

A decorative frame consisting of two horizontal lines and two vertical lines, with small circular end caps at each corner.

# Sistemas operativos

## Tema 2d: Concurrencia



# Conceptos

---

- Multiprogramación
  - Gestión de varios procesos en un sistema monoprocesador
- Multiprocesamiento
  - Gestión de varios procesos en un sistema multiprocesador
- Procesamiento distribuido
  - Gestión de varios procesos ejecutándose en sistemas de computadores múltiples o distribuidos. p.ej. Clusters

# Contextos concurrentes

---

- La concurrencia se puede presentar en tres contextos diferentes
  - Múltiples Aplicaciones
    - Compartir procesador entre varias aplicaciones activas.
  - Aplicaciones Estructuradas
    - Una misma aplicación puede ser estructurada como un conjunto de procesos concurrentes
  - Estructura del Sistema Operativo
    - Algunos S.O. están estructurados como un conjunto de procesos concurrentes o hilos.

# Principales dificultades

---

## ● Sistemas monoprocesador

- Compartir recursos globales esta lleno de riesgos
  - El orden de las lecturas y escrituras es crítico
- Difícil la asignación óptima de recursos
  - Un proceso que adquiere el control de un canal de E/S y antes de usarlo se suspende. ¿Qué hace el S.O. con el canal?
- Difícil la depuración
  - Los resultados de distintas ejecuciones no son ni deterministas ni reproducibles.

## ● Sistemas multiprocesador

- Además se problemas propios por la ejecución simultanea de procesos

# Ejemplo

- Sea un sistema multiprogramado (monoprocesador o multiprocesador)

Ejemplo:

```
global char in;  
void echo ( ) {  
    in = getchar();  
    printf("%c", in);  
}
```

- Ejecutamos 2 veces (2 procesos)
- Cambio se produce en cualquier momento
- ¿Cual será la salida? ¿Cual será la solución?

| P1               | P2               |
|------------------|------------------|
| in = getchar();  |                  |
|                  | in = getchar();  |
|                  | printf("%c",in); |
| printf("%c",in); |                  |

# Interacción entre procesos

- Los procesos no tienen conocimiento de los demás
  - Son procesos independientes que no están pensados para operar juntos.
  - Multiprogramación de procesos independientes
    - Trabajos por lotes
    - Sesiones de distintos usuarios
  - El SO debe encargarse de la competencia por los recursos
  - Se trata de **Procesos No Cooperantes. Competidores.**
- Los procesos tienen un conocimiento indirecto de los demás
  - No conocen a los otros procesos por su PID
  - Comparten el uso de algunos recursos, por ejemplo, buffers de E/S.
  - Se trata de **Procesos Cooperantes por Compartimiento**
- Los procesos tienen un conocimiento directo de los demás
  - Se conocen por su PID
  - Son capaces de comunicarse entre sí
  - Diseñados para cooperar en alguna actividad en concreto
  - Se trata de **Procesos Cooperantes por Comunicación.**

# Exclusión Mutua



# Competencia por los recursos

- **Procesos No Cooperantes. Competidores** entran en conflicto cuando compiten por el mismo recurso.
  - Los procesos utilizan los recursos y los dejan como estaban.
  - Los procesos pueden verse afectados por asignaciones de recursos.
  - Si hay competencia se plantean 3 problemas a resolver
    - Exclusión Mútua
      - Recursos Críticos
      - Sección Crítica
    - Interbloqueo
    - Inanición
  - El SO debe actuar
  - Los procesos deben ser proactivos.

```
/* program ExclusiónMutua */
const int n = /* número de procesos */;

void P(int i)
{
    while (cierto)
    {
        entrada_crítica(i);
        /* sección crítica */;
        salida_crítica(i);
        /* resto */;
    }
}

void main()
{
    parbegin(P(R1), P(R2), ..., P(Rn));
}
```

**Figura 5.1.** Exclusión mutua.

# Competencia por los recursos

- **Procesos Cooperantes por Compartición** los procesos son conscientes de que otros pueden acceder a los recursos compartidos.
- Los procesos deben cooperar para garantizar la integridad de los datos compartidos.
- Dos tipos de acceso a los datos (lectura – escritura)
  - Los problemas de Exclusión Mutua, Interbloqueo e Inanición también se dan.
  - Solo las operaciones de escritura deben ser mutuamente excluyentes.
- La coherencia de los datos debe mantenerse

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

$a = a + 1;$   
 $b = 2 * b;$   
 $b = b + 1;$   
 $a = 2 * a;$

|       |    |       |    |
|-------|----|-------|----|
| a     | 3  | a     | 3  |
| b     | 5  | b     | 5  |
| a=a+1 | 4  | a=a+1 | 4  |
| b=b+1 | 6  | b=2*b | 10 |
| b=2*b | 12 | b=b+1 | 11 |
| a=2*a | 8  | a=2*a | 8  |
| a     | 8  | a     | 8  |
| b     | 12 | b     | 11 |

# Competencia por los recursos

- **Procesos Cooperantes por Comunicación** los procesos son concedores de la identidad de los otros procesos.
  - Los procesos están diseñados para colaborar en la consecución de un fin
  - Para ello necesitan comunicarse
    - Para sincronizarse
    - Para coordinarse
  - Normalmente se utilizan primitivas de paso de mensajes
    - Del lenguaje de programación (clusters pvm mpi)
    - Del Kernel del Sistema Operativo (send receive)
  - No se comparte nada, por tanto, no es necesaria la exclusión mutua.
  - Los problemas de Interbloqueo e Inanición siguen presentes

# Requisitos de la exclusión mútua

---

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

# Exclusión Mutua: Soluciones

---

- La exclusión mutua se puede conseguir con
  - Soluciones Software (Espera activa)
    - Algoritmo de Dekker
  - Soluciones Hardware
  - Soluciones con soporte del S.O. o el Lenguaje de programación

A decorative frame consisting of two horizontal lines and two vertical lines, with small circular end caps at each corner.

# Exclusión Mutua

Soluciones Software



# Algoritmo de Dekker

## ● Espera activa:

- Un proceso siempre está esperando a entrar en su sección crítica.
- Un proceso no puede hacer nada productivo hasta que obtiene permiso para entrar en su sección crítica.

| <code>/* PROCESO 0 */</code>            | <code>/* PROCESO 1 */</code>            |
|-----------------------------------------|-----------------------------------------|
| <code>...</code>                        | <code>...</code>                        |
| <code><b>while</b> (turno != 0 )</code> | <code><b>while</b> (turno != 1 )</code> |
| <code>    /* no hacer nada*/;</code>    | <code>    /* no hacer nada*/;</code>    |
| <code>/*sección crítica*/;</code>       | <code>/*sección crítica*/;</code>       |
| <code>turno = 1 ;</code>                | <code>turno = 0 ;</code>                |
| <code>...</code>                        | <code>...</code>                        |

Garantiza la exclusión mutua.

## Problemas

1. Alternancia estricta. El ritmo de ejecución lo marca el más lento
2. Un fallo en un proceso provoca un Interbloqueo. Tanto si el fallo es en la Sección Crítica o no.

## ● Corrutinas

- Diseñadas para poder pasar el control de la ejecución entre ellas.
- No es una técnica apropiada para dar soporte al procesamiento concurrente.

# Algoritmo de Dekker

| <code>/* PROCESO 0 */</code>                    | <code>/* PROCESO 1 */</code>                    |
|-------------------------------------------------|-------------------------------------------------|
| <code>...</code>                                | <code>...</code>                                |
| <code><b>while</b> (señal[1] )</code>           | <code><b>while</b> (señal[0] )</code>           |
| <code>  <code>/* no hacer nada*/;</code></code> | <code>  <code>/* no hacer nada*/;</code></code> |
| <code>  señal[0] := true;</code>                | <code>  señal[1] := true;</code>                |
| <code>  <code>/*sección crítica*/</code></code> | <code>  <code>/*sección crítica*/</code></code> |
| <code>  señal[0] := false;</code>               | <code>  señal[1] := false;</code>               |
| <code>...</code>                                | <code>...</code>                                |

Si el proceso falla fuera de su sección crítica, el otro puede continuar. (fallo excluye las asignaciones)

Si el proceso falla en su sección crítica, el otro se bloquea.

No garantiza la exclusión mutua

P0 while y encuentra señal[1]=false

P1 while y encuentra señal[0]=false

P0 señal[0]=true y entra

P0 señal[1]=true y entra

- Cada proceso debe tener la llave de su sección crítica. Si el otro falla podrá continuar.
- Cada proceso puede examinar el estado del otro pero no lo puede alterar.
- Cuando un proceso desea entrar en su sección crítica comprueba en primer lugar el otro proceso.
- Si no hay otro proceso en su sección crítica fija su estado para la sección crítica.
- Este método no garantiza la exclusión mutua.
- Cada proceso puede comprobar las señales y luego entra en la sección crítica al mismo tiempo.

