Sistemas Operativos

Introducción a la concurrencia

Curso 2025

Facultad de Ingeniería, UDELAR

Agenda

1. Introducción
2. Grafos de precedencia
3. Cobegin - Coend
4. Fork - Join
5. Sección Crítica
6. Algoritmo de Dekker

    Entrelazado
7. Algoritmo de Peterson
8. Sección crítica por hardware

Introducción

Procesos cooperativos

- Se llaman procesos cooperativos a aquellos que pueden afectar el estado de otros procesos o cuyo estado es afectado por otros procesos.
- Esto puede ocurrir al compartir un espacio de memoria (ej. hilos) o mediante primitivas de comunicación entre procesos.
- El acceso concurrente a los mismos datos puede generar inconsistencias si no se tiene cuidado.
    - Recordar que por ejemplo un planificador expropiativo puede quitarle la CPU a un proceso en cualquier momento.
- Las técnicas de programación concurrente permiten resolver estos problemas en forma segura y eficiente.

Ejemplo

Variable A es compartida

```
Begin
    A := 1;
    Print (A);
End
```

```
Begin
    A := 2;
    Print (A);
End
```

Resultados posibles:

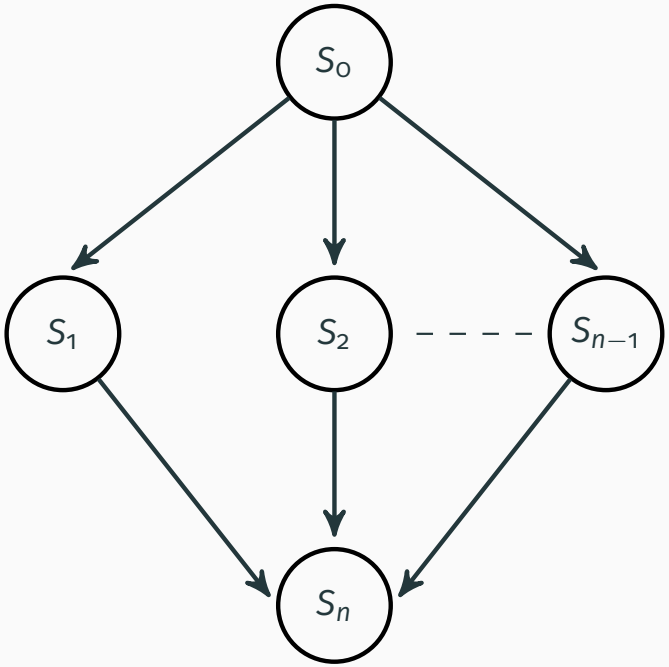
- 1, 2
- 1, 1
- 2, 1
- 2, 2

Grafos de precedencia

Grafos de precedencia

- Una posible solución a este problema es definir que tareas deben esperar por otras y cuales pueden ejecutar en paralelo.
- Esto puede aplicar para procesos, hilos de un mismo proceso o distintas secciones de código de cada proceso.
- Un grafo de precedencia es un grafo acíclico y dirigido cuyos nodos son tareas y cuyas aristas indican la precedencia.
- Un grafo de precedencia permite especificar el orden que que se deben ejecutar los procesos

Grafo de precedencia



- S_0 no depende de nadie
- $S_1, S_2, \ldots, S_{n-1}$ dependen de S_0, no hay restricciones entre ellos
- S_n depende de $S_1, S_2, \ldots, S_{n-1}$

Cobegin - Coend

Cobegin - Coend

- Una herramienta para declarar procesos concurrentes es el uso de Cobegin - Coend.
- Permite definir (algunos) grafos de precedencia declarativmente.
- Todas las sentencias dentro del bloque Cobegin - Coend se ejecutan concurrentemente
- Por ejemplo, para representar el grafo anterior:

