

Tema 4

Sincronización y Comunicación de Procesos

Fuente : Gustavo Romero

- Procesos independientes
- Procesos cooperantes
 - Cooperación directa: conocen a los otros procesos y colaboran
 - Cooperación indirecta: desconocen a los otros procesos, pero saben que acceden a recursos compartidos

Interacción entre hebras

percepción	relación	influencia	problemas
se desconocen	competencia	resultados independientes, temporización afectable	interbloqueo, inanición
percepción indirecta (recursos compartidos)	cooperación por compartición	resultados dependientes, temporización afectable	exclusión mutua, interbloqueo, inanición, coherencia de datos
percepción directa (comunicación)	cooperación por comunicación	resultados dependientes, temporización afectable	interbloqueo, inanición

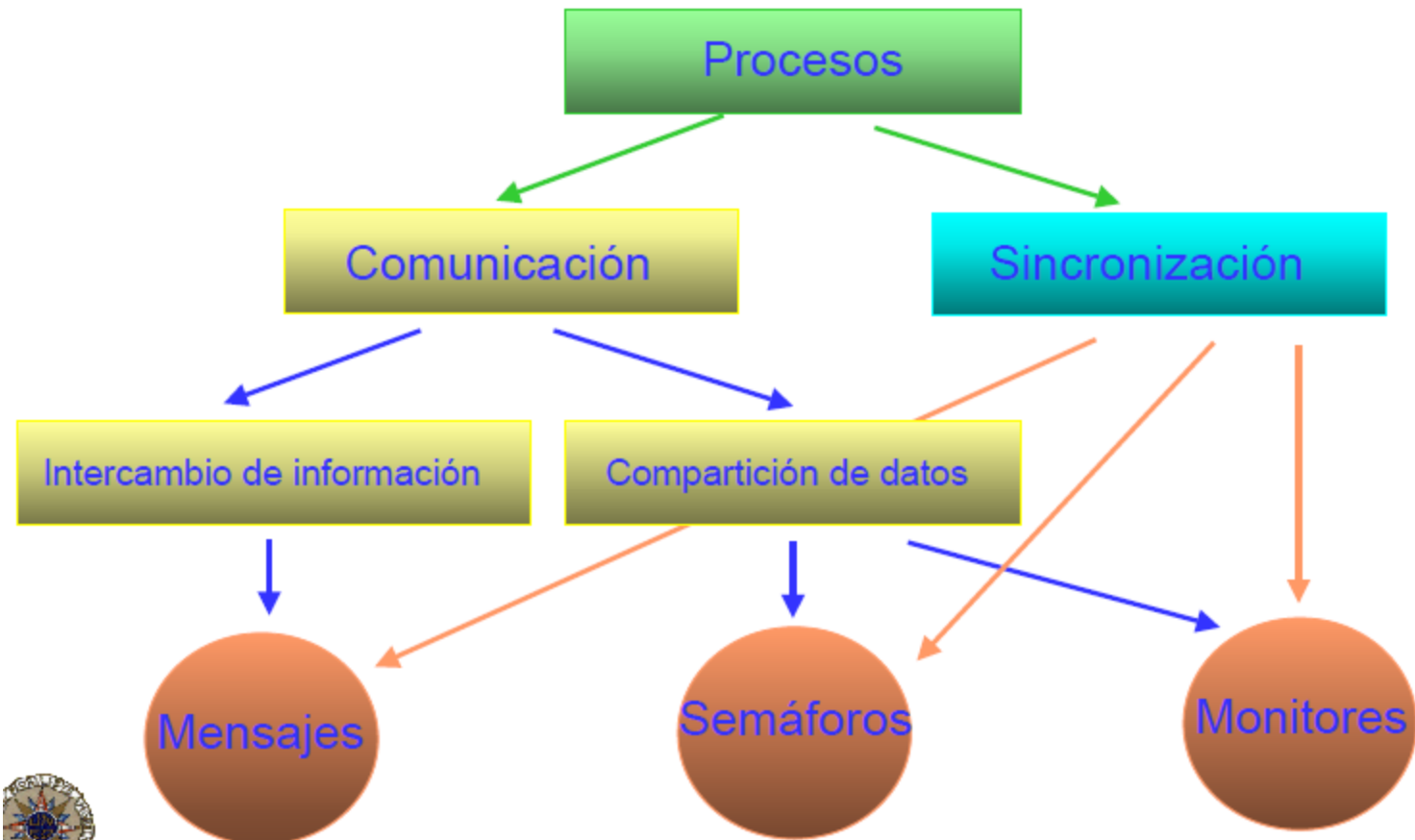
Condiciones de carrera

El problema ocurre porque dos hebras acceden a un **recurso compartido** sin **sincronizar** su uso.

- Las condiciones de carrera producen resultados **impredecibles**.
- Dependen de cuándo se ejecutan las hebras, durante cuánto tiempo y de cuándo se produce el cambio entre ellas, y por supuesto, de lo que estén haciendo.

El acceso concurrente a recursos compartidos debe ser controlado.

- Es la única forma de que podamos comprender el comportamiento de los programas.
- Necesitamos reintroducir el **determinismo** y la **predecibilidad**.

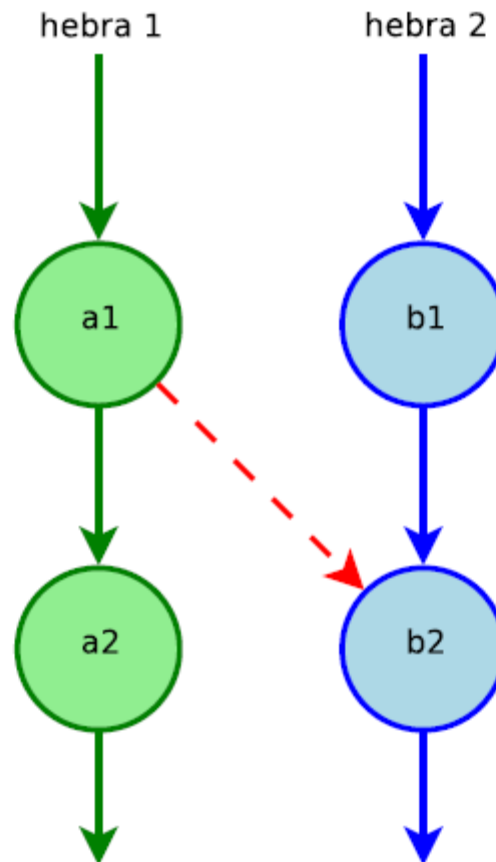


¿Por qué sincronizar?

Las hebras **cooperan** porque...

- Necesitan **coordinar** su ejecución.
 - El botón de detener del navegador cancela la descarga de una página web.
 - La hebra del interfaz del navegador necesita avisar a la hebra encargada de descargar la página web.
- Necesitan acceder a **recursos compartidos**.
 - Varias hebras de un servidor web pueden acceder a una caché común en memoria para ahorrar espacio y tiempo.

Coordinación



b2 tiene que esperar hasta que a1 haya completado su ejecución.

Recursos compartidos

- Problema básico:
 - 2 hebras acceden a una variable compartida.
 - Si la variable compartida es **leída y escrita** por ambas hebras tendremos que coordinar el acceso.
- En los próximos 2 temas estudiaremos...
 - Mecanismos para controlar el acceso a recursos compartidos:
 - Bajo nivel: cerrojos.
 - Alto nivel: semáforos, monitores, variables condición.
 - Protocolos para coordinar el acceso a recursos compartidos.
- La gestión de la concurrencia es complicada y propensa a errores.
 - En Informática se dedica una asignatura completa a su estudio.
 - Incluso las soluciones de algunos libros de texto son erróneas.

Ejemplo: cuenta bancaria (1)

procedimiento para **retirar** fondos de una cuenta bancaria

```
int retirar(CCC_t cuenta, int cantidad)
{
    int saldo = conseguir_saldo(cuenta);
    saldo = saldo - cantidad;
    almacenar_saldo(cuenta, saldo);
    return saldo;
}
```

procedimiento para **ingresar** fondos en una cuenta bancaria

```
int ingresar(CCC_t cuenta, int cantidad)
{
    int saldo = conseguir_saldo(cuenta);
    saldo = saldo + cantidad;
    almacenar_saldo(cuenta, saldo);
    return saldo;
}
```


Ejemplo: cuenta bancaria (2)

- Supongamos un saldo inicial de 1.000€.
- Estudiaremos el caso en que dos personas intentan retirar 100€ a la vez de la misma cuenta desde dos cajeros automáticos.
- Imaginemos que cada procedimiento se ejecuta como una hebra diferente.
- La ejecución de las 2 hebras podría entrelazarse debido a la planificación del sistema operativo.
- ¿Existe algún problema?
- ¿Cuál será el saldo de la cuenta tras retirar fondos?

