

# Sistemas Operativos

---

## UNIDAD 3

### EXCLUSIÓN MUTUA Y SINCRONIZACIÓN CON SEMÁFOROS

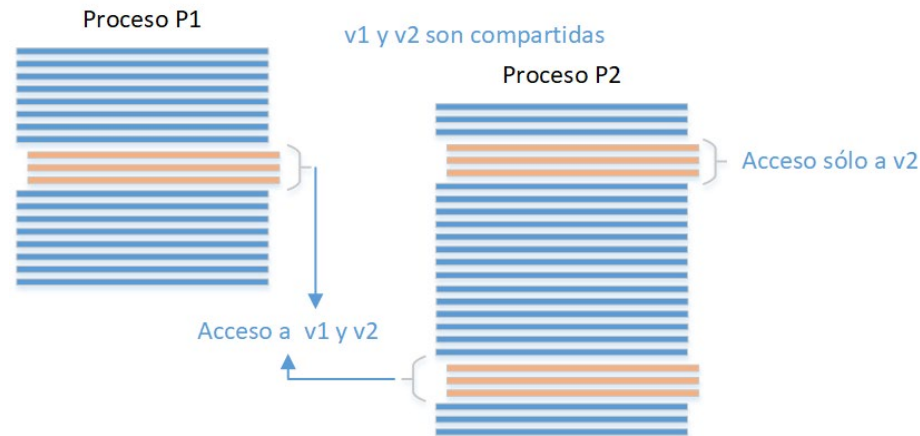


# ... ya sabemos que ...

## Sección Crítica

Es la **zona de código** (de un proceso) donde se accede a los **recursos compartidos** (con otros procesos) y que **no puede ser ejecutada cuando otro proceso esté en la misma sección crítica**.

Dos zonas de código (en procesos distintos) diremos que son la misma sección crítica si acceden a los mismos recursos compartidos (a todos o a algunos)



# ... ya sabemos que ...

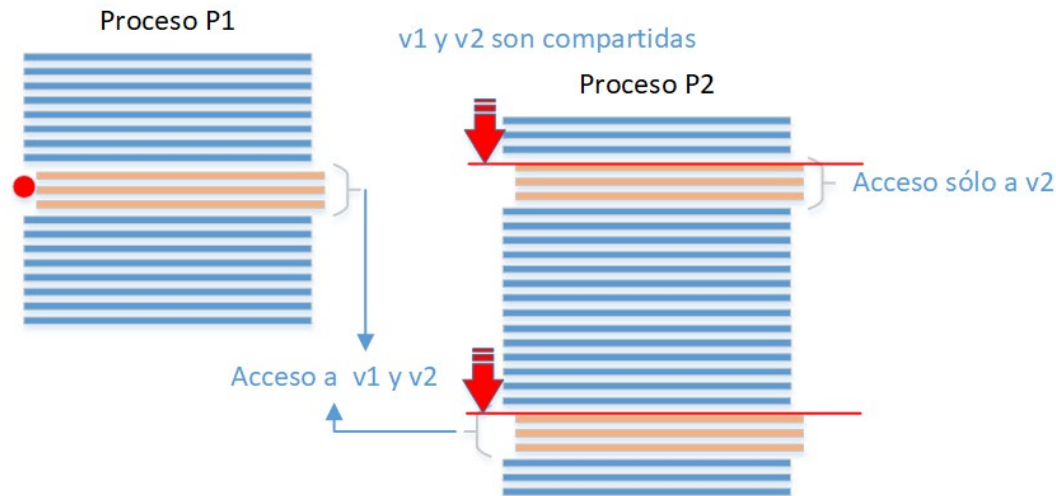
## Sección Crítica

Es la **zona de código** (de un proceso) donde se accede a los **recursos compartidos** (con otros procesos) y que **no puede ser ejecutada cuando otro proceso esté en la misma sección crítica**.

Dos zonas de código (en procesos distintos) diremos que son la misma sección crítica si acceden a los mismos recursos compartidos (a todos o a algunos)

## Exclusión Mutua

Es el requisito que garantiza que dos procesos, que comparten secciones críticas, no pueden ejecutar simultáneamente dentro de ellas.



# ... ya sabemos que ...

## Sección Crítica

Es la **zona de código** (de un proceso) donde se accede a los **recursos compartidos** (con otros procesos) y que **no puede ser ejecutada cuando otro proceso esté en la misma sección crítica**.

Dos zonas de código (en procesos distintos) diremos que son la misma sección crítica si acceden a los mismos recursos compartidos (a todos o a algunos)

## Exclusión Mutua

Es el requisito que garantiza que dos procesos, que comparten secciones críticas, no pueden ejecutar simultáneamente dentro de ellas.

## Soluciones para garantizar la Exclusión Mutua

Soportadas por el Hardware

- Instrucciones hardware atómicas (Test & Set, Exchange)
- Inhabilitación de Interrupciones

Soportadas por el Sistema Operativo y lenguajes de programación

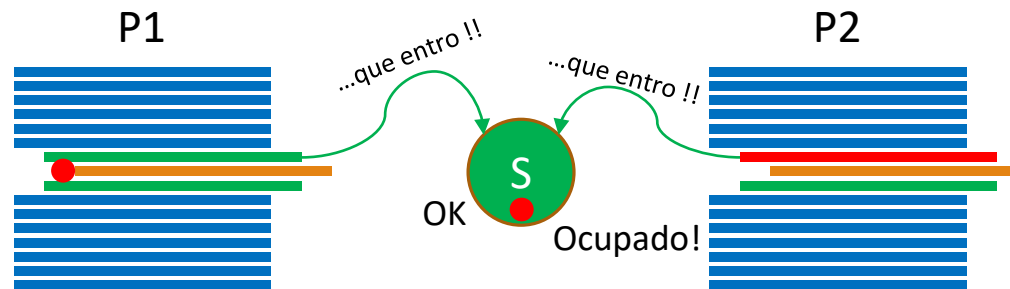
- **Semáforos**
- Monitores
- Paso de Mensajes

# Índice

- **Concepto de Semáforo**
- **Implementación por parte del Sistema Operativo**
- **Tipos de Semáforos**
  - **Binarios/Enteros**
  - **Fuertes/Débiles**
- **Exclusión mutua con Semáforos**
- **El problema del Productor Consumidor con Buffer Ilimitado**
  - **Paso a paso → Con semáforos binarios**
  - **Los “roles” de los semáforos**
  - **Con semáforos Enteros**
- **Conclusiones y estrategias**
- **El problema del Productor Consumidor con Buffer Limitado**
- **Un problema de examen**
- **Barreras**

# Concepto de Semáforo

- Es un **mecanismo** basado en señales entre procesos **para poder sincronizarse**.
- El semáforo lo crea el sistema operativo a petición de un proceso y se puede compartir.
- Los procesos pueden enviar y recibir señales a y del semáforo



# Concepto de Semáforo

- Es un **mecanismo** basado en señales entre procesos **para poder sincronizarse**.
- El semáforo lo crea el sistema operativo a petición de un proceso y se puede compartir.
- Los procesos pueden enviar y recibir señales a y del semáforo



# Concepto de Semáforo

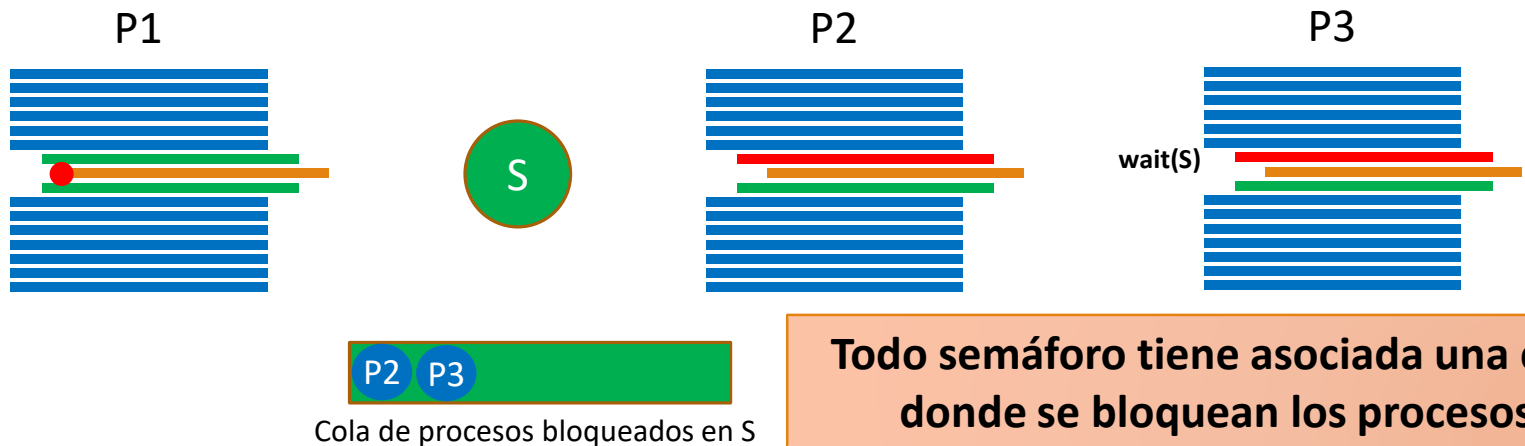
- Es un **mecanismo** basado en señales entre procesos **para poder sincronizarse**.
- El semáforo lo crea el sistema operativo a petición de un proceso y se puede compartir.
- Los procesos pueden enviar y recibir señales a y del semáforo

**Sobre un semáforo sólo se pueden hacer tres operaciones.**

- Para inicializar el semáforo llamamos a **init(S,valor)**
- Para recibir una señal (vía el semáforo S) llamamos a **wait(S)**
- Para transmitir una señal (vía el semáforo S) llamamos a **signal(S)**

**init, wait y signal son llamadas al sistema**

- El semáforo lo gestiona el S.O. por eso hay que hacer llamadas al sistema.





