Clase 3

Clase 3 **1**
Barreras 1
Locks y barreras 1
Solución de hardware: Deshabilitar interrupciones 2
Solución de grano grueso 3
Solución de grano fino: Spin Locks 4
Solución fair: Tie breaker 4
Solución fair: algoritmo ticket 6
Solución fair: algoritmo Bakery 7
Sincronización barrier 8
Barrera simétrica 9
Grano fino de Butterfly 10
Conclusión 10

Barreras

2 tipos de programa:

- Memoria compartida
  - **Variables compartidas (tema de hoy)**
  - Semáforos
  - Monitores
- Pasaje de mensajes
  - Mensajes sincrónicos y asincrónicos
  - RPC (lo vemos solo teórico)
  - Rendezvous (lo vemos en ADA)

Locks y barreras

- **Sección crítica**: Implementamos acciones atómicas en software (esto es un problema).
- **Barrera**: Punto de sincronización que todos los procesos deben alcanzar para que puedan continuar.
- **Busy waiting**: Técnica en la cual **proceso espera condición** hasta que es **verdadera** y pasa a **continuar su ejecución** (se puede implementar en cualquier instruction set). Ineficiente si hay ejecución intercalada en un CPU.

Problema de **sección crítica**: ¿Qué propiedades deben satisfacer los protocolos de entrada y salida?. (< y > respectivamente):

- Exclusión mutua (un solo proceso en su SC).
- Ausencia de deadlock (livelock).
- Ausencia de demora innecesaria.
- Eventual entrada.

SCEnter y SCExit -> Protocolos de entrada y salida.
Para implementar el <> incondicional es poner las sentencias de los protocolos:

```
1. SCEnter
2. //Sección crítica
3. SCExit
```

Para implementar el <await(cond)> debemos hacer lo siguiente:

```
1. SCEnter;
2. while (not B) {
      a. SCExit;
      b.  SCEnter;
3. }
4. S;
5. SCExit;
```

Es decir entramos en el protocolo de esperar, si cumplimos la condición debemos salir, de la sección crítica para no dejar pasar otros procesos y que no haya deadlock, y luego entramos, en cuyo caso si la condición se cumple entonces salimos del while, ejecutamos S y salimos.

Si S es skip y B cumple ASV, entonces podemos implementar mediante while (Not B) skip;
Ineficiencias de lo anterior: Entrar y salir continuamente a la zona crítica. Podemos mejorar el tema agregando un delay.

```
6. SCEnter;
7. while (not B) {
      a. SCExit;
      b. Delay;
      c.  SCEnter;
8. }
9. S;
10.  SCExit;
```

Solución de hardware: Deshabilitar interrupciones

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

Funca si solo hay una unidad de procesamiento.

Solución de grano grueso

En esta solución usamos 2 variables booleanas, que son marcadas como true por 2 procesos distintos al entrar al sección crítica, y al salir se marcan false respectivamente.

```
process SC1
 { while (true)
     { in1 = true; # protocolo de entrada
       sección crítica;
       in1 = false; # protocolo de salida
       sección no crítica;
     }
 }
```

```
process SC2
 { while (true)
     { in2 = true; # protocolo de entrada
       sección crítica;
       in2 = false; # protocolo de salida
       sección no crítica;
     }
 }
```

- No asegura el invariante MUTEX ⇒ solución de "grano grueso"

```
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;
     }
 }
```

1. No sirve porque no chequea booleanos.
2. Sirve, porque:
    a. Hay exclusión mutua.
    b. No hay deadlock, si hubiera in1 e in2 debería ser verdaderos, al intentar entrar ambas deben estar en falso.
    c. No hay demora innecesaria.
    d. Eventual entrada: Si hay algún inconveniente, no sabemos quién entrará primero a menos que la política de scheduling sea fuertemente fair.