# Algoritmo de Dekker

|                                                                                                                                              |                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>/* PROCESO 0 */ ... señal[0] := true; <b>while</b> (señal[1] )     /* no hacer nada*/; /*sección crítica*/ señal[0] := false; ...</pre> | <pre>/* PROCESO 1 */ ... señal[1] := true; <b>while</b> (señal[0] )     /* no hacer nada*/; /*sección crítica*/ señal[1] := false; ...</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|

Si el proceso falla fuera de su sección crítica, el otro puede continuar. (fallo excluye las asignaciones)

Si el proceso falla en su sección crítica, el otro se bloquea.

Garantiza la exclusión mutua

Problema

**Se provocará un interbloqueo cuando dos procesos den valor a sus señales antes de entrar en la sección crítica.**

- El anterior falla porque se permite cambiar el estado después de la consulta del otro proceso
- Dar valor a la señal para entrar en la sección crítica antes de comprobar otros procesos.
- Si hay otro proceso en la sección crítica cuando se ha dado valor a la señal, el proceso queda bloqueado hasta que el otro proceso abandona la sección crítica.

# Algoritmo de Dekker

| <code>/* PROCESO 0 */</code>      | <code>/* PROCESO 1 */</code>      |
|-----------------------------------|-----------------------------------|
| <code>...</code>                  | <code>...</code>                  |
| <code>señal[0] := true;</code>    | <code>señal[1] := true;</code>    |
| <code>while (señal[1]) {</code>   | <code>while (señal[0]) {</code>   |
| <code>  señal[0] := false;</code> | <code>  señal[1] := false;</code> |
| <code>  /*esperar*/</code>        | <code>  /*esperar*/</code>        |
| <code>  señal[0] := true;</code>  | <code>  señal[1] := true;</code>  |
| <code>}</code>                    | <code>}</code>                    |
| <code>/*sección crítica*/</code>  | <code>/*sección crítica*/</code>  |
| <code>señal[0] := false;</code>   | <code>señal[1] := false;</code>   |
| <code>...</code>                  | <code>...</code>                  |

Casi correcta. Se produce un **livelock (bloqueo vital)**

P0 pone señal[0] a cierto.

P1 pone señal[1] a cierto.

P0 comprueba señal[1].

P1 comprueba señal[0].

P0 pone señal[0] a falso.

P1 pone señal[1] a falso.

P0 pone señal[0] a cierto.

P1 pone señal[1] a cierto.

Un cambio en la velocidad relativa resuelve la situación.

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

# Algoritmo de Dekker: Solución correcta

```
boolean señal[2];
int turno;
void P0() {
    while (true){
        señal[0]:=true;
        while(señal[1])
            if turno==1{
                señal[0]:=false;
                while (turno==1)
                    /*no hacer nada*/
                señal[0]:=true;
            }
        /*sección crítica*/
        turno:=1;
        señal[0]:=false;
        /*resto*/
    }
}
```

```
void P1() {
    while (true){
        señal[1]:=true;
        while(señal[0])
            if turno==0{
                señal[1]:=false;
                while (turno==0)
                    /*no hacer nada*/
                señal[1]:=true;
            }
        /*sección crítica*/
        turno:=0;
        señal[1]:=false;
        /*resto*/
    }
}
```

```
void main(){
    señal[0]:=false;
    señal[1]:=false;
    turno:=1
    parbegin(P0, P1)
}
```

- Cada proceso tiene un turno para entrar en la sección crítica.
- Si un proceso desea entrar en la sección crítica, debe activar su señal y puede que tenga que esperar a que llegue su turno.



# Exclusión Mutua

Soluciones Hardware



# Exclusión Mutua: Soluciones Hardware

## ■ Inhabilitación de interrupciones:

- Un proceso continuará ejecutándose hasta que solicite un servicio del sistema operativo o hasta que sea interrumpido.
- Para garantizar la exclusión mutua es suficiente con impedir que un proceso sea interrumpido.

```
while true{  
    /* inhabilitar interrupciones */  
    /* seccion critica */  
    /* habilitar interrupciones */  
    /* resto; }  
}
```

## ■ Problemas:

- Se limita la capacidad del procesador para intercalar programas.
- Multiprocesador:
  - Inhabilitar las interrupciones de un procesador no garantiza la exclusión mutua.

# Exclusión Mutua: Soluciones Hardware

---

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

# Exclusión Mutua: Soluciones Hardware

## ■ La instrucción Comparar y Fijar (Test and Set)

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

```
/* program exclusionmutua */
const int n= /*numero de procesos*/
int cerrojo;
void P(int i) {
    while (true) {
        while(! TS(cerrojo))
            /*no hacer nada*/
        /*Seccion crítica*/
        cerrojo := 0;
        /*resto*/
    }
}
void main (){
    cerrojo := 0;
    parbegin(P(1), P(2), ..... , P(n));
}
```

# Exclusión Mutua: Soluciones Hardware

## ▣ La instrucción Intercambiar

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

```
/* program exclusionmutua */
const int n= /*numero de procesos*/
int cerrojo;
void P(int i) {
    int clave
    while (true) {
        clave :=1;
        while( clave != 0 )
            intercambiar(clave, cerrojo)
        /*Seccion crítica*/
        intercambiar(clave,cerrojo);
        /*resto*/
    }
}
void main (){
    cerrojo := 0;
    parbegin(P(1), P(2), ..... , P(n));
}
```

# Exclusión Mutua: Soluciones Hardware

---

## # Ventajas:

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

# Exclusión Mutua: Soluciones Hardware

---

## # Desventajas:

- La espera activa consume tiempo del procesador.
- Puede producirse inanición cuando un proceso abandona la sección crítica y hay más de un proceso esperando.
- Interbloqueo:
  - Si un proceso con baja prioridad entra en su sección crítica y existe otro proceso con mayor prioridad, entonces el proceso cuya prioridad es mayor obtendrá el procesador para esperar a poder entrar en la sección crítica.



# Exclusión Mutua

Soluciones Soporte SO



# Exclusión Mutua: Soluciones Soporte SO

---

- **SEMAFOROS**
- **MONITORES**
- **PASO DE MENSAJES**



# Exclusión Mutua

Soluciones Soporte SO  
SEMAFOROS



# Exclusión Mutua: SEMAFOROS

---

- El funcionamiento se resume así:
  - Dos o más procesos pueden cooperar por medio de señales.
  - De forma que se pueda obligar a detener a un proceso en una posición determinada hasta que reciba una señal.
- Quien detiene el proceso es el S.O.

# Exclusión Mútua: SEMAFOROS

---

- Para la señalización se utiliza una variable especial llamada **semáforo**.
- Si un proceso está esperando una señal, el proceso es suspendido hasta que tenga lugar la transmisión de la señal.
- Las operaciones *wait* y *signal* no se pueden interrumpir.
- Se emplea una cola para mantener los procesos esperando en el semáforo.

# Exclusión Mutua: SEMAFOROS

- Un semáforo es una variable que tiene un valor entero:
  - Puede iniciarse con un valor **no negativo**. Es decir  $\geq 0$
  - La operación **wait** disminuye el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta el **wait** se bloquea.
  - La operación **signal** incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea un proceso bloqueado por la operación **wait**.

```
struct semaforo {  
    int contador  
    tipoCola cola;  
}  
  
void wait(semaforo s){  
    s.contador--;  
    if (s.contador < 0){  
        poner este proceso en s.cola;  
        bloquear este proceso  
    }  
}  
  
void signal(semaforo s){  
    s.contador ++;  
    if (s.contador <=0) {  
        quitar un proceso P de la cola s.cola;  
        poner el proceso P en la cola de listos;  
    }  
}
```

# Exclusión Mutua: SEMAFOROS

## ● Semaforo binario

- Solo puede tomar valores 0 o 1
- Tambien tiene un una cola donde esperan los procesos

```
struct semaforo {  
    eunum( 0, 1) valor  
    tipoCola cola;  
}  
  
void waitB(semaforoB s){  
    if s.valor == 1  
        s.valor = 0;  
    else{  
        poner este proceso en s.cola;  
        bloquear este proceso  
    }  
}  
  
void signalB(semaforo s){  
    if s.cola.esvacia()  
        s.valor = 1;  
    else{  
        quitar un proceso P de la cola s.cola;  
        poner el proceso P en la cola de listos;  
    }  
}
```

