existe otro proceso dentro de la suya. Cuando el proceso deja la SC desbloquea a uno de los que esperan para entrar.

Protección frente al hardware

- Un proceso en su SC puede verse interrumpido por una RSI con la que comparte datos. Para proteger el acceso concurrente a las estructuras de datos del kernel por parte de las RSIs y el propio código del SO, debemos bloquear las interrupciones:

```
Deshabilitar_interrupciones();
   sección crítica;
Habilitar_interrupciones_previas();
```

Deshabilitar interrupciones

- Consideraciones:
  - Las interrupciones necesitan un servicio rápido $\Rightarrow$ no interferirlas demasiado: la SC anterior debe ser lo más breve posible.
  - Sólo bloquear las interrupciones cuyos manejadores acceden a la sección crítica.
- Esto no suele ser siempre viable en multiprocesadores: el coste de deshabilitar la interrupciones en todos los procesadores es muy elevado.

¿ Esperar o bloquear ?

- La respuesta depende de cuanto se va a tardar
  - Si la SC es corta, mejor esperar
  - Si la SC es larga, mejor bloquear
  - Duración de SC medida respecto al coste cambio de contexto y manejo de colas.
- La mayoría de los sistemas reales ofrecen ambos mecanismos, y se elige uno u otro en función del caso.

Exclusión mutua con espera ocupada

- Soluciones software: *Algoritmo de la Panadería* de L. Lamport (otros algoritmos: Dekker, Peterson, etc.). Estas soluciones no dependen de la exclusión mutua a nivel de acceso a memoria.
- Soluciones con apoyo hardware: el repertorio de instrucciones de la máquina suministra una operación del tipo *TestAndSet*.

Propuesta de solución

- Vamos a establecer diferentes soluciones al problema de la sección crítica con la estructura que anteriormente hemos indicado, es decir, vamos a construir los protocolos de entrada y de salida de la SC.
- Por el momento, y para simplificar, sólo vamos a suponer dos procesos, P_i y P_j.
- Estos procesos pueden compartir algunas variables comunes para sincronizar sus acciones.

Algoritmo 1

- Variables compartidas

  int turno:(0..1);

  turno := 0;
- turno = i ⇒ P_i puede entrar en su SC.
- Satisface la exclusión mutua pero no progresa.

P_i:

```
while (true) {
  while turno != i
        { /*nada*/};
     SeccionCritica;
  turno = j;
     RestoSeccion;
};
```

Algoritmo 2

- Variables compartidas

  int señal[2];

- señal[i] = true
  $\Rightarrow$ P_i preparado para entrar en su SC.
- Satisface la exclusión mutua pero no progresa.

P_i:

```
while (true) {
    señal[i]= true;
    while señal[j]
          {/*nada*/};
      SeccionCritica;
    señal[i]= false;
      RestoSeccion;
};
```

Problemas planteados

- Algoritmo 1 – el problema se debe a que un proceso no retiene suficiente información sobre el estado del otro proceso.
- Algoritmo 2 – la ejecución depende crucialmente de la exacta temporización de los dos procesos.
- Vamos a dar una solución que combina las dos variables compartidas de los algoritmos anteriores.

Algoritmo 3

- Variables compartidas:

```
int señal[2];
int turno=(0..1);
```

- Satisface los tres requisitos; resuelve el problema de la SC para dos procesos.

P_i:

```
while (true) {
   señal[i]= true;
   turno = j;
   while (señal[j]
       and turno=j)
       { /*nada*/};
     SeccionCritica;
   señal[i]= false;
     RestoSeccion;
};
```

Algoritmo de Lamport (para 2 procesos)

```
1 while (true) {
2  señal[0]=true;
3  turno=1;
4 while (señal[1]
  and turno=1)
  {/*nada*/};
5    SC;
6 señal[0]=false;
7    RS;
   };
```

```
a while (true) {
b señal[1]=true;
c turno=0;
d while (señal[0]
    and turno=0)
    {/*nada*/};
e    SC;
f señal[1]=false;
g    RS;
   };
```

Alg. Lamport: Código de la Hebra i-ésima

```
void CountThread(int i)
{
 int k, x, int j = 1 - i;
 senal[i] = true;
 turno = j;
 while ( senal[j] & (turno == j))
   cout << "Proceso "<<i<<"con i,j:"<<i<<j<<endl;
 for (k=0; k<250000; k++)
  {
    x=count;
    x++;
    count=x;
  }
  senal[i] = false;
}
```

Algoritmo de Lamport: Programa principal

```
volatile INT count;
bool senal[2];
int turno;
void main (void)
{
  handles[0]=CreateThread(0, 0, CountThread,
                    (void    *) 0, 0, NULL);
  handles[1]=CreateThread(0, 0, CountThread,
                    (void *) 1, 0, &threadID);
  WaitForMultipleObjects(2, handles,TRUE,INFINITE);

  cout << "El valor de Contador = " <<count<<endl;
}
/* Ejecución de Lamport.exe */
```

Algoritmo de la Panadería: n procesos

- Antes de entrar en su SC, el proceso recibe un número de tique. Entra en la SC el proceso con menor número de tique.
- Si P_i y P_j, con $i<j$, reciben el mismo número, entonces P_i es servido primero.
- El esquema de numeración siempre genera números en orden creciente de enumeración.