Ejemplo: cuenta bancaria (3)

hebra 1

```
saldo = conseguir_saldo(cuenta);  
saldo = saldo - cantidad;  
almacenar_saldo(cuenta, saldo);  
return saldo;
```

hebra 2

```
saldo = conseguir_saldo(cuenta);  
saldo = saldo - cantidad;  
almacenar_saldo(cuenta, saldo);  
return saldo;
```

saldo final: 800€

Ejemplo: cuenta bancaria (4)

hebra 1	hebra 2
saldo = conseguir_saldo(cuenta); saldo = saldo - cantidad;	
	saldo = conseguir_saldo(cuenta); saldo = saldo - cantidad; almacenar_saldo(cuenta, saldo); return saldo;
almacenar_saldo(cuenta, saldo); return saldo;	

saldo final: 900€

Ejemplo: datos compartidos (1)

```
int a = 0, b = 0; // variables compartidas
```

```
void *suma(void*)  
{  
    while(true)  
    {  
        a = a + 1;  
        b = b + 1;  
    }  
    return NULL;  
}
```

```
void *resta(void*)  
{  
    while(true)  
    {  
        a = a - 1;  
        b = b - 1;  
    }  
    return NULL;  
}
```

- Tras un tiempo... ¿seguirán a y b siendo iguales?

Ejemplo: datos compartidos (2)

```
int a = 0, b = 0; // variables compartidas
```

```
void *suma(void*)  
{  
    while(true)  
    {  
        a = a + 1;  
        b = b + 1;  
    }  
    return NULL;  
}
```

```
void *resta(void*)  
{  
    while(true)  
    {  
        a = a - 1;  
        b = b - 1;  
    }  
    return NULL;  
}
```

- El código que accede a variables compartidas son **secciones críticas** → tras cierto tiempo $a \neq b$
- Compartir datos y recursos induce problemas similares.
- Problemas: ¿CFLAGS = -O3? ¿volatile?

Ejemplo: recursos compartidos

- ¿Qué pasará al ejecutar dos copias de esta hebra?

```
void* saludo(void*)  
{  
    while(true)  
        cout << "hola, soy " << pthread_self() << endl;  
    return NULL;  
}
```

- esto...

```
hola, soy 918374512  
hola, soy 649128354
```

- o esto...

```
hola, soy hola, soy 918374512  
  
649128354
```

- Condición de carrera: ambas hebras acceden al terminal que es un recurso compartido.

Concurrencia (1)

- Niveles:
 - **Multiprogramación:** gestión de múltiples procesos.
 - **Multiprocesamiento:** gestión de múltiples hebras dentro de un proceso.
 - **Procesamiento distribuido:** gestión de múltiples procesos sobre múltiples máquinas (sin memoria compartida).
- Contexto:
 - Entre aplicaciones: originalmente ideada con este fin.
 - Dentro de una aplicación: conveniente en ciertos tipos de aplicaciones.
 - Sistema operativo: las mismas ventajas que aporta a las aplicaciones son deseables aquí.
- Tipos:
 - **Monoprocesador:** los procesos se **entrelazan** en el tiempo para ofrecer la apariencia de ejecución simultánea.
 - **Multiprocesador:** la ejecución de múltiples procesos se **solapa** en el tiempo.

Concurrencia (2)

- No puede predecirse la velocidad relativa de ejecución:
 - Eventos.
 - Políticas del sistema operativo.
- Problemas:
 - La **compartición** de recursos está plagada de peligros.
 - La **gestión** óptima de los recursos es complicada.
 - La localización de **errores** de programación es muy complicada debido a su naturaleza no determinista y no reproducible.
- Todos los sistemas comparten este tipo de problemas aunque algunos pueden verse agravados en máquinas con más de un procesador.

Sección crítica (1)

- Cuando una hebra/proceso accede a un recurso compartido diremos que está ejecutando una **sección crítica**.
- Una hebra puede tener **más de una** sección crítica.
- Las secciones críticas puede **anidarse**.
- La ejecución de una sección crítica debe ser **mutuamente excluyente**:
 - En cualquier instante de tiempo **sólo a una** hebra se le permite ejecutar su sección crítica.
 - Toda hebra debe **pedir permiso para entrar** en una sección crítica.
 - Al abandonar la sección crítica la hebra debe **asegurarse** de que otra hebra que lo necesite pueda entrar → **avisar** de que ha **salido**.
 - Es necesario que toda hebra cumpla con este **protocolo**.

Uso de secciones críticas (1)

¿Cómo resolver el problema de las secciones críticas?

- Establecer un **protocolo** de **serialización** o **exclusión mutua**.
- El resultado de las hebras involucradas no dependerán del entrelazado no determinista de sus instrucciones.

Protocolo:

- El código que desee entrar en una **sección crítica** debe solicitarlo mediante **entrar_sección_crítica**.
- Una vez ejecutada la sección crítica esta debe finalizarse con **salir_sección_crítica**.
- Al resto del código fuera de la sección crítica lo denominaremos **sección_no_crítica**.

Uso de secciones críticas (2)

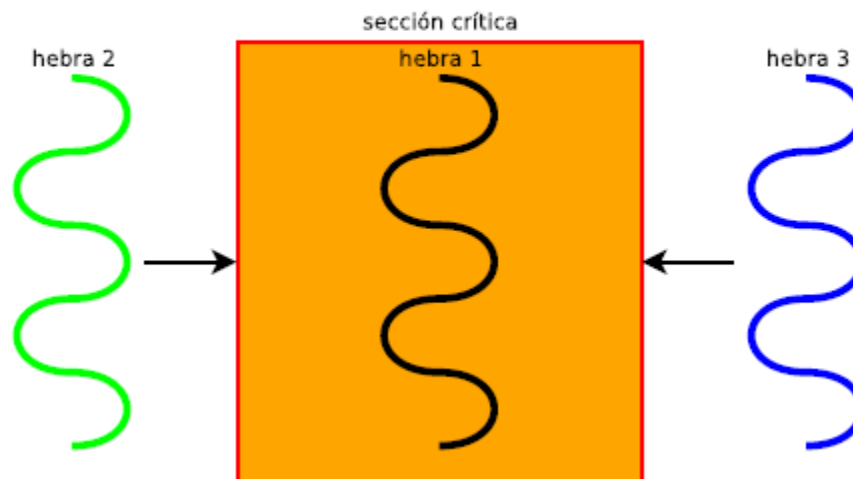
```
// patrón de comportamiento
// habitual de una hebra que
// ejecuta una sección crítica

while(true)
{
    entrar_sección_crítica
    sección_crítica
    salir_sección_crítica
    sección_no_crítica
}
```