Generalización a N procesos:

```
bool lock=false;  # lock = in1 v in2 #
```

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

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

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

Solución de grano fino: Spin Locks

Queremos hacer atómico el await de grano grueso, para ello usamos instrtuccion de test and set, fetch y add, o compare y swap (disponibles en la mayoría de procesadores):

```
1. bool TS(bool ok){
      a. <bool inicial = ok;
      b. ok = true;
      c. return inicial>
2. }
```

```
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;
    }
}
```

Solución tipo “spin locks”: los procesos se quedan iterando (spinning) mientras esperan que se limpie lock
Cumple las 4 propiedades si el scheduling es fuertemente fair.
Problema: siempre se escribe la variable lock. Conviene 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;
```

Reduce el problema de memory contention pero sigue ahí.

Solución fair: Tie breaker

Requiere scheduling débilmente fair, no usa variables especiales, es más complejo. Usa una tercera variable para determinar el último proceso que intentó entrar.
Solución gran grueso del tie-breaker:

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

process SC1 {
   while (true) {      ultimo = 1; in1 = true;
                       ⟨await (not in2 or ultimo==2);⟩
                       sección crítica;
                       in1 = false;
                       sección no crítica;
   }
}

process SC2 {
   while (true) {      ultimo = 2; in2 = true;
                       ⟨await (not in1 or ultimo==1);⟩
                       sección crítica;
                       in2 = false;
                       sección no crítica;
   }
}
```

Grano fino del tie-breaker: (cambiamos await con while con skip):

```
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;
   }
}
```

N procesos:

Problema de la Sección Crítica.
Solución Fair: algoritmo *Tie-Breaker*

Generalización a *n* procesos:

- Si hay *n* procesos, el protocolo de entrada en cada uno es un *loop* que itera a través de *n-1* etapas.
- En cada etapa se usan instancias de ***tie-breaker*** para dos procesos para determinar cuáles avanzan a la siguiente etapa.
- Si a lo sumo a un proceso a la vez se le permite ir por las *n-1* etapas ⇒ a lo sumo uno a la vez puede estar en la SC.

```
int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);
process SC[i = 1 to n] {
   while (true) {
      for [j = 1 to n] {     # protocolo de entrada
            # el proceso i está en la etapa j y es el último
            in[i] = j; ultimo[j] = i;
            for [k = 1 to n st i <> k] {
                # espera si el proceso k está en una etapa más alta
                # y el proceso i fue el último en entrar a la etapa j
                while (in[k] >= in[i] and ultimo[j]==i) skip;
            }
      }
      sección crítica;
      in[i] = 0;
      sección no crítica;
   }
}
```

Clase 3 21

Horrible.

Solución fair: algoritmo ticket

Simula cuando alguien llega a un negocio, saca un ticket y espera que lo llamen.

```
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;
   }
}
```

Problema potencial: Los valores de "próximo" y "turno" pueden crecer ilimitadamente, pero en la práctica podrían restaurarse a un valor pequeño, como 1.

Cumplimiento de propiedades:

- **Invariante global:** El predicado TICKET asegura que solo un proceso puede estar en la sección crítica (SC) porque "número" y "próximo" se manejan de forma atómica.
- **Sin deadlock y sin demora innecesaria:** Esto se garantiza porque los valores de "turno" son únicos.
- **Fair scheduling:** Con un scheduling débilmente justo, se asegura que todos eventualmente puedan entrar en la SC.
- **Implementación del : Puede hacerse con busy waiting, dado que solo referencia una variable compartida.
- **Incremento de "próximo":** Puede realizarse con una operación normal de carga/almacenamiento, ya que solo un proceso ejecuta su protocolo de salida a la vez.

Solución de grano fino:

- Sea Fetch-and-Add una instrucción con el siguiente efecto:

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;
    }
}
```

Ticket si no existe FA se debe simular con una SC y la solución puede no ser fair.

Solución fair: algoritmo Bakery

- Más complejo, fair y no requiere instrucciones especiales.
- No requiere un contador global próximo que se “entrega” a cada proceso al llegar a la SC.
- Ticket descentralizado.

