

Convivencia

Sincronización y Comunicación



Dra. Carolina Mañoso
Dpto. Informática y Automática.UNED

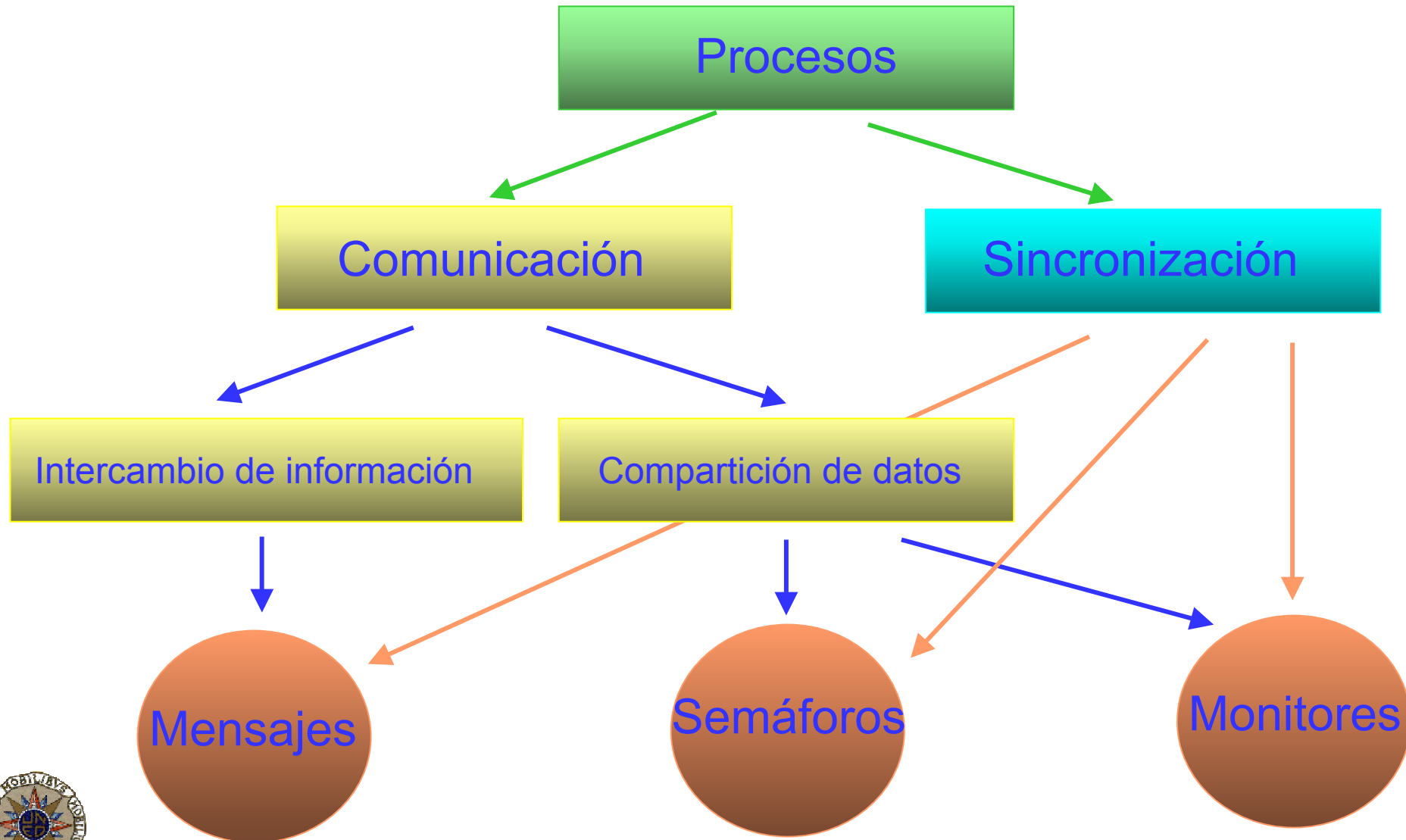
© Carolina Mañoso, 2002

Índice: Sincronización y comunicación

- Introducción
- Exclusión mutua
 - ◆ Bloqueo utilizando variables compartidas
 - ◆ Algoritmo de Peterson
 - ◆ Algoritmo de Dekker
- Semáforos
- Monitores
- Mensajes
- Interbloqueo



Introducción



Exclusión mutua: región crítica (1/5)

Ejemplo variable x compartida entre los procesos A y B

- ◆ **Problema de los Jardines:**

Se quiere tener constancia del número de visitantes que hay en un Jardín. Hay dos puertas y asociado a cada puerta un proceso.

