SISTEMAS OPERATIVOS

UNIDAD 4

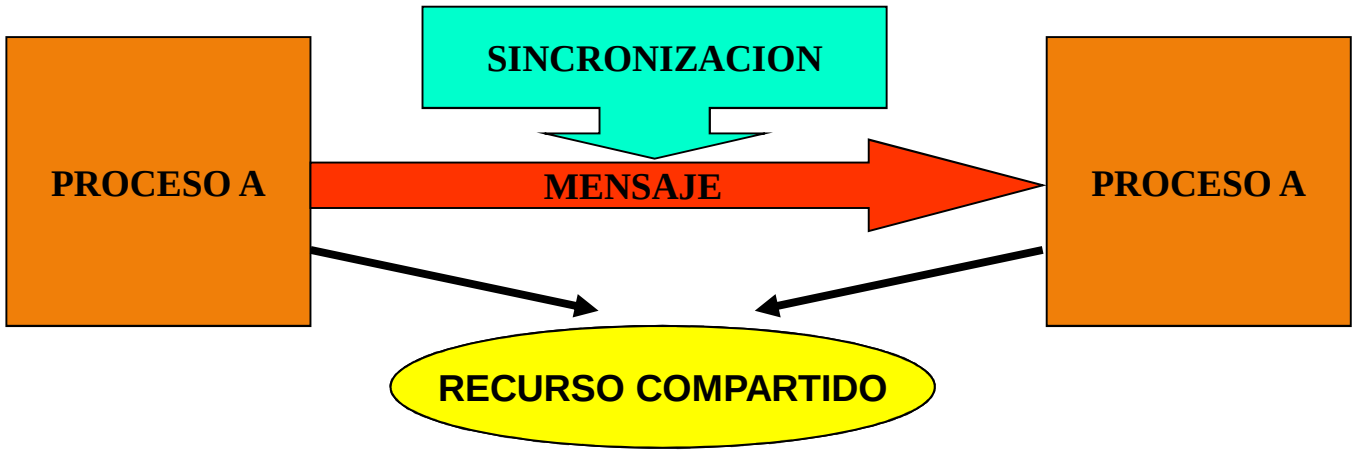
SINCRONIZACION DE PROCESOS

Cada vez que quieras conocer a fondo alguna cosa, confiasela al tiempo.
(Salvador Dali)

CONCURRENCIA –Principios generales

TIPOS DE PROGRAMAS

- **SECUENCIALES: EJEC. SECUENCIAL DE PROCESOS Y/O TAREAS**
- **CONCURRENTES: CUANDO DOS PROCESOS SEC. SE EJECUTAN EN PARALELO, SUPERPONIENDOSE EN EL TIEMPO. DETERMINAN:.**
  - **SINCRONIZACION**
  - **COMUNICACION**



CONCURRENCIA (CONCEPTOS)

- Comunicación entre procesos.
- Compartición y competencia por los recursos.
- Sincronización de la ejecución de varios procesos.
- Asignación del tiempo de procesador a los procesos.

- **Múltiples aplicaciones:**
  - **Multiprogramación.**
- **Aplicaciones estructuradas:**
  - **Algunas aplicaciones pueden implementarse eficazmente como un conjunto de procesos concurrentes.**
- **Estructura del sistema operativo:**
  - **Algunos sistemas operativos están implementados como un conjunto de procesos o hilos.**

PROBLEMAS CON LA CONCURRENCIA

- Compartir recursos globales.
- Gestionar la asignación óptima de recursos.
- Localizar un error de programación.

UN EJEMPLO

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

EJEMPLO CON DOS PROCESOS

```
  Proceso P1          Proceso P2
.                       .
ent = getchar();        .
.                       ent = getchar();
sal = ent;              sal = ent;
putchar(sal);           .
.                       putchar(sal);
.                       .
```

OPERACIONES DEL SISTEMA OPERATIVO

- Seguir la pista de los distintos procesos activos.
- Asignar y retirar los recursos:
  - Tiempo de procesador.
  - Memoria.
  - Archivos.
  - Dispositivos de E/S.
- Proteger los datos y los recursos físicos.
- Los resultados de un proceso deben ser independientes de la velocidad relativa a la que se realiza la ejecución de otros procesos concurrentes.

INTERACCION ENTRÉ PROCESOS

GRADO DE CONOCIMIENTO	RELACION	INFLUENCIA DE UN PROC EN OTROS	POSIBLES PROBL. DE CONTROL
LOS PROC. NO TIENEN CONOCIMIENTO DE LOS DEMAS	COMPETENCIA	1. Los resultados de un procesos son independientes. 2. Los tiempos de los procesos pueden verse afectados	**1. EXC. MUTUA** **2. INTERBLOQUEO (RECURSOS RENOVABLES)** **3. INANICION.**
LOS PROC. SE CONOCEN INDIRECTAMENTE (pe.. OBJ. COMPARTIDOS)	COOPERACION POR COMPARTIMIENTO	1. Los resultados de un proceso pueden depender de la infor. obt. de los otros. 2. Los tiempos de los procesos pueden verse afectados	**1. EXC. MUTUA** **2. INTERBLOQUEO (RECURSOS RENOVABLES)** **3. INANICION.** **4. COHERENCIA DE LOS DATOS**
LOS PROC. SE CONOCEN DIRECTAMENTE (EXISTEN PRIMITIVAS DE COMUNICACION)	COOPERACION POR COMUNICACION	1. Los resultados de un proceso pueden depender de la infor. obt. de los otros. 2. Los tiempos de los procesos pueden verse afectados	**1. INTERBLOQUEO (RECURSOS RENOVABLES)** **2. INANICION.**

COMPETENCIA ENTRE LOS PROCESOS

- Exclusión mutua
  - Secciones críticas:
    - Sólo un programa puede acceder a su sección crítica en un momento dado.
    - Por ejemplo, sólo se permite que un proceso envíe una orden a la impresora en un momento dado.
- Interbloqueo.
- Inanición

COOPERACION POR COMPARTIMIENTO

- Las operaciones de escritura deben ser mutuamente excluyentes.
- Las secciones críticas garantizan la integridad de los datos.

COOPERACION POR COMUNICACION

- Paso de mensajes:
  - No es necesario el control de la exclusión mutua.
- Puede producirse un interbloqueo:
  - Cada proceso puede estar esperando una comunicación del otro.
- Puede producirse inanición:
  - Dos procesos se están mandando mensajes mientras que otro proceso está esperando recibir un mensaje.

2. EXCLUSION MUTUA: REQUISITOS

1. Sólo un proceso debe tener permiso para entrar en la sección crítica por un recurso en un instante dado.
2. Un proceso que se interrumpe en una sección crítica debe hacerlo sin interferir con los otros procesos.
3. No puede permitirse el interbloqueo o la inanición.

