



## Tema 2

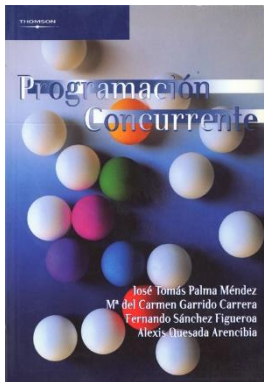
### Primeras soluciones a los problemas de sincronización




[Per90]   
Explicación  
muy sencilla  
de los intentos  
software de  
solución.




[Gal15]   
Explicación muy avanzada  
de soluciones hardware,  
spinlocks y el problema  
de las arquitecturas  
hardware modernas.



[Pal03]   
Prácticamente  
tal y como lo  
contamos en  
clase.



[Bur93]   
Bueno para  
intentos  
software

### **2.1 Solución a la condición de sincronización**

### **2.2 Soluciones a la exclusión mutua**

#### 2.2.1 Soluciones Software

Primeros intentos

Algoritmo de Peterson

Algoritmo de Dekker

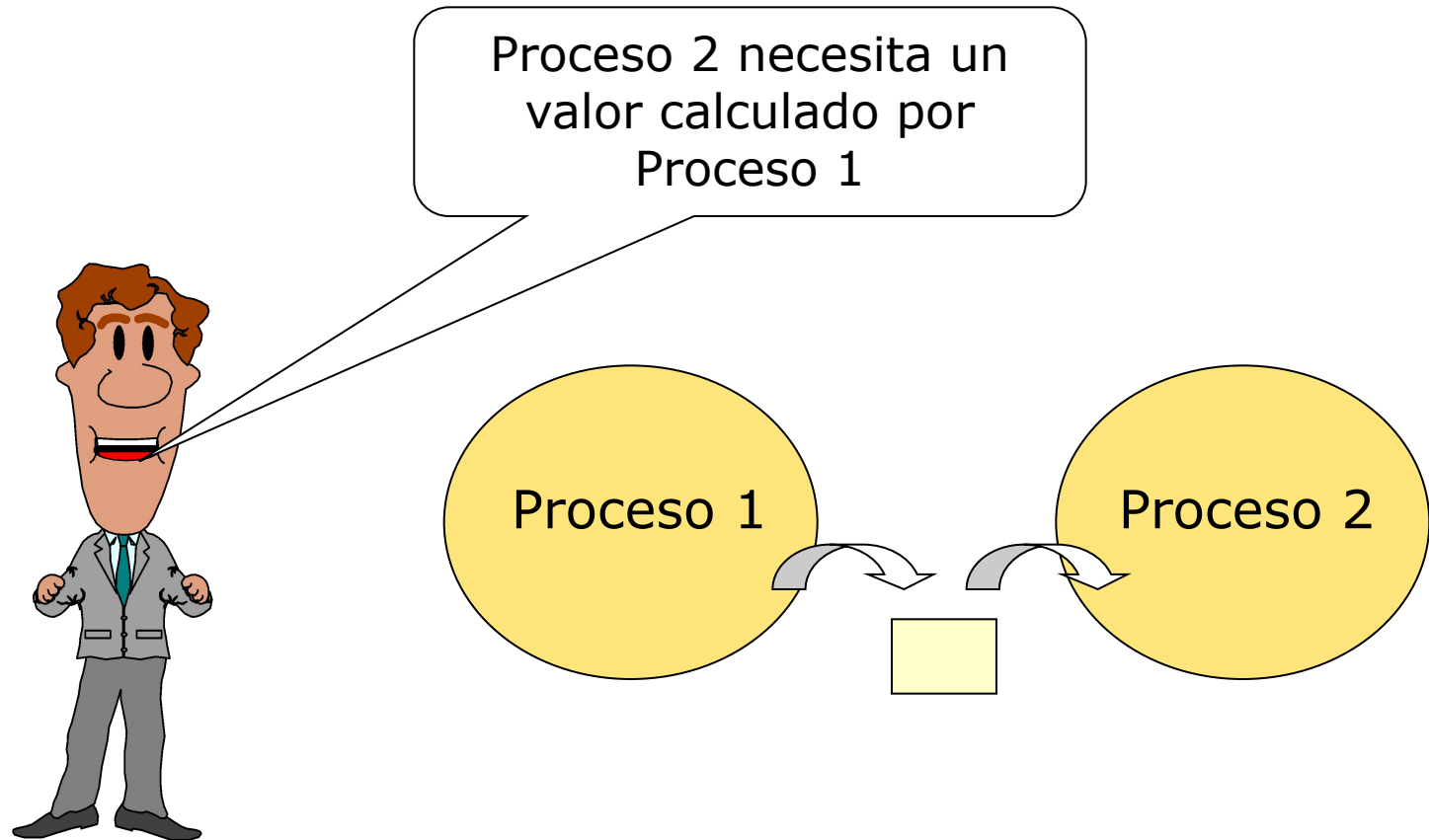
Solución para N procesos: Algoritmo de Lamport

