Clase 3

jueves, 29 de agosto de 2024 13:11

```
process SC[i=1 to n]
  { while (true)
      {  protocolo de entrada;   ⇔  <
         sección crítica;        ⇔  SC
         protocolo de salida;    ⇔  >
         sección no crítica;
      }
  }
```

Hay que implementar algo que no permita que 2 procesos estén en la parte crítica a la vez.
Con awaits, lo hago con <sección crítica> y punto.

Propiedades que debe cumplir:

- Exclusión mutua: a lo sumo un proceso está en la sección crítica a la vez
- Ausencia de deadlock: no se van a bloquear, al menos uno va a estar ejecutandose.
- Ausencia de demora innecesaria: si hay un proceso que quiere usar sección crítica y el resto está no ejectando a sección crítica o ya terminó, se puede ejecutar. Esto no se cumple si por ejemplo tengo que respetar un orden(ej le toca al 4 pero no está listo y el 5 si pero no lo puede usar porque debe respetar orden)
- Eventual entrada: en algún momento debe poder entrar. No se puede probar que esto no va a pasar a menos que haya fairness. Round robin en algunos casos no nos asegura que se cumpla

- Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional ⟨ S;⟩ ⇒ **SCEnter ; S; SCExit**

- Para una acción atómica condicional ⟨await (B) S;⟩ ⇒
  SCEnter ; while (not B) {SCExit; SCEnter;} S; SCExit;

- Si ***S*** es *skip*, y ***B*** cumple *ASV*, ⟨await (B);⟩ puede implementarse por medio de ⇒ **while (not B) skip;**

Es correcto pero no eficiente porque un proceso está entrando y saliendo de la sc hasta que uno cambie b
Con un delay en el medio a mí me sacan del procesador lo que permite que otro proceso entre y ejecute.

- Para reducir *contención de memoria* ⇒
  SCEnter ; while (not B) {SCExit; Delay; SCEnter;} S; SCExit;

Solución x hardware: no permitir interrupciones
Es ineficiente

```
process SC[i=1 to n] {
   while (true) {
        deshabilitar interrupciones;      # protocolo de entrada
        sección crítica;
        habilitar interrupciones;         # protocolo de salida
        sección no crítica;
   }
}
```

Con una máquina de un solo procesador (en multip no anda)

```
process SC1
 { while (true)
     {⟨await (not in2) in1 = true;⟩
       sección crítica;
       in1 = false;
       sección no crítica;
     }
 }
```

```
process SC2
 { while (true)
     { ⟨await (not in1) in2 = true;⟩
       sección crítica;
       in2 = false;
       sección no crítica;
     }
 }
```

Con mutex digo lo que tiene que passar, en este caso no pueden ser in1 y in2 a la vez

Ausencia de deadlock: si hay deadlock, SC1 y SC2 están bloqueados en su protocolo de entrada ⇒ **in1** e **in2** serían *true* a la vez. Esto NO puede darse ya que ambas son falsas en ese punto (lo son inicialmente, y al salir de SC, cada proceso vuelve a serlo).

Ausencia de demora innecesaria: si SC1 está fuera de su SC o terminó, **in1** es *false*; si SC2 está tratando de entrar a SC y no puede, **in1** es *true*; (¬ **in1** ∧ **in1** = *false*) ⇒ ***no hay demora innecesaria***.

Eventual Entrada:

- Si SC1 está tratando de entrar a su SC y no puede, SC2 está en SC (**in2** es *true*). Un proceso que está en SC eventualmente sale → **in2** será *false* y la guarda de SC1 *true*.
- Análogamente para SC2.
- Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito, las guardas son *true* con infinita frecuencia.

Se garantiza la eventual entrada con una política de scheduling fuertemente fair.

Para hacerlo con n procesos:

```
process SC [i=1..n]
 { while (true)
     {⟨await (not lock) lock= true;⟩
       sección crítica;
       lock = false;
       sección no crítica;
     }
 }
```

Test and set:

Hace lo que un await

Ok termina siempre en true, devuelve lo que tenia antes el inicial

```
bool TS (bool ok);
{ < bool inicial = ok;
    ok = true;
    return inicial; >
}
```

```
bool lock = false;
process SC [i=1..n]
 { while (true)
     {⟨await (not lock) lock= true;⟩
       sección crítica;
       lock = false;
       sección no crítica;
     }
 }
```

```
bool lock=false;
process SC[i=1 to n]
{ while (true)
    { while (TS(lock)) skip ;
      sección crítica;
      lock = false;
      sección no crítica;
    }
}
```

Lo que hace esto es que mientras ts devuelva verdadero(aka mientras que al entrar al ts la variable sea true) se queda ahí

