



Sistemas Operativos I

Tema 4

Comunicación y sincronización de procesos



Equipo de Sistemas Operativos DISCA / DSIC

UPV

Comunicación y sincronización de procesos



◦ Objetivos

- Presentar dos alternativas básicas para comunicación entre procesos
 - Memoria común
 - Mensajes
- Analizar las condiciones de carrera y estudiar el concepto de seriabilidad.
- Estudiar, en el caso de memoria común, el problema de la sección crítica.
- Presentar los criterios de corrección al problema de la sección crítica y sus posibles soluciones de una forma estructurada:
 - Soluciones hardware
 - Semáforos
 - Realización de semáforos mediante soluciones hardware.
- Adquirir destreza en la resolución de problemas de sincronización a través de problemas clásicos (productores y consumidores, lectores y escritores, cinco filósofos, etc.)

Comunicación y sincronización de procesos



◉ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

◉ Bibliografía

- A. Silberschatz, J. Peterson , P. Galvin. Sistemas Operativos. Conceptos Fundamentales. 5ª ed. Capítulo 6.
- A. Tanenbaum. Modern Operating Systems. Capítulos 2,11 y 12

Comunicación y sincronización de procesos



◉ Contenido



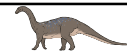
- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

1.Introducción



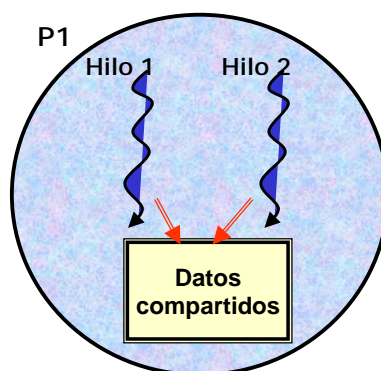
- Existe la necesidad de comunicación entre procesos.
- Los procesos requieren con frecuencia comunicación entre ellos (ejemplo: tubos).
- La comunicación entre procesos puede seguir dos esquemas básicos:
 - Comunicación por memoria común
 - Comunicación por mensajes

1.Introducción



◦ Comunicación por memoria común

- La comunicación por memoria común se puede dar en los siguientes casos:
 - Espacio de direcciones único: es el caso de los hilos de ejecución
 - El s.o. crea una zona de memoria accesible a un grupo de procesos
- **Problema de la sección crítica:** en un sistema con procesos concurrentes que se comunican compartiendo datos comunes es necesario sincronizar el acceso (lectura, escritura) a los datos compartidos para asegurar la consistencia de los mismos.

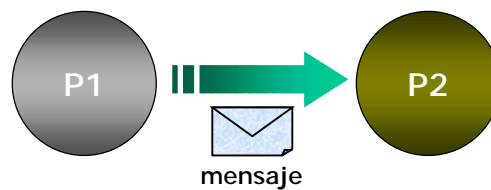


1.Introducción



◉ Comunicación por mensajes

- La comunicación por mensajes se da “normalmente” en el siguiente caso:
 - Espacio de direcciones independientes
- Sincronización en la comunicación por mensajes: Cuando dos procesos se comunican vía mensajes se necesitan mecanismos para que el proceso receptor espere (se suspenda hasta) a que el proceso emisor envíe el mensaje y éste esté disponible.
- No aparece el problema de la sección crítica.



Comunicación y sincronización de procesos



◉ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

2.Comunicación por memoria común



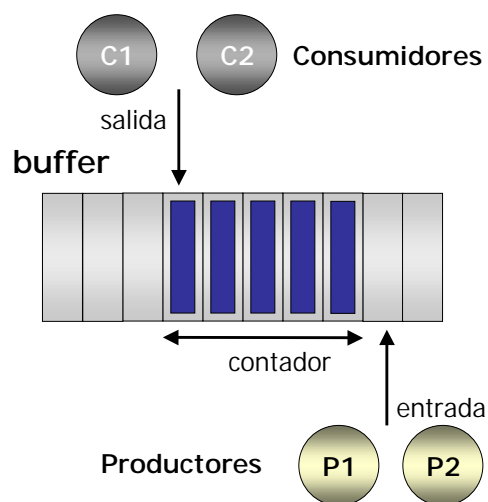
◦ El problema de los productores y consumidores con buffer acotado

- **Productor:** proceso que produce elementos (a una cierta velocidad) y los deposita en un buffer.
 - **Consumidor:** proceso que toma elementos del buffer y los consume (a una velocidad probablemente diferente a la del productor)
 - **Buffer:** Estructura de datos que sirve para intercambiar información entre los procesos productores y consumidores. Actúa a modo de depósito para absorber la diferencia de velocidad entre productores y consumidores
- Ejemplo: buffer de impresora.

2.Comunicación por memoria común



◦ El problema de los productores y consumidores con buffer acotado



2.Comunicación por memoria común



◦ Productores y consumidores con buffer acotado

• Variables compartidas:

```
var buffer: array[0..n-1]
    of elemento;
    entrada:=0, salida:=0 : 0..n-1;
    contador:= 0 : 0..n;
```

• proceso productor:

```
task productor;
    var item: elemento;
    repeat
        item := producir();
    while contador= n do no-op;
    buffer[entrada] := item;
    entrada := (entrada +1) mod n;
    contador:=contador+1;
    until false
end productor;
```

• proceso consumidor:

```
task consumidor;
    var item: elemento;
    repeat
        while contador= 0 do no-op;
        item := buffer[salida]
        salida := (salida +1) mod n;
        contador:=contador-1;
        consumir(item);
    until false
end consumidor;
```

2.Comunicación por memoria común



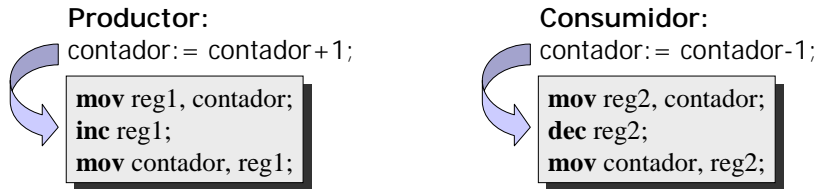
◦ Condiciones de carrera

- **Corrección en programas secuenciales:** el programa cumple con sus especificaciones (responde a unos invariantes o reglas de corrección).
- ➔ **Corrección secuencial no implica corrección concurrente:** Un programa que tiene una implementación “secuencialmente correcta” (correcta con un sólo hilo de ejecución) puede presentar problemas cuando se intenta introducir concurrencia en forma de hilos de ejecución.
- **Condición de carrera:** la ejecución de un conjunto de operaciones concurrentes sobre una variable compartida, deja la variable en un estado inconsistente con las especificaciones de corrección. Además, el resultado de la variable depende de la velocidad relativa en que se ejecutan las operaciones.
- **Peligro potencial:** Las condiciones de carrera pueden presentarse en algún escenario, pero no tienen por qué observarse en todas las posibles trazas de la ejecución del programa.

2.Comunicación por memoria común



◦ Ejemplo de un escenario de condición de carrera

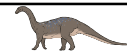


Inicialmente:
contador = 5

| T | Proceso | Operación | reg1 | reg2 | contador |
|---|---------|--------------------|------|------|----------|
| 0 | Prod. | mov contador, reg1 | 5 | ? | 5 |
| 1 | Prod. | inc reg1 | 6 | ? | 5 |
| 2 | Cons. | mov contador, reg2 | ? | 5 | 5 |
| 3 | Cons. | dec reg2 | ? | 4 | 5 |
| 4 | Cons. | mov reg2, contador | ? | 4 | 4 |
| 5 | Prod | mov reg1, contador | 6 | ? | 6 |

 incorrecto ||

2.Comunicación por memoria común



◦ Criterio de corrección en programas concurrentes

El criterio de corrección/consistencia más usual para programas concurrentes es:

- **Seriabilidad** (consistencia secuencial): El resultado de la ejecución concurrente de un conjunto de operaciones ha de ser equivalente al resultado de ejecutar secuencialmente cada una de las operaciones, en alguno de los ordenes secuenciales posibles.

Condición de carrera = no seriabilidad: no hay ninguna posible ejecución secuencial de un conjunto de operaciones que de el mismo resultado que la ejecución concurrente.

2.Comunicación por memoria común



◉ Ejemplo de seriabilidad

__La ejecución concurrente de op1, op2 y op3 :

op1 || op2 || op3

se considera correcta si el estado final de los datos compartidos es igual al estado en que quedarían después de alguna de las siguientes ejecuciones secuenciales:

op1 ; op2 ; op3 o op1 ; op3 ; op2 o
op2 ; op1 ; op3 o op2 ; op3 ; op1 o
op3 ; op1 ; op2 o op3 ; op2 ; op1 .

- Ejemplo numérico: sea una variable x con valor inicial 3 sobre la que se pueden realizar dos operaciones:
 - op1: incremento de 8
 - op2: multiplicación por 5
- Resultados correctos de op1 || op2 : 55 y 23

Comunicación y sincronización de procesos



◉ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes



3.El problema de la sección crítica



◦ Concepto de sección crítica

Una sección crítica es una zona de código en la que se accede a variables compartidas por varios procesos.

- **Problemas potenciales:** puede introducir condiciones de carrera si no se adoptan las medidas adecuadas.
- **Posible solución:** sincronizar el acceso a los datos de manera que mientras un proceso ejecuta su sección crítica ningún otro proceso ejecuta la suya (**exclusión mutua**).

