# Clase 3

[**Clase 3 1**](#_77612awltkra)

[Barreras 1](#_e2gabcv9q8rz)

[Locks y barreras 1](#_t182p3n772iy)

[Solución de hardware: Deshabilitar interrupciones 2](#_outxpnpdw9r2)

[Solución de grano grueso 3](#_pbqofx8t9bt2)

[Solución de grano fino: Spin Locks 4](#_57croascxxat)

[Solución fair: Tie breaker 4](#_f14e6c69evdj)

[Solución fair: algoritmo ticket 6](#_6jfqw9ftsqrr)

[Solución fair: algoritmo Bakery 7](#_cth4olmnxjji)

[Sincronización barrier 8](#_8ylyf6t90h6r)

[Barrera simétrica 9](#_1xrmocmwofss)

[Grano fino de Butterfly 10](#_yhl89uuwztg8)

[Conclusión 10](#_49xir0hy15em)

## 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) {
   1. SCExit;
   2. SCEnter;
3. }
4. S;
5. SCExit;

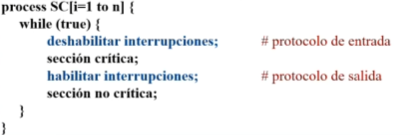
Es decir entramos en el protocolo de esperar, si cumplimos la condición debemos salir, de la sección crítica para no dejar pasar otros procesos y que no haya deadlock, y luego entramos, en cuyo caso si la condición se cumple entonces salimos del while, ejecutamos S y salimos.

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.

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

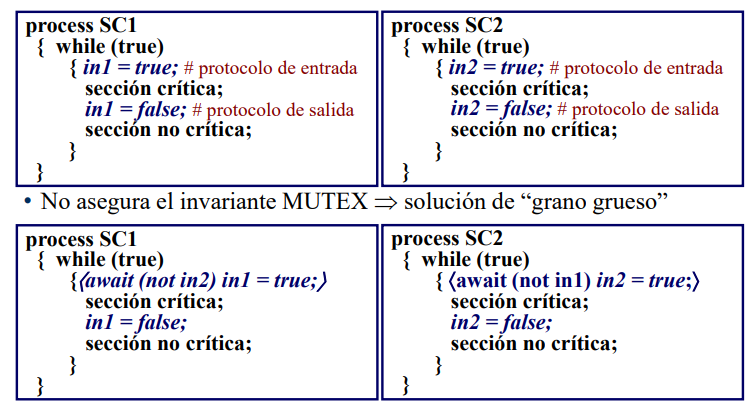
#### **Solución de hardware**: Deshabilitar interrupciones



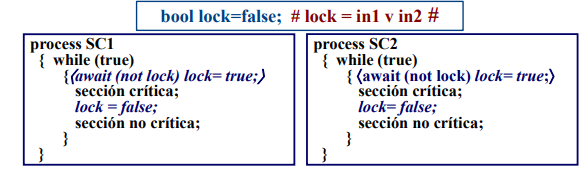
Funca si solo hay una unidad de procesamiento.

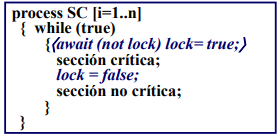
#### **Solución de grano grueso**

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.



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

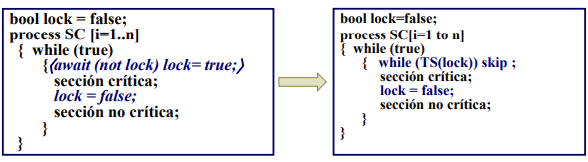
Generalización a N procesos:  




#### **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):

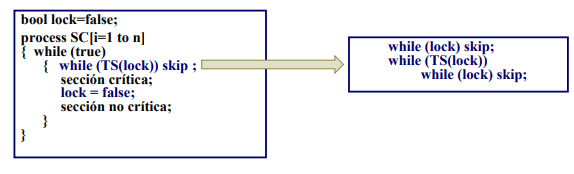
1. bool TS(bool ok){
   1. <bool inicial = ok;
   2. ok = true;
   3. return inicial>
2. }



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.

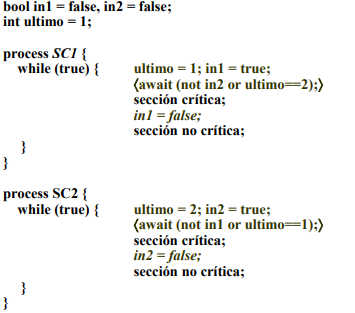
**Problema**: siempre se escribe la variable lock. Conviene Test-and-test-and-set

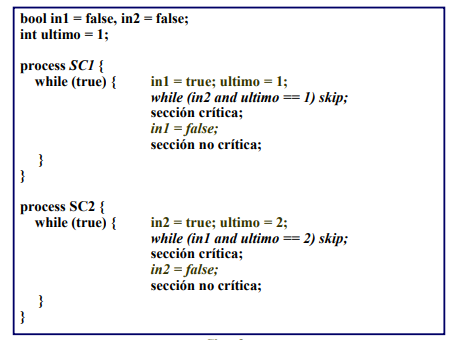
  
Reduce el problema de memory contention pero sigue ahí.