# Concepto de Semáforo

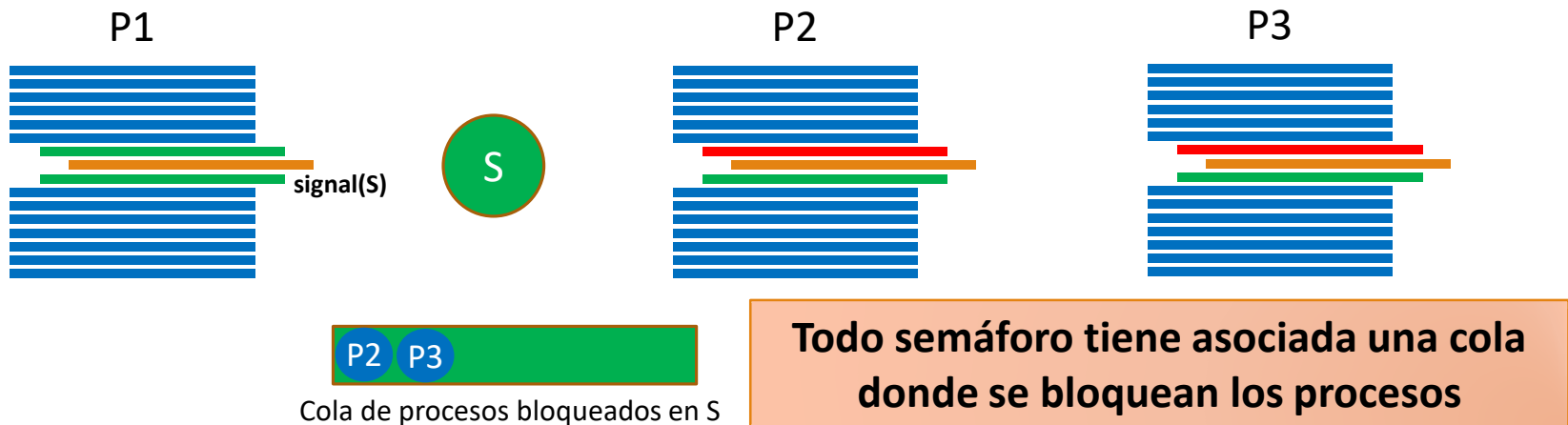
- Es un **mecanismo** basado en señales entre procesos **para poder sincronizarse**.
- El semáforo lo crea el sistema operativo a petición de un proceso y se puede compartir.
- Los procesos pueden enviar y recibir señales a y del semáforo

**Sobre un semáforo sólo se pueden hacer tres operaciones.**

- Para inicializar el semáforo llamamos a **init(S,valor)**
- Para recibir una señal (vía el semáforo S) llamamos a **wait(S)**
- Para transmitir una señal (vía el semáforo S) llamamos a **signal(S)**

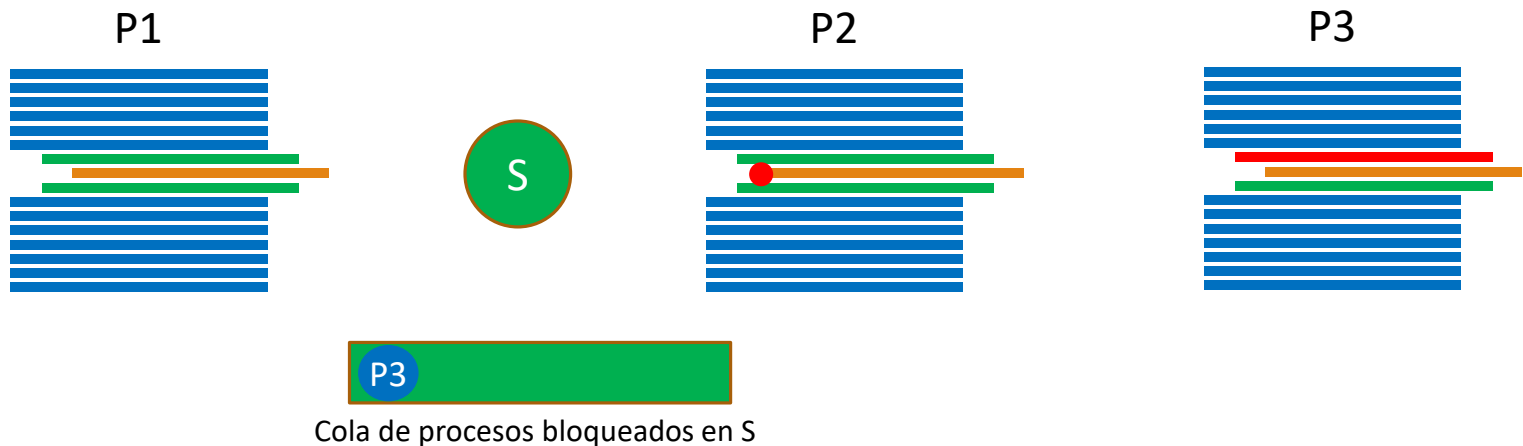
**init, wait y signal son llamadas al sistema**

- El semáforo lo gestiona el S.O. por eso hay que hacer llamadas al sistema.



# Concepto de Semáforo

- Es un **mecanismo** basado en señales entre procesos **para poder sincronizarse**.
- El semáforo lo crea el sistema operativo a petición de un proceso y se puede compartir.
- Los procesos pueden enviar y recibir señales a y del semáforo
- Sobre un semáforo sólo se pueden hacer tres operaciones.
- Para inicializar el semáforo llamamos a **init(S,valor)**
- Para recibir una señal (vía el semáforo S) llamamos a **wait(S)**
- Para transmitir una señal (vía el semáforo S) llamamos a **signal(S)**
- **init, wait y signal** son llamadas al sistema
- El semáforo lo gestiona el S.O. por eso hay que hacer llamadas al sistema.
- **Todo semáforo tiene asociada una cola donde se bloquean los procesos**




# Concepto de Semáforo

## Desde el punto de vista del programador:

- **Una variable** (del sistema) **que contiene un valor entero**
- sobre el cual sólo se pueden realizar **tres operaciones**
  1. **init** (crearlo) con un valor entero no negativo ( crea  $S \geq 0$  )
  2. **wait** decrementa el valor del semáforo (  $S = S-1$  )
  3. **signal** incrementa el valor del semáforo (  $S = S+1$  )

Por ejemplo:

```
Semaphore S = 0;  
S = 0  wait(S)  → S = -1  
S = -1 signal(S) → S = 0  
S = 0  signal(S) → S = 1  
S = 1  signal(S) → S = 2  
S = 2  wait(S)  → S = 1
```

**wait(S)**  
  
**signal(S)**

## ¿Cómo se bloquean y desbloquean los procesos en la cola?

si  $S$  queda en negativo (  $S < 0$  ) → **SE BLOQUEA**  
sino → **CONTINUA** (entraría en la Sección Crítica)

si  $S$  queda  $\leq 0$  → **LIBERAMOS DE LA COLA**  
si  $S > 0$  ... pues nada, queda incrementado

# Concepto de Semáforo

## Desde el punto de vista del programador:

- **Una variable** (del sistema) **que contiene un valor entero**
- sobre el cual sólo se pueden realizar **tres operaciones**
  1. **init** (crearlo) con un valor entero no negativo ( crea  $S \geq 0$  )
  2. **wait** decrementa el valor del semáforo (  $S = S - 1$  )
  3. **signal** incrementa el valor del semáforo (  $S = S + 1$  )

## Desde el punto de vista del proceso: (se ve más fácil)

Por ejemplo:

```
Semaphore S = 0;  
S = 0  wait(S)  → S = -1  
S = -1 signal(S) → S = 0  
S = 0  signal(S) → S = 1  
S = 1  signal(S) → S = 2  
S = 2  wait(S)  → S = 1
```

¿Cómo se bloquean y desbloquean  
los procesos en la cola?

**wait(S)** Si puedo entrar, **Paso**, sino me **Bloqueo**

**Sección Crítica**

**signal(S)** Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

# Concepto de Semáforo

$S = S - 1 \leftarrow \text{wait}(S)$  Si puedo entrar, **Paso**, sino me **Bloqueo** si queda  $S < 0$



$S = S + 1 \leftarrow \text{signal}(S)$  Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

¿Inicializamos  $S=0$  o a  $S=1$ ? ¿Para que sirve inicializar  $S>0$ ?

Si inicializamos a 0 entonces el primer  $\text{wait}(S) \rightarrow S = -1 \rightarrow$  **Se bloquea**  
**¡No pasa ni el primero que llegue!**

Si inicializamos a 1 entonces el primer  $\text{wait}(S) \rightarrow S = 0 \rightarrow$  **Pasa**  
**¡Sólo pasa el primero!** el segundo  $\text{wait}(S) \rightarrow S = -1 \rightarrow$  **Se bloquea**

Si inicializamos a 3 entonces el primer  $\text{wait}(S) \rightarrow S = 2 \rightarrow$  **Pasa**  
**¡Pasan los 3 primeros!** el segundo  $\text{wait}(S) \rightarrow S = 1 \rightarrow$  **Pasa**  
**¡El cuarto se bloquea!** el tercer  $\text{wait}(S) \rightarrow S = 0 \rightarrow$  **Pasa**  
el cuarto  $\text{wait}(S) \rightarrow S = -1 \rightarrow$  **Se Bloquea**

BANCO

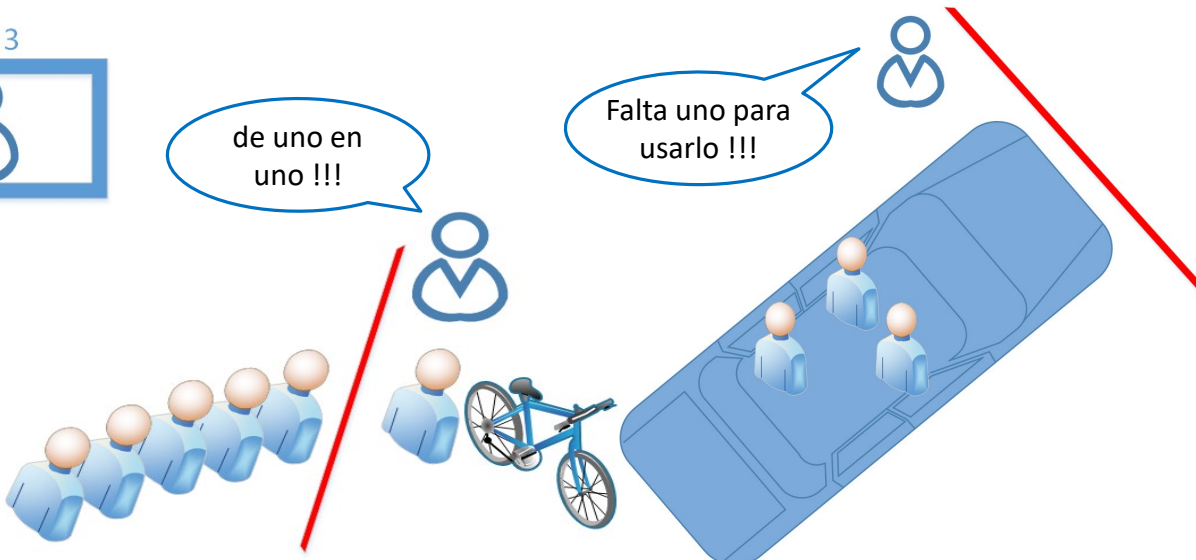
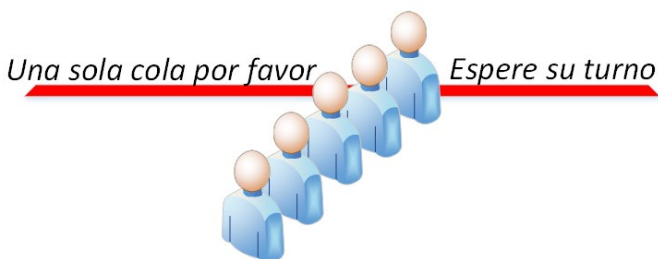
Caja 1

Caja 2

Caja 3

de uno en uno !!!