3. El problema de la sección crítica



◦ Formulación del problema de la sección crítica

- Sean n procesos compitiendo para acceder a datos compartidos
- Cada proceso tiene una zona de código, denominada sección crítica, en la que accede a los datos compartidos.
- Problema: encontrar un protocolo del tipo:

protocolo de entrada
sección CRÍTICA
protocolo de salida
sección RESTANTE

Que satisfaga las tres condiciones siguientes :

3. El problema de la sección crítica



◦ Solución al problema de la sección crítica

Cualquier solución al problema de la sección crítica ha de cumplir tres requisitos:

- **Exclusión mutua:** si un proceso está ejecutando su sección crítica ningún otro proceso puede estar ejecutando la suya.
- **Progreso:** si ningún proceso está ejecutando su sección crítica y hay otros que desean entrar a las suyas, entonces la decisión de qué proceso entrará a la sección crítica se toma en un tiempo finito y sólo puede ser seleccionado uno de los procesos que desean entrar.
- **Espera limitada:** Después de que un proceso haya solicitado entrar en su sección crítica, existe un límite en el número de veces que se permite que otros procesos entren a sus secciones críticas.

3. El problema de la sección crítica



◦ Solución al problema de la sección crítica

- Supuestos:
 - Los procesos se ejecutan a velocidad no nula.
 - La corrección no ha de depender de hacer suposiciones sobre la velocidad relativa de ejecución de los procesos.

3. El problema de la sección crítica



◦ Ejemplo: Algoritmo que soluciona el caso para dos procesos

Variables compartidas:

```
var turno : 0..1;
```

task Pi;

```
...  
while turno <> i do no-op;
```

sección CRÍTICA

```
turno := j;
```

sección RESTANTE

end Pi;

task Pj;

```
...  
while turno <> j do no-op;
```

sección CRÍTICA

```
turno := i;
```

sección RESTANTE

end Pj;

No satisface
el requisito
del progreso

Comunicación y sincronización de procesos



◦ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- ➡ 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

4. Soluciones hardware



◦ Soluciones hardware al problema de la sección crítica

Las soluciones hardware son soluciones a nivel de instrucciones del lenguaje máquina:

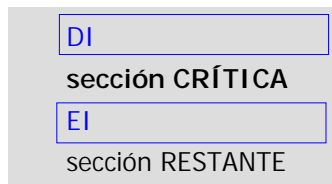
- Deshabilitación de interrupciones
- Instrucción *test_and_set* atómica (indivisible).
- Instrucción *swap* atómica.

4. Soluciones hardware



◦ Deshabilitación de interrupciones

- La deshabilitación de interrupciones se realiza utilizando las instrucciones:
 - *DI* : Deshabilitar interrupciones
 - *EI* : Habilitar interrupciones
- Se consigue la **exclusión mutua** inhibiendo los cambios de contexto durante la sección crítica, obligando así a que los procesos se ejecuten de manera atómica.
- Solución únicamente viable al nivel del núcleo del sistema operativo: puesto que no es deseable dejar el control de las interrupciones en manos de los procesos de usuario. Las instrucciones *DI* y *EI* sólo se pueden ejecutar en modo privilegiado.



4. Soluciones hardware



◦ Operación *test and set* atómica

La operación *test_and_set* permite evaluar y modificar una variable atómicamente en una sola operación de ensamblador.

La **especificación** funcional de esta operación es:

```
function testAndSet(var objetivo: boolean): boolean;  
begin  
  testAndSet := objetivo;  
  objetivo := TRUE;  
end
```

atómicamente!

4. Soluciones hardware



◦ Solución al problema de la s.c. con *test and set*

Variables compartidas:

```
var llave := FALSE : boolean;
```

Espera activa:

el proceso no se suspende por esperar la entrada a la SC

```
task Pi;
```

```
...
```

```
while testAndSet(llave) do no-op;
```

```
sección CRÍTICA
```

```
llave := FALSE;
```

```
sección RESTANTE
```

```
...
```

```
end Pi;
```

4. Soluciones hardware



◦ Solución a la s.c. con *test and set* y espera limitada

__La solución anterior no satisface el requisito de la espera limitada.____

Variables compartidas:

```
var esperando := FALSE : array[0..n-1] of boolean;  
    cerradura := FALSE : boolean;
```

4. Soluciones hardware



◦ Solución a la s.c. con *test and set* y espera limitada

```
task Pi;  
var j: 0..n-1;  
    llave: boolean;
```

Algoritmo del proceso i:

```
...  
esperando[i] := TRUE;  
llave := TRUE;  
while esperando[i] and llave do llave := testAndSet(cerradura);  
esperando[i] := FALSE;
```

sección CRÍTICA

```
j := i+1 mod n;  
while (j <> i) and (not esperando[j]) do j := j+1 mod n;  
if j=i then cerradura := FALSE else esperando[j] := FALSE;
```

sección RESTANTE

```
end Pi;
```

4. Soluciones hardware



◦ Solución a la s.c. con *test and set* y espera limitada

Proceso 0 (i=0)

```
esperando[i] := TRUE;  
llave := TRUE;  
while esperando[i] and llave do llave := testAndSet(cerradura);  
esperando[i] := FALSE;
```

```
j := i+1 mod n;  
while (j < i) and (not esperando[j]) do j := j+1 mod n;  
if j=i then cerradura := FALSE else esperando[j] := FALSE;
```

.....

Proceso 4 (i=4)

cerradura F

esperando

| | |
|-----|-----|
| F | 0 |
| F | 1 |
| F | 2 |
| F | 3 |
| F | 4 |
| F | 5 |
| F | 6 |
| F | 7 |
| ... | ... |

4. Soluciones hardware



◦ Solución a la s.c. con *test and set* y espera limitada

Proceso 0 (i=0)

```
esperando[i] := TRUE;  
llave := TRUE;  
while esperando[i] and llave do llave := testAndSet(cerradura);  
esperando[i] := FALSE;
```

```
j := i+1 mod n;  
while (j < i) and (not esperando[j]) do j := j+1 mod n;  
if j=i then cerradura := FALSE else esperando[j] := FALSE;
```

.....

Proceso 4 (i=4)

cerradura F

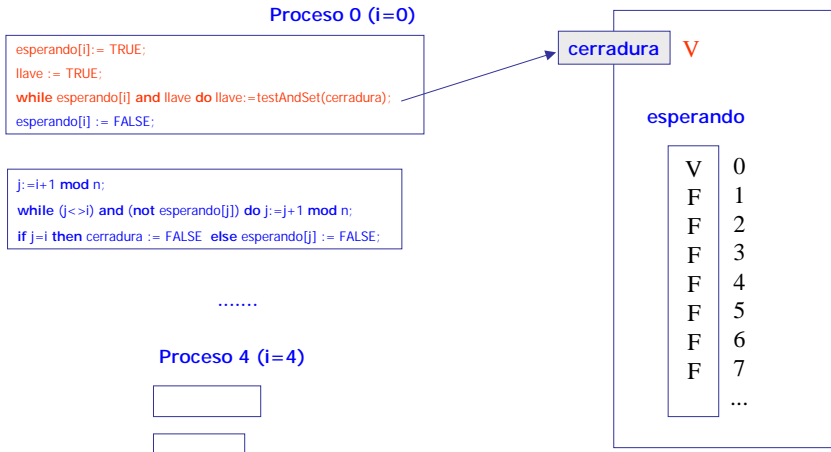
esperando

| | |
|-----|-----|
| V | 0 |
| F | 1 |
| F | 2 |
| F | 3 |
| F | 4 |
| F | 5 |
| F | 6 |
| F | 7 |
| ... | ... |

4. Soluciones hardware



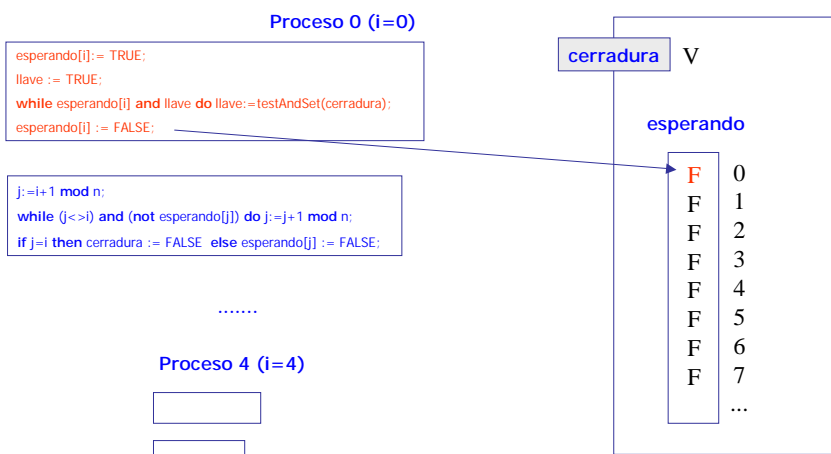
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



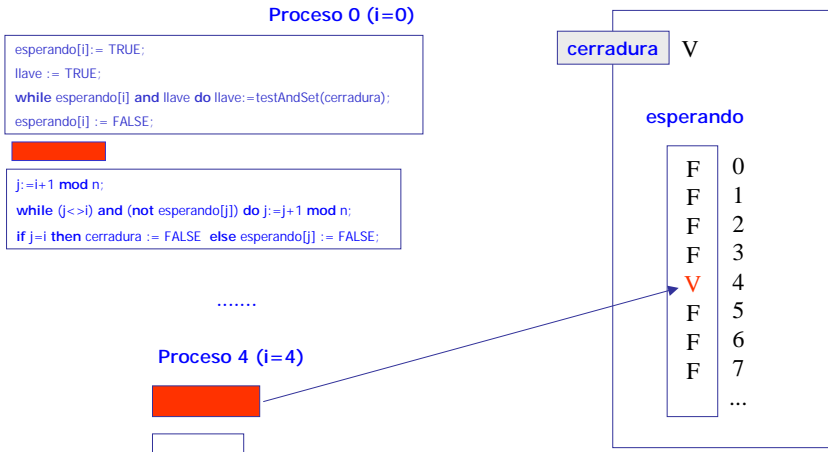
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