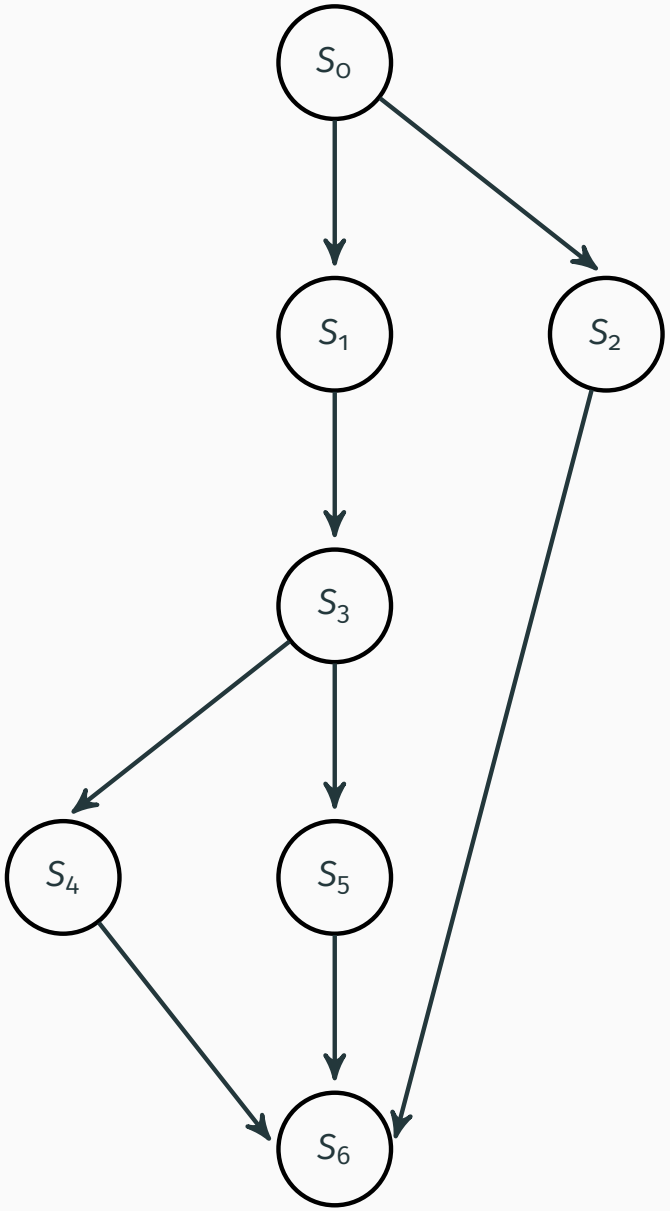
```
Begin
    S0;
    Cobegin
        S1;
        S2;
        ...
        Sn−1;
    Coend
    Sn;
End
```

Ejemplos

```
Begin
   Cobegin
      A := SemiFactorial(n, floor(n/2));
      B := SemiFactorial(floor(n/2)−1, 1);
   Coend
   Return A * B;
End
```

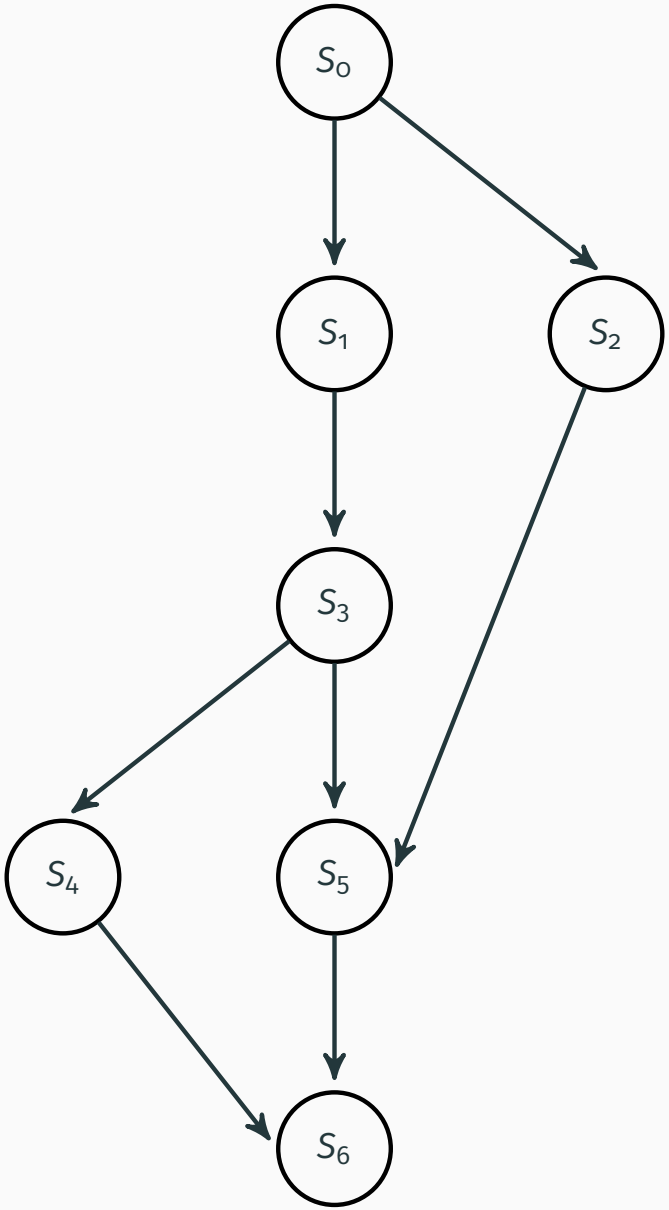
```
Begin
   A := 10
   Cobegin
      Print (A);                                  ▷ A = ?
      A := 100;
   Coend
   Print (A);                                  ▷ A = 100
End
```