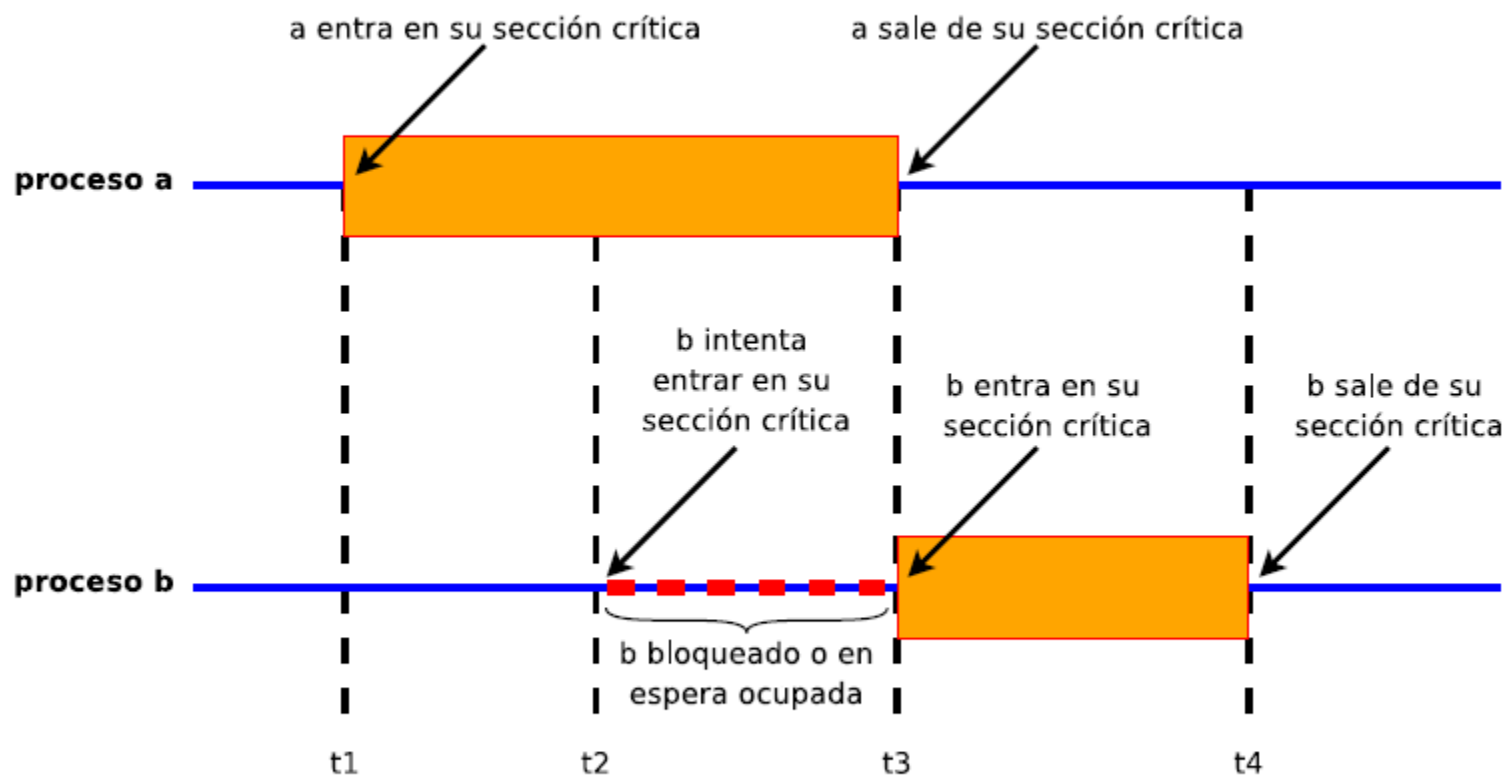
- Cada hebra se ejecuta a una **velocidad no cero** pero no podemos hacer ninguna suposición acerca de su velocidad relativa de ejecución.
- Aunque dispongamos de más de un procesador la gestión de **memoria** previene el acceso simultáneo a la misma localización.

Exclusión mutua

- Debemos utilizar **exclusión mutua** para sincronizar el acceso a los recursos compartidos, por ejemplo, **serializando** el acceso.
- Cualquier código que utilice exclusión mutua para sincronizar su ejecución es denominado **sección crítica**.
 - Sólo una hebra puede ejecutar su sección crítica.
 - El resto de hebras son forzadas a esperar a la entrada.
 - Cuando una hebra abandona su sección crítica otra puede entrar.



Protocolo de exclusión mutua



Ejecución de **secciones críticas** mediante **exclusión mutua**.

Conceptos **importantes**

- **Condición de carrera:** situación en la que el resultado de una operación depende del orden de ejecución de varias hebras.
- **Sección crítica:** sección de código que no pueden ejecutar varias hebras de forma concurrente.
- **Exclusión mutua:** requisito de acceso exclusivo a recursos durante la ejecución de una sección crítica.
- **Interbloqueo** (*"deadlock"*): situación en la que varias hebras son incapaces de avanzar porque todas ellas están esperando que alguna de las otras haga algo.
- **Inanición:** situación en la que una hebra preparada es mantenida en dicho estado indefinidamente.
- **Circulo vicioso** (*"livelock"*): situación en la que varias hebras cambian su estado pero sin llegar a realizar trabajo útil.

El problema de la exclusión mutua

1965

- Dijkstra define el problema.
- Primera solución por Dekker.
- Solución general por Dijkstra.

196x

- Soluciones por Fischer, Knuth, Lynch, Rabin, Rivest...

1974

- Algoritmo de la panadería de Lamport.

1981

- Algoritmo de Peterson.

2008

- Cientos de soluciones publicadas.
- Muchos problemas relacionados.

Soluciones para lograr exclusión mutua

Soluciones a nivel de **usuario**:

- Algoritmos no basados en el uso de instrucciones especiales del procesador ni de características del núcleo → espera ocupada.

Soluciones **hardware**:

- Instrucciones especiales del procesador.

Soluciones a nivel del **núcleo**:

- Bajo nivel: cerrojos y semáforos binarios.
- Alto nivel: semáforos y monitores.

s
o
b
r
e
c
a
r
g
a

La solución adecuada dependerá del modelo de procesamiento.

Soluciones a nivel usuario

- La sincronización se consigue a través de **variables globales**.
- Estas soluciones funcionan mediante **espera ocupada**:
`while(condición != cierto); // sólo salta`
- Simplificación:
 - Empezaremos resolviendo el problema para 2 procesos ó 2 hebras: H_0 y H_1 .
 - Cuando hablemos de más de dos hebras H_i y H_j denotarán a dos hebras diferentes ($i \neq j$).
- Exclusión mutua para 2 hebras:
 - Algoritmos de Dekker.
 - Algoritmo de Petterson.
- Exclusión mutua para más de 2 hebras:
 - Algoritmo de la panadería (Lamport).

Algoritmo 1: exclusión mutua mediante espera ocupada

variables compartidas

```
int turno = 0;
```

Hebra 0

```
while(true)
{
    while (turno != 0);
    sección_crítica();
    turno = 1;
    sección_no_crítica();
}
```

Hebra 1

```
while(true)
{
    while (turno != 1);
    sección_crítica();
    turno = 0;
    sección_no_crítica();
}
```

- Primera solución propuesta para el problema de la exclusión mutua en secciones críticas.
- Versión 1 del algoritmo de Dekker... ¡hay 5!.

Análisis del algoritmo 1: ¿funciona? (1)

variables compartidas

```
turno = 0; // compartida
```

Hebra i

```
while (true)
{
    while (turno != i);
    sección_crítica();
    turno = j;
    sección_no_crítica();
}
```

- Debemos inicializar la variable turno.
- La hebra H_i ejecuta su sección crítica si y sólo si $\text{turno} == i$.
- H_i se mantiene en **espera ocupada** mientras H_j está en su sección crítica → **exclusividad satisfecha**.
- El requisito de **progreso no es satisfecho** porque estamos suponiendo la alternancia estricta de las 2 hebras.

progreso no satisfecho → solución no válida

Análisis del algoritmo 1: ¿es correcto? (2)

- Supongamos SNC_0 larga y SNC_1 corta.
- Si $turno == 0$, H_0 entrará en su SC_0 , la abandonará haciendo $turno = 1$ y comenzará a ejecutar su larga SNC_0 .
- Ahora H_1 entrará en su SC_1 , la abandona haciendo $turno = 0$ y ejecuta su corta SNC_1 .
- Si de nuevo H_1 intenta entrar en su SC_1 , debe esperar hasta que H_0 finalice su larga SNC_0 y pase de nuevo por SC_0 .

Análisis del algoritmo 1

requisito	cumple	motivo
exclusión mutua	si	debido a turno sólo una hebra entra en su sección crítica
progreso	no	alternancia estricta, además el tamaño de la sección_no_crítica afecta a la otra hebra
espera acotada	si	espera igual al tiempo de ejecución de la otra sección crítica
portabilidad	no	1 procesador + prioridad estática = círculo vicioso
eficiencia	no	espera ocupada
escalabilidad	no	el problema se acentúa al aumentar el número de hebras

Algoritmo de Peterson

variables compartidas

```
bool bandera[2] = {false, false};  
int turno = 0; // desempate
```

hebra_i

```
while (true)  
{  
    bandera[i] = true;  
    turno = i;  
    while (bandera[j] && turno == j);  
    sección_crítica();  
    bandera[i] = false;  
    sección_no_crítica();  
}
```

- La hebra i muestra su deseo de entrar en su sección crítica haciendo `bandera[i] = true`.
- Si ambas hebras intentan entrar a la vez en su sección crítica la variable `turno` decide quien va a pasar.
- En comparación con el algoritmo de Dekker es más eficiente, más simple y extensible a más de 2 hebras.

```

#define FALSE 0
#define TRUE 1
int turno;
int petición_rec[2];
void acceso_sc(int pid)
{
    int otro_pid;
    otro_pid = 1 - pid;
    petición_rec[pid]=TRUE;
    turno = otro_pid;
    while(turno == otro_pid && petición_rec[otro_pid] == TRUE);
}

void salida_sc(int pid)
{
    petición_rec[pid] = FALSE;
}

void procesoA()
{
    while(TRUE)
    {
        /* sección no crítica */
        acceso_sc(0);
        /* sección crítica*/
        salida_sc(0);
        /* sección no crítica */
    }
}

void procesoB()
{
    while(TRUE)
    {
        /* sección no crítica */
        acceso_sc(1);
        /* sección crítica */
        salida_sc(1);
        /* sección no crítica */
    }
}

void main()
{
    petición_rec[0] = FALSE;
    petición_rec[1] = FALSE;
    ejecución_concurrente(procesoA, procesoB);
}

```

Algoritmo de Peterson: ¿es correcto?