# Exclusión Mutua: SEMAFOROS

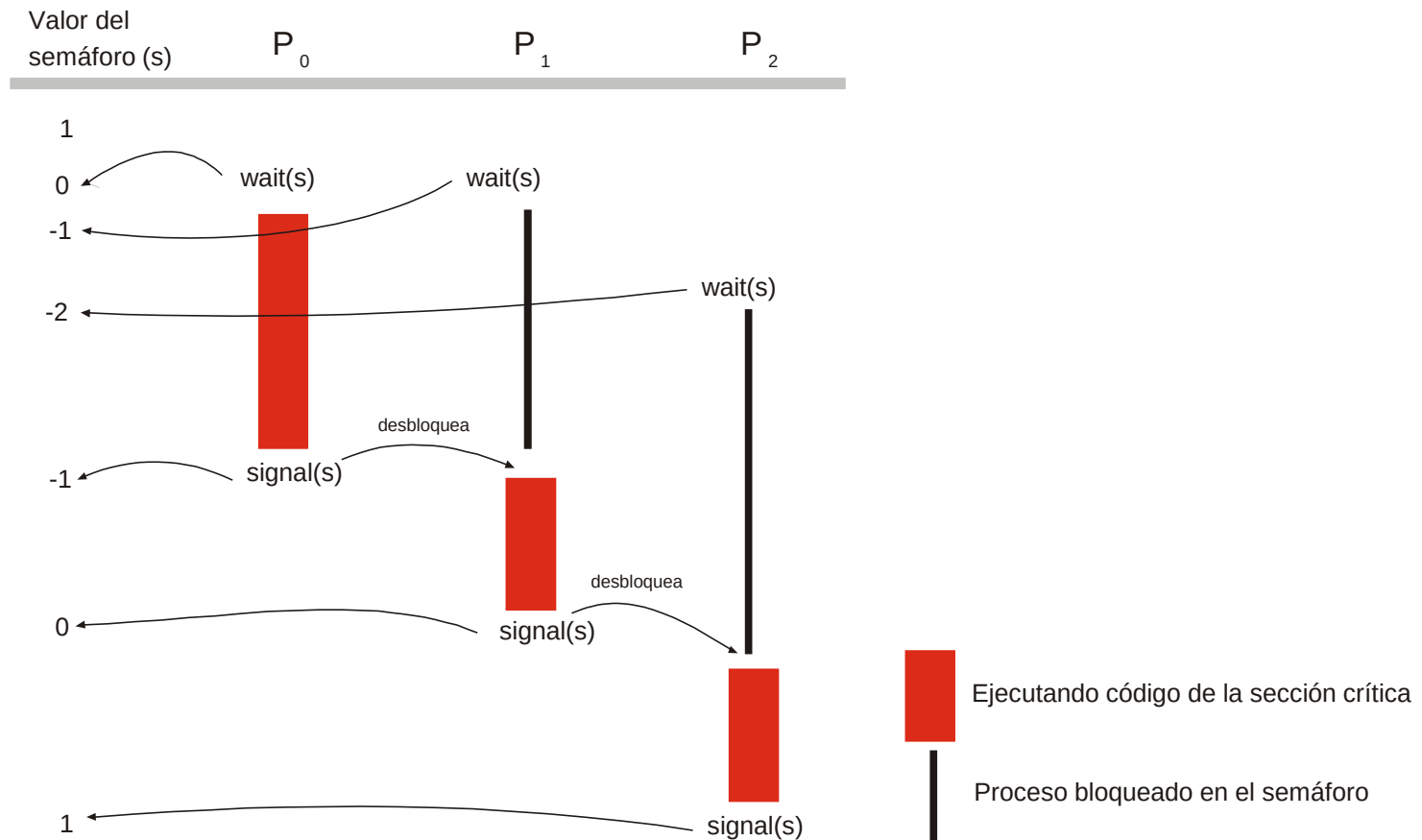
```
/* program exclusión m  tua */  
const int n = /*numero de procesos*/;  
semaforo s = 1;  
  
void main(){  
    parbegin( P(0), P(1) .... P(n) );  
}
```

```
void P(int i){  
    while true {  
        wait(s);  
        /* seccion critica*/;  
        signal(s);  
        /*resto*/  
    }  
}
```

- El sem  foro se inicializa a **1** as  , el primer proceso que ejecute wait podr   entrar poniendo s=0
- Cualquier otro proceso que quiera entrar se bloquear   en el sem  foro.
- Cualquier numero de procesos puede querer entrar, se encolar  n en el sem  foro decrementando s
- Cuando el primer proceso salga de la secci  n cr  tica incrementara s. Si es negativa, es decir, si hay procesos esperando en la cola del sem  foro se seleccionar   uno para su paso a Listo.

# Exclusión Mutua: SEMAFOROS

```
wait(s); /* entrada en la seccion critica */  
< seccion critica >  
signal(s); /* salida de la seccion critica */
```



# Exclusión Mutua: SEMAFOROS

---

- Se pueden clasificar los semáforos en función de si establecen o no la política de selección de procesos de su cola.
  - Semáforos robustos
    - Definen una política de extracción que garantiza la ausencia de inanición.
  - Semáforos débiles
    - No definen una política de extracción
- La política más equitativa es la FIFO

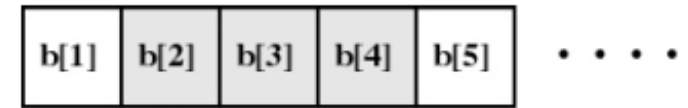
# Exclusión Mutua: SEMAFOROS

## PROBLEMA

### PRODUCTOR/CONSUMIDOR

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

Suponemos inicialmente un buffer ilimitado `b[]`



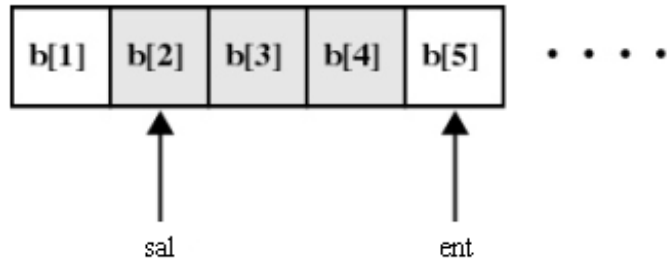
**productor:**

```
while (true) {  
    /* producir elemento v */  
    b[ent] = v;  
    ent++;  
}
```

**consumidor:**

```
while (true) {  
    while (ent <= sal)  
        /*no hacer nada */;  
    w = b[sal];  
    sal++;  
    /* consumir elemento w */  
}
```

# Exclusión Mútua: SEMAFOROS



Buffer Ilimitado

```
/* program productor_consumidor */  
int n;  
semaforoB s = 1;  
semaforoB retraso = 0;  
  
void main(){  
    n=0;  
    parbegin(productor, consumidor);  
}
```

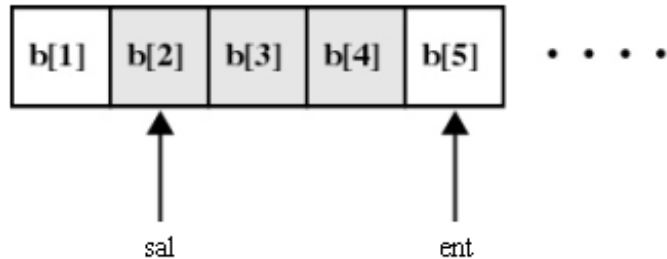
```
void productor(){  
    while true {  
        producir();  
        waitB(s);  
        añadir();  
        n++;  
        if n==1  
            signalB(retraso);  
        signalB(s);  
    }  
}
```

```
void consumidor () {  
    waitB(retraso);  
    while true {  
        waibB(s);  
        coger();  
        n--;  
        signalB(s);  
        consumir();  
        if n==0  
            waitB(retraso);  
    }  
}
```

# Exclusión Mutua: SEMAFOROS

## Buffer Ilimitado

Una solución correcta con  
Semáforos Binarios



```
/* program productor_consumidor */  
int n;  
semaforoB s = 1;  
semaforoB retraso = 0;  
  
void main(){  
    n=0;  
    parbegin(productor, consumidor);  
}
```

```
void productor(){  
    while true {  
        producir();  
        waitB(s);  
        añadir();  
        n++;  
        if n==1  
            signalB(retraso);  
        signalB(s);  
    }  
}
```

```
void consumidor () {  
    waitB(retraso);  
    while true {  
        waibB(s);  
        coger();  
        n--;  
        m=n;  
        signalB(s);  
        consumir();  
        if m==0  
            waitB(retraso);  
    }  
}
```