Falta uno para usarlo !!!



# Concepto de Semáforo

$S = S - 1 \leftarrow \text{wait}(S)$  Si puedo entrar, **Paso**, sino me **Bloqueo** si queda  $S < 0$

Sección Crítica

$S = S + 1 \leftarrow \text{signal}(S)$  Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

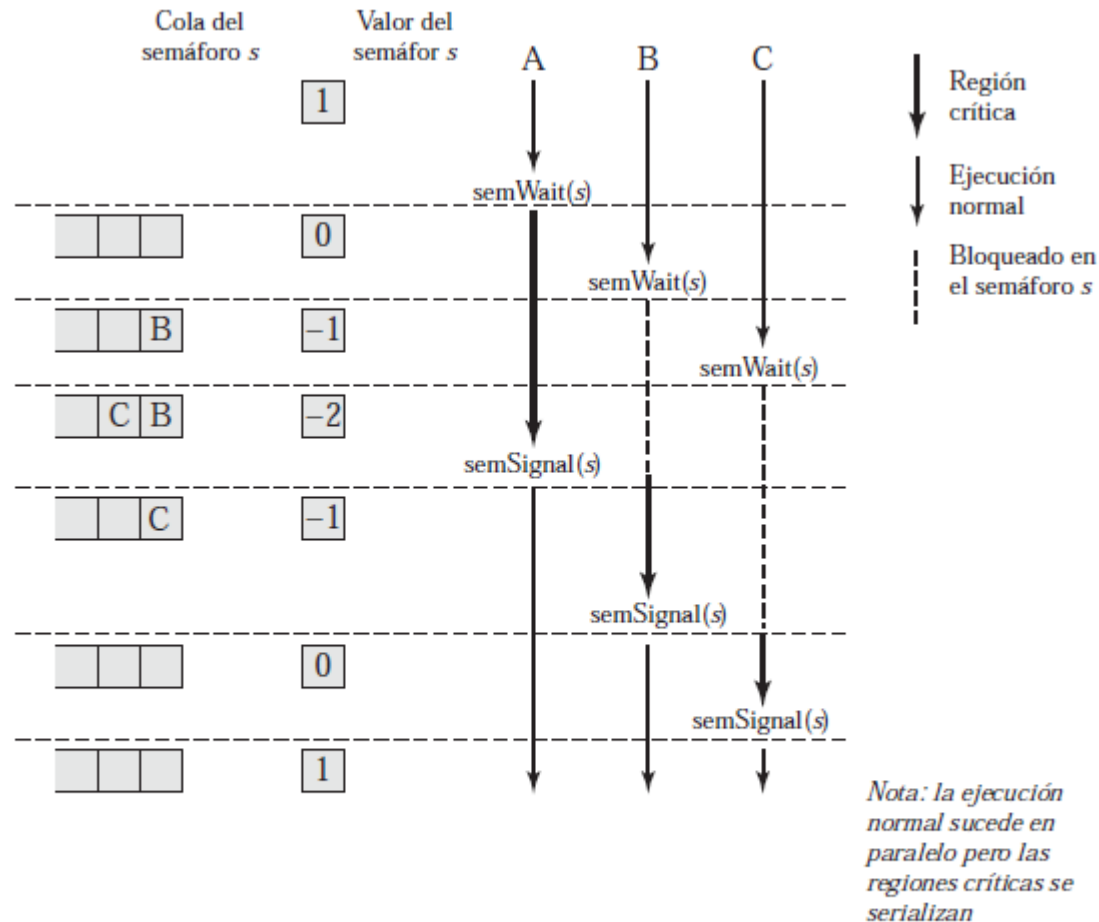


Figura 5.7. Procesos accediendo a datos compartidos protegidos por un semáforo.

# Implementación

$S = S - 1 \leftarrow \text{wait}(S)$  Si puedo entrar, **Paso**, sino me **Bloqueo**  
si queda  $S < 0$

Sección Crítica

$S = S + 1 \leftarrow \text{signal}(S)$  Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

```
struct semaphore {  
    int cuenta;  
    queueType cola;  
}  
  
void semWait(semaphore s)  
{  
    s.cuenta—;  
    if (s.cuenta < 0)  
    {  
        poner este proceso en s.cola;  
        bloquear este proceso;  
    }  
}  
  
void semSignal(semaphore s)  
{  
    s.cuenta++;  
    if (s.cuenta <= 0)  
    {  
        extraer un proceso P de s.cola;  
        poner el proceso P en la lista de listos;  
    }  
}
```

Es una estructura con  
un valor (contador) y una cola

Dos operaciones, **wait** y **signal**

**wait** decrementa  
y si  $< 0$  **Bloquea**  
sino, **Pasa**

**signal** incrementa  
Si queda alguien lo **Desbloquea**

La **Lista de Listos** del SO es donde  
están los procesos que están  
preparados para tomar la CPU

Figura 5.3. Una definición de las primitivas del semáforo.

# Implementación

$S = S - 1 \leftarrow \text{wait}(S)$  Si puedo entrar, **Paso**, sino me **Bloqueo**  
si queda  $S < 0$

Sección Crítica

$S = S + 1 \leftarrow \text{signal}(S)$  Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

```
struct semaphore {  
    int cuenta;  
    queueType cola;  
}  
void semWait(semaphore s)  
{  
    s.cuenta—;  
    if (s.cuenta < 0)  
    {  
        poner este proceso en s.cola;  
        bloquear este proceso;  
    }  
}  
void semSignal(semaphore s)  
{  
    s.cuenta++;  
    if (s.cuenta <= 0)  
    {  
        extraer un proceso P de s.cola;  
        poner el proceso P en la lista de listos;  
    }  
}
```

Es una estructura con  
un valor (contador) y una cola

Dos operaciones, **wait** y **signal**

**wait** decrementa  
y si  $< 0$  **Bloquea**  
sino, **Pasa**

**signal** incrementa  
Si queda alguien lo **Desbloquea**

¿Qué pasa en el sistema  
cuando un proceso se bloquea?

Que el planificador de procesos  
selecciona uno de la Lista de Listos  
y le asigna el procesador

Figura 5.3. Una definición de las primitivas del semáforo.



# Tipos de Semáforos

- **SEMÁFOROS “GENERALES”** o “CON CONTADOR” o “**ENTEROS**”
- **SEMÁFOROS “BINARIOS”**

# Tipos de Semáforos

- SEMÁFOROS “GENERALES” o “CON CONTADOR” o “ENTEROS”
- SEMÁFOROS “BINARIOS”

```
struct binary_semaphore {  
    enum {cero, uno} valor;  
    queueType cola;  
};  
void semWaitB(binary_semaphore s)  
{  
    if (s.valor == 1)  
        s.valor = 0;  
    else  
    {  
        poner este proceso en s.cola;  
        bloquear este proceso;  
    }  
}  
void semSignalB(binary_semaphore s)  
{  
    if (esta_vacia(s.cola))  
        s.valor = 1;  
    else  
    {  
        extraer un proceso P de s.cola;  
        poner el proceso P en la lista de listos;  
    }  
}
```

El valor del semáforos solo puede ser 0 o 1

## wait

si es 1 lo pone a 0 y **Pasa**  
sino lo deja en 0 y **Bloquea** al proceso

## signal

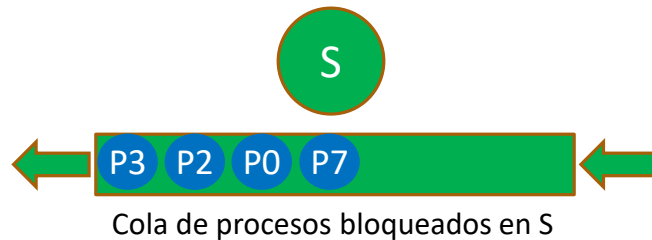
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y **Pasa**)

Figura 5.4. Una definición de las primitivas del semáforo binario.

# Tipos de Semáforos

- SEMÁFOROS FUERTES
- SEMÁFOROS DÉBILES

## SEMÁFORO FUERTE

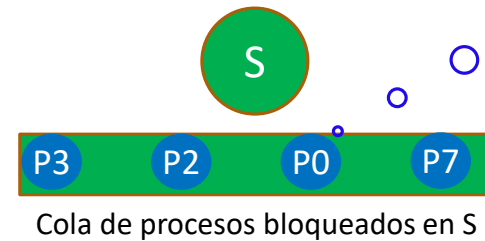


Planificación de la Cola  
FIFO  
(First In – First Out)

Los habituales en la  
mayoría de los S.O.

**Están libres de Inanición**

## SEMÁFORO DÉBIL



Planificación de la Cola  
mediante alguna política  
de planificación

**Pueden causar Inanición**

### La Inanición

Es un problema en sistemas multitarea, donde a un proceso o un hilo de ejecución se le deniega siempre el acceso a un recurso compartido. Sin este recurso, la tarea sigue bloqueada y no puede ser nunca finalizada.