Exclusión mutua:

- H_0 y H_1 sólo pueden alcanzar su sección crítica si `bandera[0] == bandera[1] == true` y sólo si `turno == i` para cada hebra H_i , lo que es imposible.

Progreso y espera acotada:

- H_i podría evitar que H_j entre en su sección crítica sólo si está atrapado en `while (bandera[j] && turno == j);`.
- Si H_j no está intentando entrar en su sección crítica `bandera[j] == false` y H_i puede entrar en su sección crítica.
- Si H_j ha hecho `bandera[j] == true` y está atrapado en el bucle `while` entonces `turno == i` o `turno = j`.
 - Si `turno == i` H_i podrá entrar en su sección crítica.
 - Si `turno == j` H_j entrará en su sección crítica y hará `bandera[j] = false` a la salida y permitirá a H_i entrar.
- Si a H_j le da tiempo a hacer `bandera[j] = true`, hará `turno = i`.
- Como H_i no cambia el valor de `turno` mientras está esperando en el `while`, H_i podrá entrar en su sección crítica tras como mucho una ejecución de la sección crítica de H_j .

Análisis del algoritmo de Peterson

requisito	cumple	motivo
exclusión mutua	si	
progreso	si	
espera acotada	si	
portabilidad	no	
eficiencia	no	espera ocupada
escalabilidad	si	pero requiere recodificación

Hardware

Implementación de operaciones atómicas

El soporte hardware puede facilitar la tarea:

- **Deshabilitar las interrupciones:** `cli/sti`

- ¿Por qué funciona?
- ¿Por qué puede esta acción evitar el cambio de hebra?
- ¿Funcionaría en cualquier tipo de sistema? → **no**, válido sólo en sistemas con un **único procesador**.

- **Instrucciones atómicas:**

- Existen instrucciones para las que el **procesador** y el **bus** garantizan su ejecución atómica.

TAS Test And Set: `lock xchg %al, (%edx)`

FAA Fetch And Add: `lock xadd %eax, (%edx)`

CAS Compare And Swap: `lock cmpxchg %ecx, (%edx)`

LL/SC Load Linked/Store Conditional: `ldrex/strex`

Primera solución: un simple cerrojo (1)

- Un cerrojo es un **objeto en memoria principal** sobre el que podemos realizar dos operaciones:
 - `adquirir()`: antes de entrar en la sección crítica.
 - Puede requerir algún tiempo de espera a la entrada de la sección crítica.
 - `liberar()`: tras abandonar la sección crítica.
 - Permite a otra hebra acceder a la sección crítica.
- Tras cada `adquirir()` debe haber un `liberar()`:
 - Entre `adquirir()` y `liberar()` la hebra **posee** el cerrojo.
 - `adquirir()` sólo retorna cuando el llamador se ha convertido en el dueño del cerrojo... ¿Por qué?
- ¿Qué podría suceder si existieran llamadas a `adquirir()` sin su correspondiente `liberar()` emparejado?
- ¿Qué pasa si el propietario de un cerrojo trata de adquirirlo por segunda vez?

Usando un cerrojo (1)

retirar fondos de una cuenta bancaria

```
while (true)
{
    cerrojo.adquirir();
    saldo = conseguir_saldo(cuenta);
    saldo = saldo - cantidad;
    almacenar_saldo(cuenta, saldo);
    cerrojo.liberar();
    return saldo;
}
```

Usando un cerrojo (2)

Hebra 1

```
cerrojo.adquirir();
```

```
saldo = conseguir_saldo(cuenta);  
saldo = saldo - cantidad;
```

```
almacenar_saldo(cuenta, saldo);
```

```
cerrojo.liberar();
```

```
return saldo;
```

Hebra 2

```
cerrojo.adquirir();
```

```
saldo = conseguir_saldo(cuenta);  
saldo = saldo - cantidad;  
almacenar_saldo(cuenta, saldo);
```

```
cerrojo.liberar();
```

```
return saldo;
```

```
/* acceso_sección_critica */  
[1a]    ccc r1, cerrojo  
[1b]    comparar r1, #0, [1d]  
[1c]    saltar [1a]  
[1d]    volver  
  
/* salida_sección_critica */  
[2a]    mover cerrojo, #0  
[2b]    volver
```

Figura 4.6 – Obtención de la exclusión mutua mediante el uso de la instrucción máquina especial del tipo comprobar y configurar (ccc)

Bloqueo mediante interrupciones

Mecanismo primitivo y sólo válido en sistemas monoprocesador.

- No cumple con el requisito de la portabilidad.
- Deshabilitar las interrupciones en un procesador no evita que en otro se ejecute la sección crítica.
- Sólo en el caso de secciones críticas **extremadamente cortas** dentro del **núcleo** y en sistemas **monoprocesador** será una solución válida.
- Si no podemos atender la interrupción del reloj no podremos cambiar de hebra tal y como hacen la mayoría de los sistemas de tiempo compartido.

Protocolo:

- Antes de que una hebra entre en su sección crítica deshabilita las interrupciones (`cli`).
- Al salir de la sección crítica se han de volver a habilitar las interrupciones (`sti`).

Deshabilitar las interrupciones a nivel de usuario (1)

monoprocesador

hebra;

```
while(true)
{
    deshabilitar_interrupciones();
    sección_crítica();
    habilitar_interrupciones();
    sección_no_crítica();
}
```

Se preserva la exclusión mutua pero se degrada la eficiencia del sistema: mientras está en la sección crítica no se atenderán interrupciones.

- No habrá tiempo compartido.
- El retraso en la atención de las interrupciones podría afectar al sistema completo. $\iff$ ¿Por qué?
- Las aplicaciones podrían abusar o contener fallos $\implies$ cuelgue del sistema.

Deshabilitar las interrupciones a nivel de usuario (2)

multiprocesador

Multiprocesador:

- No es efectivo en absoluto porque...
 - Aunque las deshabilitásemos en todos los procesadores no arregla nada porque...
 - Varios procesadores podría acceder a la vez a un mismo recurso → no garantiza la exclusión mutua.

En resumen:

- Los efectos laterales son inaceptables en general.
- La buena noticia es que esta operación es **privilegiada** :)

Semáforos con contador (Dijkstra)

objetos señal del núcleo

Definición: un semáforo es una variable entera, que aparte de la inicialización, sólo puede ser modificada mediante dos operaciones atómicas y mutuamente excluyentes.

p() / decrementar() / esperar()

Operación que decrementa el valor del semáforo.

v() / incrementar() / señalar()

Operación que incrementa el valor del semáforo.

Semáforos con contador

```
class semáforo
{
public:
    semáforo(int _valor = 0):
        valor(_valor),
        bloq(vacia) {}

    void p() // esperar()
    {
        valor = valor - 1;
        if (valor < 0)
        {
            bloq.añadir(esta);
            bloquear(esta);
        }
    }
```

```
    }

    void v() // señalar()
    {
        hebra h;
        valor = valor + 1;
        h = bloq.borrar_primero();
        desbloquear(h);
    }

private:
    int valor;
    cola<hebra> bloq;
};
```

Versión general de semáforos

- Sirve para proteger un conjunto de recursos similares
- Se inicializa con el número total de recursos disponibles, N
- “espera” y “señal” se diseñan para impedir el acceso al recurso cuando el semáforo es menor o igual a cero
- Cuando se solicita y obtiene un recurso el semáforo se decrementa
- Cuando se libera un recurso se incrementa
- La ejecución de las operaciones son indivisibles

Operaciones sobre semáforos generales

Operación de inicializar

inicializa (S: SemaforoBinario; v: integer);

Pone el valor del semáforo S al valor de v (n)
n_suspendidos := 0

Operación de espera

espera (S)

```
if S > 0 then S := S - 1
else
  n_suspendidos := n_suspendidos + 1;
  suspender la tarea que hace la
  llamada y ponerla en la cola de
  tareas
```

Operación de señal

señal (S)

```
if n_suspendidos > 0 then
  n_suspendidos := n_suspendidos - 1;
  pasar al estado preparado un
  proceso suspendido
else
  S := S + 1
```