La realidad del hardware moderno

#### 2.2.2 Soluciones Hardware

Instrucción TestAndSet

#### 2.2.3 Otras soluciones: deshabilitación de interrupciones



```
program  
condicionSincronizacion;
```

```
  int valor = 0;  
  flag = false;
```

```
process uno;  
begin  
  ...  
  valor = calcularValor;  
  flag = true;  
  ...  
end
```

```
process dos;  
int dato;  
begin  
  ...  
  while (! flag)  
    sleep(0);  
  dato = valor;  
  write (dato);  
  ...  
end;
```

```
begin  
  cobegin  
    uno;  
    dos;  
  coend  
end.
```

Es una notación  
para indicar que  
se ejecutan  
concurrentemente

### 2.1 Solución a la condición de sincronización

### 2.2 Soluciones a la exclusión mutua

#### 2.2.1 Soluciones Software

- Primeros intentos

- Algoritmo de Peterson

- Algoritmo de Dekker

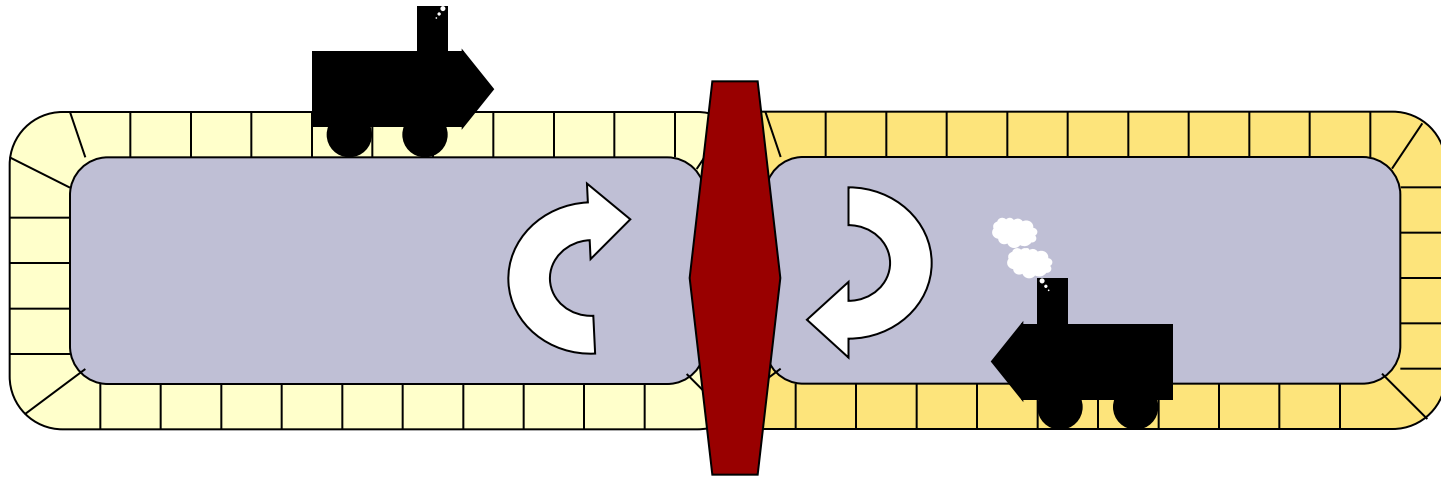
- Solución para N procesos: Algoritmo de Lamport