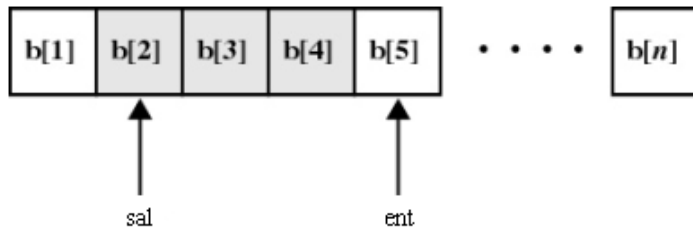
# Exclusión Mutua: SEMAFOROS

## PROBLEMA

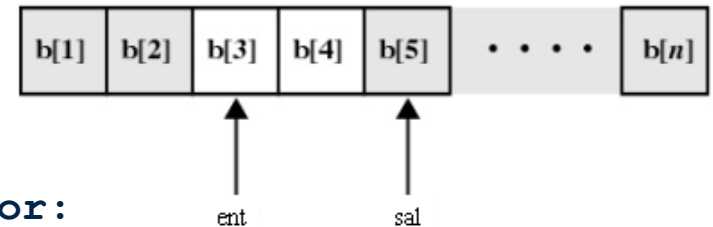
### PRODUCTOR/CONSUMIDOR

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

Suponemos un buffer circular limitado  $b[]$

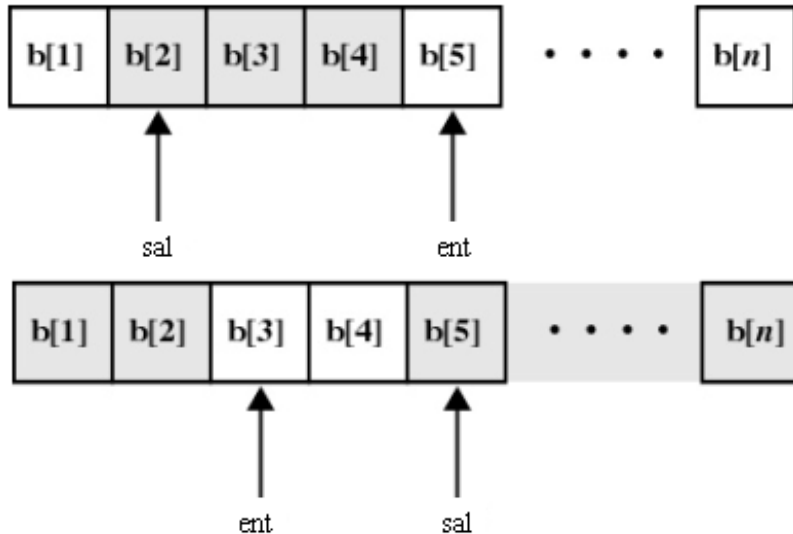


```
productor:
while (true) {
    /* producir elemento v */
    while((ent+1)%n == sal)
        /*no hacer nada*/
    b[ent] = v;
    ent=(ent+1)%n;
}
```



```
consumidor:
while (true) {
    while (ent == sal)
        /*no hacer nada */;
    w = b[sal];
    sal=(sal+1)%n;
    /* consumir elemento w */
}
```

# Exclusión Mutua: SEMAFOROS



Buffer Circular Limitado

Una solución correcta con Semaforos

```
/* program productor_consumidor */
int n;
semaforo s = 1;
semaforo e = 0;
semaforo e = tamañobuffer;

void main(){
    parbegin(productor, consumidor);
}
```

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

```
void consumidor () {
    while true {
        wait(n);
        wait(s);
        coger();
        signal(s);
        signal(e);
        consumir();
    }
}
```

- Implementación de semáforos
  - Es imprescindible que las operaciones wait y signal sean implementadas como primitivas atómicas.
    - Implementación Hardware o Firmware
    - Utilización de Test & Set o Intercambiar
    - Inhabilitación de Interrupciones

# Exclusión Mutua: SEMAFOROS

## ● 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);
}
```

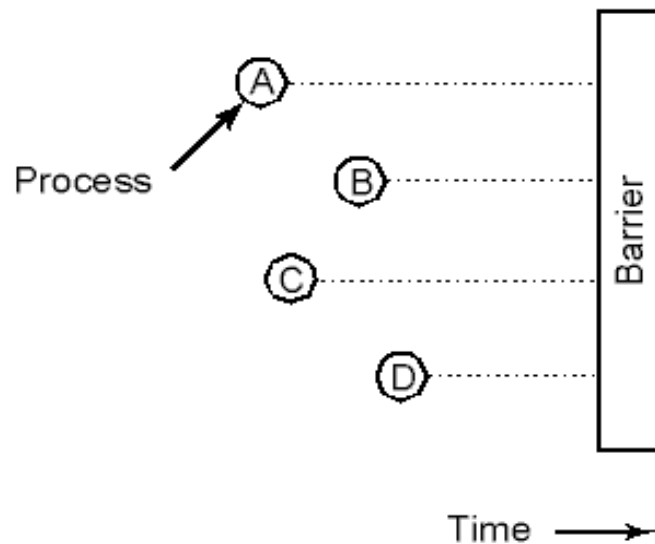
```
PROCESO 2
{
wait(p1);
<codigo proceso2>
}
```

El semáforo se inicializará a 0

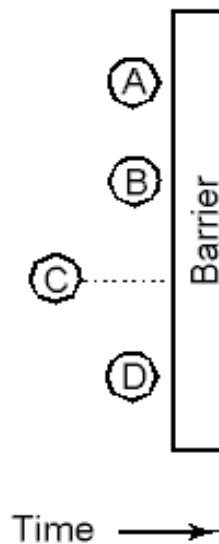
# Exclusión Mutua: SEMAFOROS

## ● Barreras

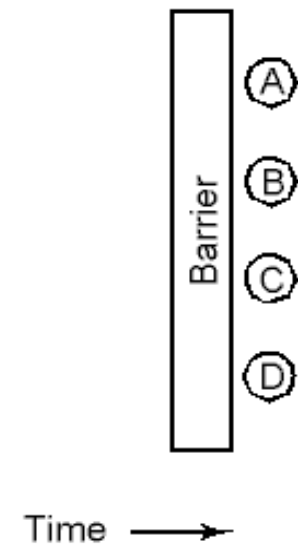
- Una barrera es el punto en el código tal que ningún proceso lo traspasa hasta que todos los procesos han llegado a su ejecución.
- Si y solo si todos los procesos llegan a la barrera, continúan ejecutándose.



(a)



(b)



(c)

# Exclusión Mutua: SEMAFOROS

## ● 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: SEMAFOROS

---

## ● Ejercicio:

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

Explicar detalladamente la solución alcanzada.

# Exclusión Mutua: SEMAFOROS

## ● 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 número de procesos.

### PROCESO $P_i$