SEMÁFOROS

- Los semáforos binarios fueron introducidos por Dijkstra en 1968
- Un **semáforo binario** es un indicador (S) de condición que registra si un recurso está disponible o no
- El semáforo binario toma dos valores:
 - ◆ $S = 0 \Rightarrow$ El recurso no está disponible
 - ◆ $S = 1 \Rightarrow$ El recurso está disponible
- Los semáforos se implementan con una **cola** a la cual se añaden los procesos que están en espera del recurso

Operaciones sobre semáforos

Operación de inicializar

inicializa (S: SemaforoBinario; v: integer);

Pone el valor del semáforo al valor de v (0 o 1)

Operación de espera

espera (S)

if S=1 then

S:=0

else

suspender la tarea que
hace la llamada y ponerla en
la cola de tareas

Operación de señal

señal (S)

if la cola de tarea está vacía then

S:=1

else

reanudar la primera tarea de la
cola de tareas

SEMÁFOROS: implementación

- Estas operaciones son procedimientos que se implementan como **acciones indivisibles** y por ello el cambio de valor del indicador se efectúa de manera real como una sola operación
- Para ello, en sistemas monoprocesador basta con inhibir las interrupciones durante la ejecución de estas operaciones
- En sistemas multiprocesador, es necesario utilizar instrucciones especiales hardware, o introducir soluciones software como las vistas anteriormente

Los semáforos son un mecanismo de sincronización entre procesos que gestiona el sistema operativo.

Un proceso simplemente tiene que solicitar el uso de un semáforo o de un conjunto de semáforos al que le asignará el nombre que desee realizando la llamada al sistema correspondiente

```

struct semáforo{
    int contador;
    /* Otras variables para implementar el semáforo y su cola
       (dependen de cada sistema operativo) */

void init_sem(semáforo S, int valor)

    S.contador=valor;

void wait_sem(semáforo S)

    /* Bloqueo de las interrupciones */

    S.contador=S.contador-1;

    if (S.contador < 0)
    {
        /* Añadir el proceso a la cola del semáforo */
        /* Bloquear el proceso */
        /* Desbloqueo de las interrupciones */
    }
    else
    {
        /* Desbloqueo de las interrupciones */
    }

void signal_sem(semáforo S)

    /* Bloqueo de las interrupciones */

    S.contador=S.contador+1;

    if (S.contador <= 0)
    {
        /* Eliminar un proceso P de la cola del semáforo */
        /* Añadir el proceso P a la cola de procesos preparados */
    }
    /* Desbloqueo de las interrupciones */

```

Figura 4.7 – Primitivas de las operaciones `init_sem`, `wait_sem` y `signal_sem` de un semáforo general que utilizan el bloqueo de interrupciones para garantizar la exclusión mutua

```

struct semáforo{
    int contador;
    int ocupado;
    /* Otras variables para implementar el semáforo y su cola
       (dependen de cada sistema operativo) */
}

void init_sem(semáforo S, int valor)
{
    S.contador=valor;
}

void wait_sem(semáforo S)
{
    while (!test_set(S.ocupado)); /* Bucle de espera activa */

    S.contador=S.contador-1;

    if (S.contador < 0)
    {
        /* Añadir el proceso a la cola del semáforo */
        /* Bloquear el proceso */
        S.ocupado=0;
    }
}

void signal_sem(semáforo S)
{
    while (!test_set(S.ocupado)); /*Bucle de espera activa */

    S.contador=S.contador+1;

    if (S.contador <= 0)
    {
        /* Eliminar un proceso P de la cola del semáforo */
        /* Añadir el proceso P a la cola de procesos preparados */
    }

    S.ocupado=0;
}

```

Figura 4.8 – Primitivas de las operaciones `init_sem`, `wait_sem` y `signal_sem` de un semáforo general que utilizan una instrucción máquina especial para garantizar la exclusión mutua

Exclusión mutua con semáforos

- Se usa “espera” como procedimiento de bloqueo antes de la sección crítica
- Se usa “señal” como procedimiento de desbloqueo

```
process P1
begin
  loop
    espera(S);
    Sección Crítica
    señal(S);
    (*resto del proceso*)
  end
end P1
```



- ◆ Se emplearán tantos semáforos como clases de secciones críticas se establezcan

```

#define TRUE 1
semáforo S;

void procesoA()
{
    while(TRUE)
    {
        /* sección no crítica */
        wait_sem(S);
        /* sección crítica asociada al recurso R */
        signal_sem(S);
        /* sección no crítica */
    }
}

void procesoB()
{
    while(TRUE)
    {
        /* sección no crítica */
        wait_sem(S);
        /* sección crítica asociada al recurso R */
        signal_sem(S);
        /* sección no crítica */
    }
}

void main()
{
    init_sem(S,1);
    ejecución_concurrente(procesoA,procesoB);
}

```

Figura 4.9 – Solución mediante un semáforo a la exclusión mutua para el caso de dos procesos concurrentes

SINCRONIZACIÓN con semáforos (1/2)

- El uso de semáforos permite la sincronización entre tareas
- Las operaciones de “espera” y “señal” no se utilizan dentro de un mismo proceso sino que se ejecutan en dos procesos separados
- El que ejecuta “espera” queda bloqueado hasta que el otro ejecuta la operación de “señal”

SINCRONIZACIÓN con semáforos (2/2)

```
module Sincronización;  
var sincro:semaforo;
```

```
process P1 (*proceso que espera*)
```

```
begin
```

```
...
```

```
espera(sincro);
```

```
...
```

```
end P1;
```

```
process P2 (*proceso que señala*)
```

```
begin
```

```
...
```

```
señal(sincro);
```

```
...
```

```
end P2;
```

```
begin (*sincronización*)
```

```
inicializa(sincro,0);
```

```
cobegin
```

```
P1;
```

```
P2;
```

```
coend
```

```
end Sincronización
```

```

#define TRUE 1
semáforo S;

void procesoA()
{
    while(TRUE)
    {
        ...
        wait_sem(S); /* Para bloquearse en espera de evento */
        ...
    }
}

void procesoB()
{
    while(TRUE)
    {
        ...
        signal_sem(S); /* Notificación del evento */
        ...
    }
}

void main()
{
    init_sem(S,0);
    ejecución_concurrente(procesoA,procesoB);
}

```

Figura 4.10 – Sincronización de dos procesos concurrentes usando un semáforo

Problema del productor-consumidor (1/3)

```
module Productor_consumidor;  
var BufferComun:buffer;
```

```
process Productor;
```

```
var x:dato;
```

```
begin
```

```
  loop
```

```
    produce(x);
```

```
    while Lleno do
```

```
      (*espera*)
```

```
    end;
```

```
    Poner(x);
```

```
  end
```

```
end Productor;
```

```
process Consumidor;
```

```
var x:dato;
```

```
begin
```

```
  loop
```

```
    while Vacio do  
      (*espera*)
```

```
    end;
```

```
    Tomar(x);
```

```
    Consume(x);
```

```
  end
```

```
end Consumidor;
```

```
begin
```

```
  cobegin
```

```
    Productor;
```

```
    Consumidor;
```

```
  coend
```

```
end Productor_consumidor;
```



Problema del productor-consumidor (2/3)

- Esta solución no es satisfactoria:
 - ◆ Las funciones Poner(x) y Tomar(x) utilizan el mismo buffer lo que plantea el problema de la exclusión mutua
 - ◆ Ambos procesos utilizan una espera ocupada cuando no pueden acceder al buffer

- Soluciones a los dos problemas:
 - ◆ Utilización de un semáforo para proteger el acceso al buffer
 - ◆ Utilización de un semáforo para sincronizar la actividad de los dos procesos

```

#define TRUE 1
#define N 100
semaforo_binario acceso;
semaforo ocupado, disponible;

void productor()
{
    int dato;
    while(TRUE)                /* Bucle infinito */
    {
        dato=producir_dato(); /* Generar dato */
        wait_sem(disponible); /* Comprobar si está lleno */
        wait_sem(acceso);     /* Entrar en sección crítica */
        escribir_buffer(dato); /* Escribir dato en el buffer */
        signal_sem(acceso);    /* Salir de sección crítica */
        signal_sem(ocupado);    /* Avisar al consumidor */
    }
}

void consumidor()
{
    int dato;
    while(TRUE)                /* Bucle Infinito */
    {
        wait_sem(ocupado);     /* Comprobar si está vacío */
        wait_sem(acceso);     /* Entrar en sección crítica */
        leer_buffer(dato);     /* Escribir dato en el buffer */
        signal_sem(acceso);    /* Salir de sección crítica */
        signal_sem(disponible); /* Avisar al productor */
        dato=consumir_dato();  /* Consumir dato */
    }
}

void main()
{
    init_sem(acceso,1);
    init_sem(ocupado,0);
    init_sem(disponible,N);
    ejecución_concurrente(productor, consumidor);
}

```

Arrastre el cursor al final que desea capturar.