- La realidad del hardware moderno

#### 2.2.2 Soluciones Hardware

- Instrucción TestAndSet

#### 2.2.3 Otras soluciones: deshabilitación de interrupciones



### Estructura de la solución

#### process Tren1

repeat

sección no crítica

*protocolo de entrada*

\*\*\* sección crítica \*\*\*

*protocolo de salida*

sección no crítica

forever

#### process Tren2

repeat

sección no crítica

*protocolo de entrada*

\*\*\* sección crítica \*\*\*

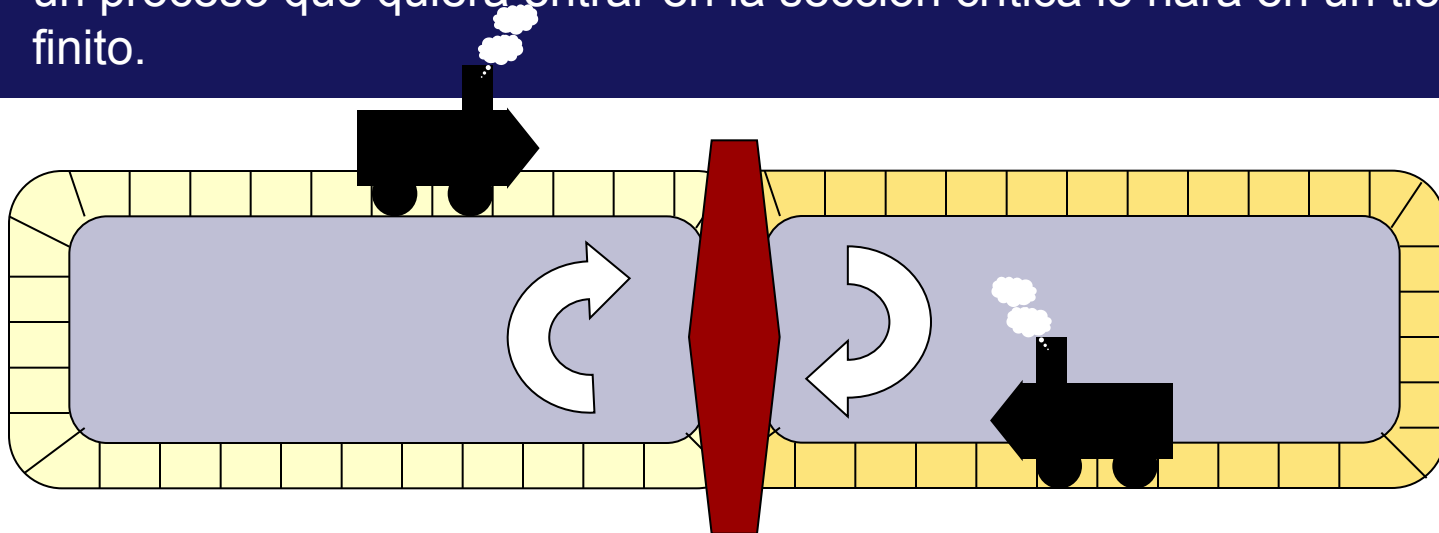