Ejemplo de grafo



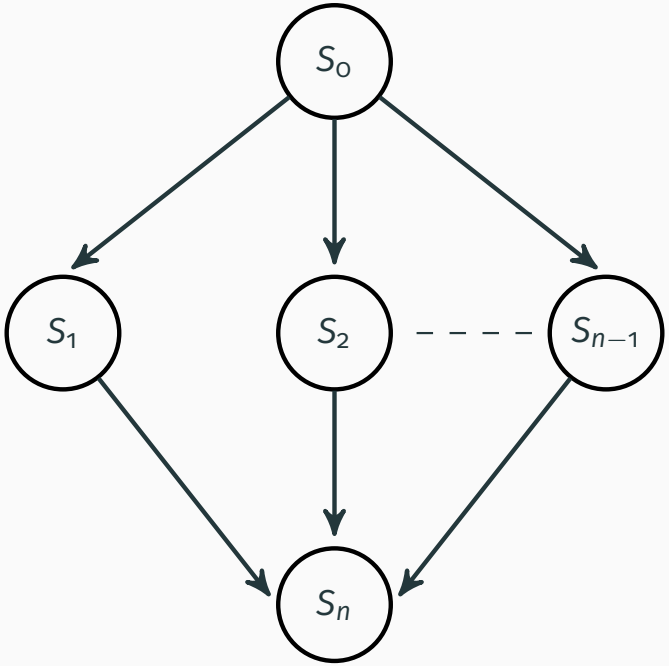

```
Begin
    S0 ;
    Cobegin
        Begin
            S1 ;
            S3 ;
            Cobegin
                S4 ;
                S5 ;
            Coend
        End
        S2 ;
    Coend
    S6 ;
End
```

Ejemplo no representable



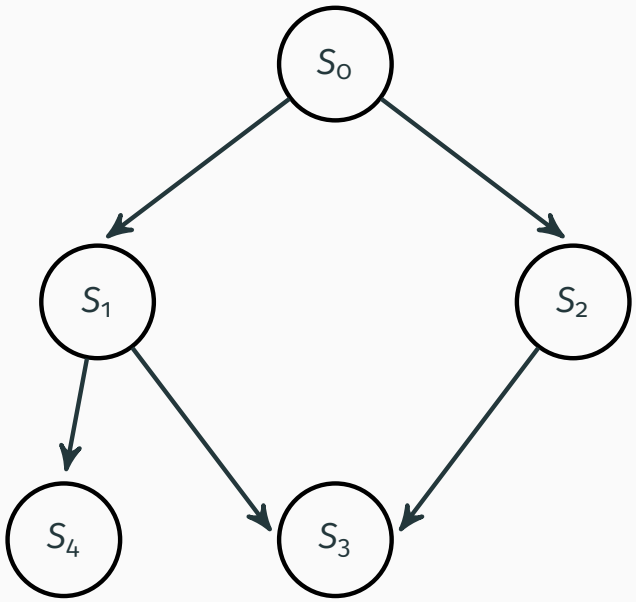
Grafos representables

- Los grafos representables con cobegin-coend tienen que tener el siguiente formato.