- ◆ Cuando entra una persona el proceso P1 incrementa x : $x:=x+1$.
- ◆ Cuando sale una persona el proceso P2 decrementa x : $x:=x-1$

- Instrucciones para la actualización de la variable:

- ◆ Copiar el valor de x en un registro del procesador
- ◆ Incrementar el valor del registro
- ◆ Almacenar el resultado en la dirección donde se guarda x



Exclusión mutua: región crítica (2/5)

- En esta situación se puede producir una mala actualización de la variable x: El planificador de procesos puede permitir el entrelazado de las operaciones elementales anteriores de cada uno de los procesos:
 - ◆ P1 carga el valor de x en un registro de su procesador
 - ◆ P2 carga el valor de x en un registro de su procesador
 - ◆ P1 incrementa el valor de su registro
 - ◆ P2 decrementa el valor de su registro
 - ◆ P1 almacena el valor de su registro en la dirección de memoria de x
 - ◆ P2 almacena el valor de su registro



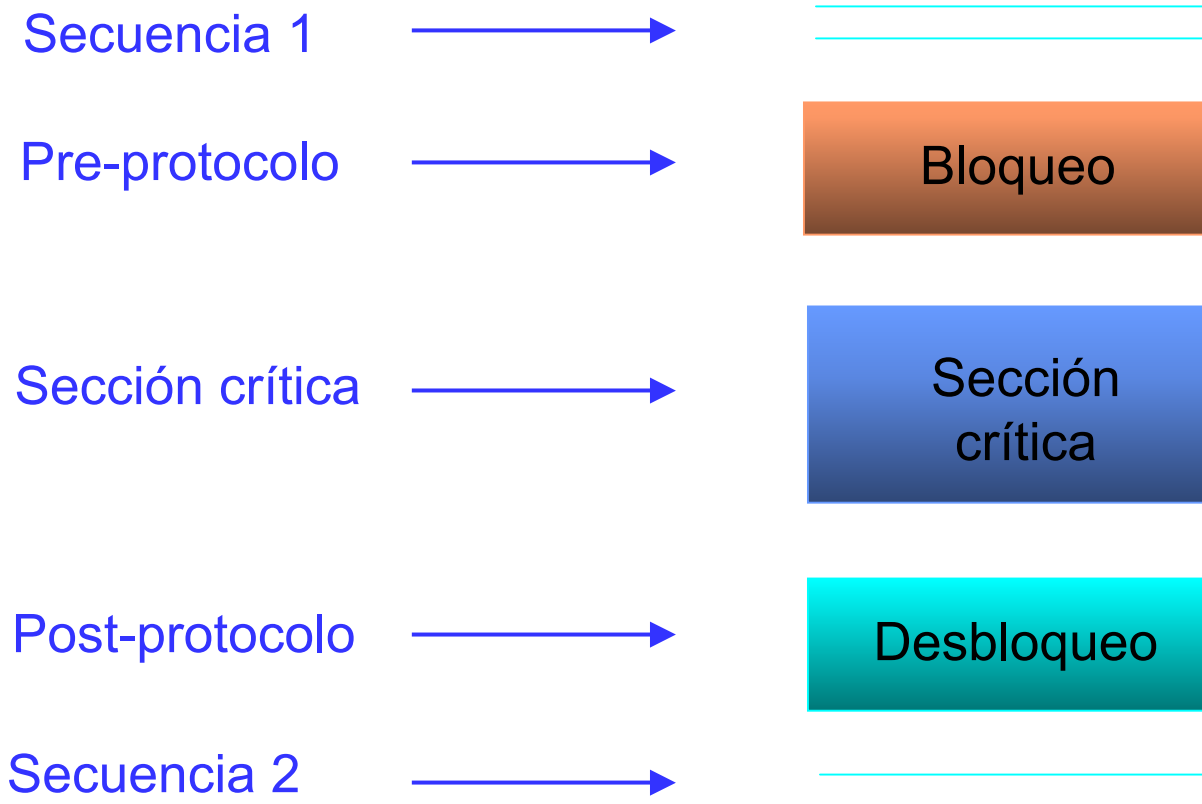
Exclusión mutua: región crítica (3/5)