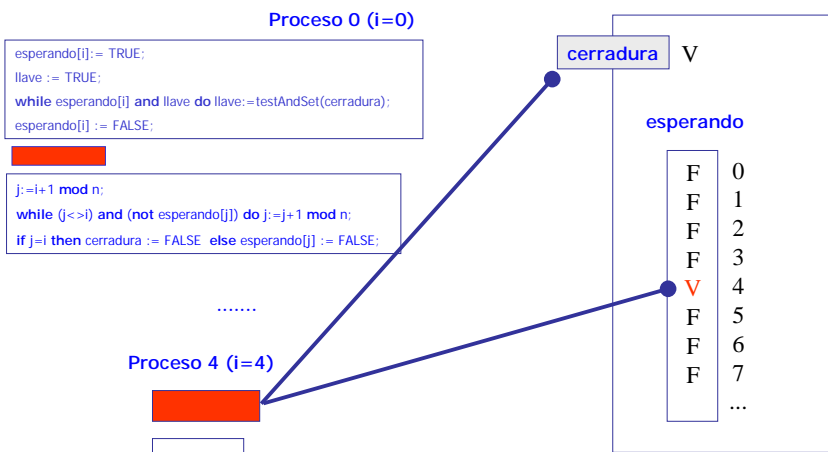
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



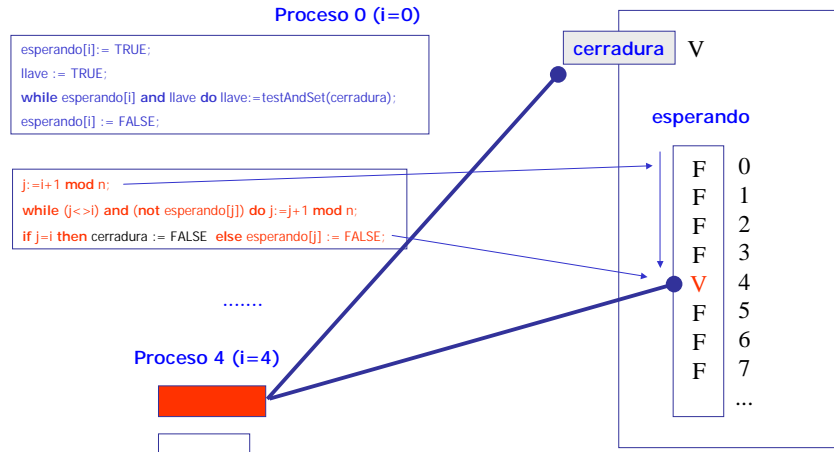
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



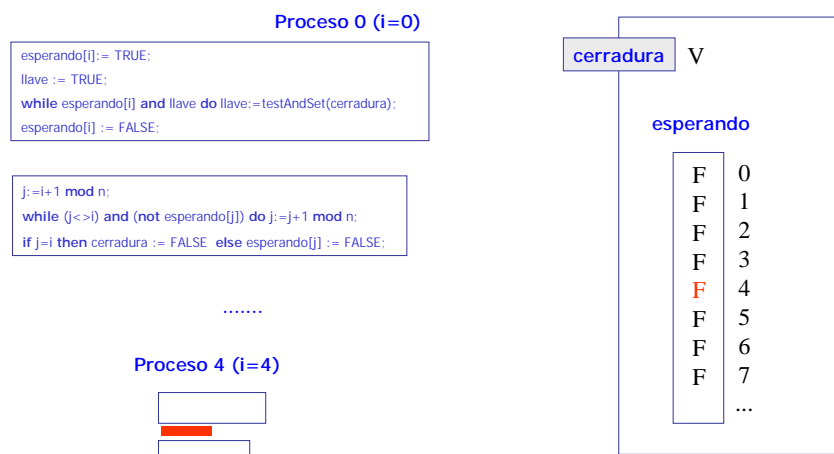
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



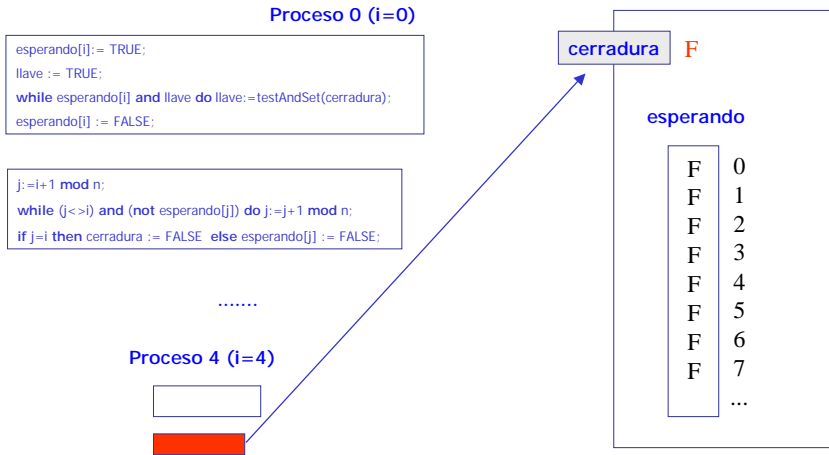
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



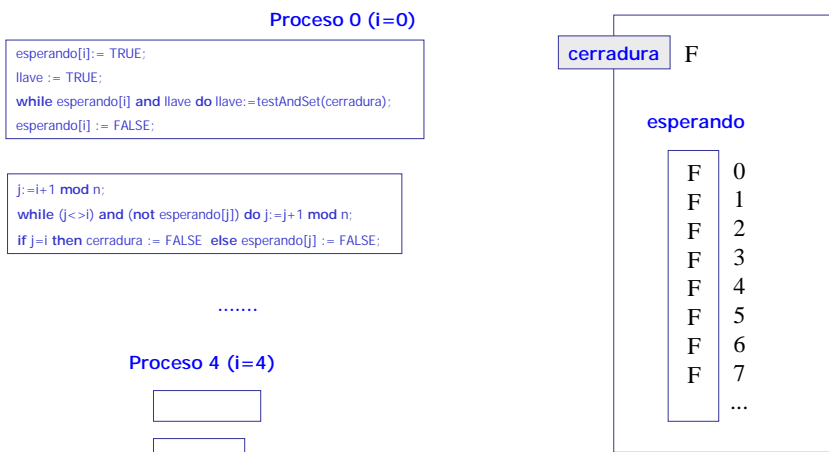
◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



◦ Solución a la s.c. con *test and set* y espera limitada



4. Soluciones hardware



◦ Operación *swap* atómica

La operación *swap* permite intercambiar atómicamente dos variables en una sola operación de ensamblador.

La **especificación** funcional de esta operación es:

```
function swap(var a,b: boolean);  
var temp : boolean;  
begin  
  temp:= a;  
  a:=b;  
  b := temp;  
end
```

atómicamente!

4. Soluciones hardware



◦ Solución a la s.c con *swap*

Variables compartidas:

```
var cerradura := FALSE : boolean;
```

Espera activa

```
task Pi;  
  var llave: boolean;  
  ...  
  llave := TRUE  
  while swap(cerradura, llave) until llave = FALSE;  
  sección CRÍTICA  
  cerradura := FALSE;  
  sección RESTANTE  
  ...  
end Pi;
```

Comunicación y sincronización de procesos



◦ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes



5. Semáforos



◦ Definición de semáforo

- Tipo de datos que toma valores enteros y sobre el que se definen las siguientes operaciones atómicas:

▫ S: semáforo(N); (* semáforo S con valor inicial $N \geq 0$ *)

▫ P(S) $S := S - 1;$
 if $S < 0$ then esperar(S);

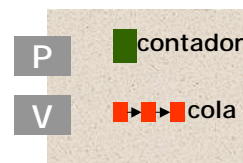
atómicamente!

▫ V(S) $S := S + 1;$
 if $S \leq 0$ then despertar(S);

atómicamente!

- La operación esperar(S) suspende al proceso que ha invocado P() y lo introduce en una cola de espera asociada a S.
- La operación despertar(S) extrae un proceso de la cola de espera asociada a S y lo activa.

