Programación Concurrente 2019

Clase 3

Facultad de Informática
UNLP

Resumen de la clase anterior

- **Conceptos**
  - **Prioridad / Granularidad / Manejo de recursos / Deadlock**
- **Aspectos de programación secuencial - concurrencia**
  - **secuencia**
  - **decisión, iteración – no determinismo**
  - **for-all**
  - **co y process**
- **Paradigmas de resolución de programas concurrentes**
  - **paralelismo iterativo (ejemplo)**
  - **paralelismo recursivo (ejemplo)**
  - **productores y consumidores**
  - **clientes y servidores**
  - **pares que interactúan**
- **Clasificaciones de hardware**
  - **por espacio de direcciones / por mecanismo de control / por granularidad / por red de interconexión**
- **Concurrencia y sincronización**
  - **Acciones atómicas e historias de un programa concurrente**
  - **Interferencia - Rol de la sincronización**
  - **Atomicidad de grano fino y de grano grueso**
  - **Propiedad de ASV**
  - **Especificación de la sincronización. AA condicionales e incondicionales**
- **Propiedades de programa. Seguridad y vida. Scheduling**

Acciones atómicas y Sincronización. Propiedad de *"A lo sumo una vez"*

Referencia crítica en una expresión ⇒ referencia a una vble que es modificada por otro proceso.
Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

*Una sentencia de asignación **x = e** satisface la propiedad de A lo sumo una vez si*
*(1) **e** contiene a lo sumo una referencia crítica y **x** no es referenciada por otro proceso, o*
*(2) **e** no contiene referencias críticas, en cuyo caso **x** puede ser leída por otro proceso*

Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez.
Una def. similar se aplica a expresiones que no están en sentencias de asignación (satisface ASV si no contiene más de una ref. crítica)

Acciones atómicas y Sincronización. Propiedad de *"A lo sumo una vez"*

EFECTO: Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución *parece* atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

int x=0, y=0; co x=x+1 // y=y+1 oc;	No hay ref. críticas en ningún proceso `x = 1 e y = 1 ∀ historia`
int x=0, y=0; co x=y+1 // y=y+1 oc;	El 1er proc tiene 1 ref. crítica. El 2do ninguna. y = 1 siempre, x = 1 o 2
int x=0, y=0; co x=y+1 // y=x+1 oc;	Ninguna asignación satisface ASV Posibles: x=1, y=2 / x=2, y=1 / x=1, y=1 Pero no podría ocurrir!!!

Especificación de la sincronización

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente

En general, es necesario ejecutar secuencias de sentencias como una única acción atómica

Una acción atómica de grano grueso es una especificación en alto nivel del comportamiento requerido de un programa que puede ser implementada de distintas maneras, dependiendo del mecanismo de sincronización disponible.

⇒ Mecanismo de sincronización para construir una acción atómica *de* grano grueso (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles

Ej: BD con dos valores que en todo momento deben ser iguales

Ej: Productor y consumidor con una lista enlazada

Especificación de la sincronización

⟨**e**⟩ indica que la expresión ***e*** debe ser evaluada atómicamente

⟨**await (B) S;**⟩ se utiliza para especificar sincronización

La expresión booleana B especifica una condición de demora.
S es una secuencia de sentencias que se garantiza que termina.
Se garantiza que B es true cuando comienza la ejecución de S.
Ningún estado interno de S es visible para los otros procesos.

Ej: ⟨**await (s>0) s=s-1;**⟩

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de await (EM y SxC) es alto

Especificación de la sincronización

Sólo exclusión mutua ⇒ ⟨S⟩
Ej: ⟨x = x + 1; y = y + 1⟩
El estado interno en el cual x e y son incrementadas resulta invisible a otros procesos que las referencian

Sólo sincronización por condición ⇒ ⟨await (B)⟩
Ej: ⟨await (count > 0) ⟩

Si B satisface ASV, puede implementarse como *busy waiting* o *spinning*: **do (not B) → skip od** **(while (not B);)**

Acciones atómicas incondicionales y condicionales

Seguridad y vida

Una *propiedad* de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo

Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida.
Son dos aspectos complementarios de la *corrección*

seguridad **(safety): n**ada malo le ocurre a un objeto: asegura estados consistentes

- una *falla de seguridad* indica que algo anda mal
- Ej: ausencia de deadlock y ausencia de interferencia (exclusión mutua) entre procesos.