Figura 4.11 – Una posible solución al problema de los productores y los consumidores

```

#define TRUE 1
semaforo_binario S1, S2;
int contador=0;

void lector()
{
    while(TRUE)      /* Bucle infinito */
    {
        wait_sem(S2);
        contador=contador+1;
        if (contador==1) wait_sem(S1);
        signal_sem(S2);
        leer_dato();
        wait_sem(S2);
        contador=contador-1;
        if (contador==0) signal_sem(S1);
        signal_sem(S2);
        procesar_dato();
    }
}

void escritor()
{
    while(TRUE)      /* Bucle infinito */
    {
        producir_adquirir_dato();
        wait_sem(S1);
        escribir_dato();
        signal_sem(S1);
    }
}

void main()
{
    init_sem(S1,1);
    init_sem(S2,1);
    ejecución_concurrente(lector, escritor);
}

```

Arrastre el cursor al área que desee capturar.

Figura 4.12 – Una posible solución al problema de los lectores y los escritores

En resumen...

- Los semáforos son una herramienta de coordinación primitiva:
 - para garantizar la **exclusión mutua**.
 - para **sincronizar** hebras.
- Las llamadas a `esperar()` y `señalar()` suelen estar dispersas por varias hebras con lo que es difícil comprender todos sus efectos.
- El uso debe ser correcto en todas estas las hebras.
- Una hebra defectuosa o maliciosa puede estropear el funcionamiento del resto.

Algunos autores recomiendan evitar su uso $\implies$ ¿Qué utilizar en su lugar? ¿Existen mejores métodos de sincronización?

Monitores

- Construcciones de alto nivel de algunos lenguajes de programación.
 - Funcionamiento parecido al de los semáforos binarios.
 - Gestión automática por parte del lenguaje.
- Ofrecido en lenguajes de programación concurrente como Java, Modula-3, Pascal concurrente,...
- Internamente puede ser implementado mediante semáforos u otros mecanismos de sincronización.

Definición de monitor

- Un módulo software (clase) que contiene:
 - Una interfaz con uno o más **procedimientos**.
 - Una secuencia de **inicialización**.
 - **Variables** de datos locales.
- Características:
 - Las variables locales sólo son accesibles mediante los procedimientos del monitor.
 - Una hebra entra en el monitor mediante la invocación de uno de sus procedimientos.
 - Sólo una hebra puede estar ejecutando algún procedimiento del monitor → los monitores proporcionan **exclusión mutua** de forma automática.

Características de los monitores

- Los monitores aseguran la **exclusión mutua** $\implies$ **no** es necesario programar esta restricción **explícitamente**.
- Los **datos compartidos** son **protegidos automáticamente** si los colocamos dentro de un monitor.
- Los monitores **bloquean** sus datos cuando una hebra entra.
- **Sincronización** adicional puede conseguirse **dentro** de un monitor mediante el uso de **variables condición**.
- Una variable de condición representa una condición (evento) que tiene que cumplirse antes de que una hebra pueda **continuar** la ejecución de alguno de los procedimientos del monitor.

- Los monitores no proporcionan por sí mismos un mecanismo para la sincronización de tareas, es necesario añadir variables llamadas de **condición**
 - ◆ A cada causa por la que un proceso deba esperar se asocia una variable de condición
 - ◆ Sobre ellas sólo pueden actuar dos procedimientos: espera (siempre suspende al proceso que la emite) y señal



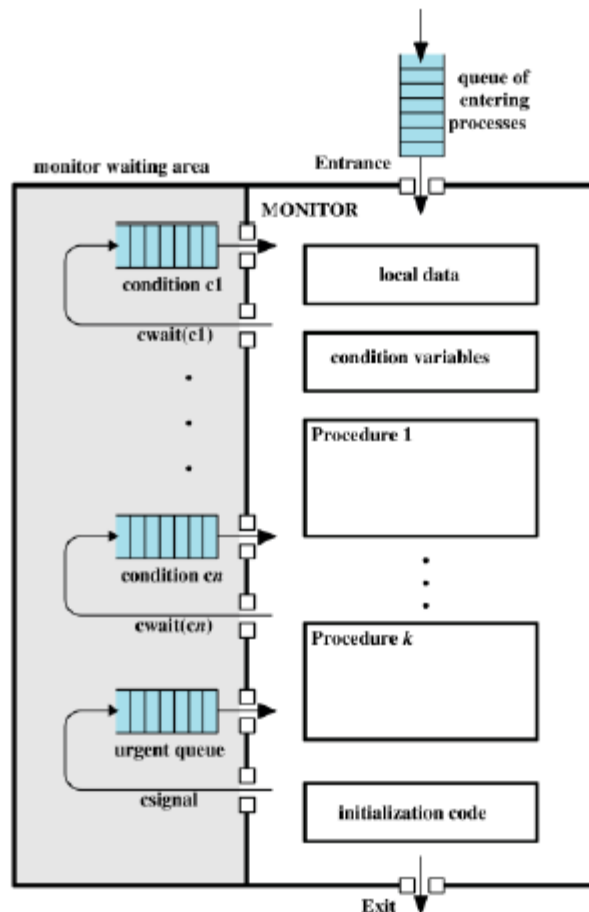
Variables condición

- Variables locales al monitor y sólo **accesibles** desde su **interior** mediante las operaciones:

`esperar()`: Bloquea la ejecución de la hebra llamadora sobre la variable condición. La hebra bloqueada sólo puede continuar su ejecución si otra ejecuta `señalar()`.

`señalar()`: Reactiva la ejecución de alguna hebra bloqueada en la variable condición. Si hay varias podemos escoger una cualquiera. Si no hay ninguna no hace nada.

Monitores y variables condición



- Las hebras pueden estar esperando en la cola de entrada o en la de alguna variable de condición.
- Una hebra se pone en la cola de espera al invocar esperar() sobre una variable de condición.
- señalar() permite a una hebra que estuviese bloqueada en la cola de una condición continuar.
- señalar() bloquea a la hebra llamadora y la pone en la cola urgente a menos que sea la última operación del procedimiento (¿?).

Sintaxis del monitor

monitor nombre_monitor;

declaración de los tipos y procedimientos que se importan y exportan /***export***/

declaración de las variables locales del monitor

declaración de las variables de condición /***condición***/

procedure Prc1 (...);

begin ... **end**;

...

procedure Prc2 (...);

begin ... **end**;

...

procedure Prcm (...);

begin ... **end**;

begin

inicialización del monitor

end

Monitor: implementación de un semáforo

```
monitor:semaforo;  
  
from condiciones import condicion, espera  
export  
    sespera, sseñal;  
var  
    ocupado: boolean;  
    :condicion  
  
procudere sseñal(var libre:condicion);  
begin  
    ocupado:=false;  
    señal(libre);  
end sseñal;
```

```
procedure sespera(var libre condicion);  
begin  
    if ocupado then  
        espera(libre);  
        ocupado:=true;  
    end  
end sespera  
  
begin  
    ocupado:=false;  
end
```

Monitor: problema del productor-consumidor

monitor: productor-consumidor;

from condiciones **import**
condicion, espera, señal;

export poner, tomar;

const tamaño=32;

var

buff:array[0..tamaño-1] of dato;
cima,base:0..tamaño-1;
espacio,itemdisponible:**condicion**
numeroenbuffer:integer;

begin (*inicialización*)

numeroenbuffer:=0;

cima:=0;

base:=0;



procedure poner(item:dato);

begin

if numeroenbuffer=tamaño **then**

espera(espacio)

end;

buff[cima]:=item;

numeroenbuffer:=

numeroenbuffer+1;

cima:=(cima+1) mod tamaño;

señal(itemdisponible);

end poner;

procedure tomar(var item:dato);

begin

if numeroenbuffer=0 **then**

espera(itemdisponible);

end;

item:=buff[base];