Semáforo



5. Semáforos



◦ Solución al problema de la sección crítica con semáforos

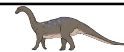
Los semáforos son un mecanismo de sincronización que no requiere espera activa

Variables compartidas:

```
var mutex: semaforo(1); (* valor inicial 1 *)
```

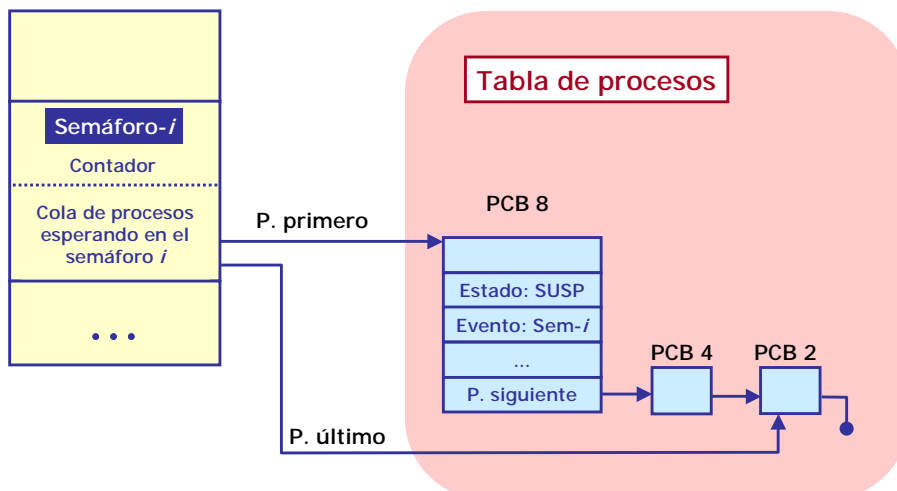
```
task Pi;  
...  
P(mutex);  
sección CRÍTICA  
V(mutex);  
sección RESTANTE  
...  
end Pi;
```

5. Semáforos

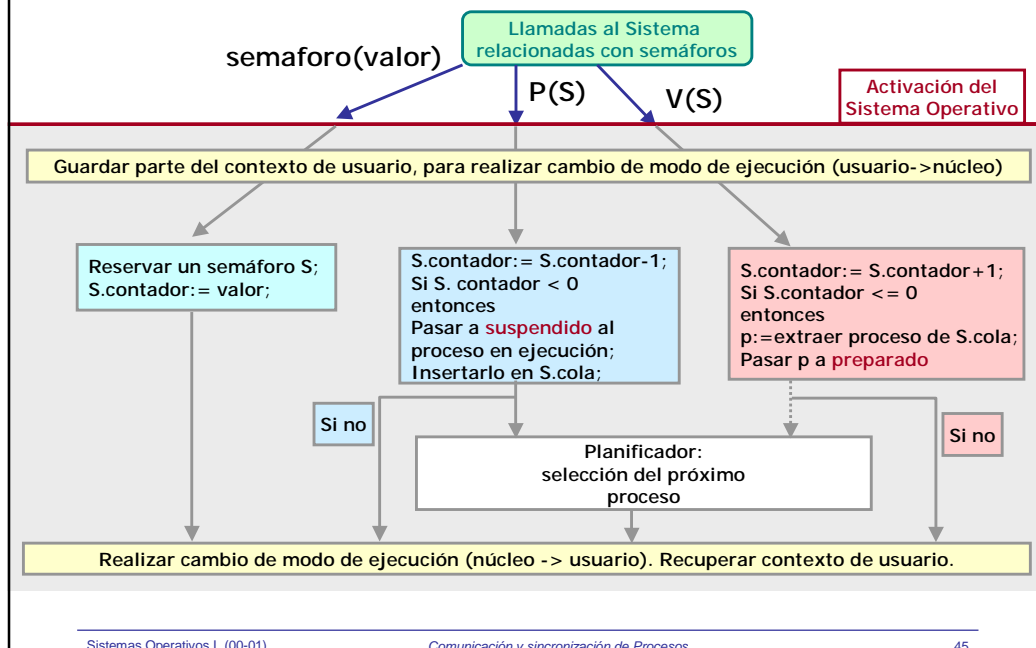


◦ Implementación de semáforos

ESTRUCTURAS DE CONTROL DEL
SISTEMA : Vector de semáforos



Flujo de Control del S.O.



Sistemas Operativos I (00-01)

Comunicación y sincronización de Procesos

45

5. Semáforos



◦ Semáforos versus espera activa

- Espera activa:
 - El proceso que espera entrar en la sección crítica "desaprovecha" el tiempo de CPU comprobando cuando puede entrar (ejecutando las instrucciones del protocolo de entrada).
- Semáforos:
 - La espera se realiza en la cola del semáforo, que es una cola de procesos suspendidos.
 - El proceso que espera entrar en la sección crítica no utiliza CPU; está suspendido.

Proceso:

```

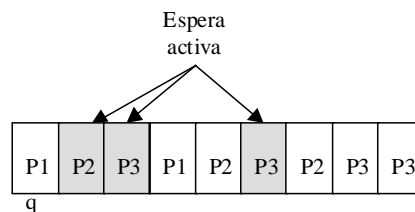
EntrdaSC;
  SecciónCrítica;
SalidaSC;
    
```

```
P1, P2, P2:Proceso;
```

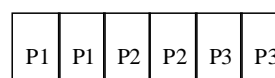
```

Duración se la
sección crítica 2q
    
```

Espera activa



Espera pasiva



Sistemas Operativos I (00-01)

Comunicación y sincronización de Procesos

46

5. Semáforos



◦ Semáforos: ejemplos de utilización

— Un semáforo es un mecanismo de sincronización de uso general:

- Para conseguir exclusión mutua.
- Para forzar relaciones de precedencia, como por ejemplo:
 - El proceso Pj debe ejecutar B después de que el proceso Pi haya ejecutado A.

```
var sinc: semaforo(0);
```

```
task Pi;  
...  
A;  
V(sinc);  
...
```

```
task Pj;  
...  
P(sinc)  
B;  
...
```

5. Semáforos



◦ Semáforos: problemas derivados de una utilización incorrecta

- **Interbloqueos:** Hay un conjunto de procesos en el que todos esperan (indefinidamente) un evento que sólo otro proceso del conjunto puede producir.
Ejemplo:

```
var s1: semaforo(1);  
s2: semaforo(1);
```

```
task P1;  
...  
P(s1);  
P(s2);  
...  
V(s1);  
V(s2);
```

```
task P2;  
...  
P(s2);  
P(s1);  
...  
V(s1);  
V(s2);
```

Escenario:


```
P1 → P(s1);  
P2 → P(s2);  
P2 → P(s1);  
P1 → P(s2);
```

Sistema en situación de interbloqueo ...

Comunicación y sincronización de procesos



◦ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
-  6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

6. Problemas clásicos de concurrencia



◦ Problemas clásicos de programación concurrente

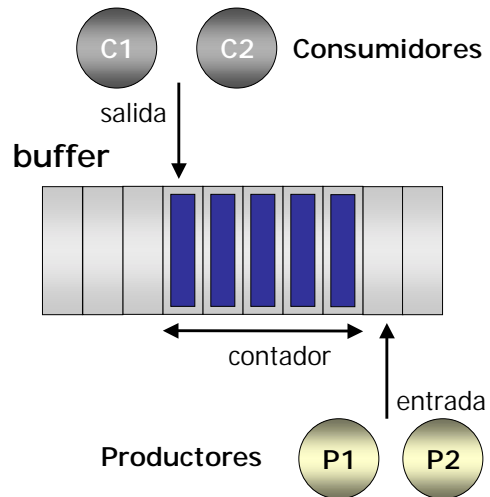
— Existe una serie de problemas clásicos que se utilizan como “banco de pruebas” de los diferentes mecanismos de sincronización entre procesos:

- Los productores y los consumidores
- Los lectores y los escritores
- Los cinco filósofos
- El barbero dormilón
- ...

6. Problemas clásicos de concurrencia



- El problema de los productores y consumidores con buffer acotado



6. Problemas clásicos de concurrencia



- Los productores y consumidores
- Variables compartidas

```
var buffer: array[0..n-1] of elemento;  
    entrada:=0, salida:=0, contador:= 0 : 0..n-1;  
lleno: semaforo(0); vacio: semaforo(n), mutex: semaforo(1);
```

6. Problemas clásicos de concurrencia



◉ Los productores y consumidores

```
task productor;  
  var item: elemento;  
  repeat  
    item := producir();  
    P(vacio);  
    P(mutex);  
    buffer[entrada] := item;  
    entrada := (entrada + 1) mod n;  
    contador:=contador+1;  
    V(mutex);  
    V(lleno);  
  until false  
end productor;
```

```
task consumidor;  
  var item: elemento;  
  repeat  
    P(lleno);  
    P(mutex);  
    item := buffer[salida];  
    salida := (salida + 1) mod n;  
    contador:=contador-1;  
    V(mutex);  
    V(vacio);  
    consumir(item);  
  until false  
end consumidor;
```

6. Problemas clásicos de concurrencia



◉ Los lectores y los escritores

◉ Problema de los lectores y los escritores

- Hay un conjunto de datos comunes (un fichero, una base de datos, etc.)
- Los procesos lectores, acceden a los datos en modo de sólo lectura
- Los procesos escritores, acceden a los datos en modo lectura-escritura

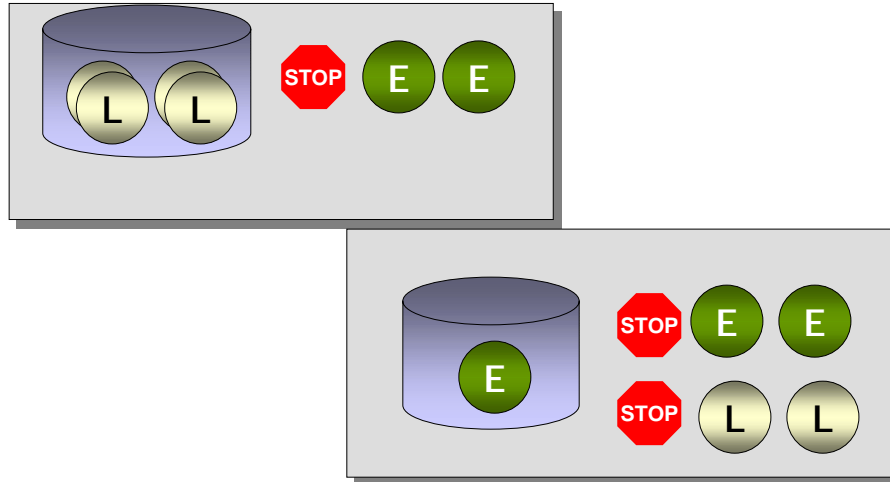
◉ Especificación del protocolo: reglas de corrección

- Diversos lectores pueden leer concurrentemente.
- Los escritores se excluyen mutuamente entre ellos
- Los escritores se excluyen mutuamente con los lectores.