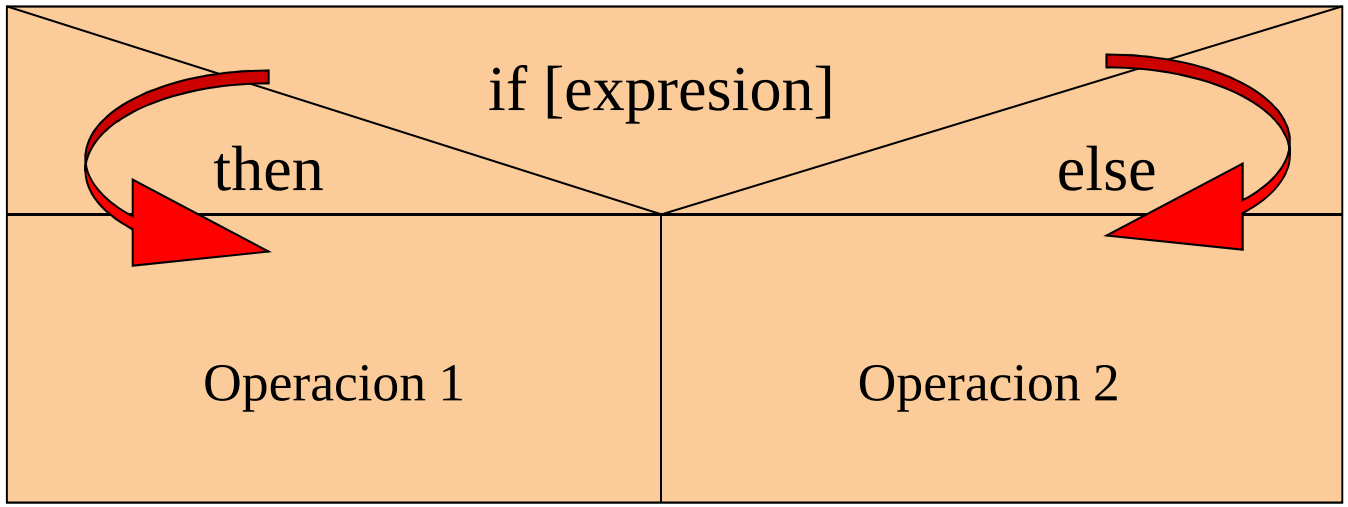
EXCLUSION MUTUA: REQUISITOS

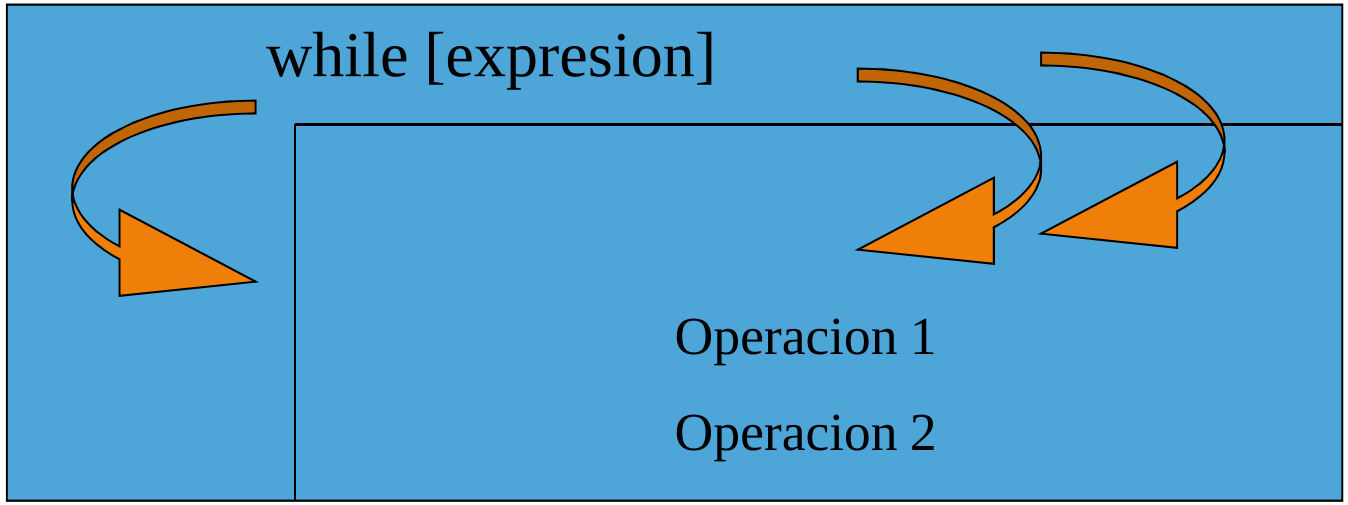
4. Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.
5. No se deben hacer suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.
6. Un proceso permanece en su sección crítica sólo por un tiempo finito.

EXCLUSION MUTUA: PSEUDOCODIGO

```
/* programa Exclusion Mutua */
const int n = /*número de procesos */;
void P(int i)
{ while(cierto)
{     entrada_critica(i);
      /*Sección Crítica*/;
      salida_critica(i);
      /*resto*/;
}}
void main()
{parbegin(P(R1),P(R2),…,P(Rn));}
```

DIAGRAMA DE ESTRUCTURAS DE CONTROL





EXCLUSION MUTUA: ALGORITMO EN BLOQUE

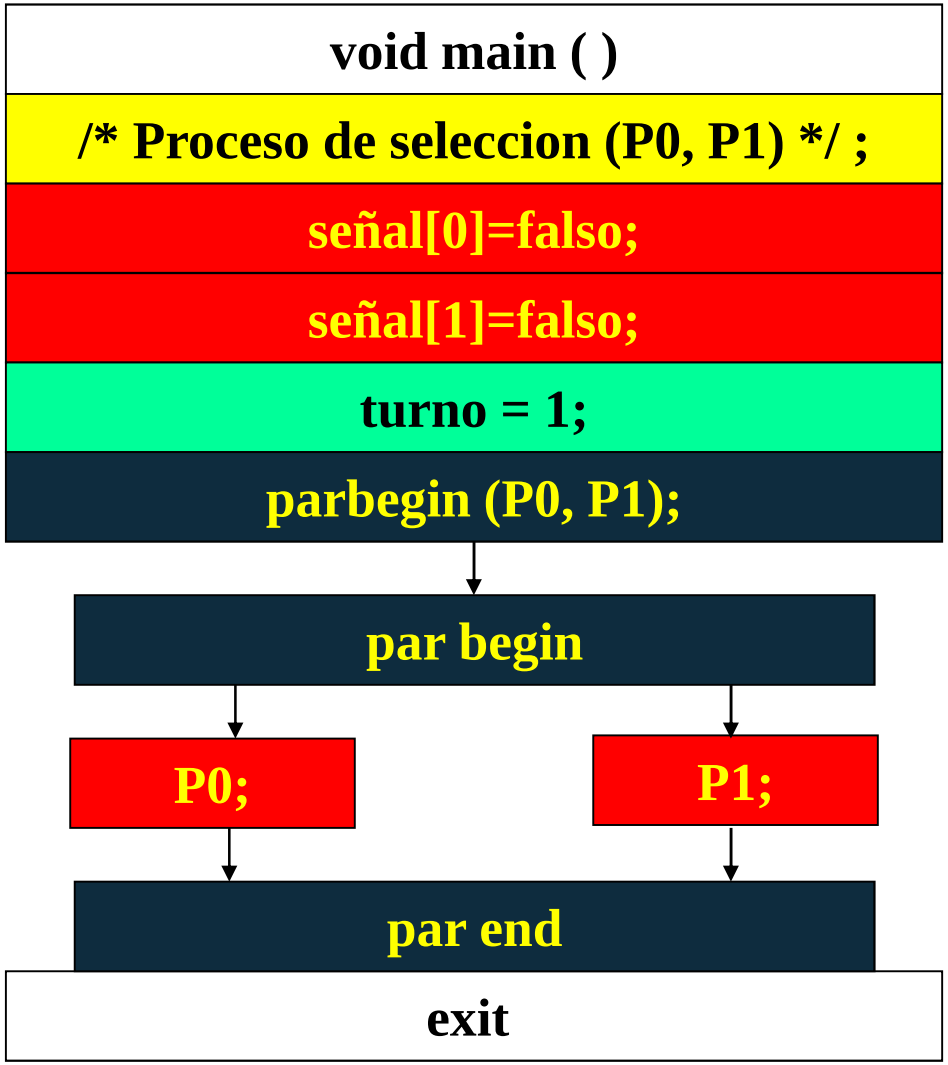
```
const  int n= ;
void P(int i)
while (cierto)
    entrada_crítica(i) ;
    /*seccion critica */ ;
    salida_crítica(i) ;
    /* resto */ ;
void main( )
parbegin(P(R1),P(R2),…,P(Rn));
exit( )
```

ALGORITMO DE DEKKER

```
Boolean señal[2];
int turno
void P0( )
while (cierto)
    señal [0] = cierto ;
    while
    (   if (turno= = 1)
        else                then
        señal [0] = falso ;
        while (turno = = 1)
            /* no hacer nada */ ;
        señal [0] = cierto ;
    /* seccion critica */ ;
    turno = 1 ;
    señal [0] = falso ;
    /* resto */ ;
```

```
void P1( )
while (cierto)
    señal [1] = cierto ;
    while
    (   if (turno= = 0)
        else                then
        señal [1] = falso ;
        while (turno = = 0)
            /* no hacer nada */ ;
        señal [1] = cierto ;
    /* seccion critica */ ;
    turno = 0 ;
    señal [1] = falso ;
    /* resto */ ;
```

ALGORITMO DE DEKKER



ALGORITMO DE DEKKER (Conclusiones)

1. El estado de ambos procesos esta por la variable señal[ ]
2. La variable turno indica cual proceso tiene prioridad para exigir la entrada a SC
3. El Algoritmo de Dekker resuelve el problema, pero es muy complejo.
4. Es difícil de demostrar la corrección.