*protocolo de salida*

sección no crítica

forever

### Condiciones

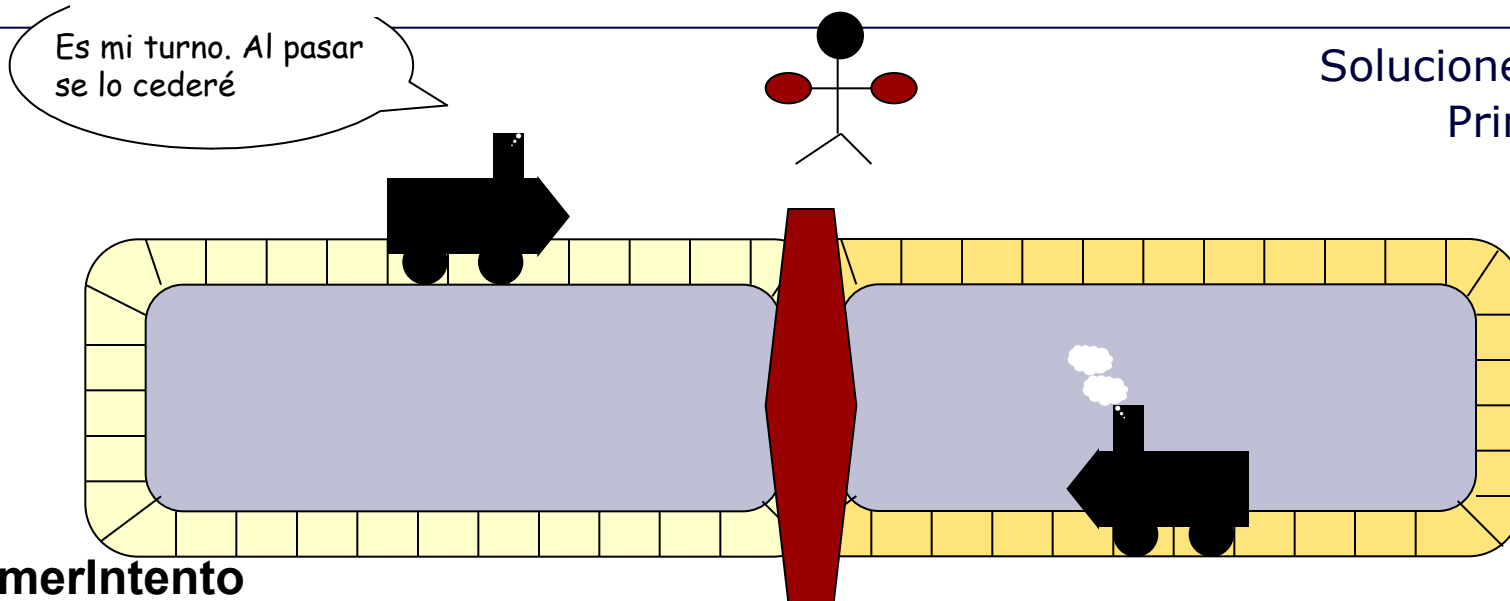
1. en **ausencia de pugna** por entrar en la sección crítica, un proceso que quiera entrar debe hacerlo.
2. No se debe producir una situación de **deadlock (interbloqueo pasivo)**.
3. No puede haber **postergación indefinida** de ningún proceso, es decir, un proceso que quiera entrar en la sección crítica lo hará en un tiempo finito.