# Exclusión mutua con Semáforos

- Tenemos n procesos identificados como P(1), P(2) ... P(n) **u otros distintos Q ...**
- Todos necesitan acceder a los mismos recursos
- Cada proceso tiene una sección crítica que accede a los recursos

## Claves para cumplir la exclusión mutua

- Declarar un semáforo s **compartido** para regular el acceso a la **sección crítica común**

En cada proceso:

- Ejecuta **wait(s)** antes de entrar en la sección crítica
- Ejecuta **signal(s)** al salir de la sección crítica
- La inicialización depende del problema
  - **s=1 si sólo puede entrar un proceso a la vez**
  - s=m si pueden entrar m procesos a la vez
  - s=0 si algún otro proceso habilita el paso a todos los procesos (levanta la barrera)

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* sección crítica */;
        semSignal(s);
        /* resto */
    }
}
void Q(int i)
{
    while (true)
    {
        semWait(s);
        /* sección crítica */;
        semSignal(s);
        /* resto */
    }
}
void main()
{
    paralelos (P(1), P(2), . . . ,P(n),Q(1),Q(2));
}
```

Figura 5.6. Exclusión mutua usando semáforos.

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**



# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos
```

**n** es compartida !!!  
pero tengo que  
proteger también las  
acciones de **añadir** y  
de **extraer**  
es decir,  
**Los accesos al buffer**

```
//Morris Lester Szyslak (Moe)
```

```
void productor(int n){  
    while (true){  
        producir();  
        añadir();  
        n++;  
    }  
}
```

```
//Homer Jay Simpson (Homer)
```

```
void consumidor(int n){  
    while (true){  
        extraer();  
        n--;  
        consumir();  
    }  
}
```

```
void main() {  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1;
```

```
//Morris Lester Szyslak (Moe)
```

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        signal(s);  
    }  
}
```

```
//Homer Jay Simpson (Homer)
```

```
void consumidor(int n){  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
    }  
}
```

```
void main()(  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**  
si es 1 lo pone a 0 y **Pasa**  
si es 0 lo deja en 0 y **Bloquea** al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y **continúa**)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

**¿Y si el consumidor llega primero? ¿Y si el consumidor va más rápido?**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1;
```

```
//Morris Lester Szyslak (Moe)
```

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        signal(s);  
    }  
}
```

```
//Homer Jay Simpson (Homer)
```

```
void consumidor(int n){  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
    }  
}
```

```
void main()(  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**  
si es 1 lo pone a 0 y **Pasa**  
si es 0 lo deja en 0 y **Bloquea** al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y **continúa**)



# ¿Semáforos? ... Un juego de Rol

Podemos pensar que :  
**los semáforos juegan dos roles distintos**

- Un rol **MUTEX** , para garantizar la exclusión mutua, protegiendo la sección crítica
- Un rol **SINCRO** , para sincronizar los procesos (ajustar su velocidad relativa, *“que se lleven bien”*)



Clancy Wiggum



Nedwar ( Ned ) Flanders

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

//Morris Lester Szyslak (Moe)

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        if (n==1)  
            signal(retardo)  
        signal(s);  
    }  
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){  
    wait(retardo);  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
    }  
}
```

```
void main(){  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

//Morris Lester Szyslak (Moe)

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        if (n==1)  
            signal(retardo)  
        signal(s);  
    }  
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){  
    wait(retardo);  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
        if (n==0)  
            wait(retraso);  
    }  
}
```

```
void main() {  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

... no nos dicen nada acerca del tamaño del buffer... pensemos que es **infinito**

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

```
//Morris Lester Szyslak (Moe)  
  
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        if (n==1)  
            signal(retardo)  
        signal(s);  
    }  
}
```

```
//Homer Jay Simpson (Homer)  
  
void consumidor(int n){  
    wait(retardo);  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
        if (n==0)  
            wait(retraso);  
    }  
}
```

```
void main(){  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**  
si es 1 lo pone a 0 y **Pasa**  
si es 0 lo deja en 0 y **Bloquea** al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y **continúa**)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/
int n; //Número de elementos producidos
BinSemaphore s=1; //MUTEX
BinSemaphore retardo=0; //SINCRO
```

//Morris Lester Szyslak (Moe)

```
void productor(int n){
    while (true){
        producir();
        wait(s);
        añadir();
        n++;
        if (n==1)
            signal(retardo)
        signal(s);
    }
}
```

```
void main(){
    n=0;
    paralelos(productor(1),productor(2), consumidor(1));
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){
    wait(retardo);
    while (true){
        wait(s);
        extraer();
        n--;
        signal(s);
        consumir();
        if (n==0)
            wait(retraso);
    }
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/
int n; //Número de elementos producidos
BinSemaphore s=1; //MUTEX
BinSemaphore retardo=0; //SINCRO
```

**iiii Las consultas a elementos compartidos deben estar también protegidas por un MUTEX !!!!**

//Morris Lester Szyslak (Moe)

```
void productor(int n){
    while (true){
        producir();
        wait(s);
        añadir();
        n++;
        if (n==1)
            signal(retardo)
        signal(s);
    }
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){
    wait(retardo);
    while (true){
        wait(s);
        extraer();
        n--;
        signal(s);
        consumir();
        if (n==0)
            wait(retraso);
    }
}
```

```
void main(){
    n=0;
    paralelos(productor(1),productor(2), consumidor(1));
}
```

Mientras consumíamos se nos ha colado un productor que nos ha cambiado la n y la consultamos después para ver si sincronizamos...

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

¿cómo lo arreglamos?

//Morris Lester Szyslak (Moe)

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        if (n==1)  
            signal(retardo)  
        signal(s);  
    }  
}
```

```
void main(){  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){  
    wait(retardo);  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        signal(s);  
        consumir();  
        if (n==0)  
            wait(retraso);  
    }  
}
```

**wait**

si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**

Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/
int n; //Número de elementos producidos
BinSemaphore s=1; //MUTEX
BinSemaphore retardo=0; //SINCRO
```

//Morris Lester Szyslak (Moe)

```
void productor(int n){
    while (true){
        producir();
        wait(s);
        añadir();
        n++;
        if (n==1)
            signal(retardo)
        signal(s);
    }
}
```

```
void main(){
    n=0;
    paralelos(productor(1),productor(2), consumidor(1));
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){
    wait(retardo);
    while (true){
        wait(s);
        extraer();
        n--;
        signal(s);
        consumir();
        wait(s)
        if (n==0)
            signal(s)
            wait(retraso);
    }
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)



# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/
int n; //Número de elementos producidos
BinSemaphore s=1; //MUTEX
BinSemaphore retardo=0; //SINCRO
```

//Morris Lester Szyslak (Moe)

```
void productor(int n){
    while (true){
        producir();
        wait(s);
        añadir();
        n++;
        if (n==1)
            signal(retardo)
        signal(s);
    }
}
```

```
void main(){
    n=0;
    paralelos(productor(1),productor(2), consumidor(1));
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){
    wait(retardo);
    while (true){
        wait(s);
        extraer();
        n--;
        signal(s);
        consumir();
        wait(s)
        if (n==0)
            wait(retraso);
        signal(s)
    }
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/  
int n; //Número de elementos producidos  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

```
//Morris Le
```

```
void
```

¿¿¿ UN WAIT DENTRO DE UNA SECCIÓN CRÍTICA!!??

Peor aún → INTERBLOQUEO

```
if (n==1)  
  signal(retraso)
```

```
  signal(s);  
}
```

```
consumir();  
wait(s);
```

```
if (n==0)  
  wait(retraso);
```

```
  signal(s);  
}
```

```
void main(){  
  n=0;  
  paralelos(productor(1),productor(2), consumidor(1));  
}
```

**wait**

si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**

Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# El problema del Productor-Consumidor

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.

Es decir, **sólo un agente** (productor o consumidor), puede acceder **al mismo tiempo**.

Instrucción	Productor	Consumidor	s	n	retraso
1			1	0	0
2	wait(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) signal(retraso)		0	1	1
6	signal(s)		1	1	1
7	producir()		1	1	1
8		wait(retraso)	1	1	0
9		wait(s)	0	1	0
10		extraer();	0	1	0
11		n--;	0	0	0
12		signal(s)	1	0	0
13		consumir();	1	0	0
14	wait(s)		0	0	0
15	añadir()		0	0	0
16	n++		0	1	0
17	if (n==1) signal(retraso)		0	1	1
18	signal(s)		1	1	1
19		if (n==0) wait(retraso)	1	1	1
20		wait(s)	0	1	1
21		extraer();	0	1	1
22		n--;	0	0	1
23		signal(s)	1	0	1
24		consumir();	1	0	1
25		if (n==0) wait(retraso)	1	0	0
26		wait(s)	0	0	0
27		extraer();	0	0	0
28		n--;	0	-1	0
29		signal(s)	1	-1	0

```
/* program productor-consumidor*/  
int n; //Número de elementos producido  
BinSemaphore s=1; //MUTEX  
BinSemaphore retardo=0; //SINCRO
```

!!! Utilizando variables locales que nadie más podrá cambiar y que me recuerdan el valor de la compartida para cuando necesite consultarlo !!!!