- Se denomina **Sección crítica** a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción
- Si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una sección crítica en la que se modifique las variables compartidas con el anterior
- Las secciones críticas se agrupan **mutuamente exclusivas** las regiones críticas de cada clase



Exclusión mutua: región crítica (4/5)

- La exclusión se consigue con protocolos que **bloqueen** el acceso a la sección crítica mientras se utiliza por un proceso



Exclusión mutua: región crítica (5/5)

- Se realiza el bloqueo de una sección crítica mediante el uso de una variable compartida de tipo booleano que se denomina **indicador** (flag)
- La acción del bloqueo se realiza con la activación del indicador y la del desbloqueo con su desactivación

Este método no resuelve el problema de la exclusión mutua, ya que **la comprobación y la puesta del indicador son operaciones separadas** y en ese caso se puede entrelazar el uso del recurso por ambos procesos



Bloqueo utilizando variables compartidas (1/4)

```
module ExclusionMutua1
```

```
var flag:boolean;
```

```
process P1
```

```
begin
```

```
  loop
```

```
    while flag=true do
```

```
      (*Espera a que quede libre el  
      dispositivo*)
```

```
    end;
```

```
    flag:=true;
```

```
    (*uso del recurso. Sección crítica*)
```

```
    flag:=false;
```

```
    (* resto del proceso*)
```

```
  end
```

```
end P1;
```

```
process P2
```

```
begin
```

```
  loop
```

```
    while flag=true do
```

```
      (*Espera a que quede libre el  
      dispositivo*)
```

```
    end;
```

```
    flag:=true;
```

```
    (*uso del recurso. Sección crítica*)
```

```
    flag:=false;
```

```
    (* resto del proceso*)
```

```
  end
```

```
end P2;
```

```
begin
```

```
  flag:=false
```

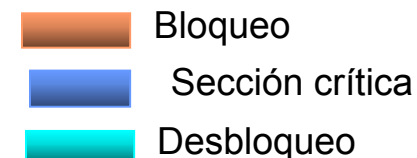
```
  cobegin
```

```
    P1;
```

```
    P2;
```

```
  coend
```

```
end
```



Bloqueo utilizando variables compartidas (2/4)

- Se realiza el bloqueo de una sección crítica mediante el uso de dos **indicadores**
- Se asocia un indicador a cada uno de los procesos. Antes de acceder al recurso un proceso debe activar su indicador y comprobar que el otro no tiene su indicador activado

- Espera activa
- **INTERBLOQUEO**: Los dos procesos llaman al “bloqueo” simultáneamente. Los dos indicadores quedan activados y los dos procesos a la espera de que se libere el recurso. Pero esto no se puede dar ya que ninguno entra en la sección crítica.



Bloqueo utilizando variables compartidas (3/4)

```
module ExclusionMutua2
var flag1,flag2:boolean;
```

```
procedure bloqueo(var mi_flag,su_flag:boolean);
begin
    mi_flag:=true; (*intención de usar el recurso*)
    while su_flag do; end (*espera liberar recurso*)
end bloqueo;
```

```
procedure desbloqueo(var mi_flag:boolean);
begin
    mi_flag:= false;
end desbloqueo;
```

```
process P1
```

```
begin
```

```
loop
```

```
    bloqueo(flag1,flag2);
```

```
    (*uso del recurso. Sección crítica*)
```

```
    desbloqueo(flag1);
```

```
    (* resto del proceso*)
```

```
end
```

```
end P1;
```

```
process P2
```

```
begin
```

```
loop
```

```
    bloqueo(flag2,flag1);
```

```
    (*uso del recurso. Sección crítica*)
```

```
    desbloqueo(flag2);
```

```
    (* resto del proceso*)
```

```
end
```

```
end P2;
```

```
begin
```

```
    flag1:=false
```

```
    flag2:=false
```

```
cobegin
```

```
    P1;
```

```
    P2;
```

```
coend
```

```
end
```