## 2.3 Soluciones a la Exclusión Mutua

### Soluciones Software Primer intento

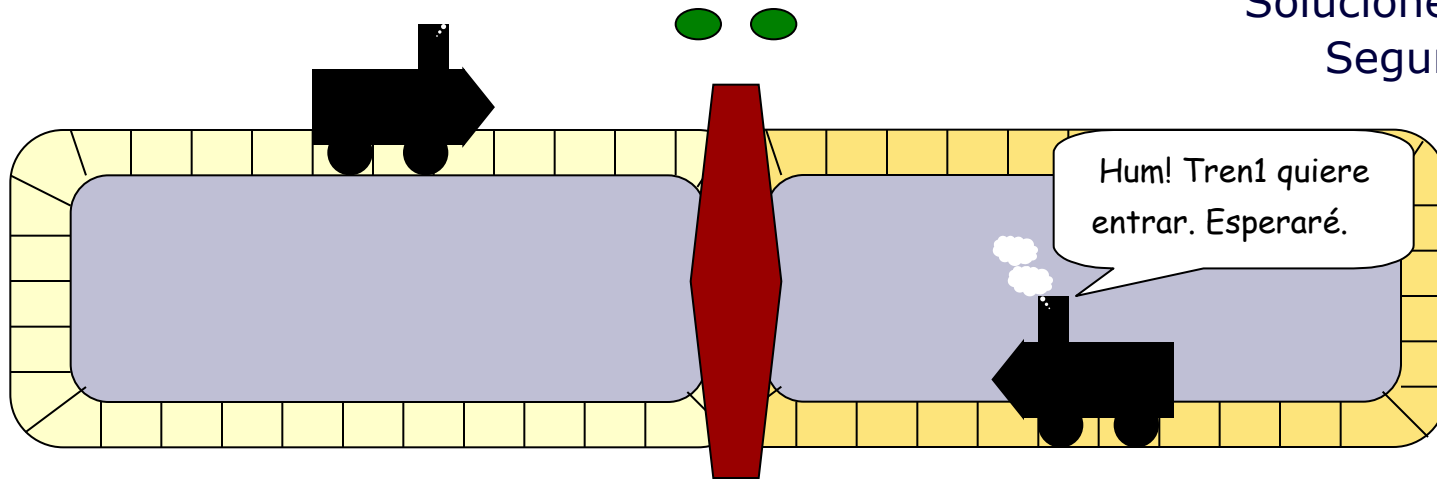


#### PrimerIntento

(\* turno es variable compartida inicialmente a 1 \*)

```
process Tren1;  
repeat  
  while (turno == 2) null; (*esperar*)  
  *** Sección Crítica ***  
  turno = 2;  
  Sección No Crítica  
forever
```

```
process Tren2;  
repeat  
  while (turno == 1) null; (* esperar *)  
  *** Sección Crítica ***  
  turno = 1;  
  Sección No Crítica  
forever
```



### SegundoIntento;

(\* Flag1 y flag2 son variables globales inicialmente a false \*)

#### process Tren1;

```
repeat
  while (flag2) null; (* esperar *)
  flag1 = true; (* quiero entrar *)
  *** Sección Crítica ***
  flag1 = false; (* salida del protocolo *)
  Sección No Crítica
forever
```

#### process Tren2;

```
repeat
  while (flag1) null; (* esperar *)
  flag2 = true; (* quiero entrar *)
  *** Sección Crítica ***
  flag2 = false; (* salida del protocolo *)
  Sección No Crítica
forever
```

```
process Tren1;  
repeat  
  flag1 = true;  
  while (flag2) null;  
  *** Sección Crítica ***  
  flag1 = false;  
  Sección No Crítica  
forever
```

```
process Tren2;  
repeat  
  flag2 = true;  
  while (flag1) null;  
  *** Sección Crítica ***  
  flag2 = false;  
  Sección No Crítica  
forever
```

```
process Tren1;
repeat
  flag1 = true;
  while (flag2) {
    (*trato de cortesía*)
    flag1 = false;
    flag1 = true;
  }
  *** Sección crítica ***
  flag1 = false;
  Sección No Crítica
forever
```

```
process Tren2;
repeat
  flag2 = true;
  while (flag1) {
    (* trato de cortesía *)
    flag2 = false;
    flag2 = true;
  }
  *** Sección crítica ***
  flag2 = false;
  Sección No Crítica
forever
```

```
process Tren1;  
repeat  
  flag1 = true;  
  turno = 2;  
  while (flag2 && (turno == 2))  
    null;  
  *** Sección Crítica ***  
  flag1 = false;  
  Sección No Crítica  
forever
```

```
process Tren2;  
repeat  
  flag2 = true;  
  turno = 1;  
  while (flag1 && (turno == 1))  
    null;  
  *** Sección Crítica ***  
  flag2 = false;  
  Sección No Crítica  
forever
```

```
process Tren1;  
repeat  
  flag1 = true;  
  while (flag2) {  
    flag1 = false;  
    while (turno == 2) null;  
    flag1 = true;  
  }  
  *** Sección Crítica ***  
  turno = 2;  
  flag1 = false;  
  Sección No Crítica  
forever
```

```
process Tren2;  
repeat  
  flag2 = true;  
  while (flag1) {  
    flag2 = false;  
    while (turno == 1) null;  
    flag2 = true;  
  }  
  *** Sección Crítica ***  
  turno = 1;  
  flag2 = false;  
  Sección No Crítica  
forever
```