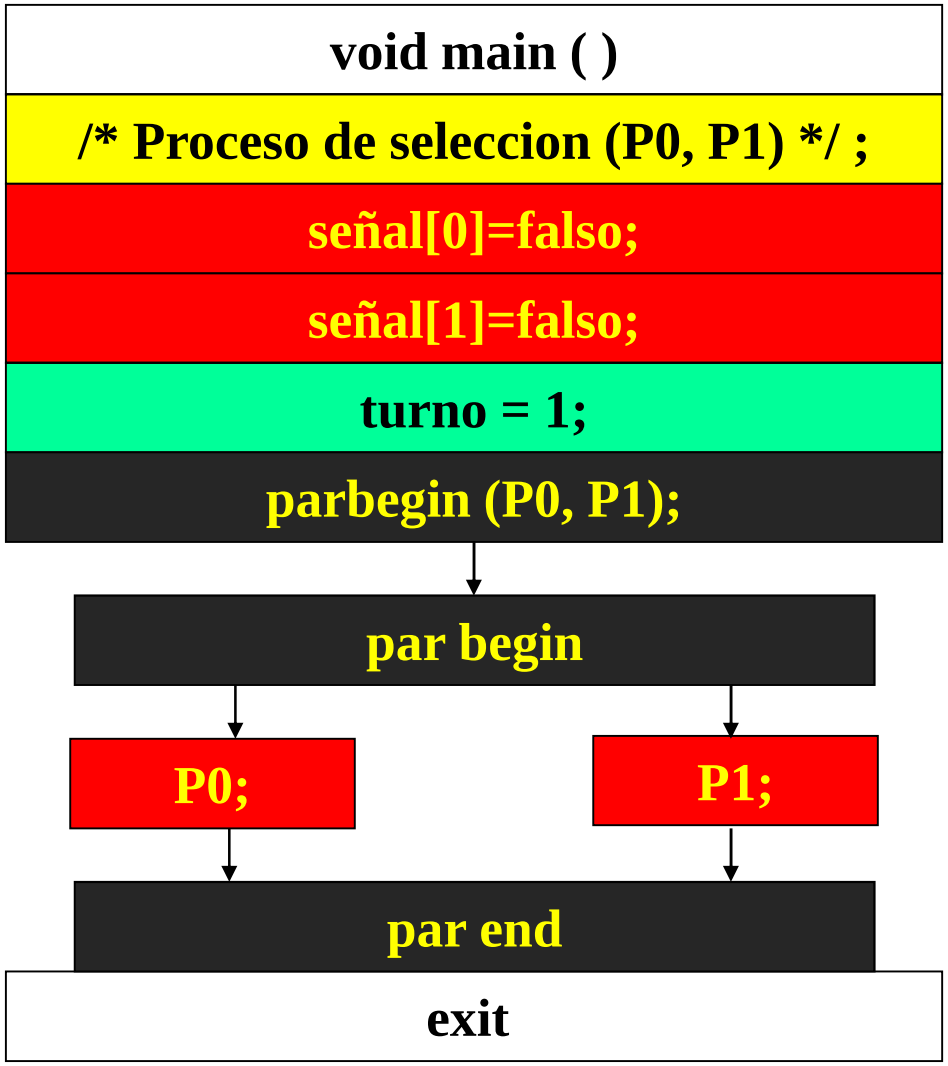
ALGORITMO DE PETERSON

```
boolean señal[2];
int turno

void P0( )
while (cierto)
    señal [0] = cierto ;
    turno = 1 ;
    while (señal[1] && turno = =
        /* no hacer nada */ ;
    /* seccion critica */ ;
    señal [0] = falso ;
    /* resto */ ;
```

```
void P1( )
while (cierto)
    señal [1] = cierto ;
    turno = 0 ;
    while (señal[0] && turno = =
        /* no hacer nada */ ;
    /* seccion critica */ ;
    señal [1] = falso ;
    /* resto */ ;
```

ALGORITMO DE PETERSON



ALGORITMO DE PETERSON (Conclusiones)

1. El estado de ambos procesos esta dado por la variable señal[ ]
2. La variable turno indica cual proceso tiene prioridad para exigir la entrada a SC

   Si P0: señal[0] = cierto => P1 no entra en SC

   Si P1: señal[1] = cierto => P0 no entra en SC.
3. Ninguno de los dos procesos puede monopolizar el Procesador, dado que P1 siempre está obligado a dar oportunidad a P0 poniendo turno = 0, antes de intentar entrar en su propia SC nuevamente.

Se puede generalizar para n-procesos.

EXM: SOLUCIONES POR Hw

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

EXM: SOLUCIONES POR Hw

Inhabilitar Interrupciones

```
while (cierto)
        /* inhabilitar IRQs */ ;
        /*seccion critica */ ;
        /* habilitar IRQs */ ;
        /* resto */ ;
```

EXM: SOLUCIONES POR Hw

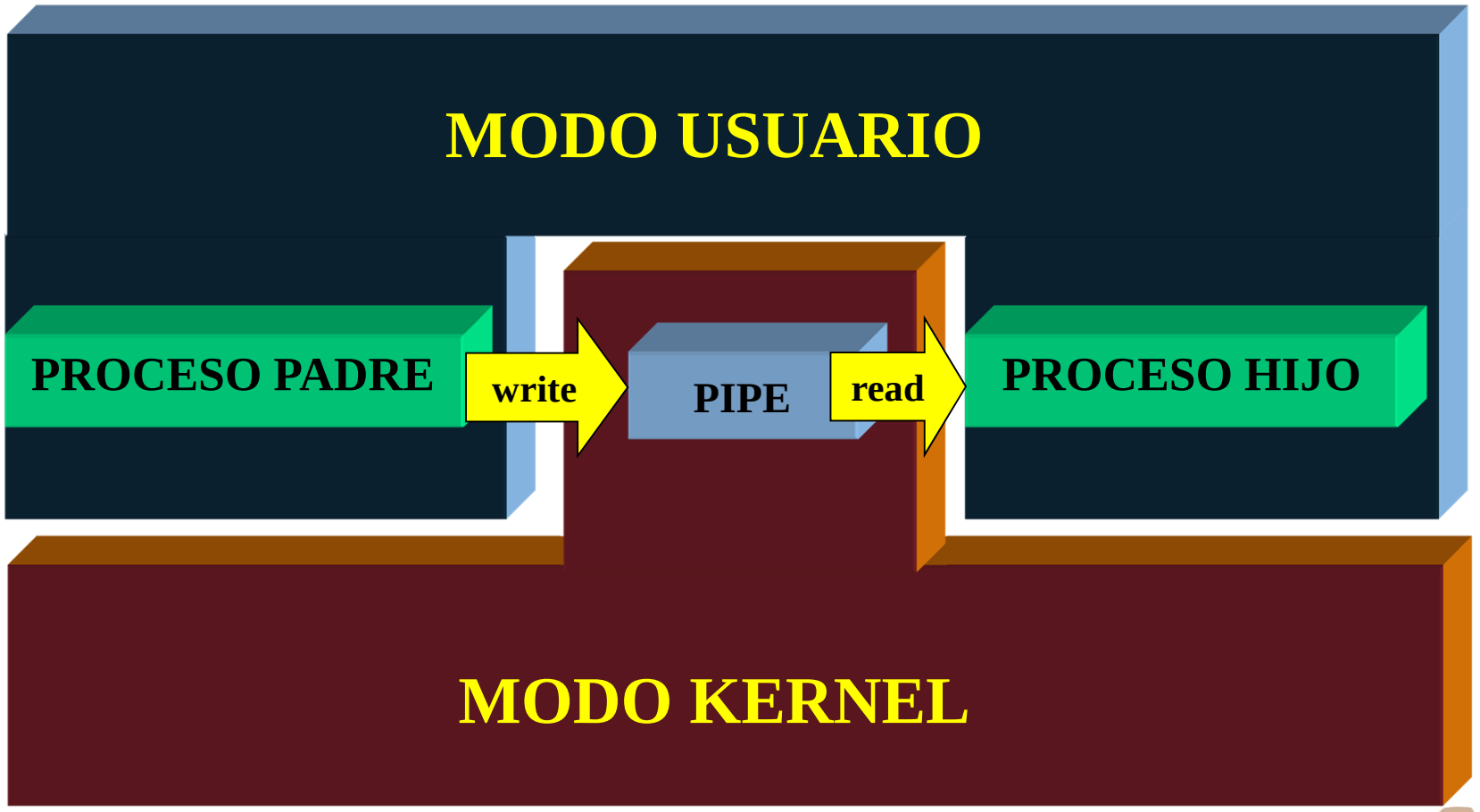
- **Instrucciones especiales de máquina:**
  - Se realizan en un único ciclo de instrucción.
  - No están sujetas a injerencias por parte de otras instrucciones.
  - Leer y escribir.
  - Leer y examinar.

IPC (INTERPROCESS COMUNICATIONS)

TIPOS

1. PIPES & FIFOs
2. MEMORIA COMPARTIDA
3. SEMAFOROS & COLAS DE MENSAJES
4. SOCKETS (PROGRAMACION)
5. PASAJE DE MENSAJES
6. IRQs & SIGNALs
7. MONITORES

PIPES & FIFOs


MODO USUARIO
PROCESO PADRE
write
PIPE
read
PROCESO HIJO
MODO KERNEL

PIPES & FIFOs

- Unidireccionales
- Tipo half-duplex
- Un lado escribe/otro lado se hace lectura
- Pipe: No se integra en el File System
- FIFO: Se registra en el File System
- Toda tubería asocia 2 descriptores y un i-nodo.
- Archivo de llegada se ejecuta en Bkground, previo al de inicio.
- Se prefiere FIFO, para comunicación entre procesos de distinto padre.