6. Problemas clásicos de concurrencia



- Los lectores y los escritores
- Especificación del protocolo: reglas de corrección



6. Problemas clásicos de concurrencia



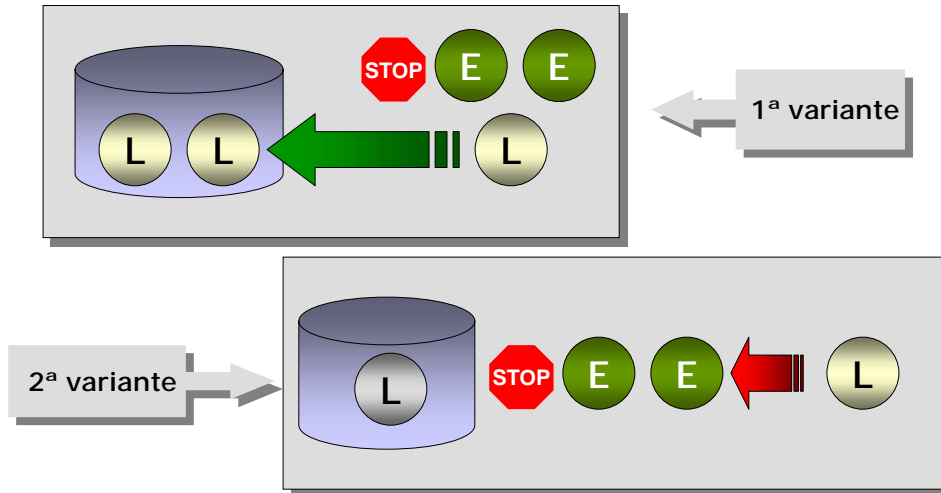
- Los lectores y escritores
- Especificación del protocolo: reglas de prioridad

- Primera variante de los Lectores-Escritores: (Prioridad a los lectores)
 - Si hay lectores leyendo y
 - escritores esperando, entonces
 - un nuevo lector tiene preferencia sobre el escritor que espera.
 - ➔ Hay inanición para los escritores, si no paran de llegar lectores
- Segunda variante de los Lectores-Escritores: (Prioridad a los escritores)
 - Si hay escritores esperando,
 - éstos siempre tienen preferencia sobre nuevos lectores que lleguen
 - ➔ Hay inanición para los lectores, si no paran de llegar escritores
- Tercera variante, sin inanición para ningún tipo de proceso

6. Problemas clásicos de concurrencia



- Los lectores y los escritores
- Especificación del protocolo: reglas de prioridad



Sistemas Operativos I (00-01)

Comunicación y sincronización de Procesos

57

6. Problemas clásicos de concurrencia



- Los lectores y los escritores (semáforos)
- Solución a la primera variante: Inanición para los escritores

Variables compartidas:

```
var mutex: semaforo(1);
    esc semaforo(1);
    nlectores:=0 : integer;
```

task escritor;

```
...
P(esc);
    escribir();
V(esc);
...
```

end escritor;

task lector;

```
...
P(mutex);
    nlectores := nlectores + 1;
    if nlectores = 1 then P(esc);
V(mutex);
    leer();
P(mutex);
    nlectores := nlectores - 1;
    if nlectores = 0 then V(esc);
V(mutex);
...
```

end lector;

Sistemas Operativos I (00-01)

Comunicación y sincronización de Procesos

58

Comunicación y sincronización de procesos



◦ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

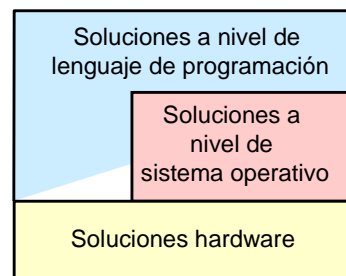


7. Construcciones lingüísticas



- Con el fin de simplificar la programación de aplicaciones concurrentes, algunos lenguajes de programación proporcionan herramientas (construcciones lingüísticas) que facilitan la sincronización entre las tareas y proporcionan un modelo de programación uniforme.

- Monitores
- Tipos *protegidos* en ADA
- Métodos *synchronized* en Java



7.1 Monitores



◦ Concepto

Un monitor es un tipo de datos que encapsula datos y operaciones.

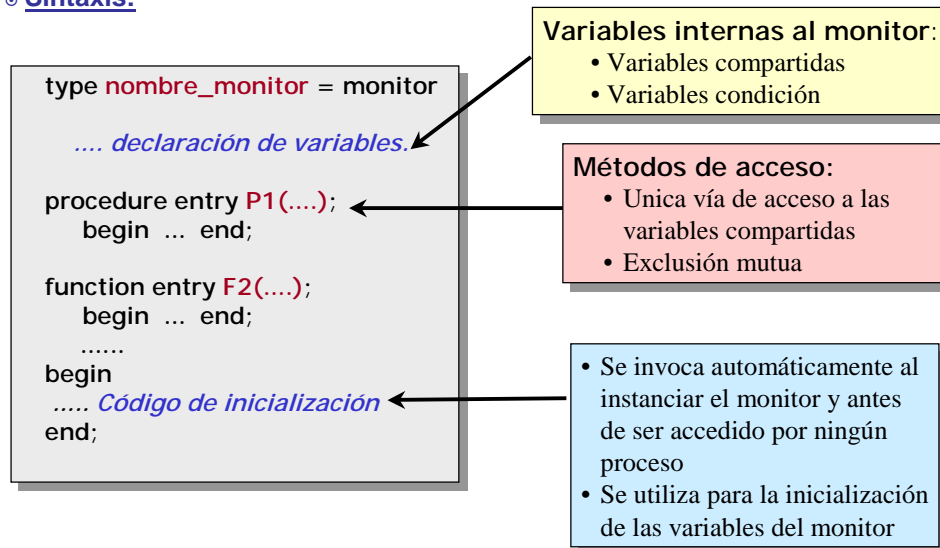
Proporciona:

- Exclusión mútua.
- Sincronización.

7.1 Monitores



◦ Sintaxis:



7.1 Monitores



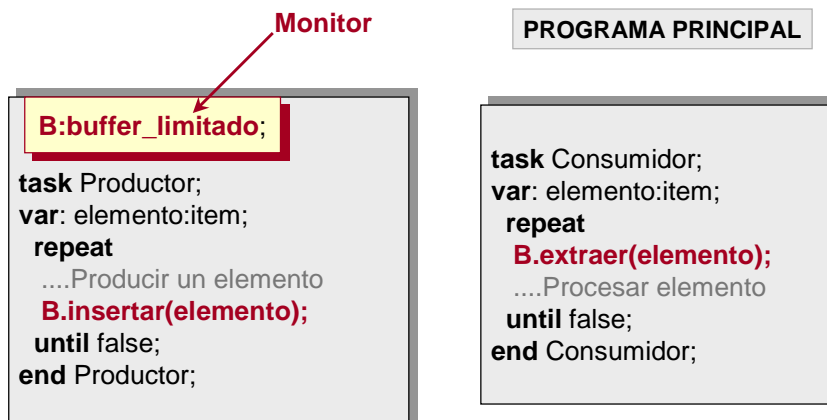
◉ Variables de tipo condición:

- Para definir esquemas complejos de sincronización, se pueden definir las variables del tipo *condition*.
 - **var x: condition;**
- Sobre las variables *condition* se pueden realizar las siguientes operaciones:
 - **x.wait;** Causa la suspensión (en una cola asociada a x) del proceso que invocó la operación.
 - **x.signal;** Se reanuda (si existe) un proceso suspendido en la cola asociada a x.
 - **x.awaited;** Indica el número de procesos suspendidos en la cola asociada a x.

7.1 Monitores



◉ Productores-consumidores con monitores



Productores-consumidores con monitores



MONITOR

```
type buffer_limitado = monitor
```

```
const n=100;  
cont, entrada, salida:integer;  
lleno, vacio: condition;  
buffer: array[0..n-1] of item;
```

```
procedure entry insertar (elem:item);  
begin  
.....  
end;
```

```
procedure entry extraer (var elem:item);  
begin  
.....  
end;
```

```
begin  
  entrada:=0; salida:=0;cont:=0;  
end;
```

```
procedure entry insertar (elem:item);  
begin  
  if cont=n then lleno.wait;  
  cont:=cont+1;  
  buffer[entrada]:=item;  
  entrada:=(entrada + 1) mod n;  
  vacio.signal;  
end;
```

```
procedure entry extraer(var elem:item);  
begin  
  if cont=0 then vacio.wait;  
  cont:=cont-1;  
  item:=buffer[salida];  
  salida :=(salida + 1) mod n;  
  lleno.signal;  
end;
```

7.1 Monitores



Variantes

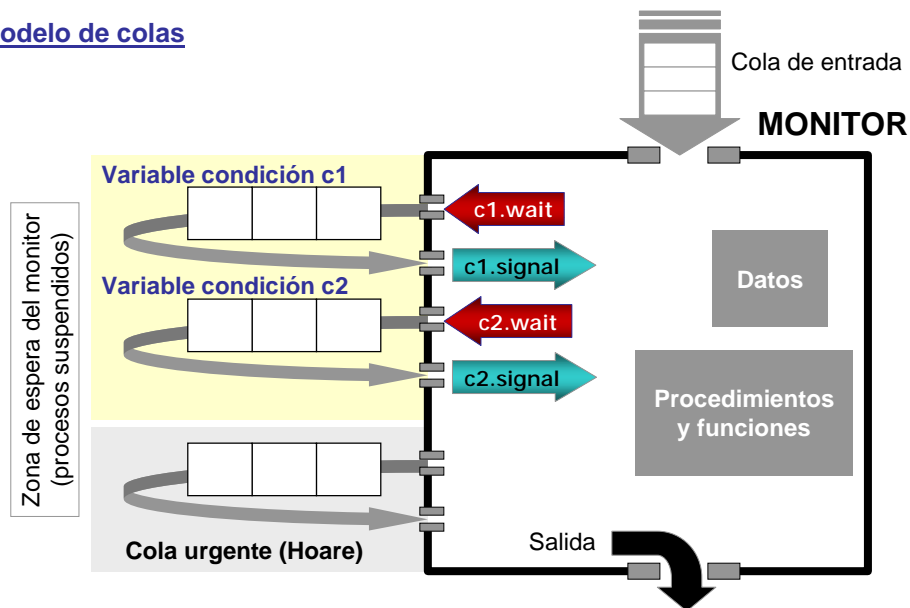
Existen diferentes variantes en la definición de un monitor según resuelvan el siguiente problema:

- Un proceso **P** ejecuta **x.signal** y activa a otro proceso **Q**, → Potencialmente **P** y **Q** pueden continuar su ejecución dentro del monitor. ¿Cómo se garantiza la ejecución en exclusión mutua en el monitor?
 - **Modelo de Hoare**: El proceso que invoca la operación **x.signal (P)** se suspende (en una cola de “urgencia”) de forma que el proceso reanudado por la operación **x.signal (Q)** pasa a ejecutarse dentro del monitor.
 - **Modelo de Lampson y Redell**: El proceso **P** continúa ejecutándose y **Q** se ejecutará cuando el monitor quede libre.
 - **Modelo de Brinch-Hansen**: La operación **x.signal** ha de ser la última operación que un proceso invoca antes de salir del monitor.

7.1 Monitores



◦ Modelo de colas



7.1 Monitores



◦ Implementación de monitores con semáforos

◦ Variables globales, por cada monitor se definen:

```
var mutex: semaforo(1);  
urgente: semaforo(0); (*cola de urgencia*)  
cont_urg: integer:=0;
```

◦ Para cada procedimiento o función F, se genera:

```
P(mutex);  
..... Cuerpo de F;  
if cont_urg > 0  
then V(urgente)  
else V(mutex);
```

7.1 Monitores



- Implementación de monitores con semáforos
- Para cada variable condición X, se define:

```
var x_sem: semaforo(0);  
x_cont: integer:=0;
```

Operación X.wait

```
x_cont:= x_cont+1;  
if cont_urg > 0  
then V(urgente)  
else V(mutex);  
P(x_sem);  
x_cont:= x_cont-1;
```

Operación X.signal

```
if x_cont > 0  
then begin  
    cont_urg:= cont_urg + 1;  
    V(x_sem);  
    P(urgente);  
    cont_urg:= cont_urg - 1;  
end;
```

7.1 Monitores



- Lectores-escriptores con monitores

Monitor

PROGRAMA PRINCIPAL

```
var le:le-control;
```

```
task lector();  
begin  
    repeat  
        le.pre-leer();  
        leer();  
        le.post-leer();  
    until false;  
end lector;
```

```
task escritor();  
begin  
    repeat  
        le.pre-escribir();  
        escribir();  
        le.post-escribir();  
    until false;  
end escritor;
```

7.1 Monitores



◦ Lectores-escriptores con monitores

MONITOR

```
type le-control = monitor;  
var lectores, escritores : Integer;  
    leer, escribir : condition;  
  
procedure entry pre-leer();  
    if escritores > 0 then leer.wait;  
    lectores := lectores+1;  
    leer.signal; (* despertar al siguiente *)  
end;  
  
procedure entry post-leer();  
    lectores := lectores-1;  
    if lectores = 0 then escribir.signal;  
end;
```

Inanición para escritores

```
procedure entry pre-escribir();  
    if escritores > 0 or lectores > 0  
    then escribir.wait;  
    escritores := escritores+1;  
end;  
  
procedure entry post-escribir();  
    escritores := escritores-1;  
    if leer.awaited > 0  
    then leer.signal;  
    else escribir.signal;  
end;  
  
begin  
    lectores:=0; escritores:=0;  
end.
```

7.1 Monitores



◦ Lectores-escriptores con monitores

MONITOR

```
type le-control = monitor;  
var lectores, escritores : Integer;  
    leer, escribir : condition;  
  
procedure entry pre-leer();  
    if escritores > 0 or escribir.awaited > 0  
    then leer.wait;  
    lectores := lectores+1;  
    leer.signal; (* despertar al siguiente *)  
end;  
  
procedure entry post-leer();  
    lectores := lectores-1;  
    if lectores = 0 then escribir.signal;  
end;
```

Inanición para lectores

```
procedure entry pre-escribir();  
    if escritores > 0 or lectores > 0  
    then escribir.wait;  
    escritores := escritores+1;  
end;  
  
procedure entry post-escribir();  
    escritores := escritores-1;  
    if escribir.awaited > 0  
    then escribir.signal;  
    else leer.signal;  
end;  
  
begin  
    lectores:=0; escritores:=0;  
end.
```

7.1 Monitores



◦ Lectores-escritores con monitores

MONITOR

```
type le-control = monitor;
var lectores, escritores : Integer;
    leer, escribir : condition;

procedure entry pre-leer();
if escritores > 0 or escribir.awaited > 0
then leer.wait;
lectores := lectores+1;
leer.signal; (* despertar al siguiente *)
end;

procedure entry post-leer();
lectores := lectores-1;
if lectores = 0 then escribir.signal;
end;
```

Sin Inanición

```
procedure entry pre-escribir();
if escritores > 0 or lectores > 0
then escribir.wait;
escritores := escritores+1;
end;

procedure entry post-escribir();
escritores := escritores-1;
if leer.awaited > 0
then leer.signal;
else escribir.signal;
end;

begin
    lectores:=0; escritores:=0;
end.
```

7.1 Monitores



◦ Monitores: espera condicional

Cuando se produce una llamada a **x.signal**, ¿qué proceso se activa?

- El monitor permite una variante de la operación **wait** sobre una variable condición para que, cuando se produzca la operación **signal**, se active a los procesos suspendidos por orden de prioridad.
- Se define la siguiente construcción:
 - **x.wait(c)**
 - donde:
 - c es una expresión entera que se evalúa cuando se realiza la operación **wait**.
 - El valor de c se almacena junto al proceso suspendido.
 - Al invocar la operación **signal**, se activa el proceso que tenga mayor prioridad (menor valor numérico de c).

7.1 Monitores



◦ Monitores espera condicional

◦ Asignación de recursos basada en tiempo de uso.

- De aquellos procesos que se encuentren esperando por el recurso ocupado, se reanudará aquel que lo vaya a usar durante menos tiempo.

```
type asignacion-recurso = monitor
var
  ocupado: boolean;
  x: condition;
procedure entry adquirir(tiempo: integer);
begin
  if ocupado then x.wait(tiempo);
  ocupado:= true;
end;
procedure entry liberar;
begin
  ocupado:= false;
  x.signal;
end;
begin
  ocupado:= false;
end.
```

7.2 Objetos protegidos



◦ Objetos protegidos de ADA : declaración

- Un objeto protegido encapsula datos que son accedidos simultáneamente por varias tareas.
- Las tareas acceden a los datos del objeto utilizando operaciones protegidas, las cuales pueden ser **funciones**, **procedimientos** y **entradas**.

```
protected [type]
  Nombre[ (Discriminantes) ]
is
  procedure Pnombre(Params);
  function Fnombre(Params) return Un_Tipo;
  entry Enombre(Params);

private
  Datos_Protegidos : Su_Tipo;
end Nombre;
```

7.2 Objetos protegidos



◦ Objetos protegidos de ADA

◦ La especificación tiene dos partes:

- Una pública, donde se declaran las operaciones protegidas.
- Una privada donde se declaran los datos protegidos. La parte privada no es visible por los clientes del objeto protegido.

```
protected [type]
  Nombre[(Discriminantes)]
is
  procedure Pnombre(Params);
  function Fnombre(Params) return Un_Tipo;
  entry Enombre(Params);

private
  Datos_Protegidos : Su_Tipo;
end Nombre;
```

especificación

7.2 Objetos protegidos



◦ Objetos protegidos de ADA: declaración

- En el cuerpo del objeto se implementan las operaciones protegidas.

```
protected body Nombre
is
  ---
end Nombre;
```

cuerpo

- ADA permite declarar tipos protegidos o instancias individuales.

```
Objeto_1: Nombre;
Objeto_2: Nombre;
```

instanciación

```
protected [type]
  Nombre[(Discriminantes)]
is
  procedure Pnombre(Params);
  function Fnombre(Params)
    return Un_Tipo;
  entry Enombre(Params);

private
  Datos_Protegidos : Su_Tipo;
end Nombre;
```

especificación

7.2 Objetos protegidos



Objetos protegidos de ADA: control de concurrencia

- Las **funciones** permiten consultar el objeto protegido y, por tanto, pueden ejecutarse simultáneamente varias funciones.
- Cuando una tarea está ejecutando un **procedimiento** o una **entrada** del objeto protegido, otra tarea interesada en acceder al objeto protegido queda detenida hasta que éste queda libre.
- Las **entradas** tienen asociadas una condición lógica denominada **barrera (o guarda)**. Si en el momento de llamar al objeto la barrera es cierta, la tarea ejecuta la entrada. En caso contrario, la tarea queda detenida en dicha entrada.

```
entry Una_Entrada (Params) when Alguna_Condición is
  var_locales;
begin
  ...
end Una_Entrada;
```

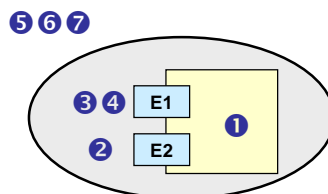
La expresión de una barrera no puede contener parámetros de la entrada. !