Bloqueo utilizando variables compartidas (4/4)

■ Una solución al interbloqueo

```
procedure bloqueo(var mi_flag,su_flag:boolean);  
begin  
  mi_flag:=true;  
  while su_flag:=false;  
    mi_flag:=false;  
    mi_flag:=true;  
  end  
end bloqueo
```

Puede que un proceso deje su sección crítica y vuelva a entrar mientras que el otro proceso desactiva su indicador en la sección de bloqueo. El que un proceso no pueda progresar porque se lo impida otro se denomina **CIERRE** (lockout)



Algoritmo de *PETERSON* (1/2)

- Estrategia presentada por Peterson, 1981
- Se introduce una variable adicional denominada *turno* que solamente resulta útil cuando se produce una petición simultanea de acceso a la región crítica
- Si ambos intentan entrar en la sección crítica, el valor de turno se pondrá a 1 y 2 pero sólo un valor de ellos permanecerá al escribirse sobre el otro, permitiendo el acceso de un proceso a su sección crítica



Algoritmo de PETERSON (2/2)

```
module ExclusionMutua_Peterson
```

```
var flag1,flag2:boolean;
```

```
    turno:integer;
```

```
procedure bloqueo(var  
    mi_flag,su_flag:boolean,su_turno:integer);
```

```
begin
```

```
    mi_flag:=true;
```

```
    turno:=su_turno;
```

```
    while su_flag and ((turno=su_turno)) do; end  
end bloqueo;
```

```
procedure desbloqueo(var mi_flag:boolean);
```

```
begin
```

```
    mi_flag:= false;
```

```
end desbloqueo;
```

```
process P1
```

```
begin
```

```
    loop
```

```
        bloqueo(flag1,flag2,2);
```

```
        (*uso del recurso. Sección crítica*)
```

```
        desbloqueo(flag1);
```

```
        (* resto del proceso*)
```

```
    end
```

```
end P1;
```

```
process P2
```

```
begin
```

```
    loop
```

```
        bloqueo(flag2,flag1,1);
```

```
        (*uso del recurso. Sección crítica*)
```

```
        desbloqueo(flag2);
```

```
        (* resto del proceso*)
```

```
    end
```

```
end P2;
```

```
begin
```

```
    flag1:=false;
```

```
    flag2:=false;
```

```
    cobegin
```

```
        P1;
```

```
        P2;
```

```
    coend
```

```
end
```



Algoritmo de *DEKKER* (1/2)

- Estrategia atribuida por Dekker y presentada por Dijkstra en 1968
- Se introduce una variable adicional denominada *turno* que sirve para establecer la prioridad relativa de los dos procesos y su actualización se realiza en la sección crítica, lo que evita que pueda haber interferencias entre los procesos
- Da prioridad al proceso P1, al iniciar el valor de *turno* a 1



Algoritmo de DEKKER (2/2)

```

module ExclusionMutua_Dekker
var flag1,flag2:boolean;
    turno:integer;
procedure bloqueo(var
    mi_flag,su_flag:boolean,su_turno:integer);
begin
    mi_flag:=true;
    while su_flag do (*otro proceso en la sec. cr.*)
        if turno=su_turno then mi_flag:=false;
        while turno=su_turno do
            (*espera a que el otro termine*) end
        mi_flag:=true;
    end
end
end bloqueo;

procedure desbloqueo(var
    mi_flag:boolean;s_u_turno:integer);
begin
    turno:=su_turno;
    mi_flag:= false;
end desbloqueo;

```

```

process P1
begin
    loop
        bloqueo(flag1,flag2,2);
        (*uso del recurso. Sección crítica*)
        desbloqueo(flag1);
        (* resto del proceso*)
    end
end P1;

process P2
begin
    loop
        bloqueo(flag2,flag1,1);
        (*uso del recurso. Sección crítica*)
        desbloqueo(flag2);
        (* resto del proceso*)
    end
end P2;