Algoritmo (cont.)

- Notación: $<$ ≡ orden lexicográfico (#tique, #pid)
  - `(a,b) < (c,d)` **si** `a<c` **ó si** `a=c y b<d`
  - max(a_o,...,a_{n-1}) es un número `k` tal que `k` ≥ a_i para `i=0,...,n-1`.
- Declaración de variables

```
var elec: array[0..n-1] of boolean;
    num: array[0..n-1] of integer;
```

- Valores iniciales

```
elec[i]:= false, i=0,1,...,n-1;
num[i]:= 0, i=0,1,...,n-1;
```

Código para P_i (cont.)

```
repeat
  elec[i]:=true;
  num[i]:=max(num[0],...,num[n-1])+1;
  elec[i]:=false;
  for j:=0 to n-1 do begin
    while elec[j] do no-op;
    while num[j]≠0 and
   (num[j],j)<(num[i],i) do no-op;
  end;

      SeccionCritica;
  num[i]:=0;
      RestoSeccion;
until false;
```

Sincronización hardware

- Se simplifica la solución con una función hardware que compruebe y modifique el contenido de una palabra de memoria atómicamente (el hardware garantiza la atomicidad de las dos asignaciones).

```
function TestAndSet(var objetivo:
         boolean):boolean;
    begin
          TestAndSet := objetivo;
          objetivo:= true;
    end;
```

Exclusión mutua con *TestAndSet*

- Datos compartidos:

```
int cerrojo: boolean
cerrojo := falso
```

- Proceso P_i:

```
while (true) {
    while TestAndSet(cerrojo) {/*nada*/};
          SeccionCritica;
    cerrojo = falso;
};
```

Cerrojo de espera ocupada

- Con **`TestAndSet`** podemos construir un mecanismo de sincronización denominado cerrojo de espera ocupada (*spin lock*).
- Cuando un proceso no puede entrar en su SC espera en un bucle.
- El problema del mecanismo es que la espera no esta acotada.
- En monoprocesadores, un proceso con un cerrojo de espera ocupada no puede ceder la CPU,¿por qué?

Spin lock

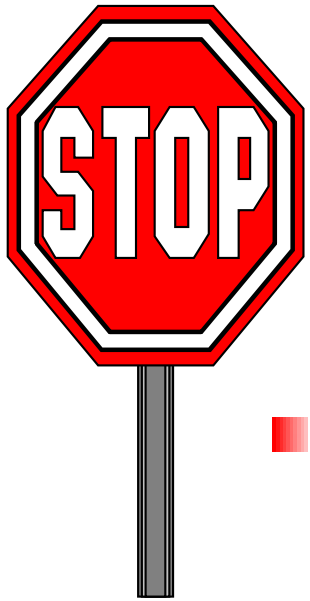
- Las primitivas para el cerrojo son:

```
void spin_lock (spinlock_t *s)  {
      while (test_and_set (s) != 0);
      }
void spin_unlock (spinlock_t *s) { *s =
 0 }
```

- Re-escribimos la solución de la SC como:

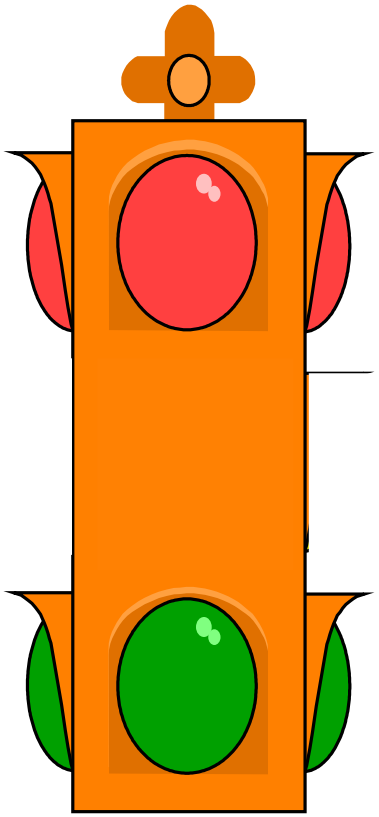
```
spinlock_t s;
spin_lock (&s);
sección crítica;
spin_unlock (&s);
```

Soluciones con bloqueo



- Existen multitud de primitivas de sincronización que implican algún tipo de bloqueo, como son:
  - Regiones críticas
  - Monitores
  - Transacciones atómicas, etc.
- Nosotros estudiaremos con detalle:
  - Semáforos
  - Paso de mensajes

Semáforos



- Definición e implementación
- Solución a problema de la exclusión mutua
- Problemas clásicos de sincronización y sus soluciones con semáforos.