Cuando entró e inicial es false va a seguir avanzando

Solución es tipo spin locks, los procesos iteran hasta que se limpie el lock. Cumple las propiedades si scheduling es fuertemente fair

TS escribe siempre en *lock* aunque el valor no cambie ⇒ Mejor ***Test-and-Test-and-Set***

```
bool lock=false;
process SC[i=1 to n]
{ while (true)
    { while (TS(lock)) skip ;
      sección crítica;
      lock = false;
      sección no crítica;
    }
}
```

```
while (lock) skip;
while (TS(lock))
      while (lock) skip;
```

Mejor porque no ando escribiendo inutilmente. Solo llamo al ts si es falso el lock

Formalmente no me asegura lo de eventual entrada porque no controla ele orden de los procesos demorados.

Algoritmo Tie-Breaker

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
   while (true) {      in1 = true; ultimo = 1;
                       while (in2 and ultimo == 1) skip;
                       sección crítica;
                       in1 = false;
                       sección no crítica;
   }
}

process SC2 {
   while (true) {      in2 = true; ultimo = 2;
                       while (in1 and ultimo == 2) skip;
                       sección crítica;
                       in2 = false;
                       sección no crítica;
   }
}
```

Le da un orden a los procesos asegurando que nunca me va a quedar uno esperando eternamente
Para dos procesos está joya, para n es un quilombo.

Algoritmo ticket:

Se reparten números y se espera a que sea el turno. Es tipo sacar número irl

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
   }
}
```

NO NECESARIO

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
   }
}
```

NO NECESARIO

Podría ser una variable local a cada proceso.

Fetch and add:

FA(var,incr): < temp = var; var = var + incr; return(temp) >

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
process SC [i: 1..n]
{ while (true)
   {  turno[i] = FA (numero, 1);
      while (turno[i] <> proximo) skip;
      sección crítica;
      proximo = proximo + 1;
      sección no crítica;
   }
}
```

fetch and add me permite que cuando alguien más se fija cuanto vale el numero todavía no esté incrementado pero cuando sale si se incrementa
En el hipotetico caso de que e ejecute infinitamente puede explotar porque nunca las reinicio

Algoritmo bakery
Hace que no haya una variable centralizada con el valor del proximo sino que distribuye el peso entre los procesos. Cuando en algún momento no se esté ejecutando nada, se resetea en 0

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
      {   ⟨ turno[i] = max(turno[1:n] + 1; ⟩
          for [j = 1 to n st j <> i] ⟨ await (turno[j] == 0 or turno[i] < turno[j]); ⟩
          sección crítica
          turno[i] = 0;
          sección no crítica
      }
}
```

No puedo hacer lo del max turno de forma atómica, lo que se hace entonces es permitir repetidos (sacar los mayor menor al carajo)
Bakery no usa funciones raras y no me van a explotar las variables. Ineficiente pq tiene que recorrer todo el vector
Me asegura que se cumple eventual entrada con round robin común

Barrera

Todos los procesos tienen que llegar hasta cierto punto antes de seguir avanzando.

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {    código para implementar la tarea i;
             ⟨ cantidad = cantidad + 1; ⟩
             ⟨ await (cantidad == n); ⟩
        }
}
```

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {  código para implementar la tarea i;
           FA (cantidad, 1);
           while (cantidad <> n) skip;
        }
}
```

Piola si uso la barrera una sola vez, sino necesito resetearla
No puedo resetearla después del while pq lo veo yo, no el resto.
Para eso lo que se hace es cambiar un contador por un vectore de n posiciones con 0 o 1

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {    código para implementar la tarea i;
             arribo[i] = 1;
             ⟨ await (continuar[i] == 1); ⟩
             continuar[i] = 0;
        }
}

process Coordinador
{ while (true)
        {    for [i = 1 to n]
                  { ⟨ await (arribo[i] == 1); ⟩
                    arribo[i] = 0;
                  }
             for [i = 1 to n] continuar[i] = 1;
        }
}
```

Hay un coordinador que está todo el tiempo viendo si todos los trabajadores ya terminaron

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {     código para implementar la tarea i;
              arribo[i] = 1;
              while (continuar[i] == 0) skip;
              continuar[i] = 0;
        }
}

process Coordinador
{ while (true)
        {     for [i = 1 to n]
                  { while (arribo[i] == 0)  skip;
                    arribo[i] = 0;
                  }
              for [i = 1 to n] continuar[i] = 1;
        }
}
```

Barrera

Un problema es que necesita coordinador. Eso es costoso pq necesita un core solo para él

Faltan cositas acá

Todo esto es busy waiting
Complejas de implementar
Un garrón basicamente
Se necesitan herramientas más potentes----> semáforos :)
Se malgasta tiempo de procesadores
Mejor estar dormido hasta que me den el procesador