7.2 Objetos protegidos



Objetos protegidos de ADA: control de concurrencia

- Las barreras se revalúan cuando finaliza la ejecución de una entrada o un procedimiento. Si alguna barrera es ahora cierta se elige una tarea detenida por dicha barrera y reanuda su ejecución.
- Las tareas detenidas en barreras tienen prioridad cuando son despertadas frente a aquellas que están esperando fuera del objeto protegido.



7.2 Objetos protegidos



Construcción de semáforos con objetos protegidos de ADA

especificación

```
protected type Semaforo (Inicial:integer:=1)
is
  entry P;
  procedure V;
private
  contador:Integer:=Inicial;
end Semaforo;
```

cuerpo

```
protected body Semaforo is
  entry P when contador>0 is
  begin
    contador:=contador-1;
  end P;

  procedure V is
  begin
    contador:=contador+1;
  end V;
end Semaforo;
```

utilización

```
Un_semaforo: Semaforo(5);
....
Un_semaforo.P;
....
Un_semaforo.V;
```

7.2 Objetos protegidos



Ejemplo en ADA: el buffer acotado

procedure Prodcons is

```
n : constant Integer := 10;
subtype Indice is Integer range 0..n-1;
subtype Cantidad is Integer range 0..n;
subtype Item is Integer ;
type Buffers is array (Indice) of Item;
```

```
task type Productor;
```

```
task type Consumidor;
```

```
Un_Buffer: Buffer;
Prod_1: Productor;
Prod_2: Productor;
Cons : Consumidor;
```

```
protected type Buffer is
  entry Poner (x : in ITEM);
  entry Quitar (x : out ITEM);
private
  Buf: Buffers;
  Entrada : Indice := 0;
  Salida : Indice := 0;
  Contador: Cantidad := 0;
end Buffer ;
```

7.2 Objetos protegidos



◉ Ejemplo en ADA: el buffer acotado

```
protected body Buffer is

  entry Poner (x : in Item) when Contador < n is
  begin
    Buf(entrada) := x;
    Entrada := (Entrada + 1) mod n;
    Contador := Contador + 1;
  end Poner;

  entry Quitar (x : out Item) when Contador > 0 is
  begin
    x := Buf(Salida);
    Salida := (Salida + 1) mod n;
    Contador := Contador - 1;
  end Quitar;

end Buffer ;
```

7.2 Objetos protegidos



◉ Ejemplo en ADA: el buffer acotado

```
task body Productor is
  x : ITEM;
begin
  loop
    x := ...;
    Un_Buffer.Poner(x);
    -- otras acciones
  end loop;
end Productor ;

task body Consumidor is
  x : ITEM;
begin
  loop
    Un_Buffer.Quitar(x);
    -- otras acciones
  end loop;
end Consumidor;

begin
  null;
end ProdCons;
```

Comunicación y sincronización de procesos



◉ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes



8. Sincronización en POSIX



POSIX ofrece dos primitivas de sincronización de hilos: los *mutex* (o cerrojos) y las variables condición.

◉ Mutex (cerrojos)

- Tienen dos estados posibles:
 - **Abierto → ningún hilo es el propietario**
 - **Cerrado → el hilo que lo cierra es el propietario. Un *mutex* no puede tener dos hilos propietarios simultáneamente.**
- Cuando un hilo intenta cerrar un *mutex* que ya ha sido cerrado, se suspende hasta que el hilo propietario lo abra.
- Sólo el hilo propietario puede abrir el *mutex*.
- Creación e inicialización de los *mutex*:
 - **`p_thread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;`**

8. Sincronización en POSIX



| Llamada | Descripción |
|---|--|
| <code>pthread_mutex_lock (mut)</code> | Si el mutex <i>mut</i> estaba abierto, lo cierra y el hilo que lo cierra es el propietario del mutex. Si estaba cerrado, suspende(*) al hilo hasta que se abra el mutex. |
| <code>pthread_mutex_unlock (mut)</code> | Abre el mutex <i>mut</i> . Solo el hilo propietario del mutex puede abrirlo (**). |
| <code>pthread_mutex_trylock(mut)</code> | Igual que <code>pthread_mutex_lock</code> pero en lugar de suspender al hilo si el semáforo estaba cerrado, retorna inmediatamente con un código de error. |

(*) Hay tipos de mutex, que le permiten al propietario cerrarlo varias veces sin bloquearse. Con este tipo de mutex, el propietario debe abrirlo tantas veces como lo hubiera cerrado para dejarlo finalmente abierto.

(**) Hay sistemas, como Redhat 5.1, que permiten que otros hilos abran el mutex, sin embargo este funcionamiento no es portable.

8. Sincronización en POSIX



◦ Variables condición en POSIX

Las variables condición son un tipo abstracto de datos con tres operaciones básicas:

- **wait**: suspende al proceso que la invoca en la condición.
- **signal**: activa un proceso suspendido en la condición
- **broadcast**: activa a todos los procesos suspendidos en la condición

Las variables condición siempre deben estar asociadas a un *mutex*.

Además existen métodos para limitar el tiempo que un hilo está bloqueado en una variable condición.

◦ Creación e inicialización de atributos:

▫ `p_thread_cond_t c1 = PTHREAD_COND_INITIALIZER;`

8. Sincronización en POSIX



◉ Variables condición en POSIX

| Llamada | Descripción |
|---|--|
| <code>pthread_cond_wait(cond, mut)</code> | De forma atómica, realiza la operación <code>pthread_mutex_unlock</code> sobre <code>mut</code> , y bloquea al hilo en la variable condición <code>cond</code> . Cuando es despertado, el hilo cierra nuevamente el mutex <code>mut</code> (realizando la operación <code>pthread_mutex_lock</code>). |
| <code>pthread_cond_signal(cond)</code> | Despierta a uno de los hilos que están bloqueados en la variable condición. Si no hay hilos bloqueados, no sucede nada. |
| <code>pthread_cond_broadcast(cond)</code> | Despierta todos los hilos bloqueados sobre <code>cond</code> . |
| <code>pthread_cond_timedwait</code> (<code>cond, mut, duracion</code>) | Igual que <code>pthread_cond_wait</code> pero si antes de <i>duracion</i> no se ha despertado al hilo, la llamada finalizará con un código de error. Al despertar, el hilo que invoca la llamada, vuelve a cerrar el mutex <code>mut</code> . |

8. Sincronización en POSIX



◉ El productor consumidor en POSIX (i)

Variables globales:

```
#define n 10

int entrada, salida, contador;
int buffer[n];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t lleno = PTHREAD_COND_INITIALIZER;
pthread_cond_t vacio = PTHREAD_COND_INITIALIZER;
```

8. Sincronización en POSIX



◉ El productor consumidor en POSIX (i)

```
void *productor(void * arg){
    int i;
    for (i=0; i<100; i++) Insertar(i);
    pthread_exit(0);
}
```

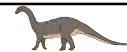
```
void *consumidor(void * arg){
    int i;
    for (i=0; i<100; i++)
        printf("%d ", Extraer());
    pthread_exit(0);
}
```

```
main() {
    pthread_t th_a, th_b;

    entrada = salida = contador = 0;

    pthread_create(&th_a, NULL, productor, NULL);
    pthread_create(&th_b, NULL, consumidor, NULL);
    pthread_join(th_a, NULL);
    pthread_join(th_b, NULL); exit(0);
}
```

8. Sincronización en POSIX



◉ El productor consumidor en POSIX (ii)

```
Insertar (int dato) {
    pthread_mutex_lock(&mutex);
    while (contador >= n) pthread_cond_wait(&lleno, &mutex);
    buffer[entrada] = dato; entrada = (entrada+1) % n;
    contador = contador+1;
    pthread_cond_broadcast(&vacio);
    pthread_mutex_unlock(&mutex);
}
```

```
int Extraer () {
    int dato;
    pthread_mutex_lock(&mutex);
    while (contador == 0) pthread_cond_wait(&vacio, &mutex);
    dato = buffer[salida]; salida = (salida+1) % n;
    contador = contador - 1;
    pthread_cond_broadcast(&lleno);
    pthread_mutex_unlock(&mutex);
    return dato;
}
```

Comunicación y sincronización de procesos



◦ Contenido

- 1.- Introducción
- 2.- Comunicación por memoria común
- 3.- El problema de la sección crítica
- 4.- Soluciones hardware
- 5.- Semáforos
- 6.- Problemas clásicos de programación concurrente
- 7.- Construcciones lingüísticas
- 8.- Sincronización en POSIX
- 9.- Comunicación por mensajes

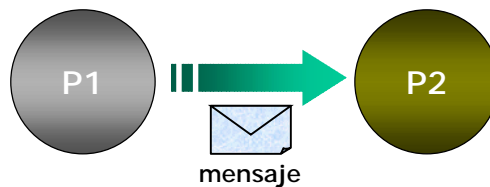


9. Comunicación por mensajes



◦ Sistemas de mensajes

- Los sistemas de mensajes permiten la comunicación entre procesos con espacios de direcciones distintos, bien sean locales o remotos.
- Operaciones básicas (proporcionadas por el Sistema Operativo):
 - `send(dst,mensaje)`: enviar un mensaje a un destino.
 - `receive(org,mensaje)`: recibir un mensaje de un origen.