[Pal03] explicación detallada de los algoritmos. Alternativamente [Bur93] o [Per90]



Soluciones Software  
La realidad del hardware moderno

La dura realidad es que si ejecutamos el algoritmo de Peterson o el de Dekker en un ordenador moderno..... ¡¡ Tampoco funcionará !!



Pero por ejemplo en una Raspberry Pi con procesador ARM6, ¡¡Sí!!

Los procesadores modernos no garantizan por defecto que los procesos ejecuten las instrucciones en el mismo orden que aparecen en el programa. No se asegura consistencia secuencial de acceso a memoria.



Y sólo vamos por el tema 2....

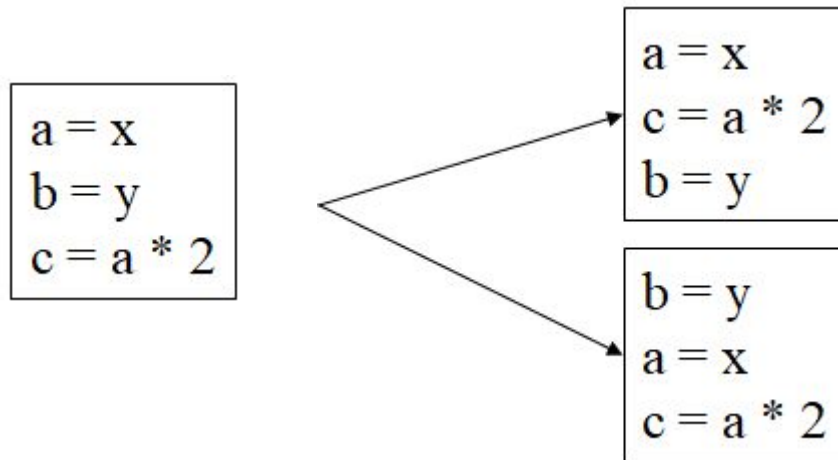
Las 3 razones por las que no se asegura:

- Optimizaciones del compilador
- Incoherencia de caché
- Ejecución fuera de orden



[Gal2015], pág. 47.  
Detalla los 3 problemas

## Ejecución fuera de orden



```
flag1 = true;  
turno = 2;  
while (flag2 && (turno == 2))  
    null;
```



```
turno = 2;  
while (flag2 && (turno == 2))  
    null;  
flag1 = true;
```

El procesador no detecta que las variable se modifican por diversos threads.  
No detecta la dependencia causal => Error !!!!



repeat

núm[i] = max(núm[0], núm[1],..., núm[n-1]) + 1;

for (int j=0; j++; j<n {

while núm[j] ≠ 0 && (núm[j]<núm[i] or núm[j]==núm[i] and j<i) null;  
}

sección crítica;

núm[i] =0;

resto proceso

forever

**Problema 1:** Dos procesos pueden coger el mismo número. No exclusión mutua

P(1) ejecuta el for y entra. No hay nadie menor que él.  
P(0) también entra por la misma razón.