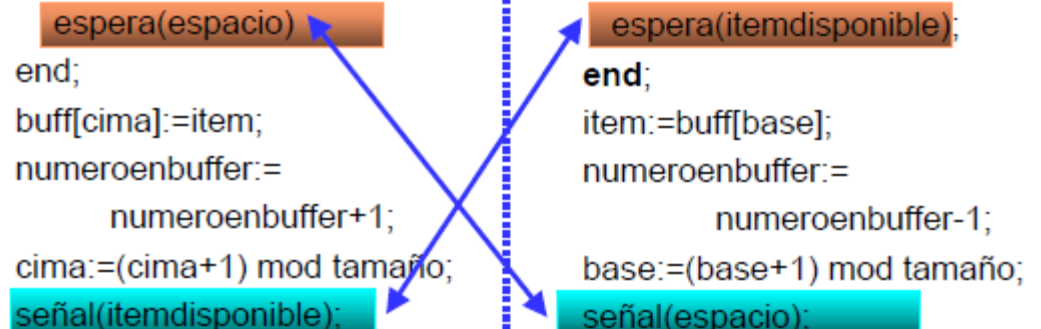
numeroenbuffer:=

numeroenbuffer-1;

base:=(base+1) mod tamaño;

señal(espacio);

end tomar;



```

#define TRUE 1
#define N 100 /* Capacidad del buffer */
monitor mon1
    condición lleno, vacio;
    int contador;
    char buffer[N];
    void escribir(int dato) /* Procedimiento del monitor */
    {
        if (contador==N) wait_mon(lleno);
        insertar_dato(buffer,dato);
        contador=contador+1;
        if (contador==1) signal_mon(vacio);
    }
    void leer(int dato); /*Procedimiento del monitor */
    {
        if (contador==0) wait_mon(vacio);
        extraer_dato(buffer, dato);
        contador=contador-1;
        if (contador==N-1) signal(lleno);
    }
    {
        contador=0; /* Inicialización del monitor */
    }
end monitor

void productor()
{
    while(TRUE)
    {
        dato=producir_dato();
        mon1.escribir(dato);
    }
}

void consumidor()
{
    while(TRUE)
    {
        mon1.leer(dato);
        consumir_dato(dato);
    }
}

void main()
{
    ejecución_concurrente(productor, consumidor)
}

```

Figura 4.15 – Solución con un monitor al problema de los productores y los consumidores

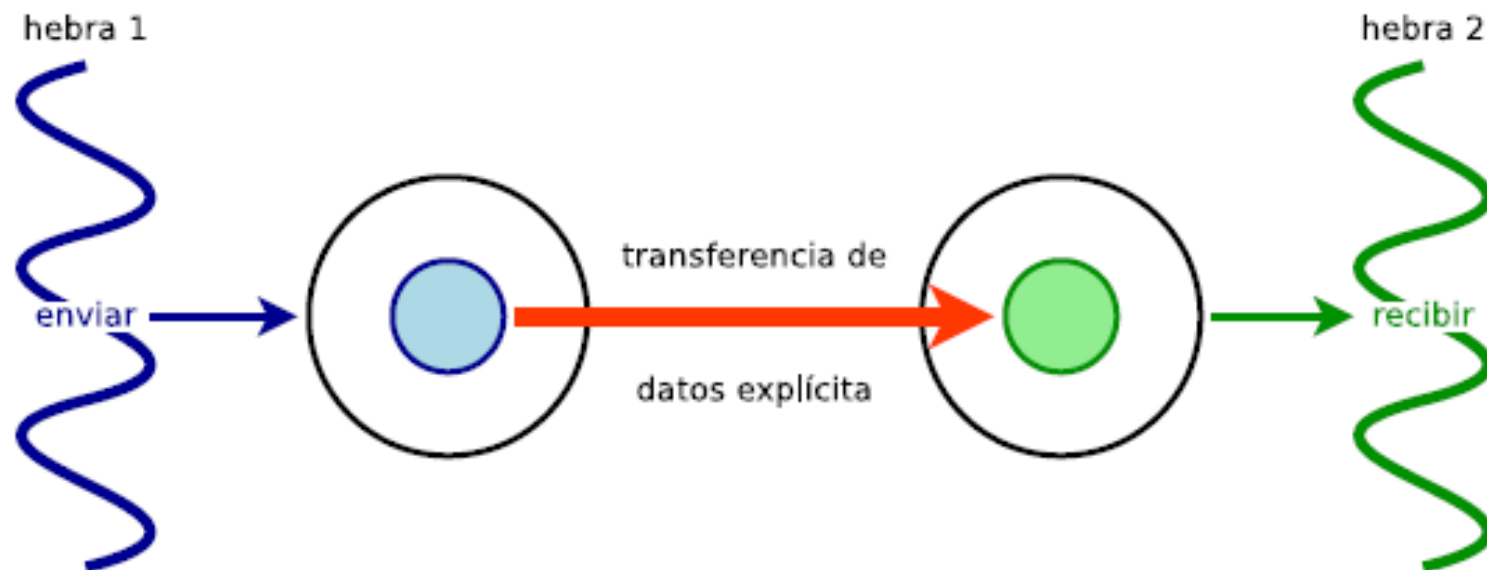
Paso de mensajes

- ¿De verdad necesitamos otro mecanismo de interacción como el **paso de mensajes**?
- **Si**, por supuesto, porque...
 - Los mecanismos ya vistos requieren **memoria compartida** $\Rightarrow$ estas soluciones no funcionarán en **sistemas distribuidos**.
 - Las hebras de diferentes aplicaciones necesitan **protección** $\Rightarrow$ incluso en sistemas con memoria compartida no siempre queremos abrir nuestro espacio de direcciones a terceros en los que podríamos **no confiar**.
 - Para conseguir que el **núcleo** del sistema operativo sea **muy pequeño** algunos arquitectos de sistemas sólo ofrecen paso de mensajes como mecanismo de comunicación entre procesos.

Paso de mensajes

- La comunicación entre procesos (*"Inter-Process Communication"* o IPC) se usa entre hebras dentro de...
 - un sistema distribuido.
 - un mismo ordenador.
 - un mismo espacio de direcciones.
- El motivo es poder realizar...
 - Intercambio de información.
 - Otra forma de sincronización y paso de señales.
- Requiere de al menos dos primitivas:
 - `enviar(receptor, mensaje)`
 - `recibir(emisor, mensaje)`
- ¿Cómo implementarlo?
 - ¿Llamada al sistema?
 - Pista: la hebras en comunicación pueden bloquearse.

Principios básicos del paso de mensajes



- Las hebras de **un proceso** cooperan mediante **variables globales**, ej: el búfer en el problema productor/consumidor.
- Las hebras de **diferentes procesos** interaccionan mediante **paso de mensajes** si no pueden establecer regiones de **memoria compartida** o si no quieren compartir regiones de memoria.

Problemas del paso de mensajes

La **inconsistencia** de datos sigue siendo un problema porque...

- Los mensajes pueden llegar **desordenados** $\implies$ incluso aunque cada mensaje sea correcto, su concatenación es incorrecta.
- Los mensajes pueden estar **incompletos** porque el receptor no tenga suficiente espacio libre en el búfer de recepción.
- Los mensajes pueden **perderse**.
- Los mensajes pueden estar **desfasados**, ej: transcurrido cierto tiempo un mensaje puede dejar de reflejar el estado del emisor.
- Cada mensaje podría suponer un problema de **seguridad**: goteo de información, desbordamiento, DoS $\implies$ por este motivo deberíamos poder controlar si los mensajes deben ser enviados o no.

MENSAJES

- Permiten sincronización y comunicación entre procesos
- La comunicación entre mensajes necesita de un proceso **emisor** y de uno **receptor** y la información que debe intercambiarse
- Las operaciones básicas son:
 - ◆ **enviar**(mensaje)
 - ◆ **recibir**(mensaje)
- La comunicación por mensajes requiere que se establezca un enlace entre el receptor y el emisor

Modos de nombrar los mensajes

- **Comunicación directa:** nombran de forma explícita al proceso con el que se comunican
 - ◆ `enviar(Q, mensaje)` $\Rightarrow$ Envía un mensaje al proceso Q
 - ◆ `recibir(P, mensaje)` $\Rightarrow$ Recibe un mensaje del proceso P
- **Comunicación indirecta** los mensajes se envían y reciben a través de un **buzón o puerto**:
 - ◆ `enviar(buzónA, mensaje)` $\Rightarrow$ Envía un mensaje al buzón A
 - ◆ `recibir(buzónA, mensaje)` $\Rightarrow$ Recibe un mensaje del buzón A