Grafos representables

- No se pueden representar grafos con la siguiente estructura general.

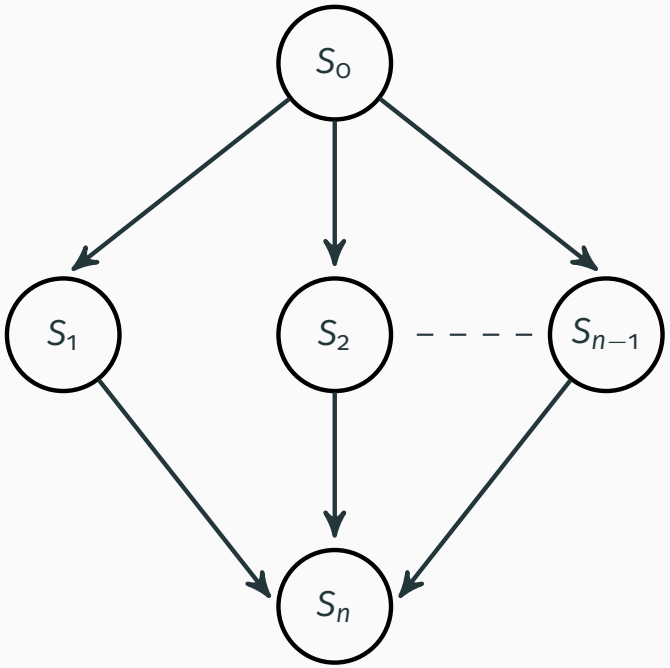


Fork - Join

Fork - Join

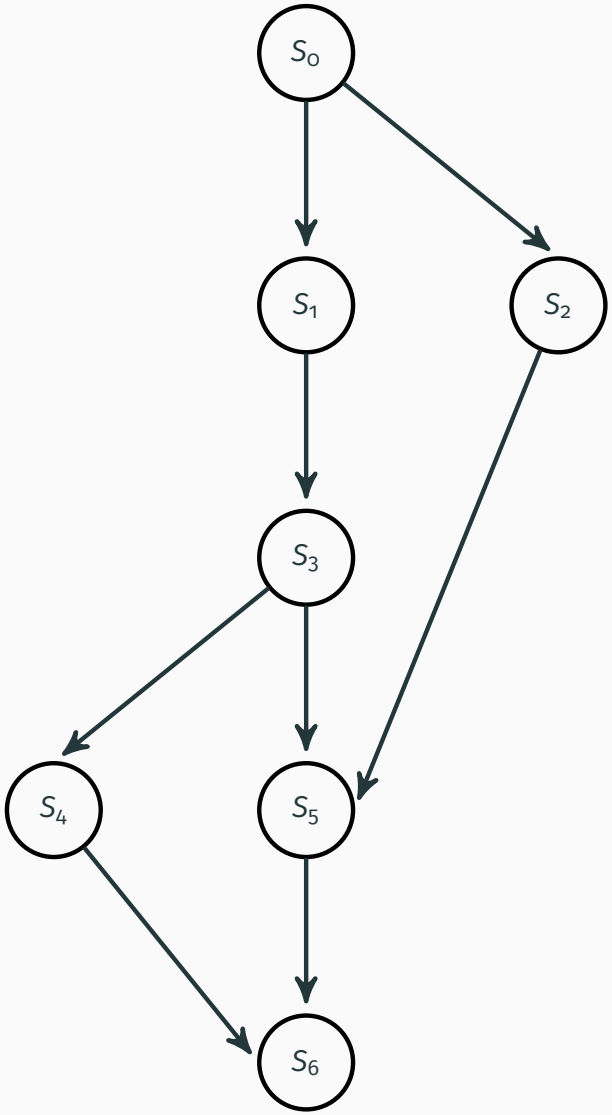
- Se necesitan más herramientas para poder programar cualquier grafo de dependencias.
- No es lo mismo que el fork-wait de unix aunque es la misma idea. Es una abstracción.
- **Fork** <**etiqueta**> Divide en dos el proceso. Uno de ellos continúa ejecutando luego del Fork y el otro salta a la etiqueta.
- **Join** <**contador**> <**etiqueta**> Cada vez que un proceso ejecuta Join, el contador (que es una variable global) se decrementa uno. Si el contador queda en 0 el proceso salta a la etiqueta, en otro caso termina.
- **Goto** <**etiqueta**> El proceso salta a la etiqueta.
- **Quit** El proceso termina.