```

```

begin
    flag1:=false;
    flag2:=false;
    turno:=1;
    cobegin
        P1;
        P2;
    coend
end

```



Peterson-Dekker

- No es fácil extender estos algoritmos al caso en que haya n procesos ejecutándose de manera concurrente
- La espera de acceso a un recurso siempre se realiza de forma “ocupada”, el proceso siempre se queda comprobando una variable, lo que supone un derroche de los recursos del sistema



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

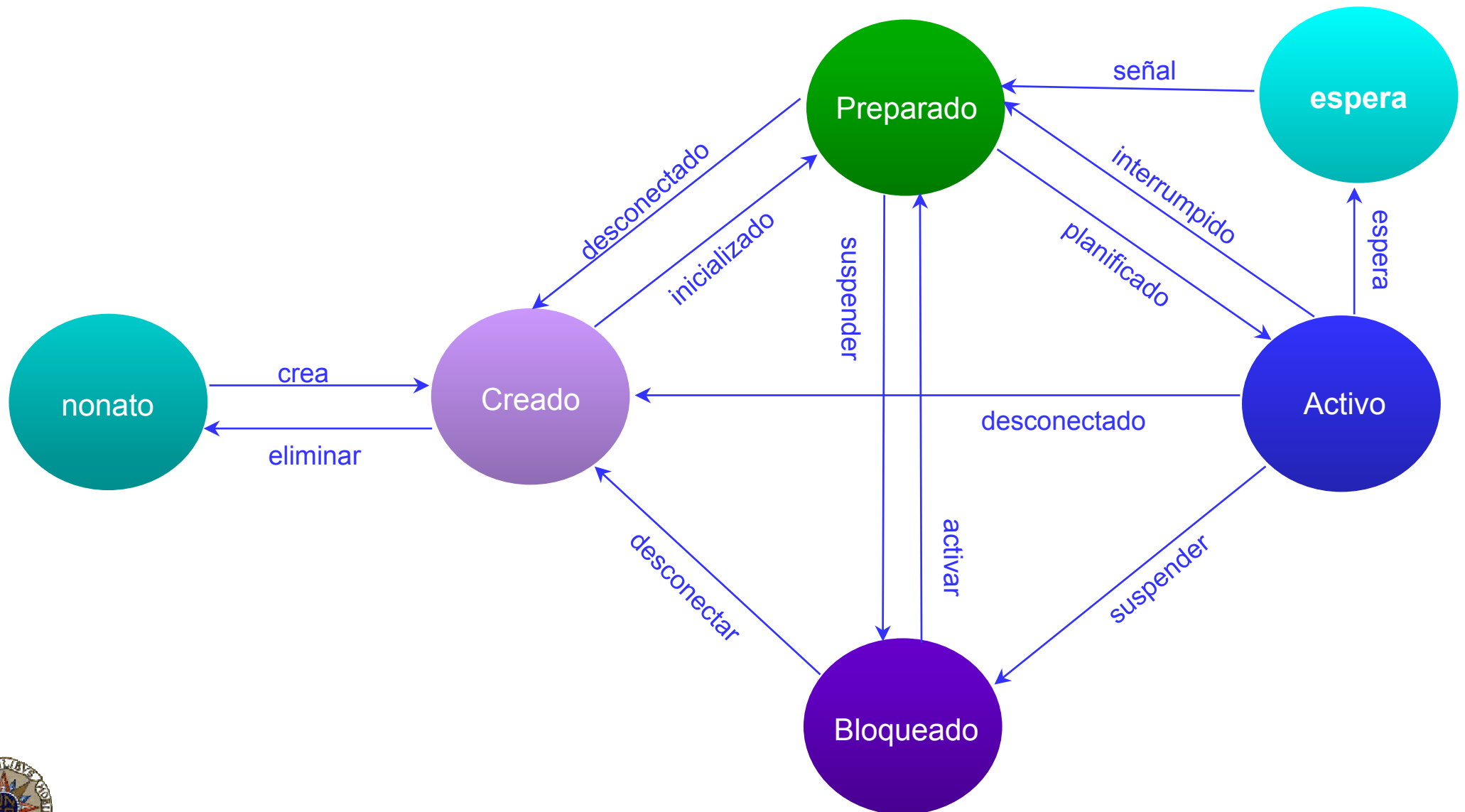


SEMÁFOROS: implementación

- 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



Un estado más: el estado espera



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



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
```



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



Problema del productor-consumidor (3/3)