Tipos de direccionamiento

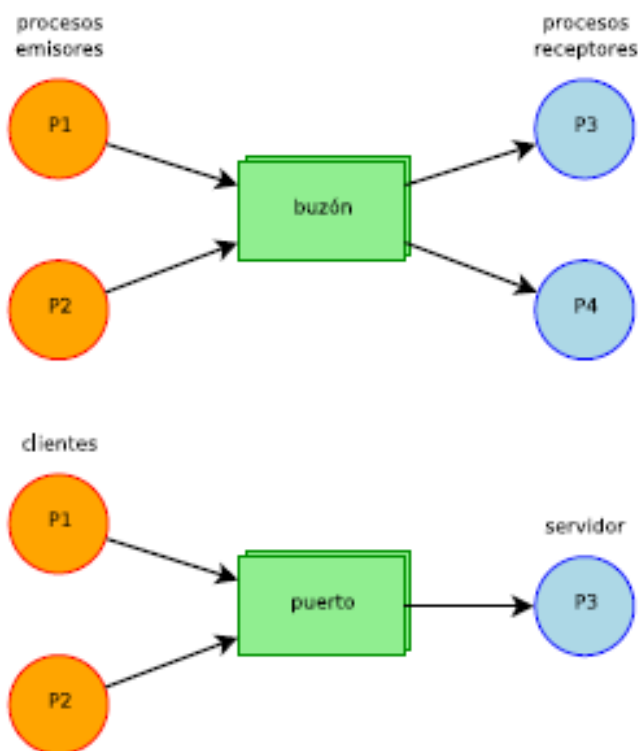
Direccionamiento **directo**:

- `enviar(hebra, mensaje)`
- `recibir(hebra, mensaje)`
- `enviar(filtro(hebra), mensaje)`
- `recibir(filtro(hebra), mensaje)`

Direccionamiento **indirecto**:

- `enviar(buzón, mensaje)`
- `recibir(buzón, mensaje)`
- `enviar(puerto, mensaje)`
- `recibir(puerto, mensaje)`

Buzones/canales frente a puertos



- Un buzón puede ser privado a un par de emisores/receptores.
- El mismo **buzón** puede ser compartido por **múltiples emisores y receptores**.
- Algunos SO permiten distinguir los mensajes por su tipo (Unix).
- Un **puerto** en cambio es un buzón asociado exactamente a un **receptor** y a cualquier número de emisores.
- Los puertos suele emplearse en aplicaciones cliente/servidor en las que el único receptor es el servidor.

Modelos de sincronización (1/2)

- **Síncrona:** El proceso que envía sólo prosigue su ejecución cuando el mensaje ha sido recibido
- **Asíncrona:** El proceso que envía sigue su ejecución sin preocuparse de si el mensaje se recibe o no
- **Invocación remota:** El proceso que envía el mensaje sólo prosigue su ejecución cuando ha recibido una respuesta del receptor

Modelos de sincronización (2/2)

- En la operación de recibir si el mensaje no está presente
 - ◆ El proceso se suspende (bloqueo) o
 - ◆ El proceso devuelve un mensaje de error (sin bloqueo): *Aceptar*
 - ◆ Se especifica un tiempo máximo de espera, pasado ese tiempo el s.o. desbloquea al proceso suspendido y envía un mensaje de error

Estructura de los mensajes

- El intercambio de información puede ser:

- ◆ Por valor
- ◆ Por referencia

- La estructura de los mensajes:

- ◆ Longitud fija
- ◆ Longitud variable
- ◆ De tipo definido

Cabeza

Cuerpo



Mensaje: implementación de un semáforo

```
module semaforo;
```

```
type mensaje=...; (tipo del mensaje*)
```

```
const nulo=...; (*mensaje vacío*)
```

```
procedure espera(var S:semaforo);
```

```
  var temp:mensaje;
```

```
  begin
```

```
    recibir(Sbuzon,temp);
```

```
    S:=0;
```

```
  end espera;
```

```
procedure señal (var S:integer);
```

```
  begin
```

```
    enviar(Sbuzon,nulo);
```

```
  end señal;
```

```
procedure inicializa (varS:integer; valor:boolean);
```

```
  begin
```

```
    if valor=1 then
```

```
      enviar(Sbuzon,nulo);
```

```
    end
```

```
    S:=valor;
```

```
  end inicializa;
```

```
begin
```

```
  crear_buzon(Sbuzon);
```

```
end {semaforo}
```

```

#define TRUE 1
#define N 5 /* Número de procesos */

void Proceso(int i)
{
    mensaje testigo;
    while(TRUE)
    {
        /* sección no critica */
        receive(buzon1, testigo);
        /* sección critica */
        send(buzon1, testigo);
        /* sección no critica */
    }
}

void main()
{
    mensaje nulo;
    crear_buzón(buzon1);
    send(buzon1, nulo);
    ejecución_concurrente(proceso(1), ..., proceso(N));
}

```

Figura 4.16 – Exclusión mutua mediante paso de mensajes

```

#define TRUE 1;

void ProcesoA()
{
    mensaje X;

    while(TRUE)
    {
        /* Código previo */
        receive(buzon1,X); /* Esperar a que el proceso B realice una acción */
        /* Código posterior */
    }

}

void ProcesoB()
{
    mensaje Y;

    while(TRUE)
    {
        /* Realizar acción para el proceso A */
        send(buzon1,Y); /* Avisar al proceso A */
        /* Código posterior */
    }

}

void main()
{
    crear_buzón(buzon1);
    ejecución_concurrente(procesoA, procesoB);
}

```

Figura 4.17 – Sincronización de dos procesos mediante paso de mensajes

4. (2 p) 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 solo uno de ellos puede columpiarse. Escribir el pseudocódigo basado en C de un programa que usando **paso de mensajes** coordine la actividad de los canarios. Suponer que la comunicación es indirecta a través de buzones y que se dispone de la operación `send` sin bloqueo y de la operación `receive` con bloqueo. Dicho programa debe tener tres partes: declaración de variables, código del proceso canario, y código de la función principal para inicializar los buzones y lanzar la ejecución concurrente de los procesos.

Solución Ejercicio 4

La solución que se propone en la Figura 1 para este problema utiliza los siguientes buzones:

- B1. Buzón que se utiliza para garantizar la exclusión mutua en el uso del plato. Se inicializa con tres mensajes ya que solo pueden comer del plato tres canarios simultáneamente.
- B2. Buzón que se utiliza para garantizar la exclusión mutua en el uso del columpio. Se inicializa con un mensaje ya que solo puede columpiarse un canario simultáneamente.

```
void main() /* Inicialización de buzones y ejecución concurrente */
{
    int h;
    mensaje nulo;

    crear_buzón(B1);
    for(h=1;h<=3;h++)
    {
        send(B1,nulo);
    }

    crear_buzón(B2);
    send(B2,nulo);

    ejecución_concurrente(canario,canario,canario)
}

void canario() /* Proceso canario */
{
    mensaje m;

    receive(B1,m);
    comer_del_plato();
    send(B1,m);

    receive(B2,m);
    columpiarse();
    send(B2,m);
}
```


Mensajes: problema del productor-consumidor

module: productor-consumidor;

type mensaje=...; (*tipo del mensaje*)

const nulo=...; (*mensaje vacío*)
tamañoQ=...; (*tamaño de la cola
de los buzones*)

var n: integer;

begin (*inicialización*)

crear_buzon(buzonP);

crear_buzon(buzonC);

for n:=1 to tamañoQ do

 enviar(buzonP, nulo)

end

cobegin

 Productor1;

 Consumidor1;

...

coend

end

process Productor1;

var x: mensaje;

begin

loop

 recibir(buzonP,x);

 produce_datos_y_mensaje;

 enviar (buzonC,x);

 resto de operacionesP;

end

end Productor1;

process Consumidor1;

var y: mensaje;

begin

loop

 recibir(buzonC,y);

 consume_datos_y_mensaje;

 enviar (buzonP,y);

 resto de operacionesC;

end

end Consumidor1;



```

#define TRUE 1
#define N 100 /* Tamaño del buffer */

void productor()
{
    mensaje X;

    while(TRUE)
    {
        receive(buzón_produzca,X);
        producir_dato(X);
        send(buzón_consuma,X);
    }
}

void consumidor()
{
    mensaje Y;

    while(TRUE)
    {
        receive(buzón_consuma,Y);
        consumir_dato(Y);
        send(buzón_produzca,Y);
    }
}

void main()
{
    int h;
    mensaje nulo;

    crear_buzón(buzón_consuma);
    crear_buzón(buzón_produzca);

    for(h=1; h<=N,h++)
    {
        send(buzon_produzca,nulo);
    }

    ejecución_concurrente(productor, consumidor);
}

```

Figura 4.18 – Solución mediante paso de mensajes al problema de los productores y los consumidores