Ejemplo




```
Begin
   S0;
   total := N − 1;
   Fork L2;
   Fork L3;
   ...
   Fork LN−1;
   S1;
   Join total FIN;
L2:
   S2;
   Join total FIN;
   ...
LN−1:
   SN−1;
   Join total FIN;
FIN:
   SN;
End
```

Ejemplo




```
Begin
   total1 := 2;
   total2 := 2;
   S0;
   Fork L2;
   S1;
   S3;
   Fork L4;
   Join total1 NEXT;
L2:
   S2;
   Join total1 NEXT;
NEXT:
   S5;
   Goto L5;
L4:
   S4;
L5:
   Join total2 FIN;
FIN:
   S6;
End
```

Sección Crítica

Problema de la sección crítica

- Cuando se tienen varios procesos cooperativos cada uno de ellos tiene una sección crítica donde se modifican datos comunes a todos los procesos.
- Para garantizar que los procesos cooperan correctamente, cuando un proceso está ejecutando su sección crítica ningún otro proceso puede estar ejecutando código de su sección crítica.
- La sección critica no tiene por que tener el mismo código para todos los procesos, solamente tienen en común que es donde se accede a los datos compartidos.
- Un proceso puede tener más de una sección crítica si accede a varios datos compartidos diferentes.

Estructura de un proceso

- La estructura general de un proceso que usa una región crítica es:

```
loop
   Ingreso                    ▷ Sección de ingreso
        Sección crítica;
   Egreso                     ▷ Sección de egreso
   Otras tareas;
end loop
```

Requerimientos I

- Una solución al problema de la sección crítica debe satisfacer los siguientes requerimientos:

Mutua exclusión

Solo un proceso ejecuta la región critica en un momento dado.

Progreso

Si uno o más procesos quieren acceder a la sección crítica y esta se libera la misma debe ser asignada a alguno de los procesos que están esperando y esta decisión no se debe dilatar indefinidamente (deadlock)

Requerimientos II

Espera acotada

Debe haber una cota máxima para la cantidad de procesos que acceden a la región crítica una vez que un proceso lo ha solicitado y antes de que se lo deje entrar (evitar posposición indefinida)

Algoritmo de Dekker

Introducción

- Implementa una posible solución al problema de la sección crítica.
- Es independiente del sistema operativo (no usa system calls) por lo que usa busy waiting para esperar.
- No se usa en la práctica pero es útil entenderlo para ver problemas típicos de algoritmos concurrentes.
- Imaginemos que hay dos vecinos Alicia (A) y Bernardo (B) que comparten un jardín. Ambos tienen perro pero no pueden sacarlos juntos al jardín porque se pelean.
- El acceso al jardín es la sección crítica.

Primera solución

```
procedure Alicia
   loop
      while Turno = 2 do;
         ▷ Espera Turno = 1
      Pasear perro;
      Turno := 2;
      Otras tareas;
   end loop
end procedure
```

```
procedure Bernardo
   loop
      while Turno = 1 do;
         ▷ Espera Turno = 2
      Pasear perro;
      Turno := 1;
      Otras tareas;
   end loop
end procedure
```

```
Begin
   Turno := 1;
   Cobegin
      Alicia;
      Bernardo;
   Coend
End
```

Segunda solución

```
procedure Alicia
   loop
      while flag_B do;
      flag_A := True;
      Pasear perro;
      flag_A := False;
      Otras tareas;
   end loop
end procedure
```