Semáforos

- Semáforo =variable entera accesible sólo a través de las operaciones atómicas:
  - *wait(S)*:
```
s := s - 1;
if s<0 then suspender(s);
```
  - *signal(S)*:
```
s := s + 1;
if s≤0 then reanudar(S);
```
  - *crear-inicializar(S)*: crea semáforo con valor inicial S (S ≥ 0).
  - *destruir(S)*: destruye el semáforo S.

Semáforos (cont.)

- *suspende(S)* –operación del SO que bloquea al proceso que la invoca. El proceso se encola en una cola asociada al semáforo S.
- *reanuda(S)* –reanuda la ejecución de un único proceso que invocó a *suspende(S).*
- Observaciones:
  - **¡ La única forma de manipular la variable semáforo es a través de las operaciones `wait` y `signal` !**
  - **¡ Sólo podemos asignar el valor de S (positivo) en su creación !**

Sección crítica para n procesos

- Código del proceso P_i:

```
semaforo_t mutex=1;
/*Asignar valor solo en declaración*/
while (true) {
    wait(mutex);
        SeccionCritica;
    signal(mutex);
        RestoSeccion;
};
```

Ejemplo de contador: *main*

```
void main(void) {
  HANDLE handles[2], semaforo;
  DWORD threadID;

  semaforo = CreateSemaphore(0,1,1,0);

  handles[0]=CreateThread(0, 0, CountThread,
       (void *) 250000, 0, &threadID);
  handles[1]=CreateThread(0, 0, CountThread,
       (void *) 250000, 0, &threadID);
  WaitForMultipleObjects(2, handles, TRUE, INFINITE);
  cout << "El valor de Contador = " << count ;

  CloseHandle(semaforo);
}
```

Valor inicial

Máx. valor

Ejemplo de contador: *hebra*

```
void CountThread(int i) {
 int k, x;
 LONG ContadorSemaforo;

 WaitForSingleObject(semaforo, INFINITE);

 for (k=0; k<i; k++) {
    x=count;
    x++;
    count=x; }

 ReleaseSemaphore(semaforo, 1, &ContadorSemaforo);

}      /* Una ejecución del mismo aquí */
```

Incremento de S

Devuelve valor S

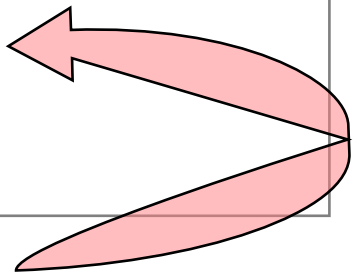
Implementación de *wait* y *signal*

- *wait* y *signal* deben ser atómicas.
  - Monoprocesadores, habilitar/ deshabilitar las interrupciones al principio/fin del código que implementa las operaciones.
  - Multiprocesadores:
    - Sin apoyo hardware –solución software al problema de la SC (SCs=wait y signal)
    - Con apoyo hardware – `TestAndSet` o similar.

Ejemplo: implementación de `wait` con `TestAndSet`

- Implementación de wait utilizando TestAndSet:

```
int cerrojo = false;
while TestAndSet(cerrojo) {/*nada*/};
    S=S-1;
    if (S < 0) {
          cerrojo = false;
          bloquea(S);
    } else
          cerrojo = falso;
```



- ¡¡Existe condición de carrera!!