```
module Productor_consumidor;  
var BufferComun:buffer;  
AccesoBuffer, Nolleno,  
Novacio:semaforo;
```

```
begin  
  inicializa(AccesoBuffer,1);  
  inicializa(Nolleno,1);  
  inicializa(Novacio,0);  
cobegin  
  Productor;  
  Consumidor;  
coend  
end Productor_consumidor;
```

```
process Productor;  
var x:dato;  
begin  
  loop  
    produce(x);  
    espera(AccesoBuffer);  
    if Lleno then  
      señal(AccesoBuffer);  
      espera(Nolleno);  
    end;  
    Poner(x);  
    señal(AccesoBuffer);  
    señal(Novacio);  
  end  
end Productor;
```

```
process Consumidor;  
var x:dato;  
begin  
  loop  
    espera(AccesoBuffer);  
    if Vacio then  
      señal(AccesoBuffer);  
      espera(Novacio);  
    end Tomar(x);  
    señal(AccesoBuffer);  
    señal(Nolleno);  
    Consume(x);  
  end  
end Consumidor;
```



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
```

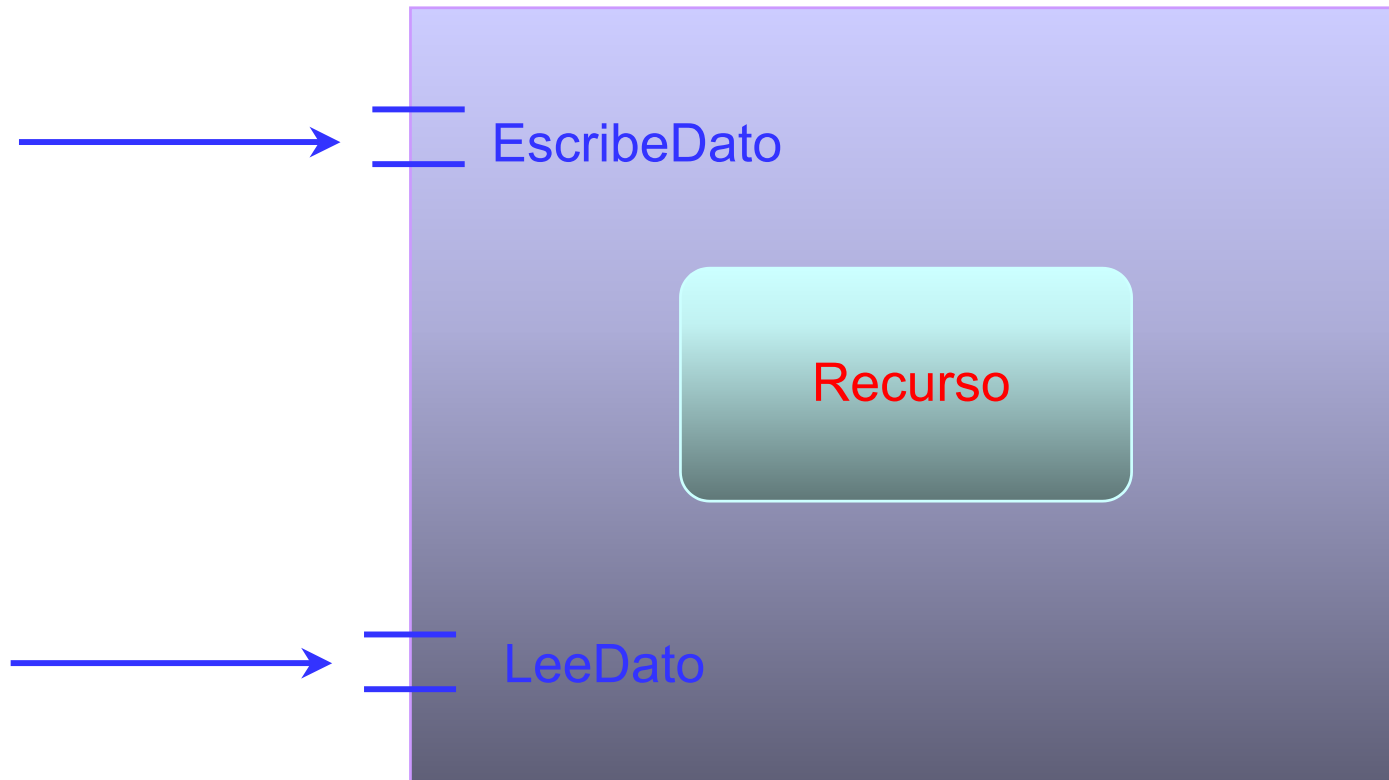


MONITORES (1/4)

- Un **monitor** es un conjunto de procedimientos que proporcionan el acceso con exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos
- Los procedimientos van **encapsulados dentro de un módulo** que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor
- El monitor se puede ver como una valla alrededor del recurso (o recursos), de modo que los procesos que quieran utilizarlo deben de entrar dentro de ella, pero en la forma que impone el monitor



MONITORES (2/4)

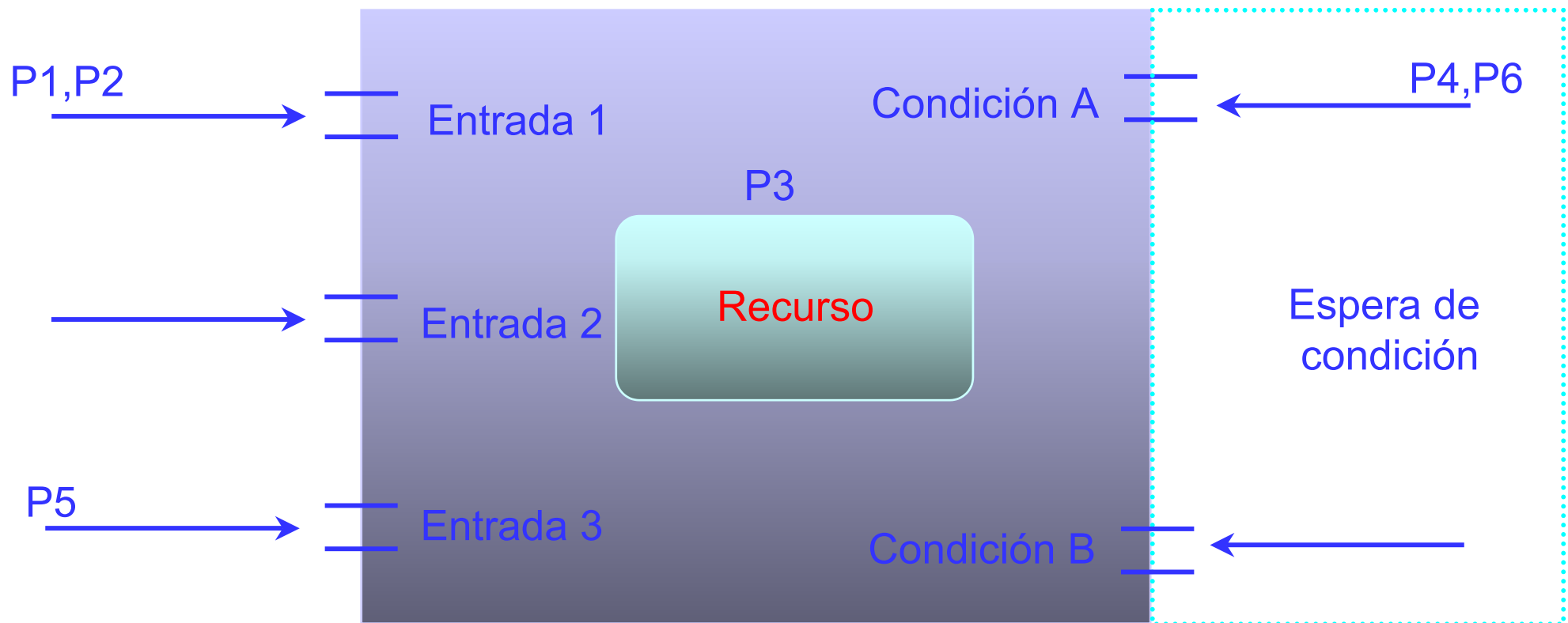


MONITORES (3/4)