vida **(liveness):** eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks

- una *falla de vida* indica que se deja de ejecutar
- Ej: *terminación*, *asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc* $\Rightarrow$ ***dependen de las políticas de scheduling.***

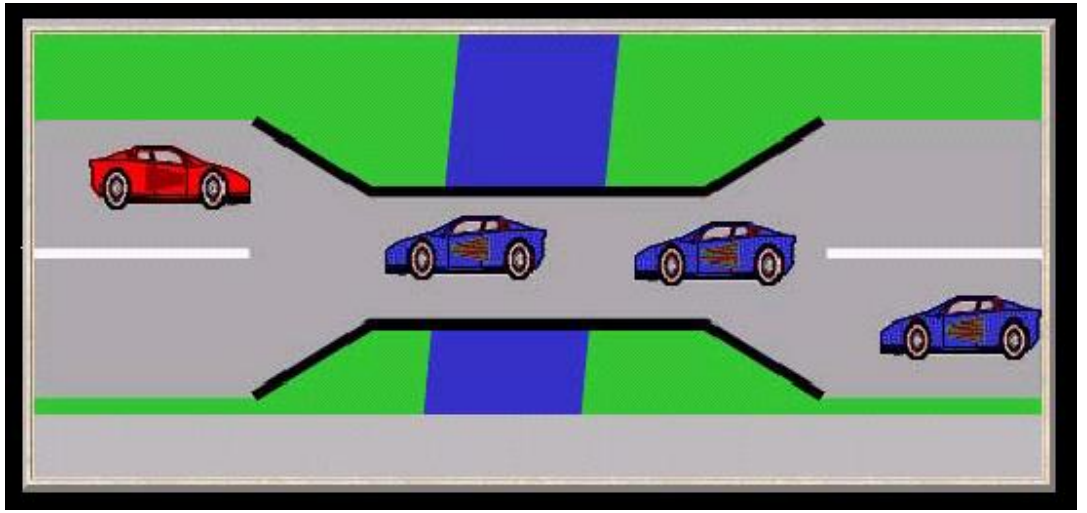
Seguridad y vida

Ejemplo: Puente de una sola vía

Puente sobre río con ancho sólo para una fila de tráfico $\Rightarrow$ los autos pueden moverse concurrentemente si van *en la misma dirección*

- Violación de *seguridad* si dos autos en distintas direcciones entran al puente al mismo tiempo

- Vida: c/ auto tendrá *eventualmente* oportunidad de cruzar el puente?



Los temas de seguridad deben balancearse con los de vida

Seguridad y vida

Seguridad → Fallas típicas (*race conditions*):

- Conflictos de read/write: un proceso lee un campo y otro lo escribe (el valor visto por el lector depende de quién ganó la "carrera").
- Conflictos de write/write: dos procesos escriben el mismo campo (quién gana la "carrera").

Vida → Fallas:

- *Temporarias*: Bloqueo temporarios, Espera, Contención de CPU, Falla recuperable.
- *Permanente*: Deadlock, Señales perdidas, Anidamiento de bloqueos, Livelock, Inanición, Agotamiento de recursos, Falla distribuida.

Fairness y políticas de scheduling

***Fairness*: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás**

Una acción atómica en un proceso es elegible si es la próxima acción atómica en el proceso que será ejecutado
Si hay varios procesos ⇒ hay *varias acciones atómicas elegibles*

Una *política de scheduling* determina cuál será la próxima en ejecutarse

Política: asignar un procesador a un proceso hasta que termina o se demora. Qué sucede en este caso??

```
bool continue = true;
co while (continue);
   // continue = false;
oc
```

Fairness y políticas de scheduling

Fairness Incondicional. Una política de scheduling es incondicionalmente fair (o *imparcial*) si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

En el ej. anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador (si ningún procesador puede monopolizar el acceso a la vble compartida)

Fairness Débil. Una política de scheduling es débilmente fair si (1) es incondicionalmente fair y (2) toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional

RR es débilmente fair en el ej anterior

No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado.

Fairness y políticas de scheduling

Fairness Fuerte**:* Una política de scheduling es ***fuertemente fair si (1) es incondicionalmente fair y (2) toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Este programa termina??