```
....  
<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);  
  }  
}
```

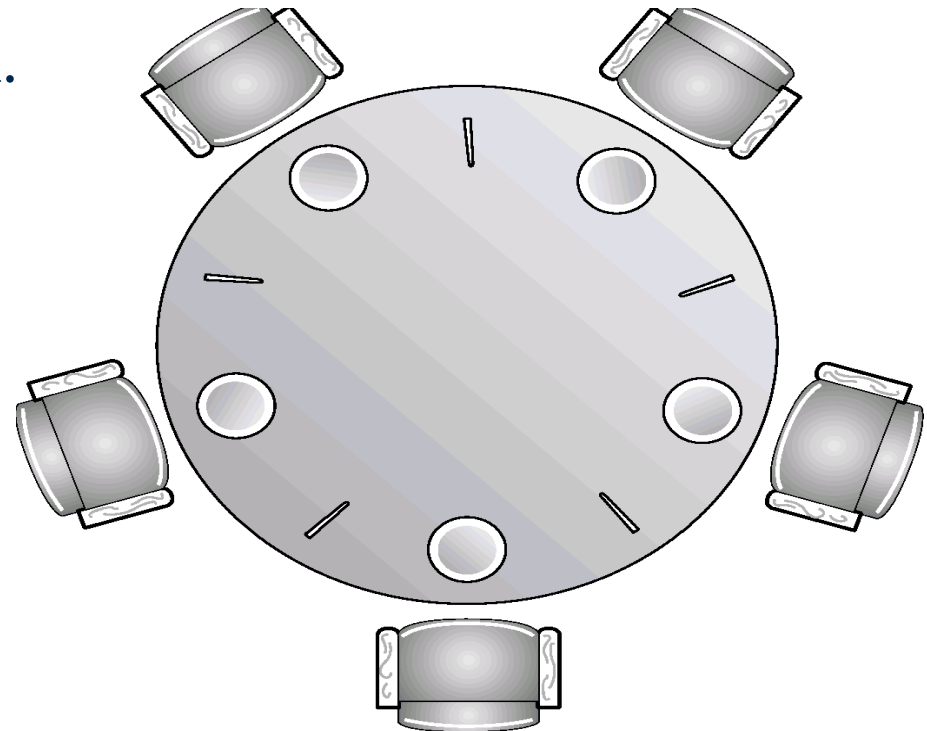
# Exclusión Mutua: SEMAFOROS

## ● 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?

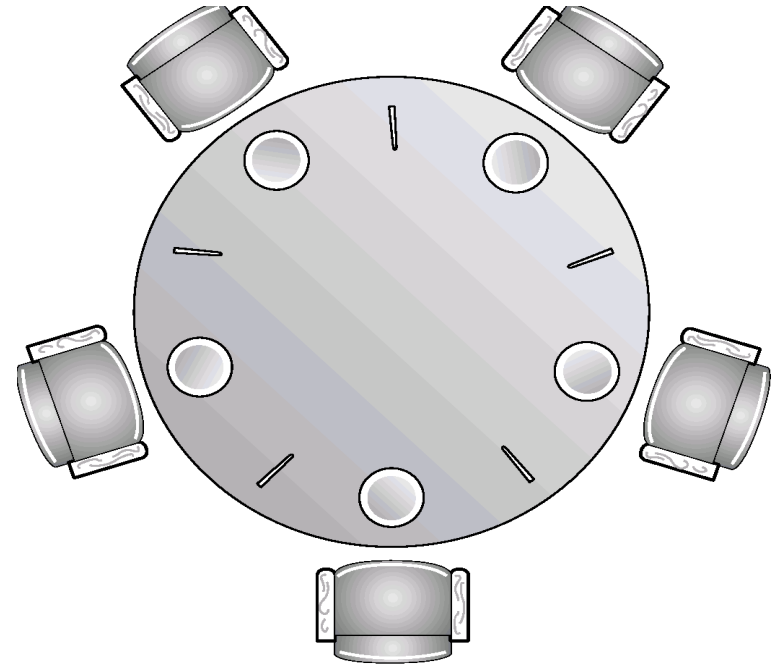


# Exclusión Mutua: SEMAFOROS

- 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]);**

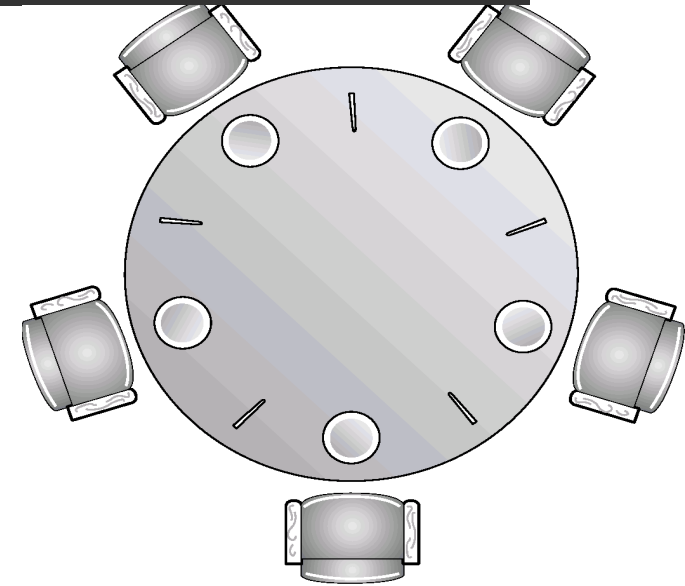
**comer();**

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

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

**signal(permiso);**

**until false** // Indefinidamente



# Exclusión Mutua: SEMAFOROS

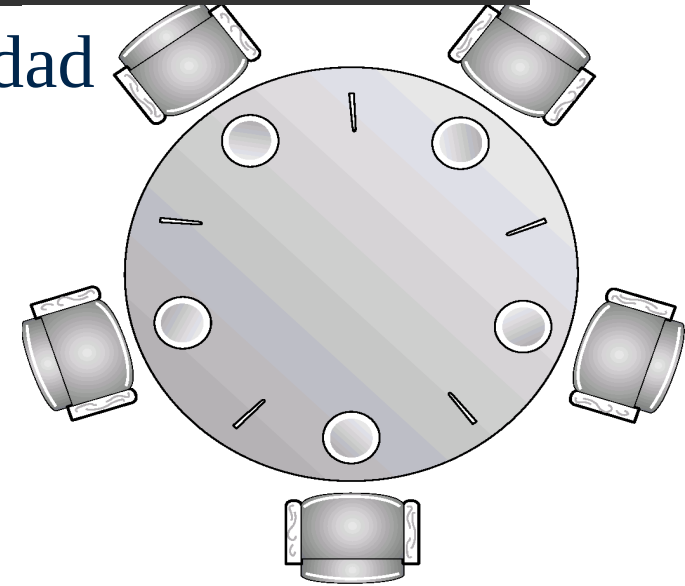
- Una posible solución: Romper la circularidad

----- 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

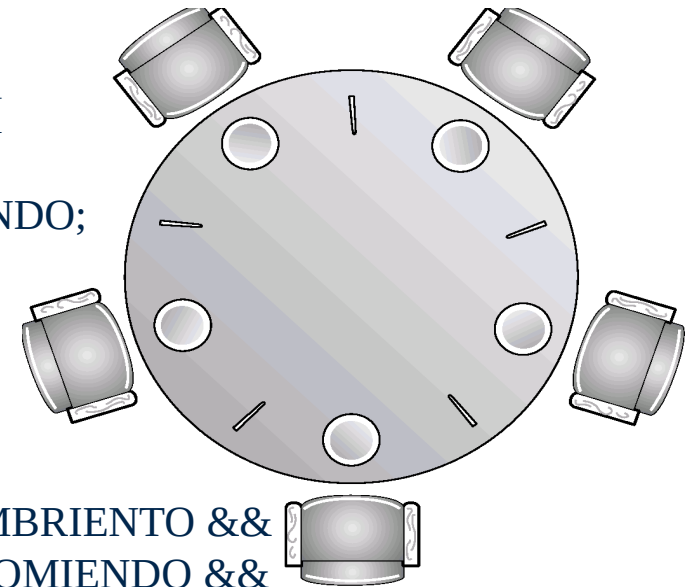
```
#define N          5
#define IZQ        (i+N-1)%N
#define DCH        (i+1)%N
#define PENSANDO   0
#define HAMBRIENTO 1
#define COMIENDO   2
typedef int semaphore;
int estado[N];
semaphore mutex = 1;
semaphore s[N];

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_lect = 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_lect = 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_lect = 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_lect);
        wait(cri_lect);
        cont_lect++;
        if (cont_lect == 1) wait(db);
        signal(cri_lect);
        signal(perm_lect);
        signal(lect);
        leer_base_datos();
        wait(cri_lect);
        cont_lect--;
        if (cont_lect == 0) signal(db);
        signal(cri_lect);
        usar_datos_leidos();
    }
}
```

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

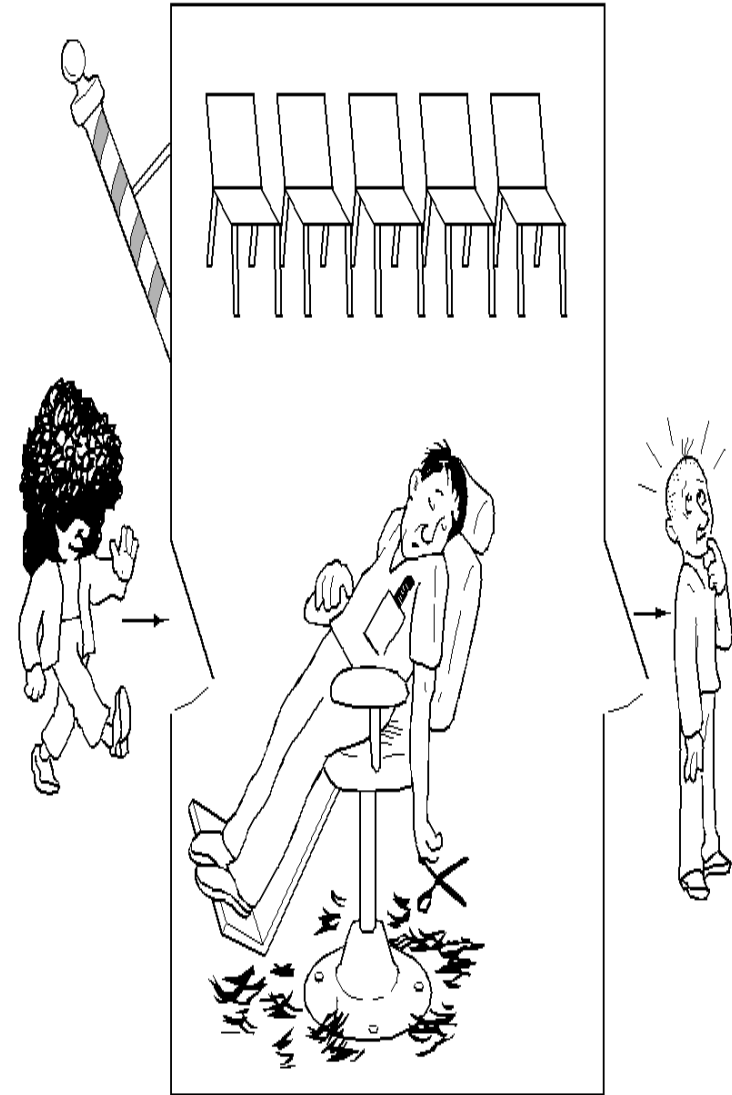
# Exclusión Mutua: SEMAFOROS

## # 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 sill as 5           //Número de sill as 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 < sill as) {
        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.

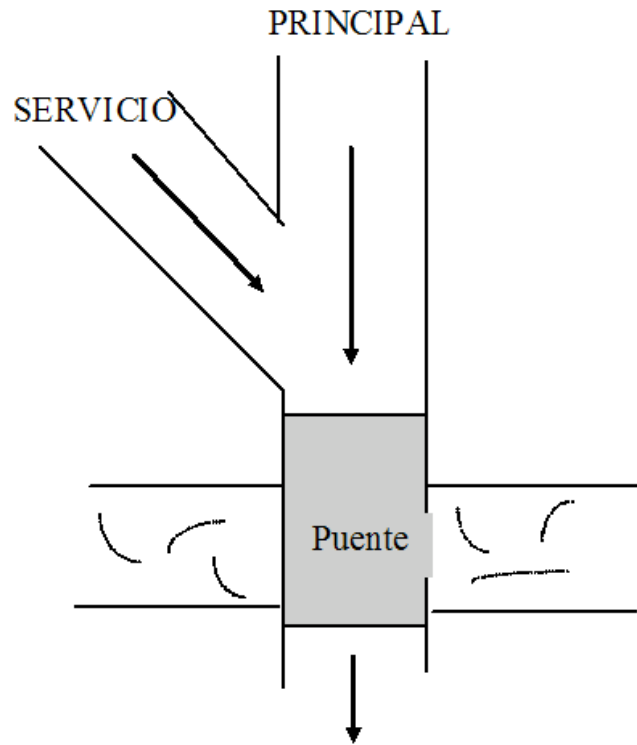
### EXCURSIONISTA

