Sistemas Operativos

Sección crítica

Curso 2024

Facultad de Ingeniería, UDELAR

Agenda

1. Introducción

2. Algoritmo de Dekker

    Entrelazado

3. Algoritmo de Peterson

4. Sección crítica por hardware

Introducción

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
```

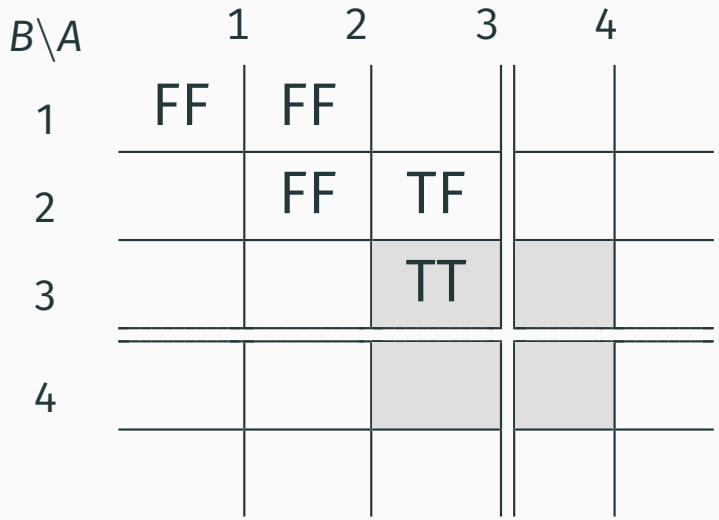
```
procedure Bernardo
   loop
      while flag_A do;
      flag_B := True;
      Pasear perro;
      flag_B := False;
      Otras tareas;
   end loop
end procedure
```

```
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;
```

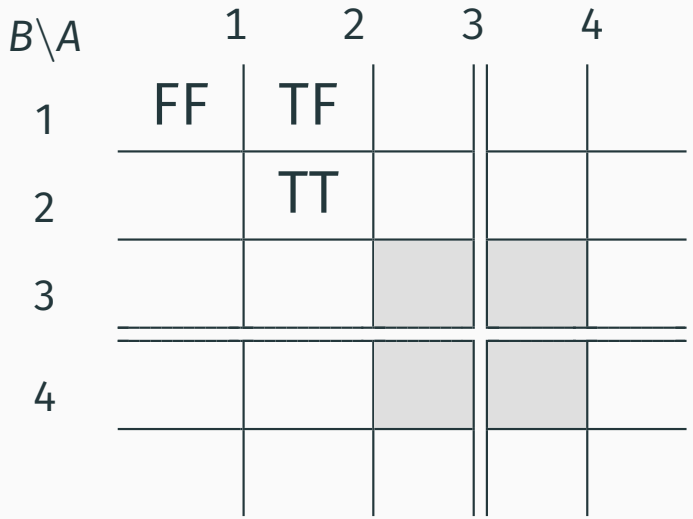
Segunda solución (otra vez)

```
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
```