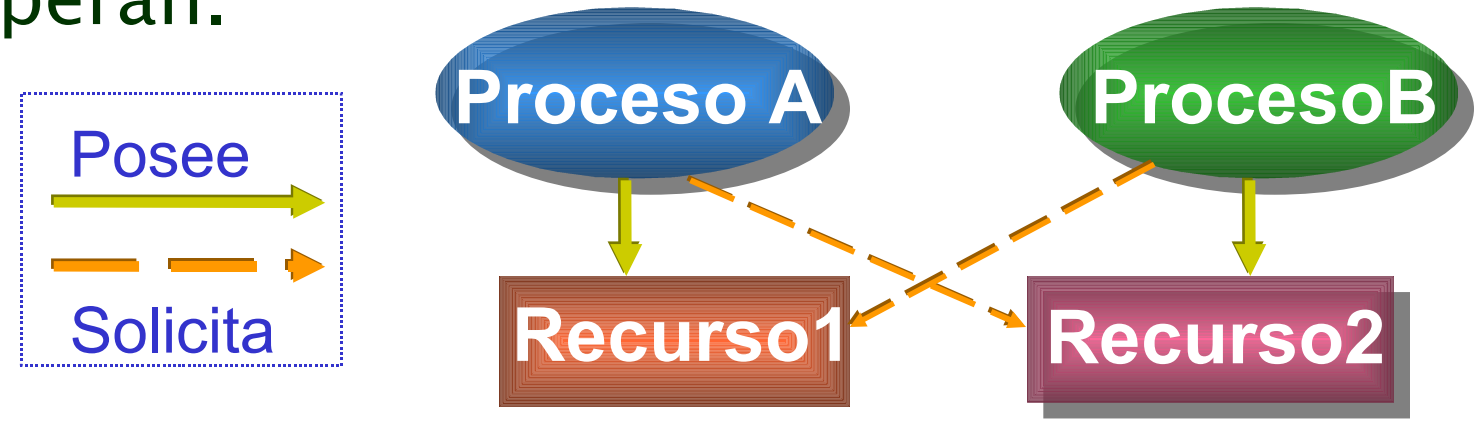
Sincronización con semáforos

- Los semáforos pueden usarse como herramienta general de sincronización: si queremos ejecutar `SeccionB` en P_j sólo después de que se ejecute `SeccionA` en P_i necesitamos un semáforo, `bandera=0` :

Código de P_i	Código de P_j
...	...
1º `SeccionA`	`wait(bandera)` 2º
`signal(bandera)`	`SeccionB`

Interbloqueo

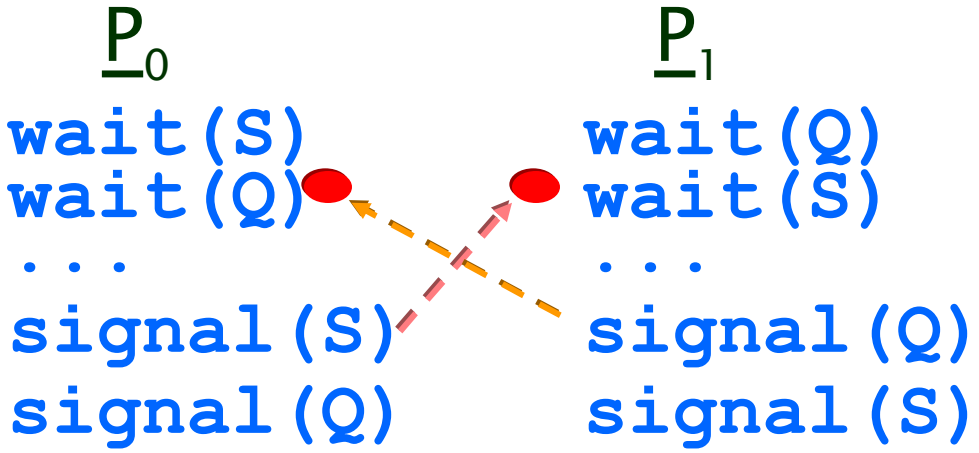
- Se produce interbloqueo (*deadlock*) cuando dos o más procesos esperan indefinida-mente por un suceso que debe ser provocado por uno de los procesos que esperan.



- Volveremos sobre esto en SOII

Interbloqueo y semáforos

- Con ellos es fácil producir interbloqueos
- Sean los semáforos *S* y *Q* inicializados a 1.



- Solución: usar siempre el mismo orden para los semáforos en todos los procesos.
- Problema: no siempre es posible hacerlo.

Inanición

- Inanición –un proceso nunca será eliminado de la cola del semáforo en el cual esta suspendido.
- Diferencia entre inanición e interbloqueo:
  - En la inanición, no es cierto que el proceso no obtendrá nunca el recurso (el proceso espera por algo que siempre que está disponible no le es asignado, p. ej., por su baja prioridad); en el interbloqueo, si.
  - En la inanición, el recurso bajo contención esta en uso continuo; en el interbloqueo, no.

Tipos de semáforos

- Semáforo contador –valor entero que puede variar en un dominio no restringido – uso: existen varias instancias de un recurso.
- Semáforo binario –valor entero que sólo puede variar entre 0 y 1 (¡seguridad!)
  - *wait*(S_{bin}):
```
if (S==1) S = 0;
else suspende(S);
```
  - *signal*(S_{bin}):
```
if (cola_vacia)  S= 1;
else reanudar(S);
```

Implementación de un semáforo contador con semáforos binarios (i)

- Se puede implementar un semáforo contador (*S*) a través de semáforos binarios, *Si*.
- Para ello necesitamos tres semáforos binarios.

```
semaforo_binarios S1, S2, S3;
int C;
```

Valores iniciales:

```
S1 = S3 = 1; S2 = 0;
```

C = valor inicial del semáforo S.

Implementación de semáforo contador con semáforos binarios (ii)

- Operación *wait (S)*:

```
wait(S3);
wait(S1);
C = C - 1;
if C < 0 {
    signal(S1);
    wait(S2);
}else signal(S1);
signal(S3);
```

- Op. *signal(S)*:

```
wait(S1);
C = C + 1;
if C ≤ 0
{ signal(S2) };
signal(S1);
;
```

Limitaciones de los semáforos

- Uso no forzado, sólo por convención.
- No son un mecanismo de comunicación.
- No esta limitado el tiempo que un proceso puede estar bloqueado en un semáforo.
- Normalmente, no poseen una operación para saber el estado sin exponernos a un bloqueo.
- ...
- En general, son mecanismo de bajo nivel.