```
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;
        }
    }
}
```

```
void CodigoPost(void) {
    wait(mutex);
    kilos+=mipeso;
    signal(c);
    signal(mutex);
}
```

### CONTROLLER

```
void Controller(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);
```

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

- 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.

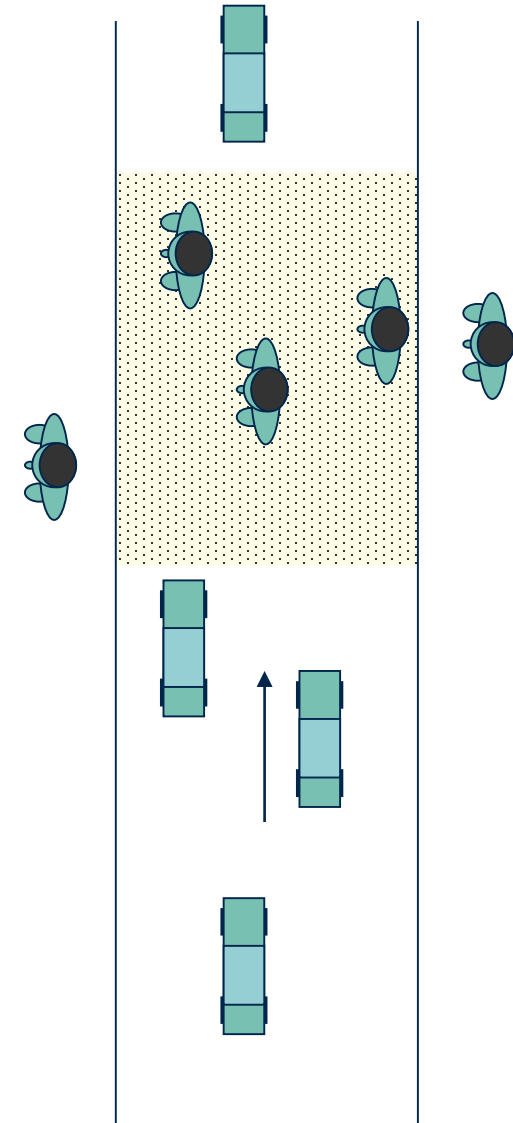
La zona de cruce es suficientemente ancha para que quepan varios coches dentro, es decir, pueden estar sobre ella varios coches a la vez y por supuesto varios peatones a la vez.

Los peatones tienen preferencia, de forma que mientras haya peatones cruzando, los coches se pararan y esperaran 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.





# Exclusión Mutua

Soluciones Soporte SO  
MONITORES



# Exclusión Mutua: MONITORES

## ● Desventajas de los semáforos

- Aunque permiten soluciones eficientes, su uso es complicado.
- Se basan en memoria compartida, en variables compartidas, por lo que resulta difícil modularizar y estructurar el código.
- El hecho de que los semáforos puedan jugar dos roles, sincronización y exclusión mutua, hace complejo el seguimiento del código
- Es delicado modificar el código, la inclusión y eliminación de sentencias puede provocar interbloqueos o hacer que la solución deje de funcionar.
- Es difícil la depuración de problemas complejos.
- No se puede saber en tiempo de compilación si se accede de manera indebida a un recurso compartido o no.
- No se pueden proteger los recursos críticos de un uso indebido del programador.

## ● Necesitamos una nueva herramienta que:

- Permita estructurar y modularizar el código
- Manteniendo un mecanismo de control sobre la sincronización
- Debe permitir encapsular los datos compartidos de forma que no se pueda hacer mal uso de los datos por parte del programador

# Exclusión Mutua: MONITORES

- Un monitor es una estructura del lenguaje de programación
- Es un módulo software que consta de uno o más procedimientos, una secuencia de inicio y datos que simbolizan el recurso compartido.
- Es un mecanismo de abstracción de datos que
  - Permite representar de forma abstracta un recurso compartido por medio de un conjunto de variables que definen su estado.
  - Los procedimientos del monitor son visibles desde el exterior
  - El acceso a las variables del monitor solo es posible mediante los procedimientos del monitor
- Un programa concurrente que utiliza monitores tiene dos tipos de procesos
  - Procesos pasivos
    - Son los que implementan los monitores y por tanto están a la espera que los activos hagan uso de los procedimientos del monitor que han sido exportados
    - Los procesos activos solo pueden interactuar con las variables compartidas del interior del monitor a través de estos procedimientos exportados
  - Procesos activos
    - Los procesos que hacen uso de los procedimientos del monitor.

# Exclusión Mutua: MONITORES

---

## ● Ventajas

- Los programadores de los procesos activos no tienen porque conocer la implementación del recurso compartido
- Si se modifica la forma de gestión del recurso compartido sin modificar la interfaz de procedimientos, los procesos activos no se ven afectados
- El programador del monitor no se preocupa de donde y como se usa el monitor, solo debe ocuparse de que funcione correctamente
- Se pueden implementar en módulos diferentes, lo que mejora la estructuración del código.

# Exclusión Mutua: MONITORES

## ● Resumen

- Un monitor es un mecanismo de abstracción de datos que contiene
  - Las variables que representan el recurso y su estado
  - Los procedimientos que realizan operaciones sobre el recurso
- Un proceso sólo puede acceder a las variables del recurso usando los procedimientos del monitor

## ● Exclusión mutua

- Un monitor está construido de tal forma que:
  - La ejecución de los procedimientos del monitor (internos o externos) no se ejecutan nunca de forma solapada
  - Se impide, por tanto, que dos procesos activos estén ejecutando simultáneamente procedimientos del monitor, (el mismo o distintos)

# Exclusión Mutua: MONITORES

## ● Esquema de un monitor

```
monitor nombre_monitor;  
  var variables locales;  
  export procedimientos exportados;  
  
  procedure proc1(parametros);  
    var: variables locales;  
    begin  
      {codigo del procedimiento}  
    end;  
  
  procedure proc2(parametros);  
    var: variables locales;  
    begin  
      {codigo del procedimiento}  
    end;  
  
begin  
  {codigo de inicialización}  
end;
```

## ● Un monitor se compone de:

- Conjunto de variables locales que se usan para almacenar el estado interno del recurso que representa el monitor o el estado de algún procedimiento interno.
- Estas variables no pueden variar entre llamadas sucesivas al monitor (a uno de los procedimientos exportados)
- El código de inicialización se ejecuta antes de la primera instrucción del programa que usa el monitor
- El conjunto de procedimientos (internos o exportados) pueden modificar las variables locales
- Los procedimientos pueden aceptar parametros

# Exclusión Mutua: MONITORES

- Un proceso activo
  - Ve al monitor como un conjunto de procedimientos que permiten usar un recurso compartido
  - En la llamada a uno de estos procedimientos está implícito el uso que se hace del recurso
  - Para invocar a un procedimiento de monitor:

```
nombre_monitor.nombre_procedimiento(parametros)
```

# Exclusión Mutua: MONITORES

- Decimos que un proceso activo está dentro del monitor cuando está ejecutando uno de los procedimientos exportados de éste
- Decimos que un proceso activo abandona el monitor cuando termina de ejecutar el código del procedimiento exportado
- El acceso exclusivo en la ejecución de los procedimientos exportados del monitor se basa en:
  - La existencia de una **cola del monitor**
    - Cuando un proceso está dentro del monitor, si otro proceso intenta acceder al monitor (usando uno de los procedimientos exportados) el código del monitor (el generado por el compilador) bloquea al segundo proceso en la cola del monitor.
    - Cuando un proceso abandona el monitor, el código del monitor selecciona el proceso en cabeza de cola del monitor y lo desbloquea, entrando dicho proceso en el monitor. Si la cola esta vacía el monitor queda libre.
- La responsabilidad de bloquear al proceso es del monitor, no del proceso activo.

# Exclusión Mutua: MONITORES

## ● Ejemplo:

- Dos procesos deben incrementar el valor de una variable compartida y poder examinar su valor en cualquier momento.

## ● Necesitamos

- Una variable permanente representa la variable compartida
- Un procedimiento para incrementar el valor
- Un procedimiento para consultar el valor
- Inicializar la variable compartida