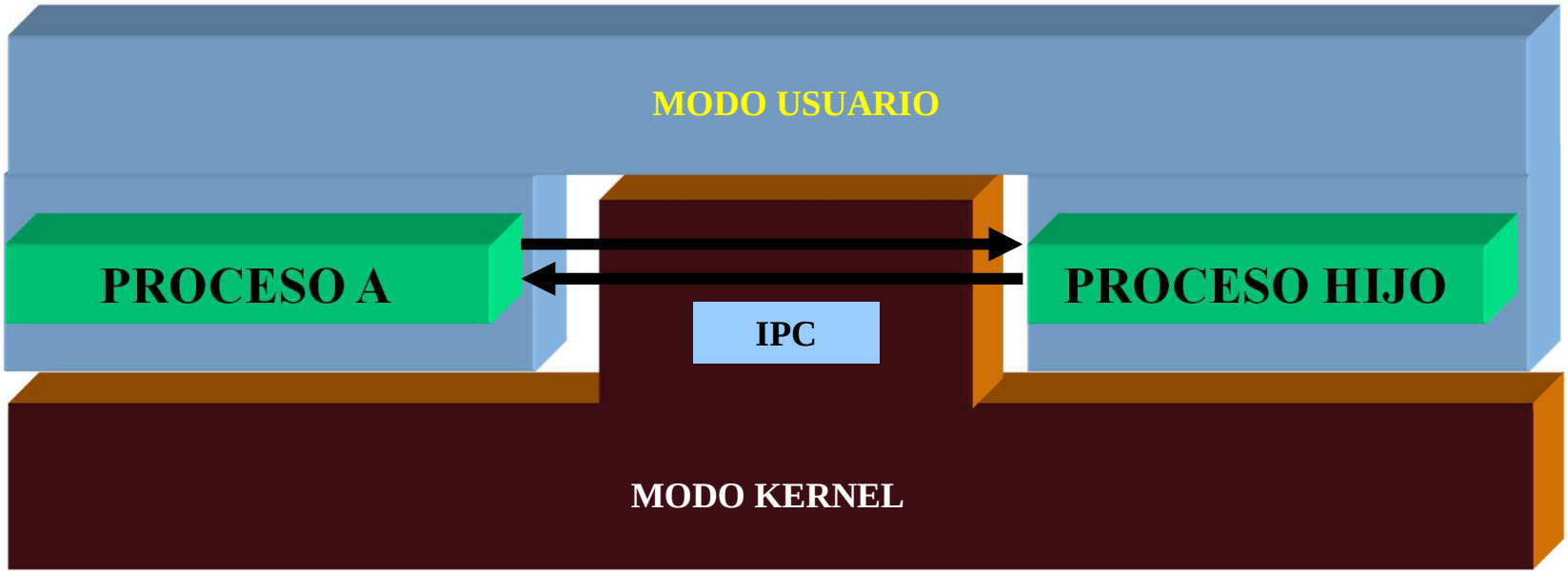
MEMORIA COMPARTIDA

TIPOS

1. SEMAFOROS
2. COLAS DE MENSAJES
3. SEGMENTOS COMPARTIDOS

- Trabajan en modo kernel
- Son IPC sys V.
- Comunican procesos no relacionados. Sin padre en común.
- Todo IPC se mapea en la memoria protegida del núcleo.
- Se crean con: semget (semáforo), msgget (cola de mensajes) y shmget (segm. Compartido)
- Pertenece a la librería <sys/types.h>
- Todo IPC tiene un IDE, único

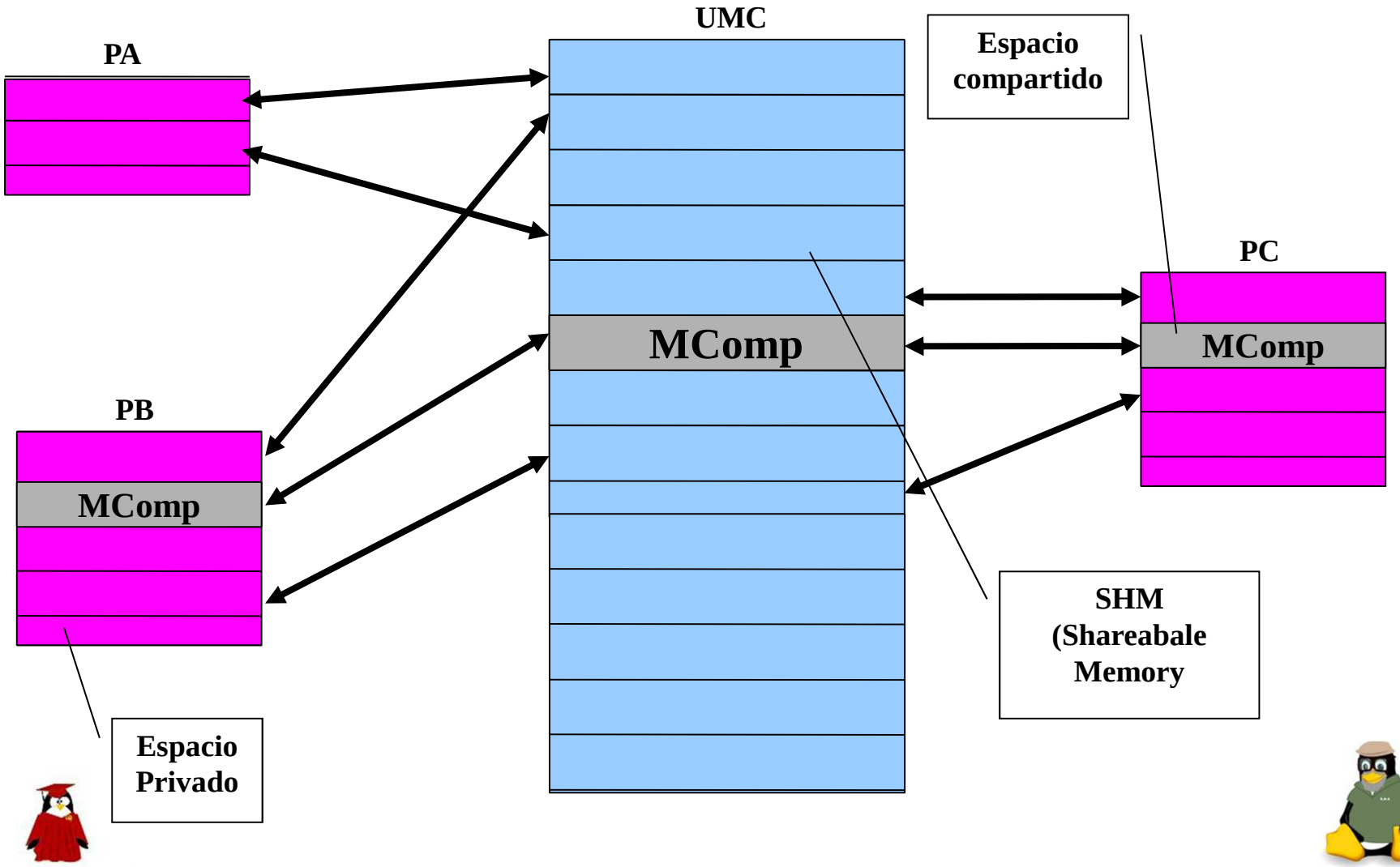
MEMORIA COMPARTIDAD



LIMITACIONES

- Interfaz de Programación Compleja
- Limitado a un número de pequeños procesos.
- No tienen contador de sucesos. No actualizan el mapeo de memoria.
- IPC es Modo Kernel. Si quiero hacer una E/S se usa otra interfaz de programación.

MEMORIA COMPARTIDAD



MEMORIA COMPARTIDA

SEMAFOROS

- Permiten acceso a recursos compartidos.
- Sincronizan procesos. Son bloqueadores de acceso.
- Tipos: binario y entero.
- Creo semáforo con semget y elimino semáforo con semctl. Y semop abre semáforos.
- Debemos usar las librerías: sys/sem.h, sys/IPC.h y sys/types.h.
- Deben ser declarados con permisos de r/w

MEMORIA COMPARTIDA

COLAS DE MENSAJES

- Lista enlazada. Usa Modo Kernel.
- Usa ID de cola de mensaje.
- Por defecto trabaja como un FIFO.
- Tambien se puede ver como Memoria Asociativa. Tal que se puede acceder a un Mensaje en forma arbitraria.
- Usa las librerías: sys/msg.h, sys/types.h y sys/IPC.h.
- Las funciones que se usan son msgget (creación) y msgctl (eliminar)

3. 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 pueden ser interrumpidas.
- Se emplea una cola para mantener los procesos esperando en el semáforo.

SEMAFOROS: CONDICIONES DE DIJKSTRA

- Un semáforo es una variable que tiene un valor entero:
  - Puede iniciarse con un valor no negativo.
  - La operación *wait* disminuye el valor del semáforo. Si el valor de *wait* 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 una operacion *wait*.

SEMAFOROS Y SC DE N PROCESOS

wait (*S*):

```
        while S≤ 0 do no-op;
        S--;
```

signal (*S*):

```
        S++;
```

- Datos compartidos:

```
semaphore mutex; //initially mutex = 1
```

- Proceso *Pi:*

```
do {
   wait(mutex);
      critical section
         signal(mutex);
      remainder section
} while (1);
```

SEMAFOROS: IMPLEMENTACION

- Definición de un Semaforo como un registro de estructura:

```
typedef struct {
   int value;
struct process *L;
} semaphore;
```

- Se asumen dos simles operaciones:
  - El bloque es suspendido por el proceso que lo invoca.
  - wakeup(*P*) resume la ejecución de un bloque de proceso P.