Problemas clásicos de sincronización

- Problema del productor-consumidor
  -modela la relación de un proceso que consume la información generada por otro.
- Problema de los lectores y escritores
  -modela la manipulación de datos compartidos por procesos con diferentes patrones de acceso: unos que los leen (lectores) y otros los escriben (escritores)
- Problema de los filósofos comensales
  -modela la compartición simultánea de varios recursos por varios procesos.

Condiciones del Productor-consumidor

- El productor no debe poner un ítem en el búfer cuando esta totalmente lleno
  ⇒*sincronización* – semáforo **`vacio`** cuenta los items vacíos. Inicializado a `N`.
- El consumidor no puede eliminar un ítem del búfer si este no tiene elementos. ⇒ *Sincronización* – semáforo **`lleno`** cuenta los items llenos. Inicializado a `0`.
- Productor y consumidor no debe escribir ó leer en/de el mismo item a la vez. *Exclusión mutua* -> semáforo **`mutex`**=`1`.

Productor-consumidor: código

- Productor

```
while (true) {
  produce itemP;
  wait(vacio);
  wait(mutex);
  bufer[i]=itemP;
  i=i+1;
  signal(mutex);
  signal(lleno);
};
```

- Consumidor

```
while (true) {
  wait(lleno);
  wait(mutex);
  itemC=bufer[i];
  i=i-1;
  signal(mutex);
  signal(vacio);
  consume itemC;
};
```

Ejemplo: productor-consumidor

```
typedef struct _NODO
{
   int dato;
   _NODO *siguiente;
} NODO;
```

```
void push(int valor){
   NODO *temp;

   temp = new NODO;
   temp->dato = valor;
   temp->siguiente =
   cima;
   cima = temp;}
```

```
int pop() {
 NODO *temp;
 int valor;
 if (cima==0)
      {
            return 0;
 }
     temp = cima;
 valor = cima->dato;
 cima = cima->siguiente;
 delete temp;
 return valor;}
```

Ejemplo: código del consumidor

```
VOID Consumidor(VOID) {
int x, p;
LONG contador;
for (x=0; x<10; x++) {
 WaitForSingleObject(Lleno, INFINITE);
 WaitForSingleObject(MutexNodo, INFINITE);
 p = pop();
 ReleaseSemaphore(MutexNodo, 1, 0);
 ReleaseSemaphore(Vacio, 1, &contador);
 WaitForSingleObject(MutexEscritura,INFINITE);
    cout << "Iteraccion de lectura: " << x
      << "; valor sacado: " << p
      << "; semaforo 'vacio' a: " << contador << endl;
 ReleaseSemaphore(MutexEscritura, 1, 0);
    Sleep(200);
  } }
```

Ejemplo: código del productor

```
VOID Productor(VOID) {
int x;
LONG contador;
for (x=0; x<10; x++) {
  WaitForSingleObject(Vacio, INFINITE);
  WaitForSingleObject(MutexNodo, INFINITE);
  push(x);
  ReleaseSemaphore(MutexNodo, 1, 0);
  ReleaseSemaphore(Lleno, 1, &contador);
  WaitForSingleObject(MutexEscritura, INFINITE);
    cout << "Iteration de escritura: " << x
      << "; semaforo 'lleno' a: " << contador <<
      endl;
  ReleaseSemaphore(MutexEscritura, 1, 0);
  Sleep(100);
  } }
```

Ejemplo: programa principal

```
void main(void) {
HANDLE des[numH];
DWORD hebraID;
//Creamos los semaforos
MutexEscritura = CreateSemaphore (0,1, 1, 0);
MutexNodo = CreateSemaphore (0,1, 1, 0);
Lleno = CreateSemaphore(0, 0, MAX_BUFERS, 0);
Vacio = CreateSemaphore(0,MAX_BUFERS, MAX_BUFERS,0);
des[0]=CreateThread(0, 0,Consumidor, 0, 0, 0);
des[1]=CreateThread(0, 0,Productor, 0, 0, 0);
WaitForMultipleObjects(numH, des, TRUE,INFINITE);
// Destruimos los semaforos
CloseHandle(Lleno);
CloseHandle(Vacio);
CloseHandle(MutexNodo);
CloseHandle(MutexEscritura);}  //Aquí para ejecutar.
```

Problema de los lectores-escritores

- Los escritores necesitan un acceso exclusivo a los datos compartidos. La lectura no plantea problemas.
- Cualquier solución involucra primar a un tipo de proceso frente al otro:
  - No hacer esperar a los lectores salvo que un escritor haya obtenido permiso para usar el objeto compartido.
  - Si un escritor espera para acceder a un objeto no se permiten nuevas lecturas.
- Cualquier solución puede provocar inanición.