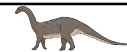
9. Comunicación por mensajes



◉ Variantes

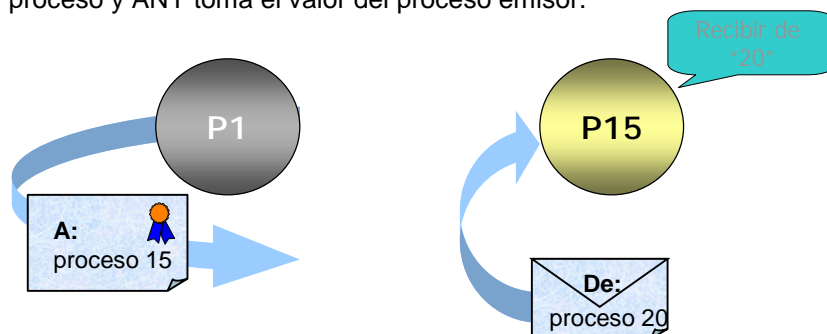
- Según el modo de nombrar origen y destino:
 - Comunicación directa: proceso a proceso.
 - Comunicación indirecta: basada en puertos o buzones.
- Según la capacidad del enlace:
 - Comunicación síncrona: el enlace tiene capacidad = 0.
 - Comunicación asíncrona: el enlace tiene capacidad > 0.

9. Comunicación por mensajes



◉ Comunicación directa

- Los mensajes se direccionan a procesos.
- Los procesos necesitan conocer sus identificadores: las direcciones **dst** y **org** son identificadores de procesos.
- **org** puede ser un comodín, ANY. En este caso se recibe de cualquier proceso y ANY toma el valor del proceso emisor.



9. Comunicación por mensajes



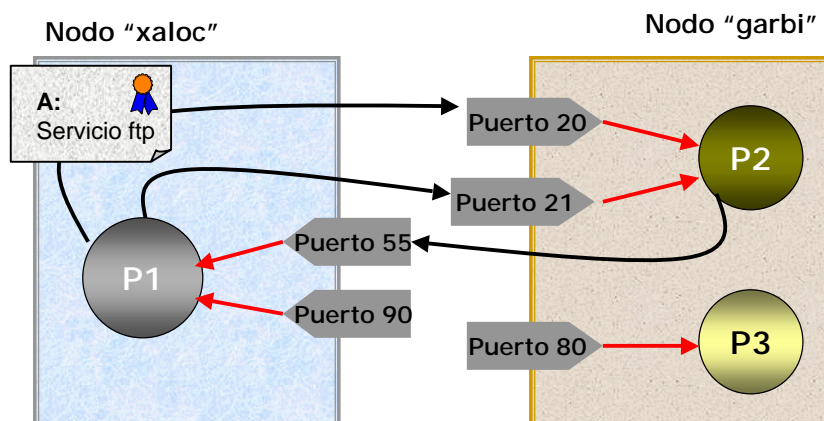
◦ Comunicación indirecta: puertos o buzones

- Los mensajes se envían a puertos: las direcciones dst y org son identificadores de puertos.
- Un puerto tiene un identificador único en el sistema.
- El protocolo TCP/IP utiliza puertos “bien conocidos” que sirven para invocar servicios de red.
 - ftp: port 20 y 21
 - www: port 80
 - ntp: port 123

9. Comunicación por mensajes



◦ Comunicación indirecta: puertos



9. Comunicación por mensajes



◦ Comunicación indirecta: puertos y buzones

- **Puertos:** permiten que varios procesos envíen, pero sólo puede haber un proceso asociado para recibir.
- **Buzones:** permiten que varios procesos se asocien para recibir. El sistema operativo selecciona qué proceso recibe el mensaje.

9. Comunicación por mensajes



◦ Operaciones sobre puertos

¿Qué proceso puede recibir del puerto? Alternativas:

- **a) El propietario:** El proceso que lo crea es el propietario del puerto y es el único que puede recibir o destruir el puerto. Cuando el proceso propietario finaliza, se destruyen todos los puertos que posee. Son necesarias las siguientes operaciones:
 - **create(port):** crea un puerto.
 - **destroy(port):** destruye un puerto.
- **b) El que se vincula al puerto:** El puerto pertenece al S.O. y su existencia es independiente de los procesos que lo utilizan. Cuando un proceso se vincula a un puerto, es el único que puede recibir de él.
 - **bind(puerto):** vincula un proceso con un puerto.

9. Comunicación por mensajes



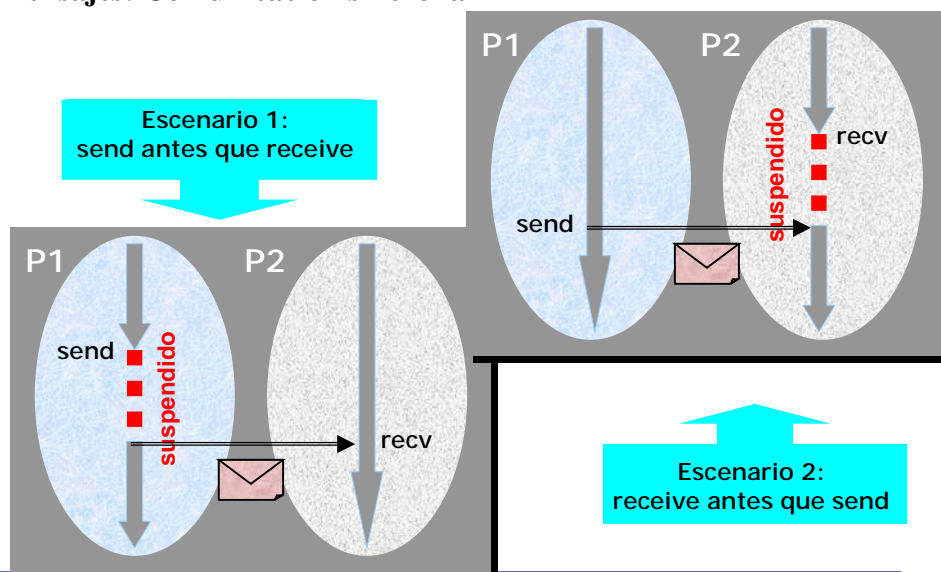
◉ Comunicación síncrona

- La capacidad del enlace es cero: El enlace no almacena mensajes; no pueden existir mensajes enviados y pendientes de entrega.
- Cita o rendez-vous: La comunicación sólo tiene lugar cuando emisor y receptor han invocado sus operaciones respectivas send y receive.
- Implementación de la cita o rendez-vous: El primer proceso que llega a la cita se suspende esperando a que el otro llegue.

9. Comunicación por mensajes



Mensajes: Comunicación síncrona



9. Comunicación por mensajes



◦ Comunicación asíncrona

La capacidad del enlace es N: **pueden haber mensajes enviados y pendientes de entrega**

• Variantes:

- **Capacidad limitada:** N está acotado.
- **Capacidad ilimitada:** N no está acotado.

• Sincronización:

- **Receive de enlace vacío:**
 - **Opción bloqueante:** suspender al proceso que la invoca.
 - **Opción no bloqueante :** devolver un error al proceso que la invoca.
- **Send a un enlace lleno:**
 - **Opción bloqueante:** suspender al proceso que la invoca.
 - **Opción no bloqueante :** devolver un error al proceso que la invoca o perder el mensaje.

9. Comunicación por mensajes



◦ El problema de los Productores y Consumidores con mensajes:

- Los espacios de direcciones son separados. No hay variables comunes.
- **Versión con:**
 - Comunicación directa.
 - Comunicación síncrona o asíncrona (send y receive bloqueantes)

```
task productor;  
var item : elemento;  
repeat  
    item := producir();  
    send( consumidor, item );  
until false;  
end productor;
```

```
task consumidor;  
var item : elemento;  
repeat  
    receive( productor, item );  
    consumir(item);  
until false;  
end consumidor;
```

9. Comunicación por mensajes



◦ El problema de los Productores y Consumidores con mensajes:

• Versión con:

- Comunicación directa.
- Comunicación asíncrona, con enlace de capacidad N (receive bloqueante, send no bloqueante)

```
task productor;  
var item : elemento;  
    m : mensaje;  
repeat  
    item := producir();  
    receive( consumidor, m );  
    construir_msj(m, item);  
    send( consumidor, m );  
until false;  
end productor;
```

```
task consumidor;  
var item : elemento;  
    m : mensaje;  
    i: Integer;  
for i:=1 to N do send( productor, m );  
repeat  
    receive( productor, m );  
    item := extraer_msj(m);  
    consumir(item);  
until false;  
end consumidor;
```

9. Comunicación por mensajes



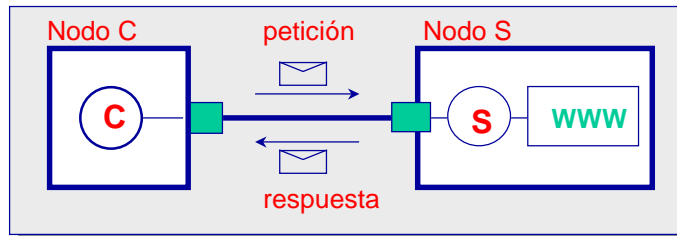
◦ Mensajes: el modelo cliente-servidor

- Modelo de comunicación especialmente adecuado para sistemas distribuidos, basado en asociar un proceso servidor al recurso que se desea compartir.
- Existen dos patrones de comportamiento diferenciados:
 - Servidores: Gestionan un recurso y ofrecen servicios relacionados con el recurso. Reciben peticiones de los clientes, las ejecutan en su nombre y responden a los clientes.
 - Clientes: Envían mensajes de petición a los servidores y esperan respuesta.

9. Comunicación por mensajes



◉ Mensajes: el modelo cliente-servidor



CLIENT

```
...  
send(serv, petición);  
recv(rem, respuesta);  
...
```

SERVIDOR

```
repeat  
  recv(serv, petición, ...);  
  respuesta= trabajar(petición);  
  rem=remite(petición);  
  send(rem, respuesta, ...);  
until false;
```