//Morris Lester Szyslak (Moe)

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        n++;  
        if (n==1)  
            signal(retardo)  
        signal(s);  
    }  
}
```

```
void main(){  
    n=0;  
    paralelos(productor(1),productor(2), consumidor(1));  
}
```

//Homer Jay Simpson (Homer)

```
void consumidor(int n){  
    int m; //local mía, privada  
    wait(retardo);  
    while (true){  
        wait(s);  
        extraer();  
        n--;  
        m=n; // me guardo el valor  
        signal(s);  
        consumir();  
        if (m==0) //consulta mi valor  
            wait(retraso);  
    }  
}
```

**wait**  
si es 1 lo pone a 0 y Pasa  
si es 0 lo deja en 0 y Bloquea al proceso

**signal**  
Si la cola está vacía lo pone a 1  
sino Desbloquea a uno de la cola  
(y continúa)

# ¿Conclusión?

- Trabajar con semáforos es altamente complejo en determinados problemas, hay que analizar bien las posibilidades .... **¿Tengo que probar todas las posibles trazas?**

## Claves para no tener que hacerlo

- Analizar bien el problema, entiende cómo se deben sincronizar los procesos
- Definir los semáforos MUTEX y SINCRO necesarios (analizar cuántos MUTEX usar)  
Si los recursos son los mismos entonces el mismo MUTEX sino creo otro ...
- Entender cuáles son los recursos críticos a proteger (funciones, variables... )
- Si accedo a una variable compartida **tanto en lectura como en escritura** debo protegerlas
- Si **accedo** en un **if** tendré que **guardarlas en temporales** mientras esté en la sección crítica
- Es **muy peligroso introducir un wait** en una sección crítica  
pero no está prohibido, deberé **garantizar que alguien rompa el posible interbloqueo**

¿Estoy usando los semáforos apropiados?

¿cuáles deben ser binarios?

¿cuáles deben ser enteros?

# Enteros vs Binarios

En principio se pueden resolver los problemas con ambos tipos de semáforos

¿Cuál elegir?

- Binarios → MUTEX para secciones críticas que permiten un único proceso  
→ SINCRO para sincronización
- Enteros → MUTEX cuando se permita entrar a varios procesos en la sección crítica  
→ SINCRO para sincronización (barreras)

## Sincronización por contador.

Si pienso que necesito un contador para sincronizar procesos en función del valor del contador entonces:

Binarios → Tengo que proteger al contador puesto que ambos procesos lo usan para saber cómo sincronizar (productor-consumidor con binarios)

Enteros → **No es necesario un contador** adicional para sincronizar, la sincronización se puede conseguir con semáforos enteros (incluyen un contador)

# Productor-Consumidor (Buffer Infinito, Enteros)

```
/* program productor-consumidor*/  
Semaphore sProd=0; //SINCRO (su contador cuenta los que hay producidos)  
BinSemaphore s=1; //MUTEX de acceso al buffer
```

```
void productor(int n){  
    while (true){  
        producir();  
        wait(s);  
        añadir();  
        signal(s);  
        signal(sProd)  
    }  
}  
  
void consumidor(int n){  
    while (true){  
        wait(sProd);  
        wait(s);  
        extraer();  
        signal(s);  
        consumir();  
    }  
}  
  
void main(){  
    n=0;  
    paralelos(productor(1),productor(2),  
    consumidor(1));  
}
```

Cuando no haya elementos producidos se bloqueará

$S = S - 1 \leftarrow \text{wait}(S)$  Si puedo entrar, **Paso**, sino me **Bloqueo** si queda  $S < 0$



$S = S + 1 \leftarrow \text{signal}(S)$  Yo **Sigo**, y si hay alguien bloqueado se **Desbloquea**

# Productor-Consumidor con Buffer Limitado

Hay uno o más procesos generando algún tipo de datos (registros, caracteres, objetos, etc.) y colocándolos en un *buffer*. Hay un único consumidor que está extrayendo datos del *buffer*. El sistema debe garantizar que se impida la superposición de operaciones sobre los datos.



# Productor-Consumidor (con Buffer Limitado)

```
/* program productor-consumidor*/  
Semaphore sHuecos=24 //SINCRO (su contador cuenta los huecos que quedan en la caja)  
Semaphore sProd=0; //SINCRO (su contador cuenta los que hay producidos)  
BinSemaphore s=1; //MUTEX de acceso al buffer
```

Cuando no haya huecos  
libres se bloqueará

```
void productor(int n){  
    while (true){  
        producir();  
        wait(sHuecos);  
        wait(s);  
        añadir();  
        signal(s);  
        signal(sProd)  
    }  
}  
  
void consumidor(int n){  
    while (true){  
        wait(sProd);  
        wait(s);  
        extraer();  
        signal(s);  
        signal(sHuecos);  
        consumir();  
    }  
}  
  
void main(){  
    n=0;  
    paralelos(productor(1),productor(2),  
    consumidor(1));  
}
```

Cuando no haya  
elementos producidos se  
bloqueará



# Ejercicio: “Los canarios en su jaula”

**E2 - ( 2,5 Ptos )** – Una persona tiene en su casa una jaula llena de canarios en la que hay un plato de alpiste y un columpio. Todos los canarios quieren primero comer del plato y luego columpiarse, sin embargo sólo tres de ellos pueden comer del plato al mismo tiempo y sólo uno de ellos puede columpiarse. Cuando terminan de columpiarse, están por la jaula hasta que les vuelve a entrar hambre y se repite su rutina. Desarrollar una solución con semáforos que coordine la actividad de los canarios.

```
/* program canarios*/
Semaphore sPlato=3
BinSemaphore sColumpio=1;

void canario (int idcanario){
    //El canario siempre hace lo mismo, primero come y luego se columpia en ese orden
    while true {
        //El canario pretende comer
        wait(sPlato)
        comer(idcanario);
        signal(plato)
        //El canario pretende columpiarse
        wait(sColumpio)
        columpiarse(idcanario);
        signal(sColumpio)
    }
    // El canario se pasea por la jaula
    pasearse(idcanario);
}

void comer (int idcanario){
    printf("El canario %d esta comiendo", idcanario);
    sleep(random); //El canario está un tiempo comiendo
    printf("El canario %d termina de comer", idcanario)
}

void columpiarse (int idcanario){
    printf("El canario %d se esta columpiando",idcanario);
    sleep(random); //El canario está un rato columpiandose
    printf("El canario %d termina de columpiarse",idcanario);
}

void pasearse (int idcanario){
    sleep(random); //El canario está un tiempo paseando
}

void main(){
    parbegin(canario(1),canario(2),...,canario(N));
}
```

# Exclusión Mutua: Barreras con Semáforos

## Secuenciación con Semáforos

- Se quieren ejecutar dos procesos y se quiere que P2 se ejecute siempre después de P1, y no sabemos cuando termina P1.
- Se utiliza un semáforo para detener a P2 hasta que P1 termine.

### PROCESO 1

```
{  
<codigo proceso 1>  
signal(p1);  
}
```

### PROCESO 2

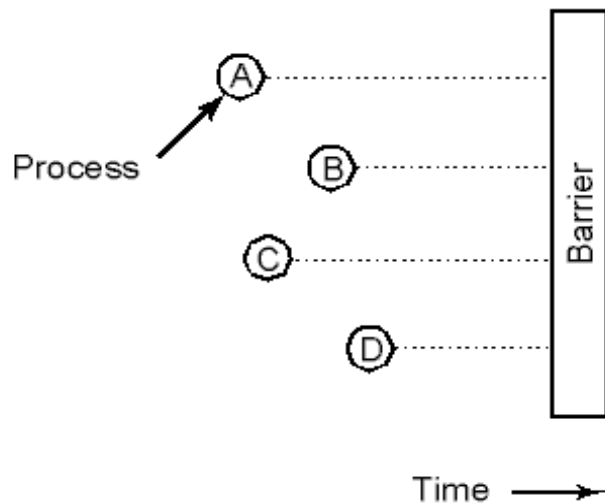
```
{  
wait(p1);  
<codigo proceso2>  
}
```

El semáforo se inicializará a 0

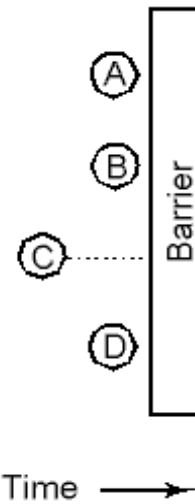
# Exclusión Mutua: Barreras con semáforos

## Barreras

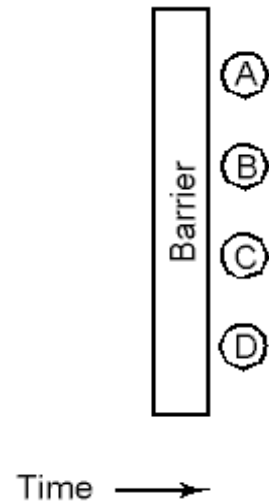
- Una barrera es el punto en el código tal que ningún proceso lo traspasa hasta que todos los procesos han llegado a su ejecución.
- Si y solo si todos los procesos llegan a la barrera, continúan ejecutándose.



(a)



(b)



(c)

# Exclusión Mutua: Barreras con semáforos

## Barreras

- Una barrera es el punto en el código tal que ningún proceso lo traspasa hasta que todos los procesos han llegado a su ejecución.
- Si y solo si todos los procesos llegan a la barrera, continúan ejecutándose.
- Supongamos 2 procesos: Ambos deben poder detenerse y notificar al otro que han llegado a la barrera

### PROCESO 1

```
....  
<codigo proceso 1>  
signal(barrera1);  
wait(barrera2);  
<codigo tras la barrera>  
....
```

### PROCESO 2

```
....  
<codigo proceso 2>  
signal(barrera2);  
wait(barrera1);  
<codigo tras la barrera>  
....
```

Ambos semáforos se inicializarán a 0

Cada proceso primero notifica que ha llegado a su barrera y luego espera en la del otro.

# Exclusión Mutua: Barreras con Semáforos

## Ejercicio:

Implementar una barrera para un numero arbitrario (n) de procesos.

Explicar detalladamente la solución alcanzada.

## Solución

- Basada en un proceso coordinador que se encarga de retener a todos los procesos en su barrera hasta que llega el último. Se precisan tantos semáforos como procesos (incluyendo al proceso coordinador). El coordinador conoce el numero de procesos.

### PROCESO P<sub>i</sub>

```
....  
<codigo proceso i>  
signal(barrera[i]);  
wait(coordinador);  
<codigo tras la barrera>  
....
```

### PROCESO COORDINADOR