Solución al primer problema

```
int mutex:semaforo:=1  /*mutex var clec*/
    wrt:semaforo:=1  /*mutex escritores*/
    clec:integer:=0 /*#lectores actual*/
```

- Escritor

```
wait(wrt);
   escribir;
signal(wrt);
```

- Lector

```
wait(mutex);
  clec:=clec+1;
  if (clec=1) wait(wrt);
signal(mutex);
  leer;
wait(mutex);
 clec:=clec-1;
 if (clec=0) signal(wrt);
signal(mutex);
```

Ejemplo: código de escritores

```
void HebraEscritora(INT id) {
  while (1)   {
    // Simular el procesamiento de escritura
    Sleep(GetTickCount() % 1000);
    // El escritor necesita escribir
    cout <<"Escritor "<<id <<"esperando”<< endl;
    WaitForSingleObject(write, INFINITE);
    // El escritor esta escribiendo
    cout << id << " esta escribiendo\n";
    Sleep(GetTickCount() % 1000);
    cout <<"Escritor "<<id<<"finalizado” << endl;
    // Se ha realizado la escritura
    ReleaseSemaphore(write, 1, 0);
  }
}
```

Ejemplo: código lectores

```
void HebraLectora(INT id) {
  while (1)
    Sleep(GetTickCount() % 100);
    WaitForSingleObject(mutex, INFINITE);
    ContadorLectores++;
    if (ContadorLectores==1)
        WaitForSingleObject(write, INFINITE);
    ReleaseMutex(mutex);
    cout <<ContadorLectores<<" lectores"<< endl;
    Sleep(GetTickCount() % 100);
    WaitForSingleObject(mutex, INFINITE);
    ContadorLectores--;
    cout <<ContadorLectores<<" lectores"<< endl;
    if (ContadorLectores==0)
      ReleaseSemaphore(write, 1, 0);
    ReleaseMutex(mutex);  } }
```

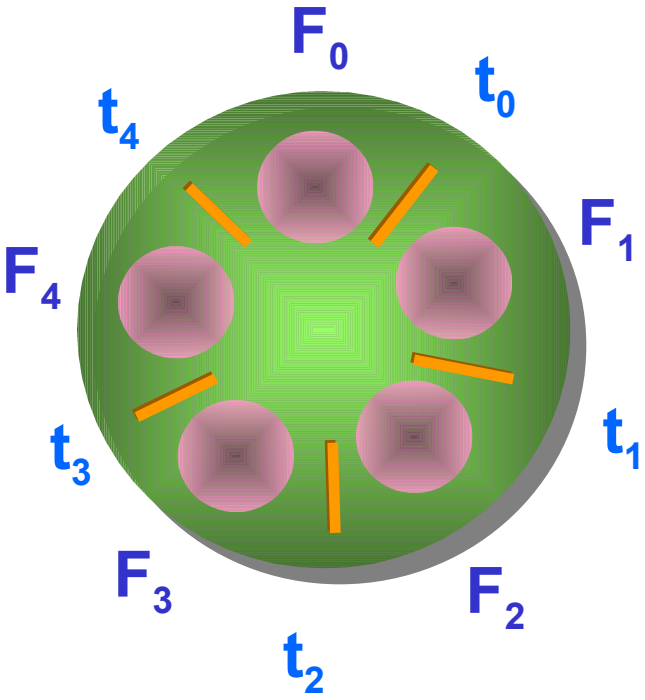
Ejemplo: programa principal

```
const INT numLect=6, numEscrit=3 ;
void main(void) {
  HANDLE hand[numLectores+numEscritores];
  DWORD threadID; INT i;
  mutex = CreateMutex(0, FALSE, 0);
  write = CreateSemaphore(0, 1, 1, 0);
  for (i=0; i<numLectores; i++)    {
    hand[i]=CreateThread(0, 0,HebraLectora,
      (VOID *) i, 0, &threadID); }
  for (i=0; i<numEscritores; i++)    {

 hand[i+numLect]=CreateThread(0,0,HebraEscritora,
      (VOID *) i, 0, &threadID);  }
 WaitForMultipleObjects(numLect+numEscrit,
    handles, TRUE, INFINITE);
  CloseHandle(mutex);
  CloseHandle(write); } // Aquí ejecución
```

Los filósofos comensales

- 5 filósofos se pasaban la vida pensando y comiendo. La comida la hacían en una mesa con cinco platos y sólo cinco tenedores. Para comer se necesitaban dos tenedores a la vez.
- Usamos los datos compartidos:

```
int tenedor[4]:
semaforos;
```



1ª solución

- Filósofo F_i

```
while (true) {
        wait(tenedor[i]);
        wait(tenedor[i+1 % 5]);
        comer;
        signal(tenedor[i]);
        signal(tendor[i+1 % 5]);
        pensar;
};
```

- Solución potencialmente interbloqueante.