**Problema 2:** Mientras uno elige, otro recorre el array. Sigue sin asegurarse exclusión mutua

P(1) está ejecutando el for cuando P(0) todavía no ha almacenado su valor en núm[0]. Entrará P(1) y luego P(0)

```
repeat
```

```
  elección[i] = verdadero;
```

```
  núm[i] = max(núm[0], núm[1],..., núm[n-1]) + 1;
```

```
  elección[i] = falso;
```

```
  for (int j=0; j++; j<n {
```

```
    while (elección[j]) null;
```

```
    while núm[j] ≠ 0 && (núm[j]<núm[i] or núm[j]==núm[i] and j<i) null
```

```
  }
```

```
  sección crítica;
```

```
  núm[i] =0;
```

```
  resto proceso
```

```
  forever
```

Elección



Num



Sale de la espera ocupada, pero vuelve a ella

Entra en la sección crítica  
Sale de la sección crítica y entra en la SC

### 2.1 Solución a la condición de sincronización

### 2.2 Soluciones a la exclusión mutua

#### 2.2.1 Soluciones Software

Primeros intentos

Algoritmo de Peterson

Algoritmo de Dekker

Solución para N procesos: Algoritmo de Lamport

La realidad del hardware moderno

#### 2.2.2 Soluciones Hardware

Instrucción TestAndSet

#### 2.2.3 Otras soluciones: deshabilitación de interrupciones

Procesadores que ofrecen instrucciones especiales que se ejecutan de forma atómica

Ejemplos:

exchange

get&Set

get&Add

test&Set

testAndSet



[Gal2015] pág. 57, a muy bajo nivel, pensado para programadores de SO, librerías cercanas al SO, programación en ensamblador, etc

#### Instrucción testAndSet (a, b)

- almacena el valor de b en a
- Pone b a verdadero

```
process procesoUno;
boolean unoNoPuedeEntrar;
{
  while (true) {
    unoNoPuedeEntrar = true;
    while (unoNoPuedeEntrar) do
      testandset (unoNoPuedeEntrar,activo);
    (** Sección crítica uno **)
    activo = false;
    otrasCosasUno;
  }
}
```

```
process procesoDos;
boolean dosNoPuedeEntrar;
{
  while (true) {
    dosNoPuedeEntrar = true;
    while (dosNoPuedeEntrar) do
      testandset (dosNoPuedeEntrar,activo);
    (** Sección crítica dos **)
    activo = false;
    otrasCosasDos;
  }
}
```

(\* activo es true cuando el otro proceso está dentro de S.C. \*)

```
begin
  activo = false;
  cobegin
    procesoUno;
    procesoDos
  coend;
end.
```

### 2.1 Solución a la condición de sincronización

### 2.2 Soluciones a la exclusión mutua

#### 2.2.1 Soluciones Software

Primeros intentos

Algoritmo de Peterson

Algoritmo de Dekker

Solución para N procesos: Algoritmo de Lamport

#### 2.2.2 Soluciones Hardware

Instrucción TestAndSet

#### 2.2.3 Otras soluciones: deshabilitación de interrupciones

#### **Process uno;**

...

Deshabilitar interrupciones

Sección crítica

Habilitar interrupciones

...

#### **Process dos;**

...

Deshabilitar interrupciones

Sección crítica

Habilitar interrupciones

...



## Conclusiones:

- Ninguna de las **construcciones software** que conocemos solucionan adecuadamente el problema de la **exclusión mutua**. Ni tan siquiera el de la **condición de sincronización**.
  - Este tipo de algoritmos que contienen **espera ocupada** ó **espera activa** suelen recibir el nombre de **spinlocks**.
  - Un tratamiento adecuado de spinlocks viene en [Gal2015], pág. 77, aunque a un nivel de implementación muy bajo.
- Tampoco las instrucciones atómicas a nivel **hardware** ayudan demasiado.
- Se hace necesaria la aparición de nuevas primitivas en los lenguajes de programación que solucionen estos problemas: son las **primitivas de sincronización**. A esto dedicaremos el tema 3.



## Tema 2

### Primeras soluciones a los problemas de sincronización