SEMAFOROS: IMPLEMENTACION

- Las operaciones con Semaforos se definen como

```
wait(S):
        S.value--;
        if (S.value < 0) {
                poner este proceso en la s.cola;
                bloquear este proceso;
        }
signal(S):
        S.value++;
        if (S.value <= 0) {
                remover el proceso P de la s.cola;
                poner el proceso P en la cola de
                listos;
        }}
```

SEMAFOROS: IMPLEMENTACION

Tipos según los valores a tomar:

a. ***Por Primitivas de operación*: wait & signal (- n <= 0 <=+n)**

b. ***Semáforos binarios*: wait & signal (0,1)**

Tipos según la administración de la cola de Espera:

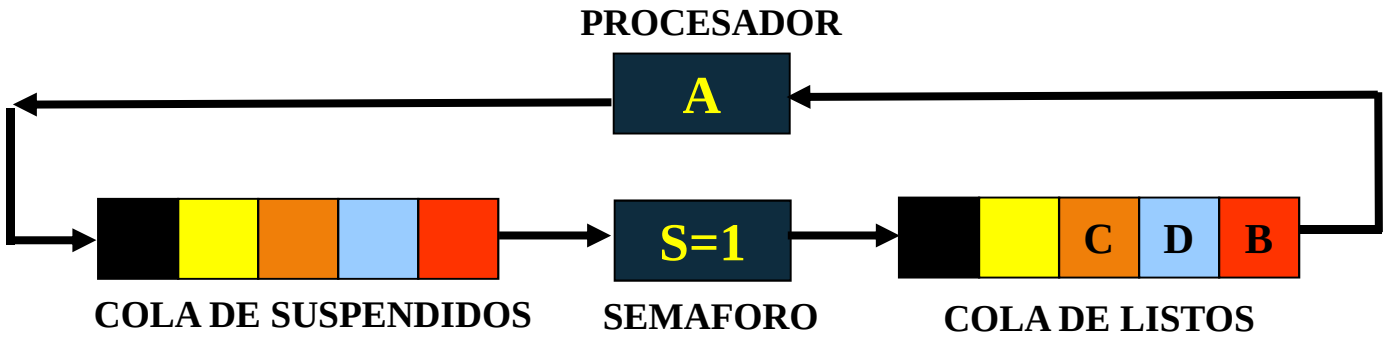
a. ***Semáforo Robusto*: FIFO**

b. ***Semáforo Débil*: Sin algoritmo específicado.**

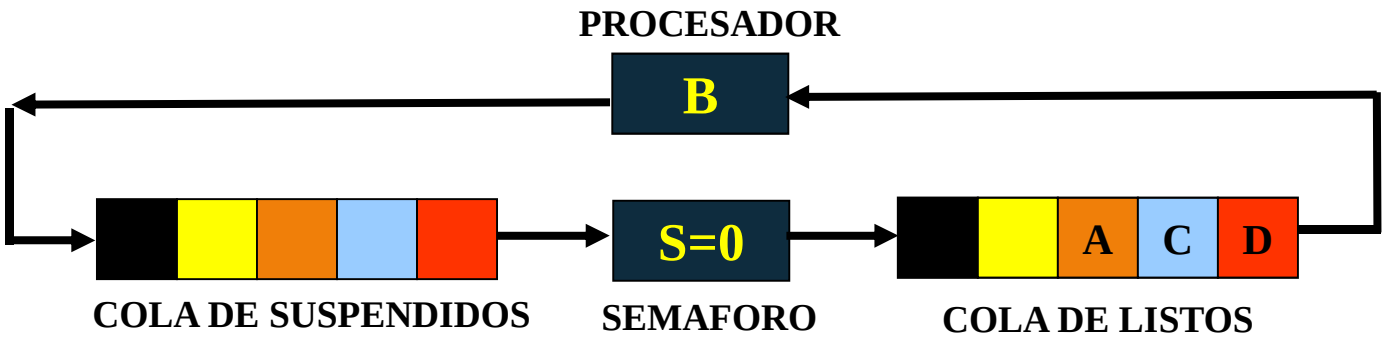
SEMAFOROS: EJEMPLO DE IMPLEMENTACION

A, B, C DEPENDEN DE UN RESULTADO DE *D*

1. *A* EN EJECUCION. *C, D* Y *B* LISTOS & EL SEMAFORO VALE 1. HAY UN RESULTADO DE *D* DISPONIBLE

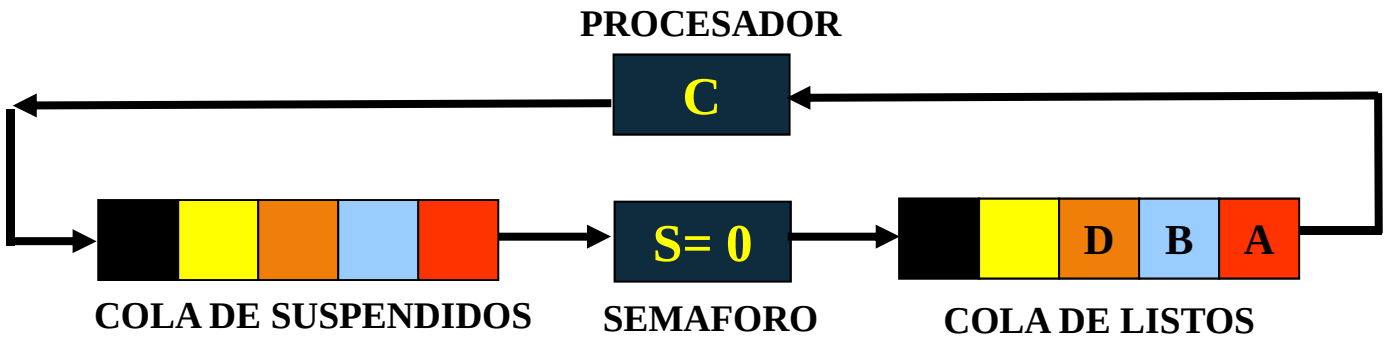


2. *A* PASA A LISTO. *A* EJECUTA UN WAIT & *B* PASA A EJECUCION

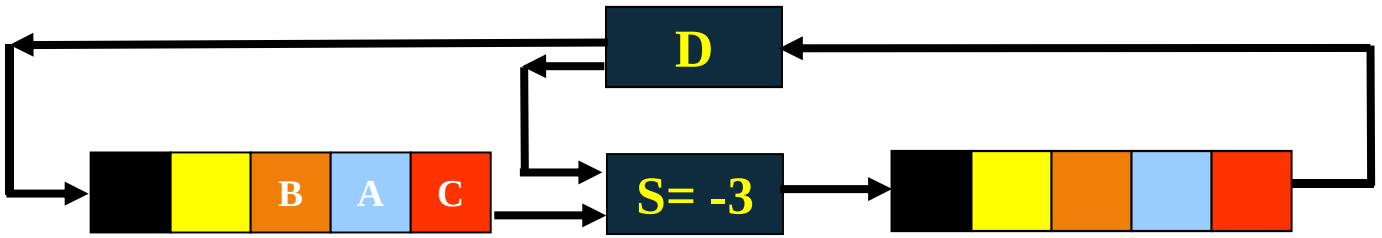


SEMAFOROS: EJEMPLO DE IMPLEMENTACION

5. *D* VUELVE A LISTOS. *C* COMIENZA SU EJECUCION.

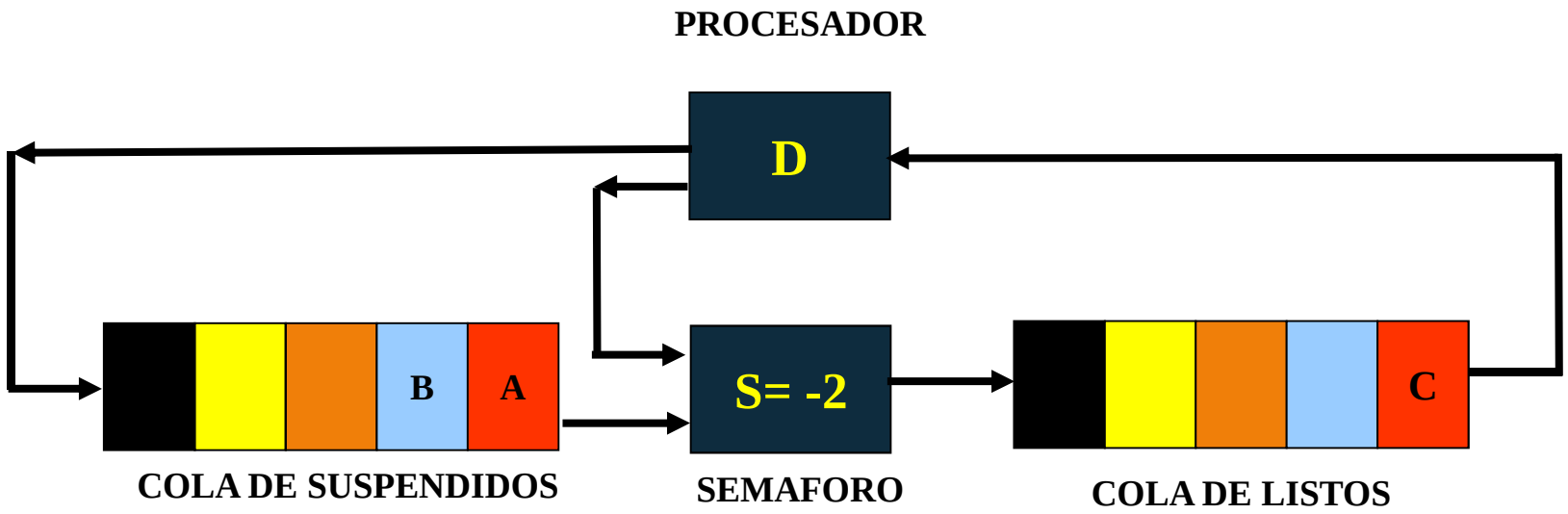


6. *C* EJECUTA UN WAIT Y PASA A SUSPENSION. OCURRE LO MISMO CON *A* Y *B*. ENTONCES *D* REINICIA SU EJECUCION.