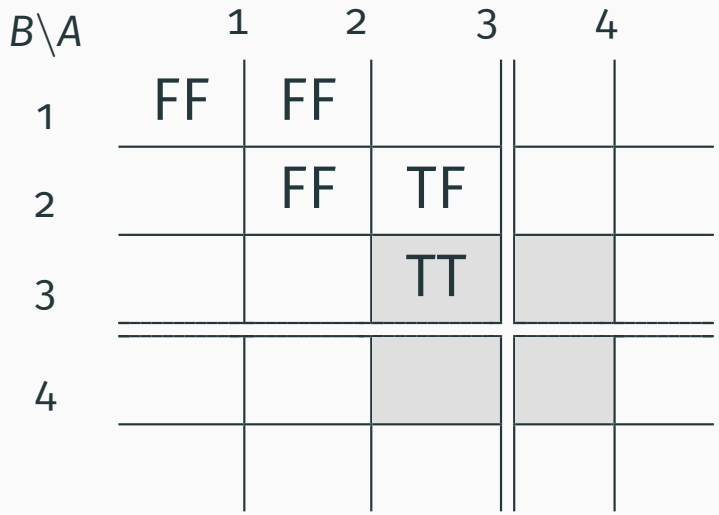
```
procedure Bernardo
   loop
      while flag_A do;
      flag_B := True;
      Pasear perro;
      flag_B := False;
      Otras tareas;
   end loop
end procedure
```

```
Begin
   flag_A := False;
   flag_B := False;
   Cobegin
      Alicia;
      Bernardo;
   Coend
End
```

Entrelazado

- Es un método gráfico para analizar el comportamiento de algoritmos concurrentes.
- Permite graficar todas las combinaciones posibles de ejecución de las instrucciones y ver si alguna entra en la región crítica.
- Si se encuentra un camino que entre en la región crítica el algoritmo está mal.
- Para mostrar que es correcto hay que graficar todas las ejecuciones posibles.
- En general se representa la región crítica como una instrucción aunque sean muchas

Ejemplo de entrelazado




```
1: while flag_B do;
2: flag_A := True;
3: Pasear perro;
4: flag_A := False;
   Otras tareas;
```

```
1: while flag_A do;
2: flag_B := True;
3: Pasear perro;
4: flag_B := False;
   Otras tareas;
```

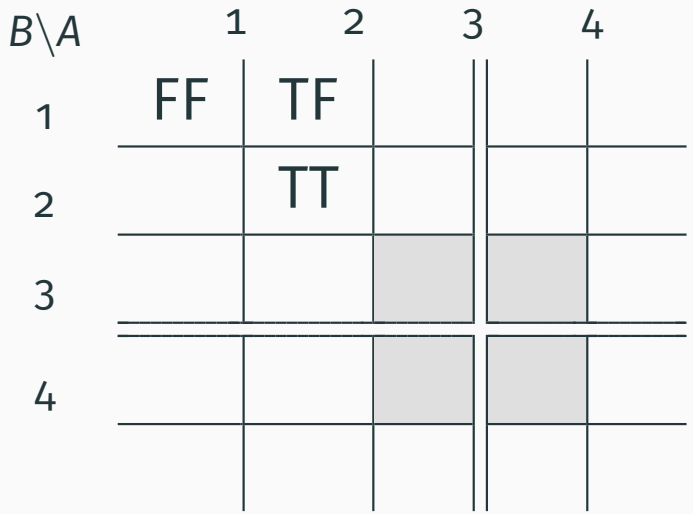
Segunda solución (otra vez)

```
procedure Alicia
   loop
      flag_A := True;
      while flag_B do;
      Pasear perro;
      flag_A := False;
      Otras tareas;
   end loop
end procedure
```

```
procedure Bernardo
   loop
      flag_B := True;
      while flag_A do;
      Pasear perro;
      flag_B := False;
      Otras tareas;
   end loop
end procedure
```

```
Begin
   flag_A := False;
   flag_B := False;
   Cobegin
      Alicia;
      Bernardo;
   Coend
End
```

Entrelazado




```
1: flag_A := True;
2: while flag_B do;
3: Pasear perro;
4: flag_A := False;
   Otras tareas;
```

```
1: flag_B := True;
2: while flag_A do;
3: Pasear perro;
4: flag_B := False;
   Otras tareas;
```

Tercera solución

```
procedure Alicia
  loop
    flag_A := True;
    while flag_B do
      if turno = 2 then
        flag_A := False;
        while turno = 2 do;
        flag_A := True;
      end if
    end while
    Pasear perro;
    turno := 2;
    flag_A := False;
    Otras tareas;
  end loop
end procedure
```

```
procedure Bernardo
  loop
    flag_B := True;
    while flag_A do
      if turno = 1 then
        flag_B := False;
        while turno = 1 do;
        flag_B := True;
      end if
    end while
    Pasear perro;
    turno := 1;
    flag_B := False;
    Otras tareas;
  end loop
end procedure
```

```
Begin
  turno := 1;
  flag_A := False;
  flag_B := False;
  Cobegin
    Alicia;
    Bernardo;
  Coend
End
```