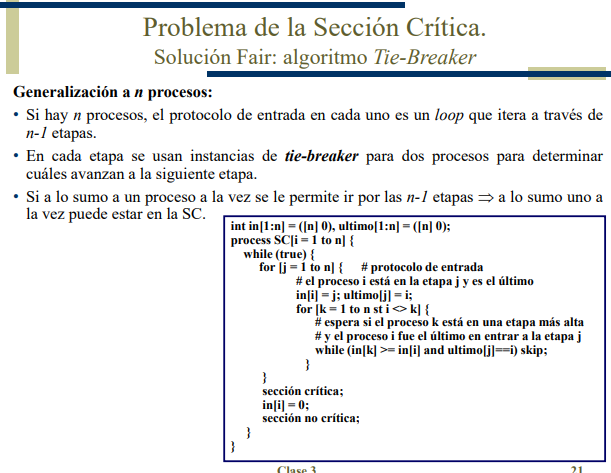
#### **Solución fair**: Tie breaker

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:



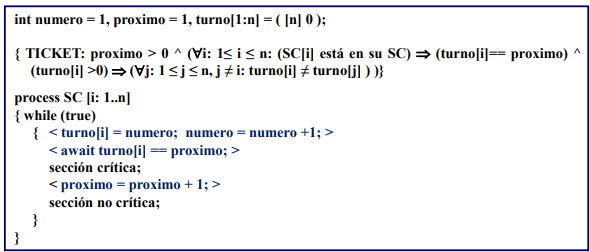
Grano fino del tie-breaker: (cambiamos await con while con skip):  


N procesos:  


Horrible.

#### **Solución fair: algoritmo ticket**

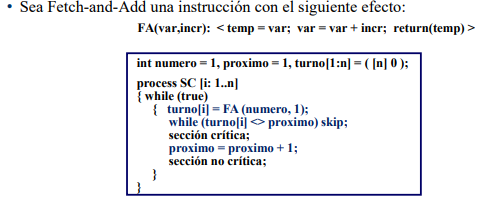
Simula cuando alguien llega a un negocio, saca un ticket y espera que lo llamen.



**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:**

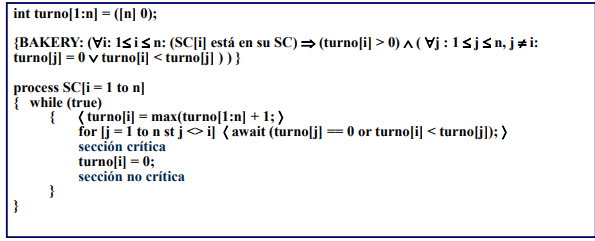
* **Invariante global:** El predicado TICKET asegura que solo un proceso puede estar en la sección crítica (SC) porque "número" y "próximo" se manejan de forma atómica.
* **Sin deadlock y sin demora innecesaria:** Esto se garantiza porque los valores de "turno" son únicos.
* **Fair scheduling:** Con un scheduling débilmente justo, se asegura que todos eventualmente puedan entrar en la SC.
* **Implementación del await:** 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:  


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

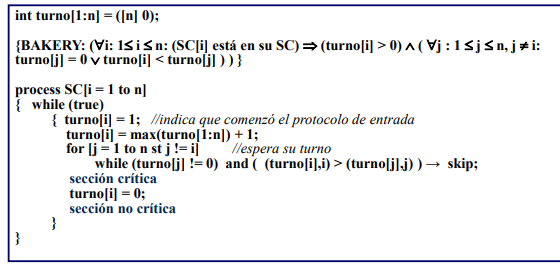
#### **Solución fair:** algoritmo Bakery

* 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.



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:  


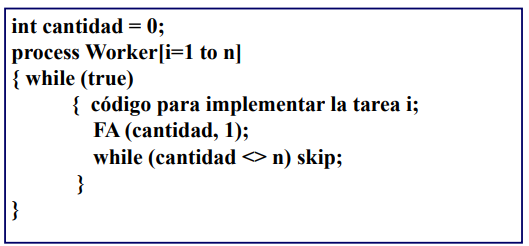
Dado que puede darle el mismo valor de turno a varios, se hace una forma de romper el empate dando el turno al que tenga menor id.

Además para que los demás procesos te cuenten como competencia ya al inicio tiene que poner su turno en uno.

### Sincronización barrier

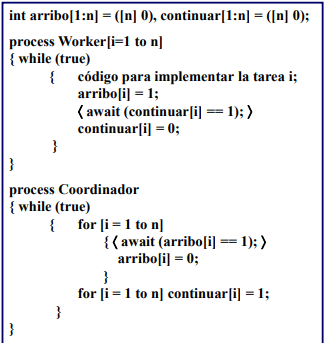
Cada proceso aumenta la variable cantidad al llegar, cuando la cantidad es igual a la cantidad de proceso entonces todos pasan.

Sirve para una sola iteración. Si queremos hacerlo multi iteraciones tenemos que ver cuando resetear cantidad para no generar deadlock.



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.

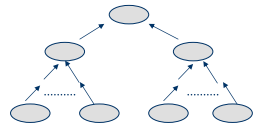


Problemas:

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

Solución:

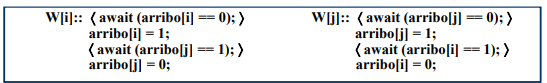
* Podemos combinar a los trabajadores como coordinadores.
* Usamos una estructura de árbol donde las señales de arribo van hacia arriba del árbol y las de continuar hacia abajo.



Problema, se suele tener que triplicar el código:

#### **Barrera simétrica**

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



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**

## 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.