SEMAFOROS: EJEMPLO DE IMPLEMENTACION

7. *D* TIENE UN NUEVO RESULTADO. *D* EJECUTA UN SIGNAL. *C* PASA A LISTO. LOS CICLOS DE *D* EN EJECUCION DE LIBERARAN LOS PROCESOS *A* Y *B*.



SEMAFOROS: PROBLEMA DEL 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.

SEMAFOROS: PROBLEMA DEL PRODUCTOR/ CONSUMIDOR

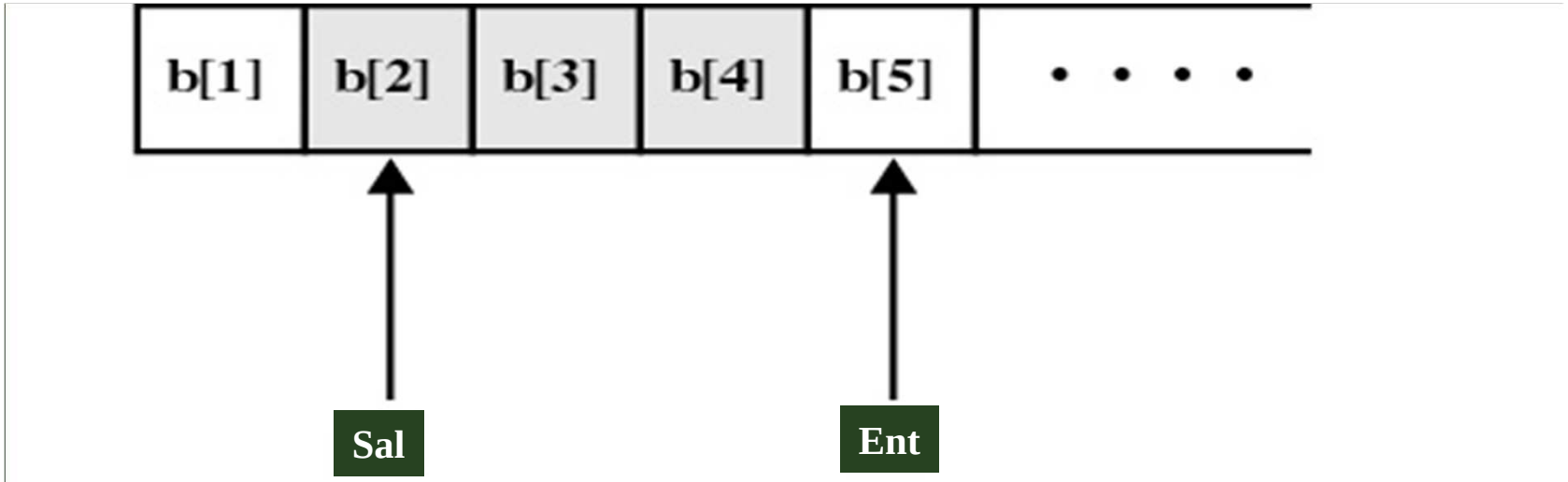
PRODUCTOR

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

CONSUMIDOR

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

SEMAFOROS: PROBLEMA DEL PRODUCTOR/ CONSUMIDOR



Nota: El área sombreada indica la parte del buffer ocupada.

Figura 5.11. Buffer ilimitado en el problema del productor/consumidor.

SEMAFOROS: PROBLEMA DEL PRODUCTOR/ CONSUMIDOR

PRODUCTOR CON BUFFER CIRCULAR

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

SEMAFOROS: PROBLEMA DEL PRODUCTOR/ CONSUMIDOR

CONSUMIDOR CON BUFFER CIRCULAR

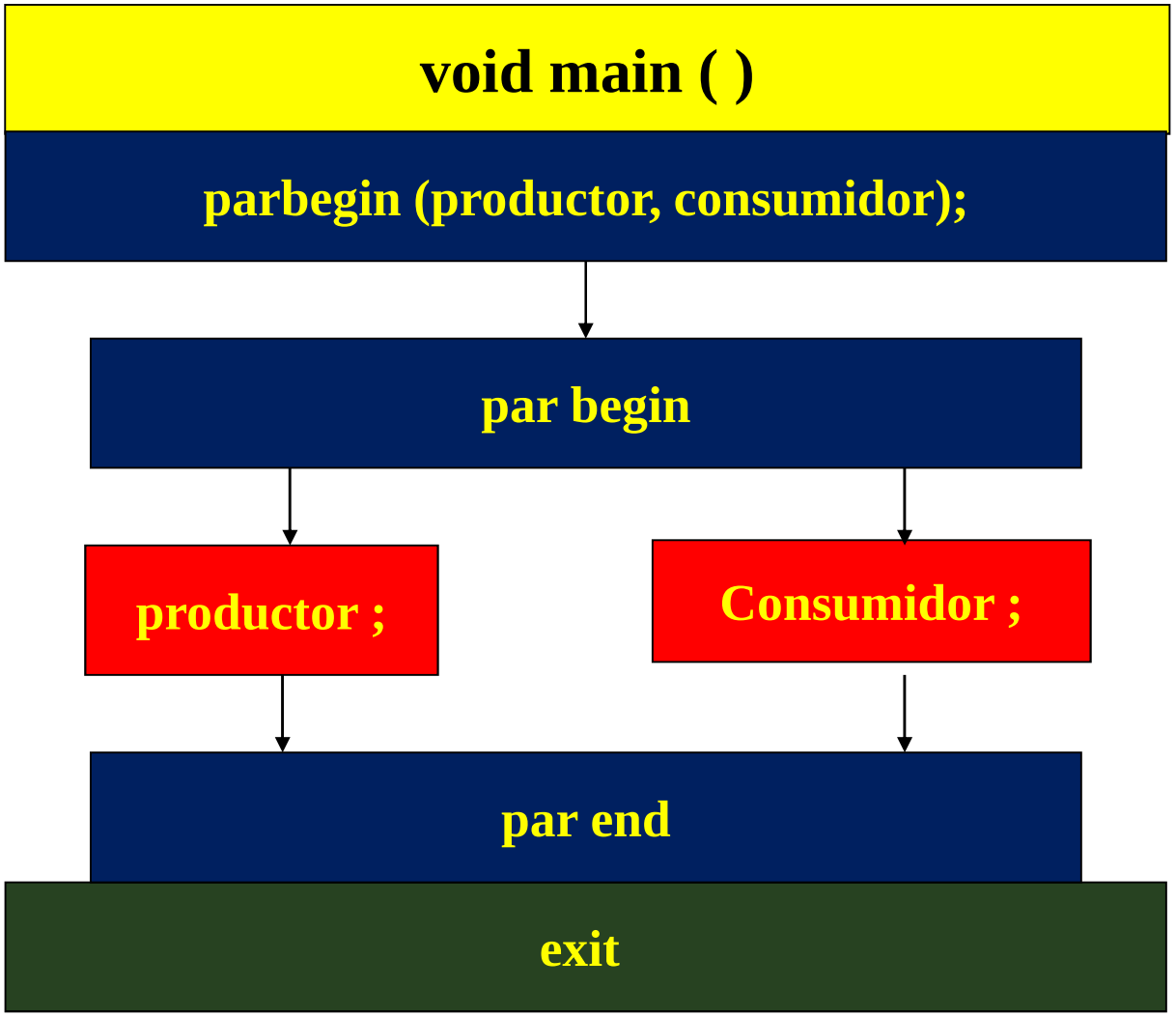
```
consumidor:
while (cierto) {
      while (ent == sal)
            /* no hacer nada */;
      w = b[sal];
      sal = (sal + 1) % n;
      /* consumir elemento w */
}
```

SOLUCION P/C USANDO SEMAFOROS

```
/* program productor-consumiror
const int tambuffer = /* tamaño buffer*/;
semaforo s =1;
semaforo n =0;
semaforo e = tambuffer;
void productor( )
while (cierto)
    producir ( ) ;
    wait (e) ;
    wait (s);
    añadir( );
    signal (s);
    signal (n) ;
```

```
void consumidor( )
while (cierto)
    wait (n) ;
    wait (s) ;
    tomar (s);
    signal (s);
    signal (e);
    consumir ( ) ;
```

SOLUCION P/C USANDO SEMAFOROS



PROGRAMAS MONITORES

- Un monitor es un módulo de software.
- Características básicas:
  - Las variables de datos locales están sólo accesibles para el monitor.
  - Un proceso entra en el monitor invocando a uno de sus procedimientos.
  - Sólo un proceso se puede estar ejecutando en el monitor en un instante dado.