```
{  
  for (i=0; i<n; i++){  
    wait(barrera[i]);  
  }  
  for (i=0; i<n; i++){  
    signal(Coordinador);  
  }  
}
```

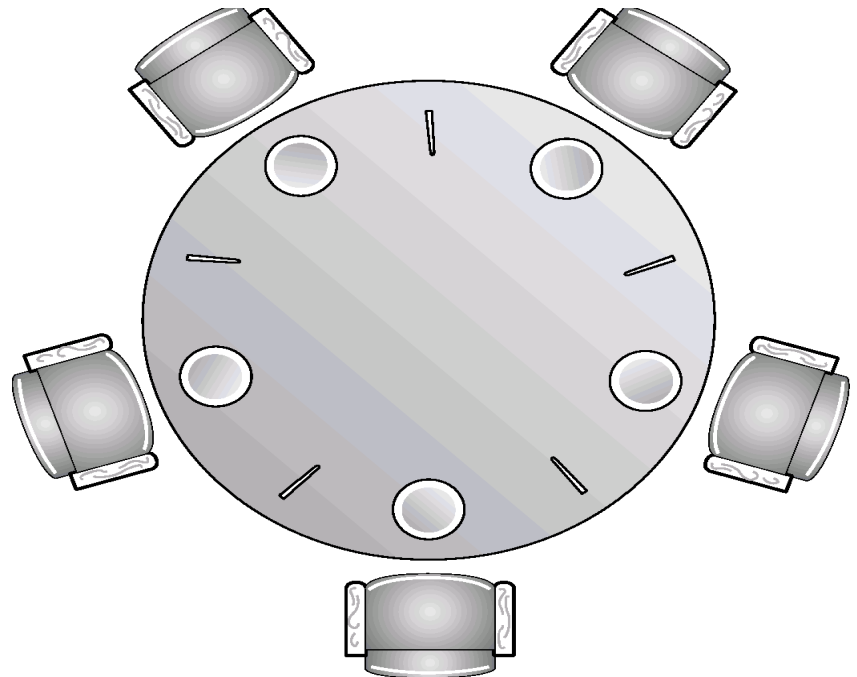
# Exclusión Mutua: SEMAFOROS

## Problema de los 5 filósofos:

- Planteado en 1965 por Dijkstra
- Ilustra problemas de exclusión mutua, interbloqueo e inanición
- Cada filósofo de vez en cuando se sienta a la mesa para comer.

Para ello necesita coger los  
tenedores de su izquierda y derecha.  
Cada tenedor es compartido con el  
filósofo adyacente.

- ¿Cuál es el Recurso crítico?



# Exclusión Mutua: SEMAFOROS

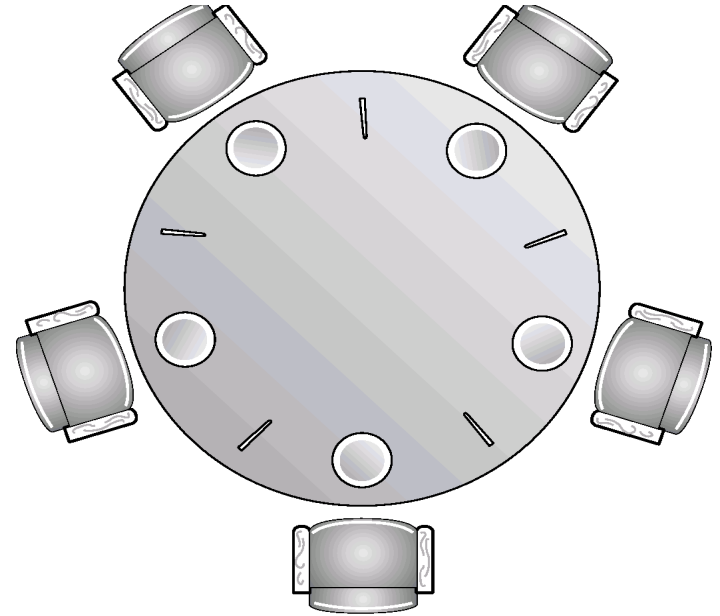
Necesitamos acceder de manera exclusiva a los tenedores, por tanto, un semáforo para cada tenedor

Un proceso para cada Filósofo

Numero de Filósofos 5:

Hay circularidad en los índices.

```
----- Filósofo i -----  
repeat           // Repetir  
    pensar();  
    wait(tenedor[i]);  
    wait(tenedor[i+1 % 5]);  
    comer();  
    signal(tenedor[i]);  
    signal(tenedor[i+1 % 5]);  
until false // Indefinidamente
```



¿Garantiza la exclusión mutua?  
¿Es correcta la solución?

# Exclusión Mutua: SEMAFOROS

Una posible solución: Permitir solo 4

----- Filósofos F1, F2, F3, F4, F5 -----

i=5

semaforo permiso=i-1;

semaforo tenedor[i];

**repeat** // Repetir

**pensar();**

**wait(permiso);**

**wait(tenedor[i]);**

**wait(tenedor[i+1 % 5]);**

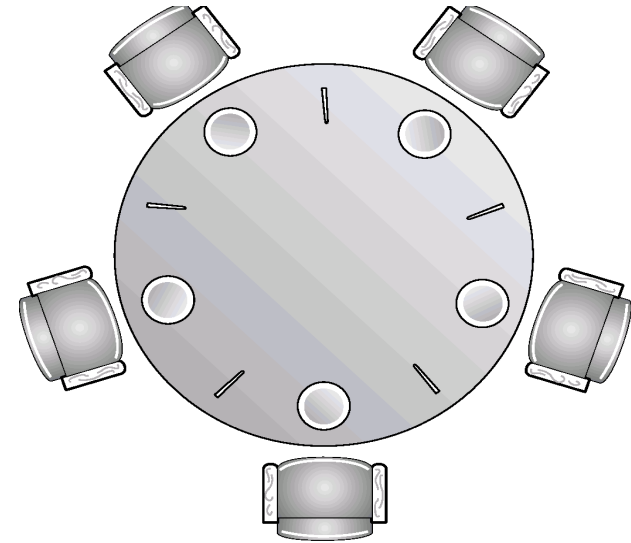
**comer();**

**signal(tenedor[i]);**

**signal(tenedor[i+1 % 5]);**

**signal(permiso);**

**until false** // Indefinidamente



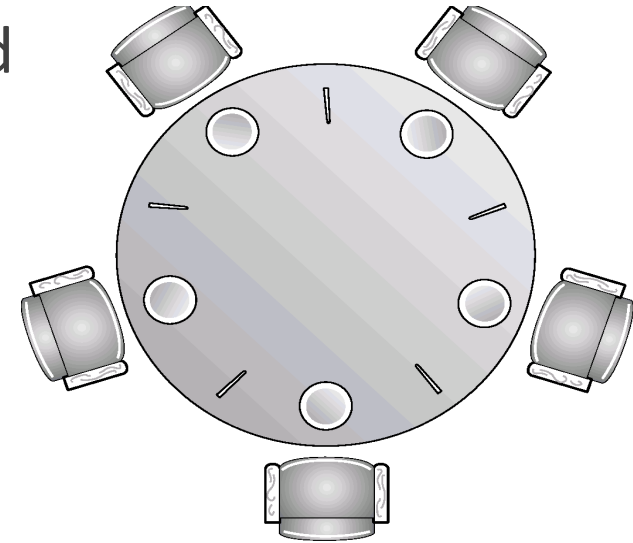


# Exclusión Mutua: SEMAFOROS

Una posible solución: Romper la circularidad

```
----- Filósofos F1, F2, F3, F4 -----  
repeat          // Repetir  
    pensar();  
    wait(tenedor[i]);  
    wait(tenedor[i+1 % 5]);  
    comer();  
    signal(tenedor[i]);  
    signal(tenedor[i+1 % 5]);  
until false    // Indefinidamente
```

```
----- Filósofo 5 -----  
repeat          // Repetir  
    pensar();  
    wait(tenedor[1]);  
    wait(tenedor[5]);  
    comer();  
    signal(tenedor[1]);  
    signal(tenedor[5]);  
until false    // Indefinidamente
```



# Exclusión Mutua: SEMAFOROS

Una solución más elaborada: coger los tenedores solo si están disponibles

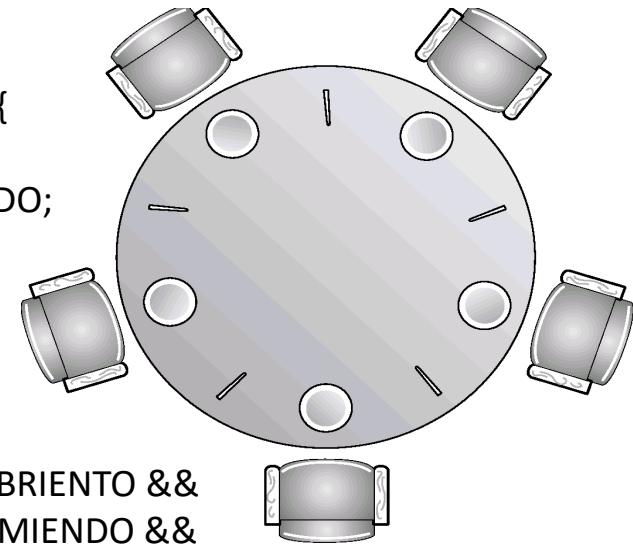
```
#define N      5
#define IZQ (i+N-1)%N
#define DCH   (i+1)%N
#define PENSANDO      0
#define HAMBRIENTO    1
#define COMIENDO      2
typedef int semaphore;
int estado[N];
semaphore mutex = 1;
semaphore s[N];

void filosofo(int i) {
    while(true){
        pensar();
        coger_tenedores(i);
        comer();
        dejar_tenedores(i);
    }
}
```

```
void coger_tenedores(i){
    wait(mutex);
    estado[i] = HAMBRIENTO;
    test(  i);
    signal(mutex);
    wait(s[i]);
}

void dejar_tenedores(i){
    wait(mutex);
    estado[i]= PENSANDO;
    test(IZQ);
    test(DCH);
    signal(mutex);
}

void test(i){
    if (estado[i]==HAMBRIENTO &&
        estado[IZQ]!=COMIENDO &&
        estado[DCH]!=COMIENDO) {
        estado[i]=COMIENDO
        signal(s[i]);
    }
}
```



# Exclusión Mutua: SEMAFOROS

## Lectores – escritores

- Existe un recurso común (fichero, base de datos, zona de memoria, banco de registros del procesador, etc...) al cual acceden varios procesos para leer y varios procesos para escribir de manera concurrente. (lectores y escritores respectivamente)
- Los lectores pueden acceder concurrentemente entre si, ya que en ningún caso pueden ocasionar problemas de inconsistencia en los datos.
- Los escritores solo pueden acceder en exclusión mutua con otros escritores y con los lectores.
- Es decir:
  1. Cualquier número de lectores puede leer el recurso simultáneamente
  2. Sólo puede escribir en el recurso un escritor en cada instante
  3. Si un escritor está accediendo al recurso, ningún lector puede leerlo.

# Exclusión Mutua: SEMAFOROS

## # Lectores – escritores: Primera solución

```
semaphore cri_lec = 1; //Controla el acceso a la región crítica de los lectores
semaphore db = 1;     //Controla el acceso a la base de datos o recurso compartido.
int cont_lec = 0;      //Contador de lectores leyendo la base de datos
```

```
void lector (void) {
    while (true) {
        wait(cri_lec);
        cont_lec++;
        if (cont_lec == 1) wait(db);
        signal(cri_lec);
        leer_base_datos();
        wait(cri_lec);
        cont_lec--;
        if (cont_lec == 0) signal(db);
        signal(cri_lec);
        usar_datos_leidos();
    }
}
```