Ejemplo de los filósofos

```
VOID Filosofo(LPVOID id) {
 int izqdo = (int) id;
 int dercho = (izqdo + 1) % NFILOSOFOS;
 while(1){
 WaitForSingleObject(hMutexTenedor[izqdo],INFINITE);
 printf("Filosofo #%d:coge tenedor izquierdo\n",(int)
  id);
 WaitForSingleObject(hMutexTenedor[dercho],INFINITE);
 printf("Filosofo #%d:coge tenedor derecho y comienza
       a comer\n",(int) id);
 ReleaseMutex(hMutexTenedor[izquierdo]);
 ReleaseMutex(hMutexTenedor[derecho]);
 printf("Filosofo #%d: libera tenedores y comienza a
  pensar\n",(int) id);
    }
}
```

Programa principal

```
#define NFILOSOFOS 5
HANDLE hMutexTenedor[NFILOSOFOS];
                                        //Aquí ejecutar
void main() {
  HANDLE hVectorHebras[NFILOSOFOS];
  DWORD IDHebra;
  int i;
  for (i=0; i<NFILOSOFOS; i++)
      hMutexTenedor[i]=CreateMutex(NULL, FALSE,NULL);
  for (i=0; i<NFILOSOFOS; i++)
      hVectorHebras[i] = CreateThread (NULL, 0,
      (LPTHREAD_START_ROUTINE)Filosofo,
      (LPVOID) i, 0, (LPDWORD)&IDHebra);
  WaitForMultipleObjects(NFILOSOFOS,hVectorHebras,
  TRUE,INFINITE);
}
```

Ejemplo de interbloqueo

Ejecución normal

Interbloqueo

Filosofos

12 x 16

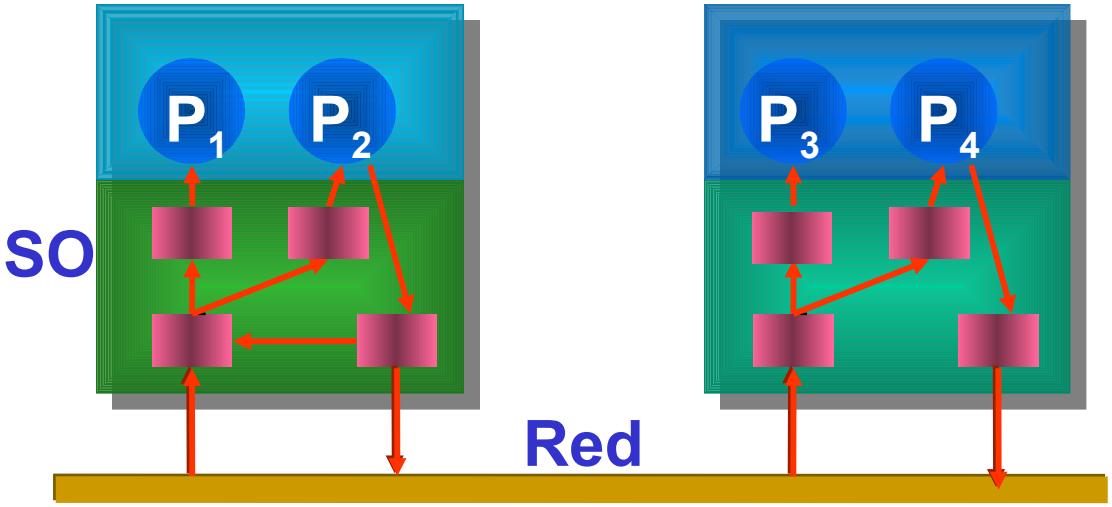
```
Filosofo #4:  coge tenedor izquierdo
Filosofo #1:  coge tenedor derecho y comienza a comer
Filosofo #1:  libera tenedores y comienza a pensar
Filosofo #3:  coge tenedor izquierdo
Filosofo #0:  coge tenedor derecho y comienza a comer
Filosofo #0:  libera tenedores y comienza a pensar
Filosofo #2:  coge tenedor izquierdo
Filosofo #4:  coge tenedor derecho y comienza a comer
Filosofo #4:  libera tenedores y comienza a pensar
Filosofo #3:  coge tenedor derecho y comienza a comer
Filosofo #3:  libera tenedores y comienza a pensar
Filosofo #0:  coge tenedor izquierdo
Filosofo #2:  coge tenedor derecho y comienza a comer
Filosofo #1:  coge tenedor izquierdo
Filosofo #1:  coge tenedor derecho y comienza a comer
Filosofo #2:  libera tenedores y comienza a pensar
Filosofo #3:  coge tenedor izquierdo
Filosofo #1:  libera tenedores y comienza a pensar
Filosofo #0:  coge tenedor derecho y comienza a comer
Filosofo #0:  libera tenedores y comienza a pensar
Filosofo #2:  coge tenedor izquierdo
Filosofo #4:  coge tenedor izquierdo
Filosofo #1:  coge tenedor izquierdo
Filosofo #0:  coge tenedor izquierdo
```

Soluciones libres de bloqueo

- Permitir que solo cuatro filósofos estén sentados simultáneamente.
- Permitir a un filósofo coger su tenedor sólo si ambos están libres (=coger tenedor es una sección crítica).
- Solución asimétrica: filósofo par coge tenedores izquierdo-derecho, filósofo impar coge tenedores derecho-izquierdo.
- Una solución libre de bloqueo no elimina necesariamente la posibilidad de inanición.