```
monitor incremento;  
  var i:integer;  
  export inc,valor;  
  
  procedure inc;  
  begin  
    i:=i+1;  
  end;  
  
  function valor:integer;  
  begin  
    valor:=i;  
  end;  
  
begin  
  i:=0;  
end;
```

```
incremento.inc();  
v:=incremento.valor();
```

# Exclusión Mutua: MONITORES

- Un monitor incluye **variables de condición** que sólo son accesibles desde dentro del monitor.
- Estas variables de condición proporcionan el mecanismo de sincronización entre procesos.
- Permiten bloquear y desbloquear un proceso que está dentro del monitor
  - Cuando el código de un procedimiento de monitor detecta una condición en el recurso compartido que hace que el proceso que invocó al procedimiento no pueda continuar
  - Cuando otro proceso al usar otro (o el mismo) procedimiento modifica el estado del recurso de tal forma que el proceso que se bloqueo pueda continuar, éste se desbloqueará.
- Una variable de condición se declara dentro del monitor

```
var
  cond: condition;
  array_cond: array [1..5] of condition;
```

- Los valores de las variables condición no son accesibles por el programador del monitor.
- Son la representación de colas FIFO, inicializadas a vacía en su declaración

# Exclusión Mutua: MONITORES

- Permiten bloquear y desbloquear un proceso que está dentro del monitor

- Operación *delay(C)* o *cwait(C)*

- Permite bloquear al proceso que ejecuta el monitor en la cola de la condición C
- Se bloquea incondicionalmente al final de la cola FIFO
- Realiza las siguientes acciones

- liberar la exclusión mutua del monitor
- bloquear el proceso llamante en la cola C

- Operación *resume(C)* o *csignal(C)*

- Permite desbloquear al proceso cabecera de la cola de la condición C
- Si la cola está vacía la operación no hace nada
- Realiza las siguientes acciones

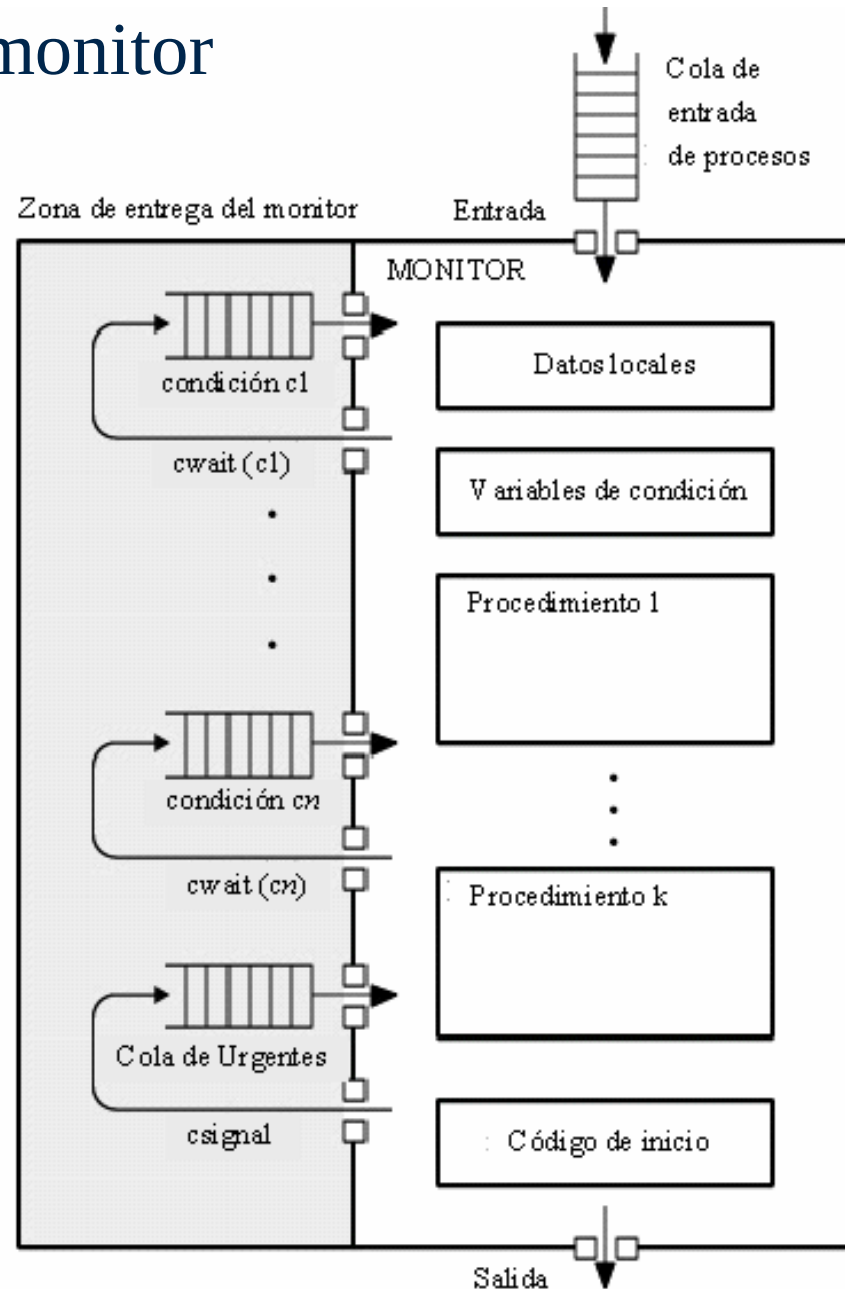
- liberar la exclusión mutua del monitor
- desbloquea al proceso en la cola C que tendrá prioridad sobre los procesos en cola de monitor
- se bloquea el proceso llamante en la cola de cortesía

# Exclusión Mutua: MONITORES

- La **cola de cortesía** o de urgentes
  - El monitor tiene una cola FIFO en la que se bloquean todos los procesos que ceden el monitor como consecuencia de la ejecución de *resume(C)*
  - Esta cola tiene prioridad frente la cola de monitor
    - Si hay procesos esperando en ambas colas, serán desbloqueados primero los procesos en cola de cortesía.
- La función *empty(C)*
  - En algunos casos es útil conocer si bajo una condición existen o no procesos bloqueados, para ello la función booleana *empty(C)* devolverá **true** o **false** si está llena o vacía respectivamente la condición C.

# Exclusión Mutua: MONITORES

## ● Esquema de un monitor



# Exclusión Mutua: MONITORES

## ● Productor/Consumidor con buffer limitado:

```
/*program productor_consumidor*/
monitor buffer_acotado;
char buffer[N];
int sigent, sigsal;
int contador;
condition lleno, no_hay;

void añadir(char x){
    if (contador == N) delay(esta_lleno);
    buffer[sigent] = x;
    sigent=(sigent + 1) % N;
    contador++;
    resume(no_hay);
}

void coger(char x){
    if (contador == 0) delay(no_hay);
    x = buffer[sigsal];
    sigsal=(sigsal + 1) % N;
    contador--;
    resume(esta_lleno);
}

{
    sigent=0; sigsal=0; contador=0;
}
```

```
void productor(){
    char x;
    while true {
        producir(x);
        añadir(x);
    }
}

void consumidor() {
    char x;
    while true {
        coger(x);
        consumir(x);
    }
}

void main(){
    parbegin(productor, consumidor);
}
```

## Ejercicio

- Implementar un semáforo binario utilizando monitores
  - Necesitaremos
    - Las operaciones
      - *wait*
      - *signal*
    - Una variable condición *csem* para bloquear a aquellos procesos que ejecuten un *wait* sobre nuestro semáforo
    - Una variable booleana *sem* para indicar si el semáforo está abierto o cerrado

# Exclusión Mutua: MONITORES

## ● Semáforo binario implementado con un monitor

```
monitor sbinario;  
  bool sem;  
  condition csem;  
  export wait,signal;  
  
  void wait(){  
    if sem  
      delay(csem);  
    sem=true;  
  }  
  
  void signal(){  
    if !empty(csem)  
      resume(csem);  
    else sem=false;  
  }  
  
begin  
  sem=false;  
end;
```

```
sbinario.wait();  
  