Algoritmo de Peterson

Introducción

- Resuelve el mismo problema pero es más sencillo
- Es más fácil de generalizar para N procesos

Algoritmo de Peterson

```
procedure Alicia
   loop
      flag_A := True;
      turno := 1;
      while flag_B AND turno = 1 do;
      Pasear perro;
      flag_A := False;
      Otras tareas;
   end loop
end procedure
```

```
procedure Bernardo
   loop
      flag_B := True;
      turno := 2;
      while flag_A AND turno = 2 do;
      Pasear perro;
      flag_B := False;
      Otras tareas;
   end loop
end procedure
```

```
Begin
   turno := 1;
   flag_A := False;
   flag_B := False;
   Cobegin
      Alicia;
      Bernardo;
   Coend
End
```

Entrelazado

$B \backslash A$	1	2	3	4	5	
1	$\frac{1}{2}$FF	$\frac{1}{2}$TF	1TF	1TF	1TF	1FF
2	$\frac{1}{2}$FT	$\frac{1}{2}$TT	1TT	1TT	1TT	1FT
3	2FT	2TT	$\frac{1}{2}$TT	2TT	2TT	2FT
4	2FT	2TT	1TT			2FT
5	2FT	2TT	1TT			2FT
	2FF	2TF	1TF	1TF	1TF	$\frac{1}{2}$FF

```
1: flag_A := True;
2: turno := 1;
3: while flag_B AND turno = 1 do;
4: Pasear perro;
5: flag_A := False;
   Otras tareas;
```

```
1: flag_B := True;
2: turno := 2;
3: while flag_A AND turno = 2 do;
4: Pasear perro;
5: flag_B := False;
   Otras tareas;
```

Sección crítica por hardware

Sincronización por hardware

- Las soluciones eficientes al problema de la región crítica requieren asistencia de hardware y por lo tanto también del sistema operativo.
- La abstracción fundamental es el lock que protege las regiones críticas.
- Los procesos deben adquirir un lock antes de entrar a la región crítica y liberarlo al salir. A veces se les llama mutex.
- A lo largo del curso veremos varias formas de implementar esta primitiva básica.

Estructura de un proceso

- La estructura general de un proceso que usa una locks es:

```
loop
   Get                                  ▷ Obtengo lock
        Sección crítica;
   Release                               ▷ Libero lock
   Otras tareas;
end loop
```

Sistema monoprocesador

- En un sistema monoprocesador para asegurar el acceso con mutua exclusión a una región crítica alcanza con deshabilitar las interrupciones
- Pero si la región crítica dura mucho tiempo se pueden perder interrupciones

```
loop
   CLI                                  ▷ Obtengo lock
        Sección crítica;
   STI                                   ▷ Libero lock
   Otras tareas;
end loop
```

Sistema multiprocesador

- En un multiprocesador no es suficiente con deshabilitar las interrupciones en todos los procesadores
- Para no tener que hacer esto el hardware proporciona instrucciones de sincronización que se pueden usar para implementar locks.

Test and set

- Permite chequear el contenido de una variable global y modificarlo en forma atómica (como si fuera una región crítica)

```
function TestAndSet(var)
   ret := var;
   var := True;
   return ret;
end function
```

Región crítica con test and set

- La solución a la región crítica es más sencilla que Dekker o Peterson pero igual requiere busy waiting.

```
lock := False;  ▷ inicialización (una vez)

repeat
   while TestAndSet(lock) do;
   Región crítica;
   lock := False;
   Otras tareas;
until False
```

Swap

- Intercambia el valor de dos variables en forma atómica.

```
procedure Swap(a, b)
    tmp := a;
    a := b;
    b := tmp;
end procedure
```

Región crítica con swap

- La solución a la región crítica con swap también requiere busy waiting.

```
lock := False;  ▷ inicialización (una vez)

repeat
   key := True;                ▷ variable local
   while key do
      Swap(lock, key);
   Región crítica;
   lock := False;
   Otras tareas;
until False
```