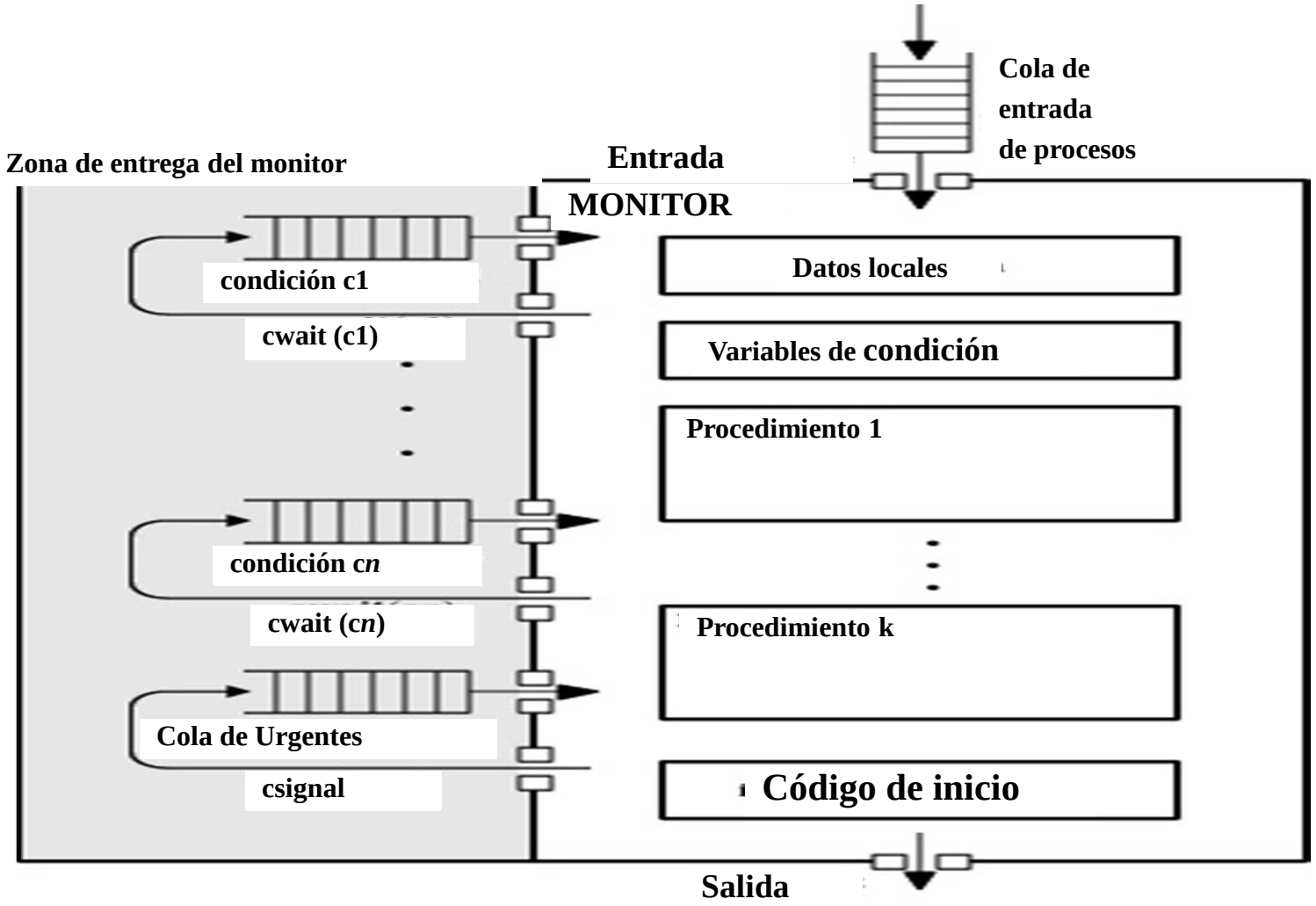


Figura 5.21. Estructura de un monitor.

PASO DE MENSAJES

- Refuerzo de la exclusión mutua.
- Intercambio de información.

```
send (destino, mensaje)
receive (origen, mensaje)
```

PASO DE MENSAJES

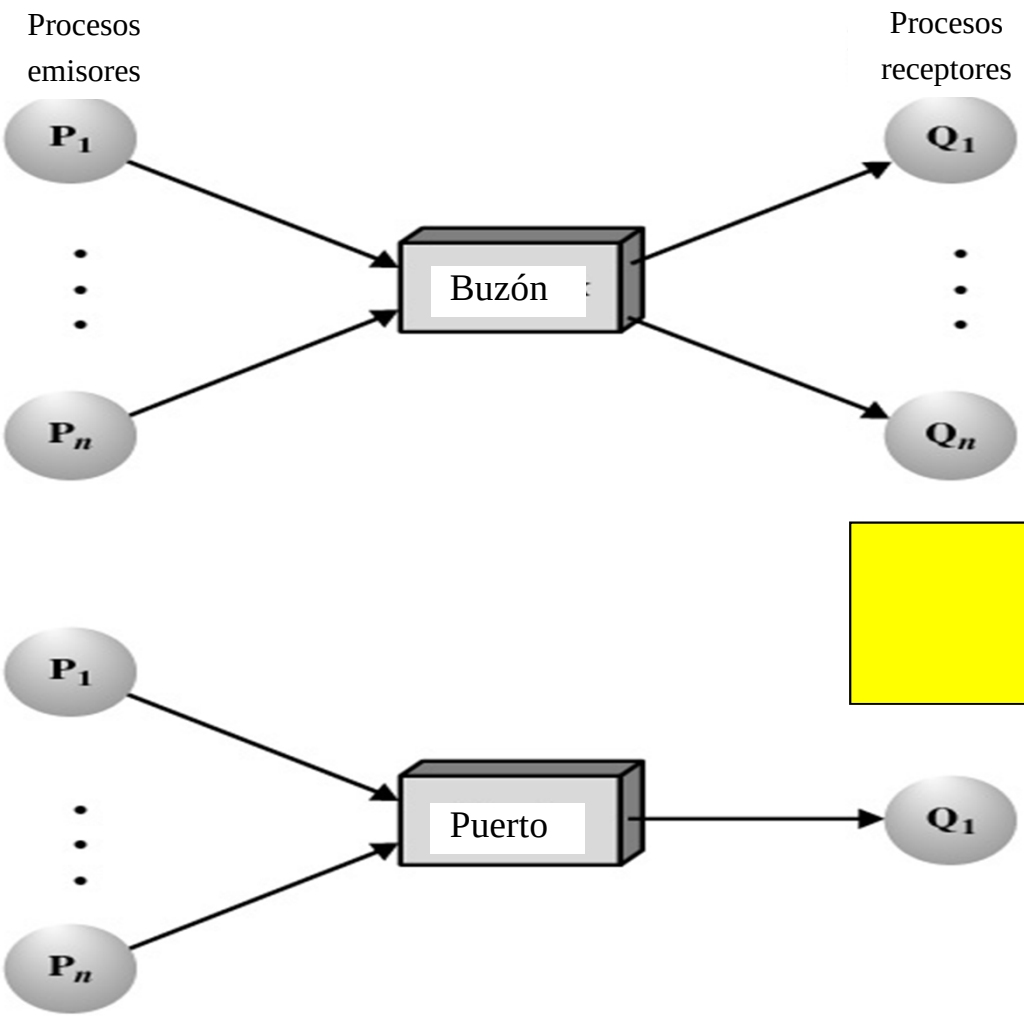
- Envío no bloqueante, recepción bloqueante:
  - Permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sera posible.
  - El receptor se bloquea hasta que llega el mensaje solicitado.
- Envío no bloqueante, recepción no bloqueante:
  - Nadie debe esperar.

PASO DE MENSAJES & SINCRONIZACION: DIRECCIONAMIENTO

- **Direccionamiento directo:**
  - **La primitiva *send* incluye una identificación específica del proceso de destino.**
  - **La primitiva *receive* puede conocer de antemano de qué proceso espera un mensaje.**
  - **La primitiva *receive* puede utilizar el parámetro *origen* para devolver un valor cuando se haya realizado la operación de recepción.**

PASO DE MENSAJES & SINCRONIZACION: DIRECCIONAMIENTO

- **Direccionamiento indirecto:**
  - **Los mensajes se envían a una estructura de datos compartida formada por colas.**
  - **Estas colas se denominan buzones (*mailboxes*).**
  - **Un proceso envía mensajes al buzón apropiado y el otro los coge del buzón.**



COMUNICACION

ENTRE PROCESOS

Williams Stallings SISTEMAS OPERATIVOS. Principios de diseño e interioridades. 4ta ed. Pearson Eduación S.A. Madrid, 2001 ISBN: 84-205-3177-4

Figura 5.24. Comunicación indirecta entre procesos.