sbinario.signal();
```

A decorative frame consisting of two horizontal lines and two vertical lines, forming a rectangle with small circular endpoints.

# Exclusión Mutua

Soluciones Soporte SO  
PASO DE MENSAJES



# Exclusión Mutua: MENSAJES

- Los semáforos y monitores:
  - Un único computador (mono o multiprocesador), compartiendo memoria
- El paso de mensajes, además:
  - Sistema distribuido (conectados en red).
- Intercambio de información. Primitivas:
  - **send (destino, mensaje)**
  - **receive (origen, mensaje)**
- Aspectos de diseño
  - Acuse de recibo. Perdida del mensaje. Perdida del acuse. Secuenciación de mensajes.
  - Verificación de autenticidad.
  - Rendimiento. En la misma máquina puede ser lento el mecanismo de mensajes frente a monitores o semáforos.
- Análisis desde las siguientes perspectivas
  - Sincronización
  - Direccionamiento
  - Formato
  - Disciplina de cola
  - Exclusión mútua con mensajes.

# Exclusión Mutua: MENSAJES

---

## ● Sincronización

### ● Primitivas bloqueantes o no bloqueantes

#### ● Envío bloqueante, recepción bloqueante (rendez-vous)

Permite sincronización estricta entre varios procesos

#### ● **Envío no bloqueante, recepción bloqueante**

Combinación más común

Requiere de mensajes de respuesta (ACK) si se quiere verificar recepción

#### ● Envío no bloqueante, recepción no bloqueante

# Exclusión Mutua: MENSAJES

## ● Direccionamiento

### ● Direccionamiento Directo.

Las primitivas incluyen identificador del proceso con el que comunicar

Requiere el conocimiento de los procesos origen y destino

### ● Direccionamiento Implícito

El identificador del proceso que recibirá mensaje incluido en el mensaje anterior

### ● Direccionamiento Indirecto.

#### ● Se usan Buzones para intercambiar los mensajes

Este esquema desacopla emisor y receptor

Los buzones pueden ser

Estáticos: El buzón se crea y se asigna a un proceso

Dinámicos: Asociación del proceso a un buzón de manera dinámica con primitivas *conectar/desconectar*

El propietario del buzón puede ser:

|              |   |                                                                                                                  |
|--------------|---|------------------------------------------------------------------------------------------------------------------|
| El emisor.   | } | Se crea el buzón y se asigna al emisor/receptor.<br>Se destruye cuando finaliza el emisor o mediante primitivas. |
| El receptor. |   |                                                                                                                  |

El Sistema operativo. En este caso se necesitan primitivas de creación y destrucción del buzón.

## Direcccionamiento Indirecto

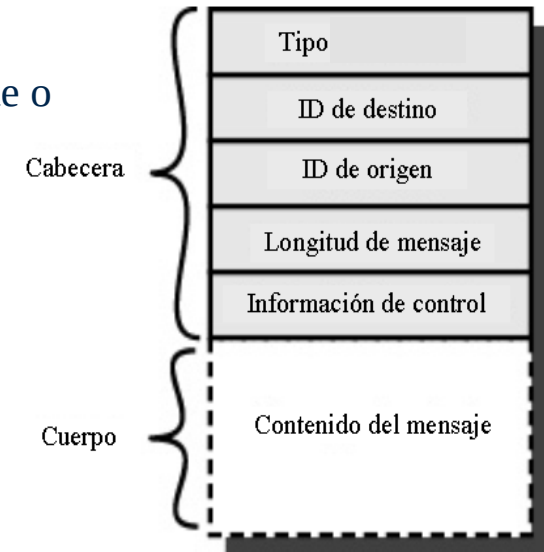
Relación entre emisores/receptores

- **Uno a Uno**  
Permite un enlace privado de comunicaciones entre dos procesos
- **Uno a Muchos**  
Útil en aplicaciones donde es necesario difundir un mensaje a n procesos, broadcast.
- **Muchos a Uno**  
El buzón se denomina puerto.  
El receptor (servidor) ofrece un servicio a varios procesos.  
El puerto está disposición de los emisores (clientes) para recibir peticiones de servicio.
- **Muchos a Muchos**

# Exclusión Mutua: MENSAJES

## Formato.

- Depende del servicio de mensajería si es sobre un sistema independiente o distribuido
- Mensajes cortos de tamaño fijo:
  - Minimizan el almacenamiento y se puede transmitir el mensaje.
- Mensajes largos
  - Los datos se ubican en un fichero y se transmite su referencia.
- Mensajes de longitud variable. Se divide en dos partes
  - Cabecera:
    - Tipo de mensaje
    - Proceso destino (buzón, puerto)
    - Proceso origen (cliente)
    - Longitud del mensaje
    - Información de control (prioridad, siguiente mensaje, contador, etc...)
  - Cuerpo: Contenido del mensaje



## Disciplina de la cola

- FIFO:** Puede no ser suficiente si hay mensajes más urgentes que otros.
- PRIORIDADES:** En función del tipo de mensaje, del destinatario o emisor.
- SELECCIÓN:** Permitir que el receptor examine la cola en busca del mensaje que quiere recibir a continuación.

# Exclusión Mutua: MENSAJES

## ● Espera selectiva – Se usa cuando

- Se envían mensajes a un buffer que sólo puede atender a un envío en cada instante
- No se conoce a priori el cliente que hace la petición al buffer en un instante dado, el servidor debe estar preparado para recibir sin importar el orden.
- Si no hay emparejamiento

```
select{  
    receive(buzón1, mensaje);  
    sentencias;  
or  
    receive(buzón2, mensaje);  
    sentencias  
or  
    ...  
}
```

## ● Espera selectiva con guarda

- Se utilizan condiciones para atender la recepción: Si se cumple la condición y entonces podemos recibir de ...
- Puede haber varias condiciones true que habilitan la recepción, sólo se atiende una (aleatoriamente)

```
select{  
    when condicion1  
    receive(buzón1, mensaje);  
    sentencias;  
or  
    when condicion2  
    receive(buzón2, mensaje);  
    sentencias  
or  
    ...  
}
```

## ● Espera selectiva: si no hay emparejamiento entonces bloqueo hasta evento

# Exclusión Mutua: MENSAJES

## ● Exclusión mutua con Mensajes

- send no bloqueante y receive bloqueante.

```
/*program exclusión mútua*/
const int n = /*Número de procesos*/

void P(int i){
    mensaje msj;
    while true {
        receive(mutex, msj);
        /*sección crítica*/
        send(mutex, msj);
        /*resto*/
    }
}

void main(){
    crear_buzon(mutex);
    send(mutex,null);
    parbegin(P(1), P(2), ..... P(n));
}
```

# Exclusión Mutua: MENSAJES

## ● Ejercicio:

- Resolver el problema del productor/consumidor con buffer ilimitado usando mensajes

Solución: *send no bloqueante, receive bloqueante*

```
/*program Productor Consumidor Buffer ilimitado*/  
msg_t /*tipo abstracto que define el item, es el mensaje a pasar */
```

```
void Productor(){  
    msg_t item;  
    while true {  
        item = producir();  
        send(buffer,item);  
    }  
}
```

```
void Consumidor(){  
    while true {  
        receive(buffer,item);  
        consumir(item)  
    }  
}
```

```
void main(){  
    crear_buzon(buffer);  
    parbegin(Productor(), Consumidor());  
}
```

# Exclusión Mutua: MENSAJES

 **Ejercicio:**  
Resolver el problema del productor/consumidor con buffer limitado por medio de mensajes.

**Solución:** *send no bloqueante, receive bloqueante*

```
/*program Productor Consumidor Buffer ilimitado*/
const int capacidad = N;
msg_t null; /*mensaje vacío*/
int i;

void Productor(){
    msg_t msjp;
    while true {
        receive(buffer_producir,msjp);
        msjp=producir();
        send(buffer_consumir,item);
    }
}

void Consumidor(){
    msg_t msjc;
    while true {
        receive(buffer_consumir,msjc);
        consumir(msjc);
        send(buffer_producir,null);
    }
}

void main(){
    crear_buzon(buffer_producir);
    crear_buzon(buffer_consumir);
    for (int i=1; i<=capacidad; i++)
        send(buffer_producir,null);
    parbegin(Productor(), Consumidor());
}
```

# Exclusión Mutua: MENSAJES

---

## ● Ejercicio:

Resolver el problema del productor/consumidor con buffer limitado por medio de mensajes con las siguientes

Consideraciones:

- Número de productores:  $P$
- Número de consumidores:  $C$
- Tamaño del buffer de Productos:  $N$
- Proponer una solución con espera selectiva (con o sin guarda) haciendo uso de un proceso coordinador del buffer de productos.
- Definir los buzones que sean necesarios.

# Exclusión Mutua: MENSAJES

---

## ● Ejercicio:

Implementar una barrera para un numero arbitrario (n) de procesos utilizando primitivas genéricas de paso de mensajes *send* y *receive*

Explicar detalladamente la solución alcanzada.

# Paso de mensajes en Unix

- Unix System V Release4 (SVR4) proporciona tres herramientas para el paso de mensajes

- Tuberías
  - Colas de Mensajes
  - Sockets
- En la misma máquina
- En la distinta máquina

## ● Tuberías

- Tuberías sin nombre: Para comunicar procesos con relación de parentesco
- Tuberías con nombre: Para comunicar procesos sin relación de parentesco

## ● Colas de Mensajes

- Buffer FIFO mantenidos por el S.O. donde puede haber varios procesos escribiendo y varios leyendo.
- La exclusión mutua al buffer está garantizada por el S.O.
- Se pueden escribir o leer estructuras con formato de mensaje tratados (leídos/escritos) como items indivisibles
- Se pueden tipar los mensajes y dotar de primitivas de selección de mensaje por tipo (esto rompe FIFO, aunque FIFO para mismo tipo)
- Primitivas principales:
  - **msgget** : Para crear una cola o habilitar acceso a una cola ya creada
  - **msgctl** : Control de colas de mensajes
  - **msgsnd**: Para escribir (send) mensaje en la cola
  - **msgrcv**: Para leer (receive) mensaje de la cola