Paso de mensajes

- La comunicación entre procesos (*IPC–Inter–Process Comunication*) mediante paso de mensajes es necesaria cuando los procesos no comparten memoria común (no comparten variables comunes).



Paso de mensajes y memoria compartida

- Los sistemas de mensajes y de memoria compartida no son mutuamente exclusivos. Estos pueden utilizarse simultáneamente dentro de un SO o un único proceso.
- El paso de mensajes entre procesos en la misma máquina puede implementarse con memoria compartida.
- La memoria compartida puede también implementarse con mensajes.

Primitivas de paso de mensajes

- **`send(mensaje)`** –envía un mensaje.
- **`receive(mensaje)`** –recibe un mensaje. Si no hay mensajes disponibles se bloquea.
- Si los procesos P y Q desean comunicarse:
  - Establecer un *enlace de comunicación*.
  - Intercambiar mensajes vía send/receive.
- Un enlace de comunicación tiene una:
  - implementación física: bus, memoria, etc.
  - implementación lógica: propiedades lógicas

Cuestiones de implementación

- ¿Cómo se establecen los enlaces?
- ¿Puede un enlace asociarse con más de dos procesos?
- ¿Cuantos enlaces pueden existir entre cada par de procesos comunicantes?
- ¿Cuál es la capacidad del enlace?
- ¿El tamaño del mensaje que el enlace puede acomodar es fijo o variable?
- ¿Es un enlace unidireccional o bidireccional?

Tipos de comunicación

- Podemos establecer dos categorías en la forma en que los procesos se comunican:
  - Comunicación directa –los procesos comunicantes se nombran explícita-mente uno a otro en las operaciones de paso de mensajes. Ej. enviar carta.
  - Comunicación indirecta –los procesos intercambian mensajes a través de un buzón. Ej. enviar carta a un apartado de correos.

Comunicación directa

- Los procesos utilizan las primitivas:
  `send(P,mensaje)`– envía mensaje a P
  `receive(Q,mensaje)`–recibe mensaje de Q
- Propiedades del enlace:
  - se establece automáticamente.
  - asociado a un par de procesos.
  - exactamente un enlace por par de procesos.
  - Unidireccional o bidireccional.

El productor-consumidor con paso de mensajes

- Productor:

```
repeat …
produce item en nextp;
… send(consumidor, nextp);
until false;
```

- Consumidor:

```
repeat …
… receive(productor, nextp);…
consume item de nextp;
until false;
```

Comunicación indirecta

- Los mensajes se dirigen/se reciben de un *buzón* (o también denominado *puerto*).
  - Cada buzón tiene un identificador único.
  - P y Q deben compartir el buzón
- Propiedades del enlace:
  - Asociado con muchos procesos.
  - Varios enlaces por par de procesos.
  - Unidireccional o bidireccional.
- Varios procesos comparten enlace ¿quién lee un mensaje?

Búfering

- Búfering –cola de mensajes ligada al enlace.
- Tres formas de implementarlo:
  - Capacidad cero -- emisor espera al receptor (*cita* o *rendezvous*).
  - Capacidad limitada –el emisor espera si el enlace esta lleno.
  - Capacidad ilimitada –el emisor nunca espera.

Condiciones de excepción

- El paso de mensajes puede producir fallos ⇒ son necesarios mecanismos para recuperarnos de ellos, denominados *gestión de condiciones de excepción*.
- Terminación de procesos –¿Qué ocurre si proceso termina antes de procesar mensaje?
- Perdida de mensajes por falta de fiabilidad de la red – Se pueden establecer un *plazo*.
- Mensajes corruptos –Detección: suma de comprobación; solución: retransmisión.

Terminación de procesos

- El emisor P termina y el receptor Q espera (se bloquea) para siempre. Soluciones:
  - El sistema operativo termina a Q.
  - El SO notifica a Q de la terminación de P.
  - Q tiene un mecanismo interno (cronómetro) que determina cuanto debe esperar por mensaje de P.
- P envía mensaje y Q termina. Soluciones:
  - El sistema notifica a P.
  - El sistema termina a P.
  - P y Q utilizan reconocimiento con plazos.

Perdida de mensajes

- Frente a la perdida de mensajes tenemos varias alternativas:
  - El SO garantiza la retransmisión.
  - El emisor es responsable de detectarla utilizando plazos.
  - El emisor obtiene una excepción.

Mensajes corruptos

- Los mensajes que llegan al receptor desde el emisor pueden estar mezclados o deteriorados debido al ruido de canal de comunicación.
- Soluciones:
  - Necesidad de un mecanismo de detección de errores, como *checksum*, para detectar cualquier error.
  - Necesidad de mecanismo de recuperación de errores, p. ej. retransmisión.