```
void escritor(void) {
    while (true) {
        preparar_datos();
        wait(db);
        escribir_base_datos();
        signal(db);
    }
}
```

¿Se puede producir inanición de los escritores?, ¿por qué?, ¿Quién tiene prioridad?

# Exclusión Mutua: SEMAFOROS

## # Lectores – escritores: Segunda solución

```
semaphore cri_lec = 1; //Controla el acceso a la región crítica de los lectores
semaphore cri_esc = 1; //Controla el acceso a la región crítica de los escritores
semaphore db = 1;      //Controla el acceso a la base de datos o recurso compartido.
semaphore perm_lec = 1; //Permite inhibir las lecturas en el momento que un escritor desee escribir.
semaphore lect = 1;     //Encola a los lectores con intención de obtener permiso de lectura
int cont_lec = 0;        //Contador de lectores leyendo la base de datos (critico lectores)
int cont_esc = 0;        //Contador de escritores con intención de escribir, incluyendo el que escribe.
```

```
void lector (void) {
    while (true) {
        wait(lect);
        wait(perm_lec);
        wait(cri_lec);
        cont_lec++;
        if (cont_lec == 1) wait(db);
        signal(cri_lec);
        signal(perm_lec);
        signal(lect);
        leer_base_datos();
        wait(cri_lec);
        cont_lec--;
        if (cont_lec == 0) signal(db);
        signal(cri_lec);
        usar_datos_leidos();
    }
}
```

```
void escritor(void) {
    while (true) {
        preparar_datos();
        wait(cri_esc);
        cont_esc++;
        if (cont_esc == 1) wait(perm_lec);
        signal(cri_esc);
        wait(db);
        escribir_base_datos();
        signal(db);
        wait(cri_esc);
        cont_esc--;
        if (cont_esc == 0) signal(perm_lec);
        signal(cri_esc);
    }
}
```

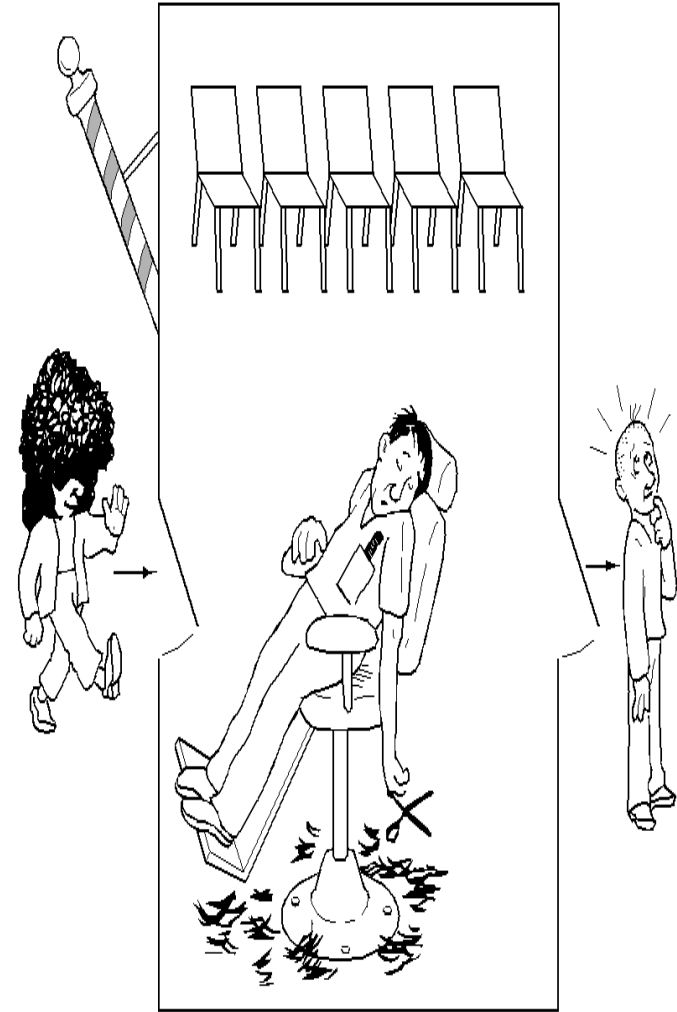
# Exclusión Mutua: SEMAFOROS

## El barbero dormilón.

En la peluquería hay 1 silla de barbero y  $n$  sillas donde los clientes esperan su turno.

El barbero se echa a dormir si no hay clientes esperando.

Cuando llega un cliente despierta al peluquero, si llegan más clientes mientras el peluquero está cortando el pelo, estos se esperan en las sillas de espera, si hay sitio, sino, salen del establecimiento.



# Exclusión Mutua: SEMAFOROS

## El barbero dormilón.

```
# define sillas 5           //Número de sillas de espera para clientes
semaphore clientes = 0;    //Número de clientes esperando ser atendidos
semaphore barberos = 0;   //Número de peluqueros que están ociosos
semaphore mutex = 1;      //Para garantizar la exclusión mutua
int clientes_esperando = 0; //Contador de número de clientes esperando
semaphore begincorte=endcorte=0;
```

```
void barbero (void) {
    while (true) {
        wait(clientes); //a dormir si no hay clientes
        wait(mutex);
        clientes_esperando--;
        signal(barberos); //ahora hay un barbero listo
        signal(mutex);
        cortar_pelo( );
    }
}
```

```
void cliente (void) {
    wait(mutex);
    if (clientes_esperando < sillas) {
        clientes_esperando++;
        signal(clientes); //despierta al barbero
        signal(mutex);
        wait(barberos); //esperar si el peluquero esta ocupado
        sentarse_y_esperar( );
    }
    else signal(mutex);
}
```

```
void cortar_pelo(void) {
    wait(begincorte);
    cortarpelo();
    signal(fincorte);
}
```

```
void sentarse_y_esperar(void) {
    sentarse();
    signal(begincorte);
    wait(fincorte);
}
```

# Exclusión Mutua: SEMAFOROS

## # El puente sobre el Amazonas

Sobre el Amazonas se ha construido un puente de un único carril para minimizar su coste.

Desde cada extremo del puente no se divisa el otro extremo del puente por lo que no se puede saber si hay coches entrando o dentro del puente cuando se intenta pasar.

En los extremos del puente existe una luz roja que indica si se puede pasar o no. Cuando un coche se encuentra la luz apagada podrá entrar al puente. Unos sensores detectarán este hecho y encenderán la luz en el otro extremo, pero se mantiene apagada la luz del extremo por donde entró el coche. Cuando el último coche salga por el otro extremo, la luz se apagará automáticamente en dicho extremo.

Plantear la solución para que puedan pasar los coches sin colisiones utilizando semáforos, decidir cuantos, su tipo y los valores de inicialización.

- No gestionar la inanición ni prioridades
- Los coches no se averían en el puente
- Se cuenta con una función para cruzar el puente.
  - $\text{Cruzar}(A,B) \rightarrow$  Cruza de la orilla A a la B
  - $\text{Cruzar}(B,A) \rightarrow$  Cruza de la orilla B a la A



# Exclusión Mutua: SEMAFOROS

## # El puente sobre el Amazonas

```
semaphore mutexA = 1; //Exclusión mutua para el contador de coches de la orilla A
semaphore mutexB = 1; //Exclusión mutua para el contador de coches de la orilla B
semaphore puente = 1; //Sincroniza el paso por el puente en un sentido u otro
int cochesA=cochesB = 0; //Contadores de numero de coches en cada orilla
```

### CODIGO DE LA ORILLA A

```
void orillaA (void) {
    while (true) {
        wait(mutexA);
        cochesA++;
        if (cochesA==1)
            wait(puente)
        signal(mutexA);
        cruzar(A,B);
        wait(mutexA);
        cochesA--;
        if (cochesA==0)
            signal(puente);
        signal(mutexA);
    }
}
```

### CODIGO DE LA ORILLA B

```
void orillaB (void) {
    while (true) {
        wait(mutexB);
        cochesB++;
        if (cochesB==1)
            wait(puente)
        signal(mutexB);
        cruzar(B,A);
        wait(mutexB);
        cochesB--;
        if (cochesB==0)
            signal(puente);
        signal(mutexB);
    }
}
```

# Exclusión Mutua: SEMAFOROS

## # El puente colgante

En una antigua senda de montaña se ha construido un puente colgante que es capaz de soportar 450Kg máximo sin riesgo de romperse.

Por la senda pasan excursionistas en ambos sentidos y el peso de cada excursionista más el de su mochila son diferentes, pero cada uno lo conoce.

Plantear una solución con semáforos que permita que se cruce el puente por el máximo número de excursionistas posible siempre que no se ponga en peligro su integridad física.

- Plantear un código que deberían ejecutar cada excursionista (proceso) antes de pasar el puente (recurso con límite de carga) independientemente del sentido de cruce.
- Indicar claramente que variables y semáforos comparten los procesos y cuáles son los valores iniciales.
- Considerar que cada proceso tiene una variable local llamada *mipeso* que tiene el peso del excursionista al que representa más el de su mochila.

# Exclusión Mutua: SEMAFOROS

## # El puente colgante

```
bool esperando = false //Indica si hay un excursionista esperando a pasar que ya restado su peso
int kilos = 750;       //Número de kilos que soporta el puente
semaphore s1=1         //Para parar a los demás excursionistas cuando uno ya detecta que no puede pasar
semaphore mutex=1      //Para la exclusión mutua
semaphore s2=0;        //Para colgar al que detecta que no puede pasar
```

EXCURSIONISTA

...

...

CodigoPre();

Cruzar();

CodigoPost();

...

...