- La espera de los procesos no se hace en colas separadas para cada procedimiento, sino que se utiliza una única cola, ya que sólo un procedimiento puede ejecutarse cada vez
- La ventaja para la exclusión mutua que presenta un monitor frente a los semáforos es que está implícita
- Los monitores no proporcionan por si 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



MONITORES (4/4)



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;

end



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;

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



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



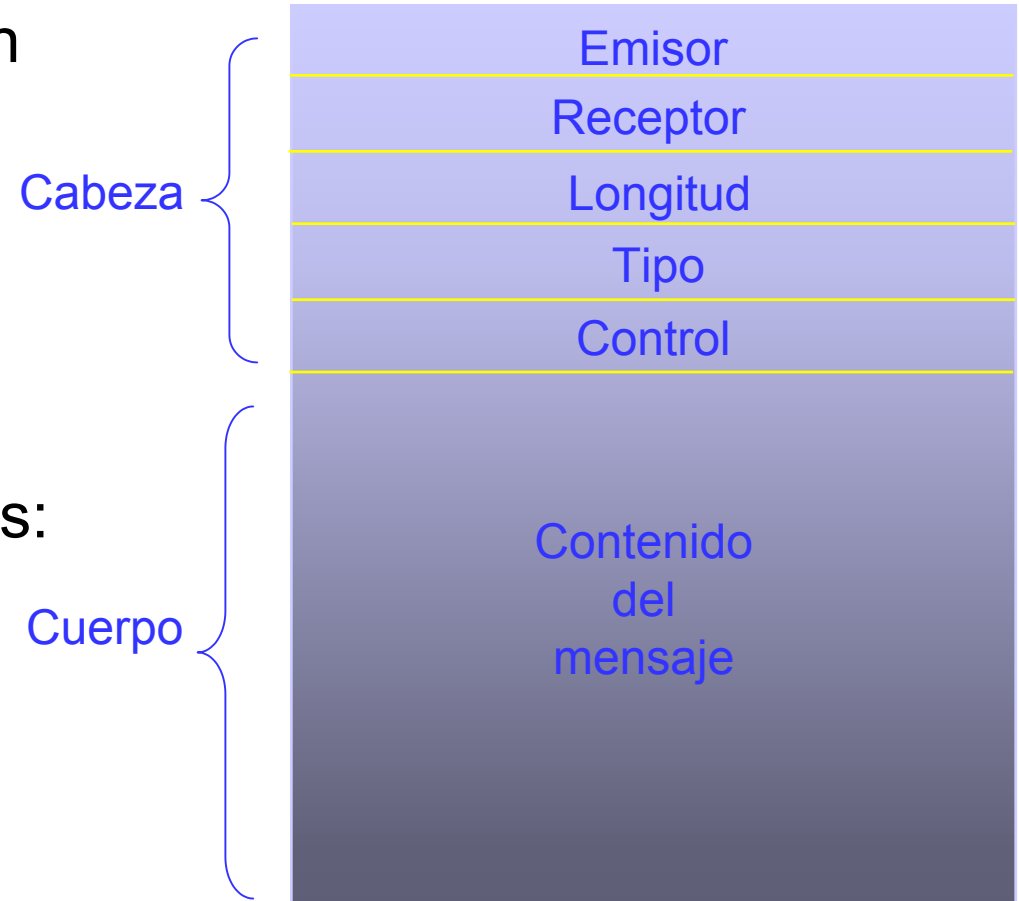
Estructura de los mensajes

- El intercambio de información puede ser:

- ◆ Por valor
- ◆ Por referencia

- La estructura de los mensajes:

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



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}
```



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;