```
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
      }
}
```

Me fijo el máximo de todos los turnos, le sumo uno y me lo tomo, hacemos busy waiting esperando a que no haya ningún turno intentando entrar en la sección crítica o sus turnos son mayores que el del proceso actual. El que termina pone su turno en 0. No se pueden implementar directamente por el primer await.

Solución de grano fino:

```
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] = 1;   //indica que comenzó el protocolo de entrada
           turno[i] = max(turno[1:n]) + 1;
           for [j = 1 to n st j != i]        //espera su turno
                 while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
           sección crítica
           turno[i] = 0;
           sección no crítica
        }
}
```

Dado que puede darle el mismo valor de turno a varios, se hace una forma de romper el empate dando el turno al que tenga menor id.
Además para que los demás procesos te cuenten como competencia ya al inicio tiene que poner su turno en uno.

Sincronización barrier

Cada proceso aumenta la variable cantidad al llegar, cuando la cantidad es igual a la cantidad de proceso entonces todos pasan.
Sirve para una sola iteración. Si queremos hacerlo multi iteraciones tenemos que ver cuando resetear cantidad para no generar deadlock.

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

Solución en caso:

Usamos flags y coordinadores, en este caso usamos un arreglo de arribos para saber quienes llegaron, y un arreglo de continuar manejado por el coordinador para indicarles que tienen que seguir.

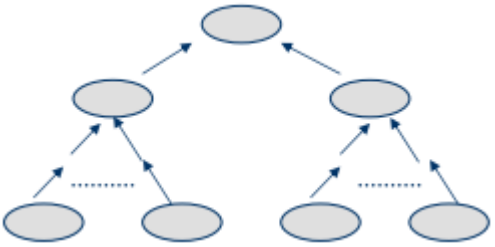
```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

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;
         }
}
```

Problemas:

- Requiere un proceso y un procesador extra.
- O(n) del coordinador.

Solución:

- Podemos combinar a los trabajadores como coordinadores.
- Usamos una estructura de árbol donde las señales de arribo van hacia arriba del árbol y las de continuar hacia abajo.



Problema, se suele tener que triplicar el código:

Barrera simétrica

Se construyen barreras para procesos a partir de pares de barreras.

```
W[i]:: ⟨ await (arribo[i] == 0); ⟩
       arribo[i] = 1;
       ⟨ await (arribo[j] == 1); ⟩
       arribo[j] = 0;

W[j]:: ⟨ await (arribo[j] == 0); ⟩
       arribo[j] = 1;
       ⟨ await (arribo[i] == 1); ⟩
       arribo[i] = 0;
```

Butterfly:

Primera etapa: Cada proceso se coordina con el que está al lado, distancia 1.
Segunda etapa: Con el que está a distancia 2.
Tercera etapa: Coordinar con los que están a distancia 4.

Pasos: log2 (n)

Grano fino de Butterfly

```
int E = log(N);
int arribo[1:N] = ([N] 0);

process P[i=1..N]
{  int j;
   while (true)
      {  //Sección de código anterior a la barrera.
         //Inicio de la barrera
         for (etapa = 1; etapa <= E; etapa++)
            {   j =  (i-1) XOR (1<<(etapa-1));    //calcula el proceso con cual sincronizar
                while (arribo[i] == 1) → skip;
                arribo[i] = 1;
                while (arribo[j] == 0) → skip;
                arribo[j] = 0;
            }
         //Fin de la barrera
         //Sección de código posterior a la barrera.
      }
}
```

Conclusión

- Los protocolos de "busy waiting" son complejos y no separan claramente las variables de sincronización de las de computación.
- La verificación y prueba de corrección son difíciles, especialmente con un número elevado de procesos.
- En entornos de multiprogramación, "busy waiting" es ineficiente, ya que desperdicia recursos de procesamiento.
- Es necesaria la creación de herramientas especializadas para diseñar protocolos de sincronización más eficientes.