```
void CodigoPre(void) {
```

```
    wait(s1);
```

```
    wait(mutex);
```

```
    kilos-=mipeso;
```

```
    if (kilos<0){
```

```
        esperando=true;
```

```
        signal(mutex);
```

```
        wait(s2);
```

```
        signal(s1);
```

```
    }
```

```
    else {
```

```
        signal(mutex);
```

```
        signal(s1);
```

```
    }
```

```
}
```

```
void CodigoPost(void) {
```

```
    wait(mutex);
```

```
    kilos+=mipeso;
```

```
    if (kilos>=0 && esperando){
```

```
        esperando=false;
```

```
        signal(s2);
```

```
    }
```

```
    signal(mutex);
```

```
}
```

Si uno ya no puede pasar inhabilita a los que vengan detrás que si pudieran pasar. Hasta que no se libere el peso suficiente (el del que espera) nadie más puede pasar.

# Exclusión Mutua: SEMAFOROS

## El puente colgante – 2ª Versión

```
int kilos = 750;           //Número de kilos que soporta el puente
semaphore mutex=1         //Para la exclusión mutua
semaphoreb s[100]={0};    //Para colgar a los que detectan que no puede pasar
semaphoreb c=0;           //Para dormir al controler
cont_waiting=0;           //Para contar los que están esperando a probar si pueden pasar
```

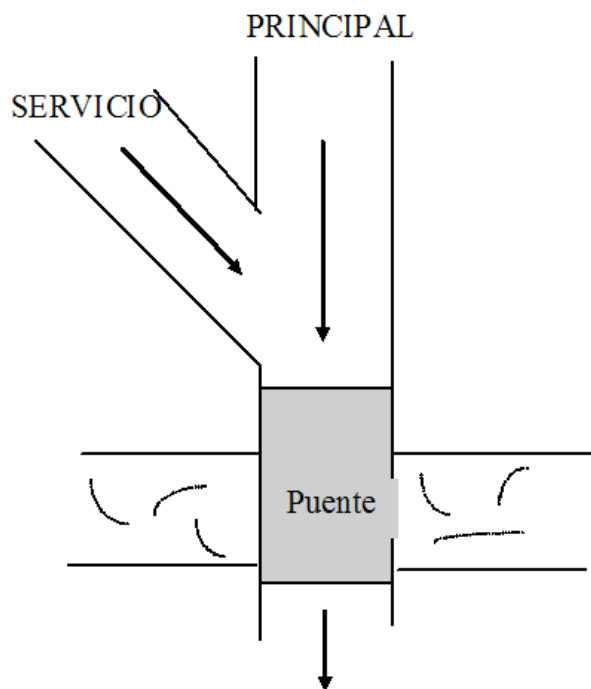
Ejecución paralela de N excursionistas y 1 controler.

### EXCURSIONISTA

```
mipeso=X;
...
CodigoPre();
Cruzar();
CodigoPost();
...
...
void CodigoPre(void) {
    int cruzando=false;
    while (!cruzando){
        wait(mutex);
        if (kilos-mipeso<0){
            cont_waiting++;
            signal(mutex);
            wait(s[cont_waiting]);
        }
        else {
            kilos-=mipeso;
            signal(mutex);
            cruzando=true;
        }
    }
}
```

### CONTROLER

```
void Controler(void){
    int i;
    while (true){
        wait(mutex);
        for (i=0;i<cont_waiting;i++){
            signal(s[i]);
        }
        cont_waiting=0;
        signal(mutex);
        wait(c);
    } //while
}
```



2. (3 ptos) En la entrada a un antiguo puente de madera unidireccional confluyen dos carreteras, la carretera principal y la carretera de servicio. Los vehículos que vienen por ambas carreteras deben cruzar el puente pero tienen que tener en cuenta las siguientes consideraciones:

- a) El puente solo puede soportar el peso de 5 vehículos
- b) Cuando haya vehículos en la carretera principal que quieran cruzar el puente tendrán siempre prioridad frente a los vehículos de la carretera de servicio
- c) Los vehículos que estén en la carretera de servicio podrán empezar a cruzar el puente cuando no exista ningún vehículo en la carretera principal que quiera cruzar el puente.

Implementa el código que gestiona la circulación de los vehículos mediante semáforos, explicando la implementación realizada.

Semáforos:

np=0; //contador de numero de coches en carretera principal

binario mutex=0; //para exclusión mutua contador

binario cs=0; //Corte de carretera de servicio

entero p=5; //Entrada al puente

binario servicio=1;

Principal

```
wait(mutex);  
  np++;  
signal(mutex);
```

```
wait(p);
```

```
wait(mutex);  
  np--;  
  if np==0 then  
    signal(cs);  
  signal(mutex);
```

```
---puente----
```

```
signal(p);
```

Servicio

```
wait(servicio)  
wait(mutex);  
if np>0 then  
  signal(mutex);  
  wait(cs);  
else  
  signal(mutex);  
signal(servicio)
```

```
wait(p);
```

```
---puente---
```

```
signal(p);
```

Var		
a,b:integer;		
s1, s2: semaforo;		
 procedure P1;	procedure P2;	Begin
begin	begin	s1:=1; s2:=1;
wait(s1),	wait(s2),	a:=1; b:=2;
a:=a+1;	wait(s1);	cobegin
b:=2*a;	a:=a+2;	p1;
signal(s1);	signal(s1);	p2;
end;	b:=2*a;	coend;
	signal(s2);	writeln(b/a);
	end;	end;

- a) El programa puede presentar interbloqueos.
- b) La ejecución del programa siempre dará como resultado la impresión de un 2.
- c) La ejecución del programa siempre dará como resultado la impresión de un 2 o un 4.
- d) La ejecución del programa no siempre dará como resultado un múltiplo de 2.

11.- Sea el siguiente programa concurrente. ¿Qué salida produciría?

```
Procedure P1;  
var  
i:integer;  
begin  
for i:=1 to 2 do  
begin  
wait(s1);  
write(i);  
signal(s2);  
end  
end;  
end;
```

```
Procedure P2;  
var  
i:char;  
begin  
for i:='A' to 'B' do  
begin  
wait(s2);  
write(i);  
signal(s1);  
end  
end;  
end;
```

```
begin  
s1:=0;  
s2:=1;  
cobegin  
p1;  
p2;  
coend;  
end.
```

- a) 1A2B
- b) A1B2
- c) AB12
- d) Cualquier combinación de los caracteres 1, 2, A y B.



Sea el siguiente programa:

Program Ejemplo;

Var

a,b:integer;

s1, s2: semaforo;

procedure P1;

begin

wait(s1),

a:=a+1;

if a==2 then wait(s2);

b:=2\*a;

signal(s2);

signal(s1);

end;

procedure P2;

begin

wait(s2),

wait(s1);

a:=a+2;

signal(s1);

b:=2\*a;

signal(s2);

end;

Begin

s1:=1; s2:=1;

a:=1; b:=2;

cobegin

p1;

p2;

coend;

writeln(b/a);

end;

- a) El programa puede presentar interbloqueos.
- b) La ejecución del programa siempre dará como resultado la impresión de un 2.
- c) La ejecución del programa siempre dará como resultado la impresión de un 2 o un 4.
- d) La ejecución del programa no siempre dará como resultado un múltiplo de 2.

# Exclusión Mutua: SEMAFOROS

## ■ Ejercicio propuesto

Sea una carretera en la que hay una zona de paso para peatones.

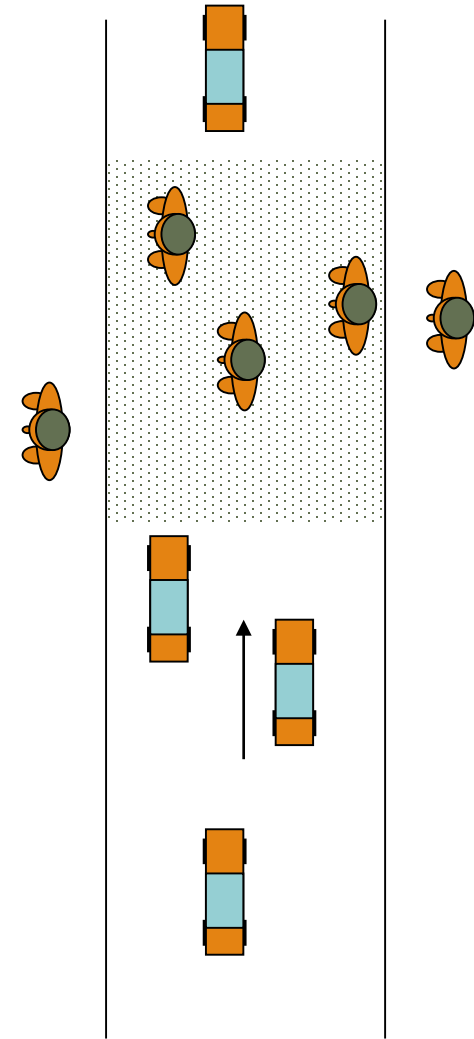
La zona de cruce es suficientemente ancha para que quepan varios coches dentro, es decir, pueden estar sobre ella varios coches a la vez y por supuesto varios peatones a la vez.

Los peatones tienen preferencia, de forma que mientras haya peatones cruzando, los coches se pararán y esperarán a que el último peatón abandone la zona de cruce.

Los coches podrán pasar mientras no llegue ningún peatón, cuando llegue uno, se pararán y esperarán a que el peatón abandone la zona de cruce.

Los peatones son desconfiados. Cuando llega un peatón a la zona de cruce, es el primero que va a pasar y hay coches en la carretera, aunque tiene prioridad y podría pasar sin dudar, el peatón se cura en salud y se para hasta que el coche pare, es decir, el peatón para si hay coches y cuando el coche para le cede el paso al peatón y este cruza.

Una vez que ya hay peatones cruzando, los siguientes peatones ya no tienen miedo y cruzan sin dudar aunque haya coches en la carretera.



# ... lo siguiente

- MONITORES

## ... más Info

En la web de la asignatura <http://so.momrach.es>

- Tenéis un [cuaderno con ejercicios de concurrencia](#) resueltos:
  - De semáforos
  - De monitores
  - De paso de mensajes
- Una entrada [Semáforos](#)
  - Con la explicación y ejemplo **funcional** de cómo se programan los semáforos en C
  - Una librería (funcional) que facilita el uso al estilo de la teoría
- Una entrada [Sincronización de Padre-Hijos con semáforos](#)
  - Con código **funcional** en C para sincronizar padre e hijos con semáforos **usando la librería anterior**

Stallings – Sistemas Operativos Principios de Diseño e Interioridades (5ª ed.)

- En el punto 5.3 se tratan los semáforos
- En el punto 5.6 se presenta el problema de los Lectores/Escritores

Jesús Carretero – Sistemas Operativos una Visión Aplicada

- En el punto 5.3.2 Explica las Tuberías y cómo se implementa la sección crítica con ellas  
Productor-Consumidor con Tuberías
- En el punto 5.3.4 Trata los semáforos y los Lectores/Escritores

# Fin

---

## UNIDAD 3

### EXCLUSIÓN MUTUA Y SINCRONIZACIÓN CON SEMÁFOROS