PASO DE MENSAJES & SINCRONIZACION: DIRECCIONAMIENTO

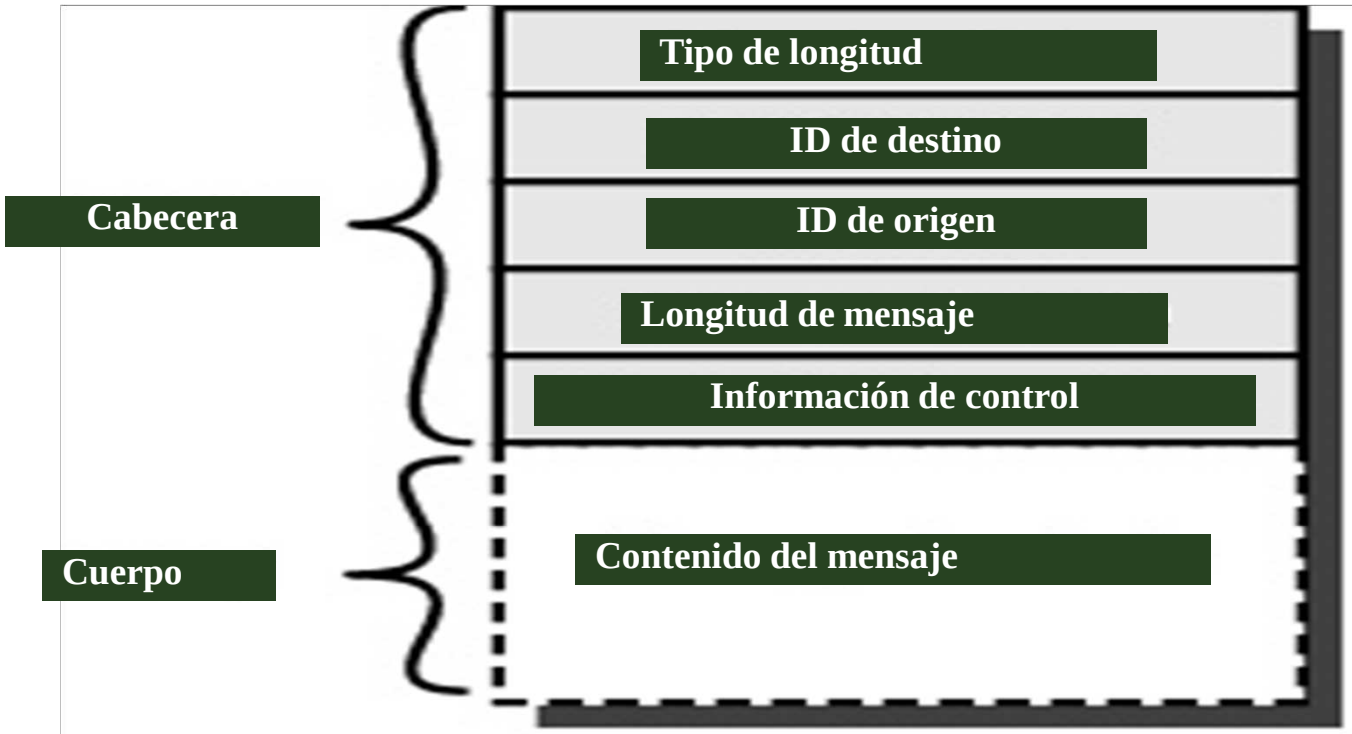
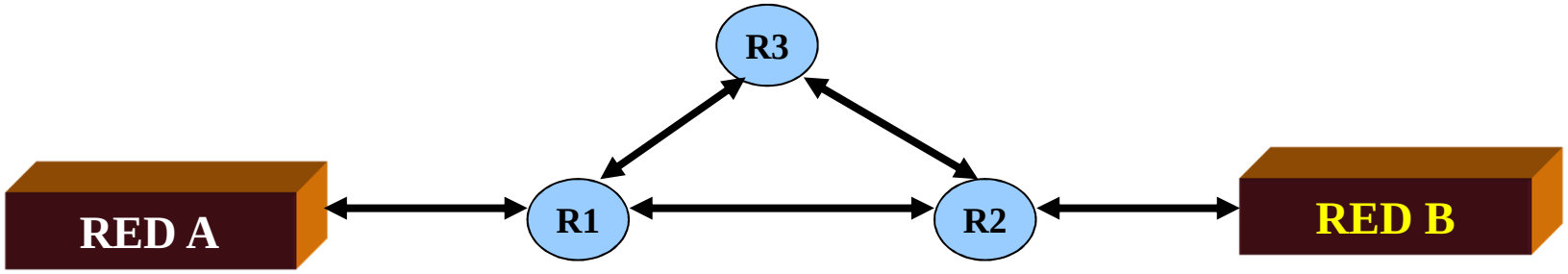


Figura 5.25. Formato típico de mensaje.

TCP/IP - SOCKETS



CONMUTACION DE PAQUETES

TIPO DE PROTOCOLOS

- Orientados a la Conexión
- Sin conexión
- Secuencia de Chars (TCP)
- Secuencia de Paquetes (UDP)

SOCKET BERKELEY

- Interfase de programación
- Struct sockaddr (dir socket)
- librería sys/socket.h

TCP/IP - SOCKETS

OPERACIONES

- Creación
- Apertura
- Cierre
- Lectura
- Escritura

Creación

Int socket (int domain, int type, int protocol)

Protocolo de Red:
PF_UNIX
PF_INET
PF_AX25
PF_IPX
PF_APPLTALK

Protocolo de secuencia o Protocolo de Datagrama

Valor O, es el valor por defecto

TCP/IP - SOCKETS

TIPOS

1. Stream (sock_stream)
2. Datagrama (sock_dgram)
3. Paq. Secuenciados (sock_seqpacket)
4. Crudo (sock_raw)
5. Sin secuencia (sock_rdm)

CONEXION

- Procesos de Server
- Procesos de Cliente

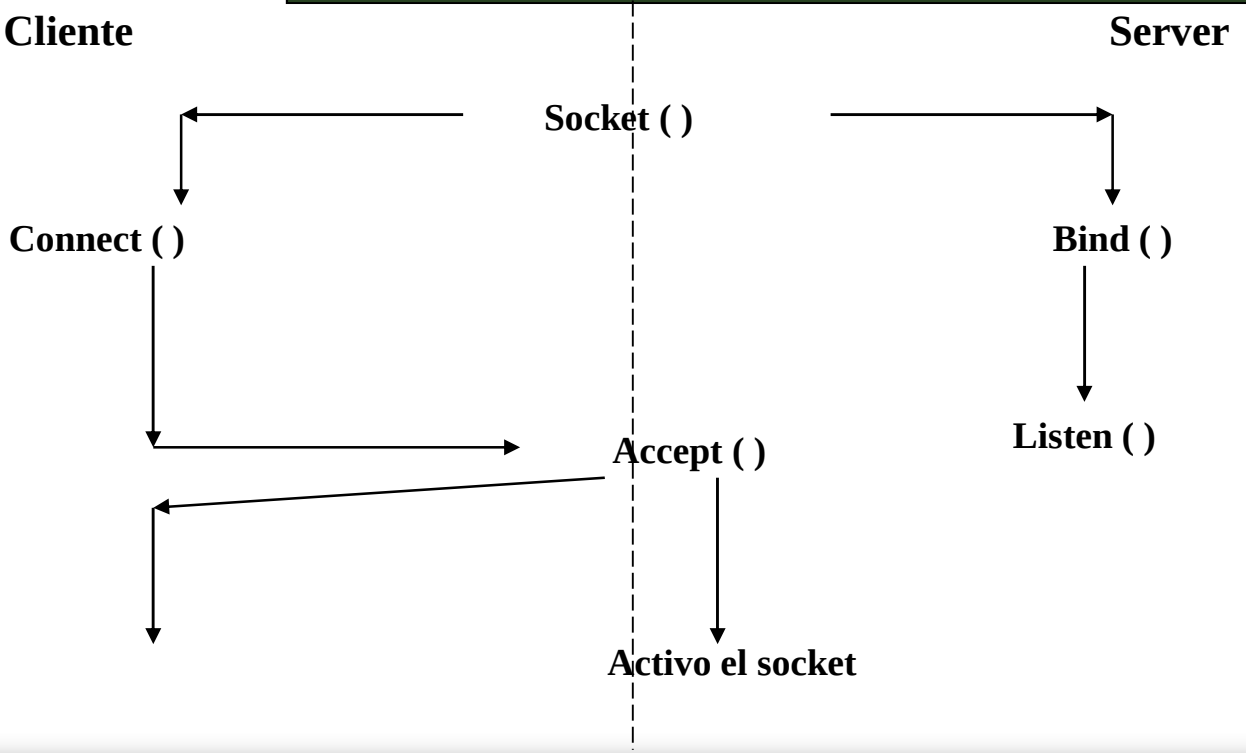
- Crean Socket para pedir información a Socket Server.
- Realizan acceso a un Recurso

- Reciben pedidos de Información
- Permiten accso a un Recurso
- Crean Sockets y experan pedidos de conexión

SOCKETS SERVER

OPERACIONES

1. Crearse
2. Linkearse a una dirección
3. Socket sobre TCP/IP es una dir IP.
4. Escucha de un Socket Cliente (bind o enlazar, listen o escuchar y accept o aceptar)



Bibliografia

1. Programación en Linux, con ejemplos. Kurt Wall. QUE, Prentice Hall. Madrid. 2000.
2. Sistemas Operativos. 5ta Ed. William Stalling. Pearson Prentice Hall. Madrid. 2006
3. Sistemas Operativos. 7ma Ed. William Stalling. Pearson Prentice Hall. Madrid. 2012
4. Sistemas Operativos Modernos. Andrew. S. Tanenbaum. Prentice-Hall. Interamericana S.A. Madrid, 2009.
5. Unix, Sistema V Versión 4. Rosen,Rozinsky y Farber.McGraw Hill. NY 2000.
6. Lunix, Edición especial. Jack Tackett, David Guntery Lance Brown. Ed. Prentice Hall. 1998.
7. El Libro de Linux. Syed M. Sarwar, Robert Koretsky y Syed. A. Sarwar. Ed. Addison Wesley. 2007. España.

FIN UNIDAD 4

SINCRONIZACION E IPC

May the force be with you