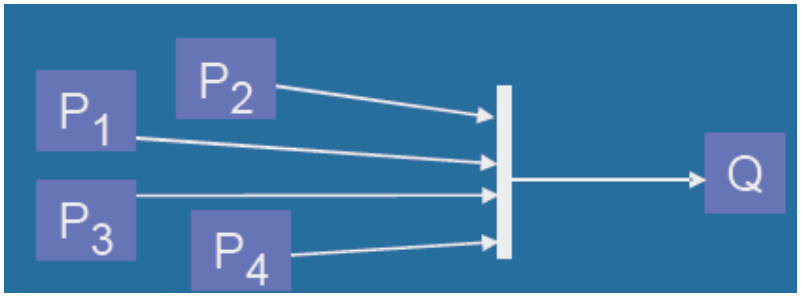
```
bool continue = true, try = false;
co while (continue) { try = true; try = false; }
 // ⟨await (try) continue = false ⟩
oc
```

No es simple tener una política que sea práctica y fuertemente fair.
En el ej anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica.
Round-robin es práctica pero no es fuertemente fair.

Tipos de Problemas Básicos de Concurrencia

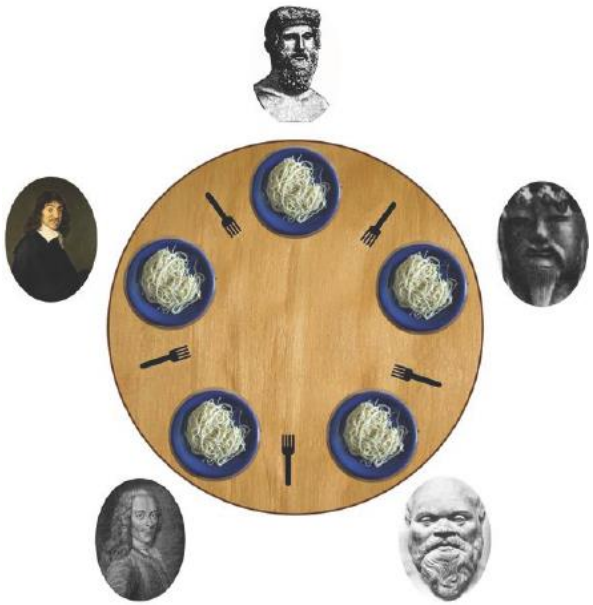
***Exclusión mutua**:* problema de la sección crítica. (Administración de recursos).

Barreras: punto de sincronización



Comunicación

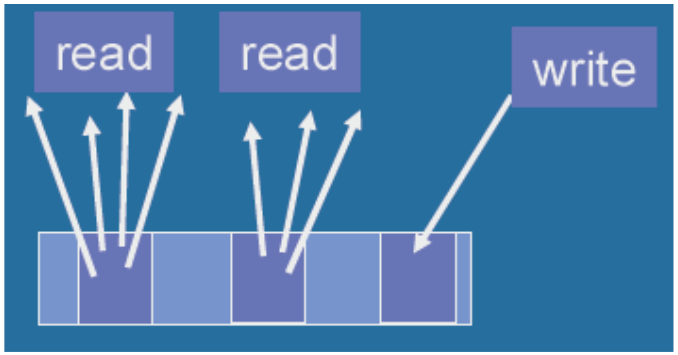
Filósofos: Dijkstra, 1971.
Sincronización multiproceso.
Evitar deadlock e inanición.
Exclusión mutua selectiva



Tipos de Problemas Básicos de Concurrencia

Productor-consumidor

Lectores-escritores



***Sleeping barber**:* Dijkstra.
Sincronización – rendezvous.

Herramientas para la concurrencia

- Memoria compartida
  - Variables compartidas
  - Semáforos
  - Regiones Críticas Condicionales
  - Monitores
- Memoria distribuida (pasaje de mensajes)
  - Mensajes asincrónicos
  - Mensajes sincrónicos
  - Remote Procedure Call (RPC)
  - Rendezvous

Sincronización por variables compartidas. Locks y barreras

***Problema de la SC**:* implementación de acciones atómicas en software (*locks*)

Barrera: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

En la técnica de *busy waiting* un proceso chequea repetidamente una condición hasta que sea verdadera.

Ventaja: implementación con instrucciones de cualquier procesador

Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es interleaved)

Aceptable si cada proceso ejecuta en su procesador.

El problema de la Sección Crítica

```
process SC[i=1 to n] {
    while (true) {
        protocolo de entrada;        /* ⟨
        sección crítica;             /*  …
        protocolo de salida;         /*   ⟩
        sección no crítica;
    }
}
```

Las soluciones a este problema pueden usarse para implementar sentencias *await* arbitrarias.

Qué propiedades deben satisfacer los protocolos de entrada y salida?

El problema de la Sección Crítica

Exclusión mutua. A lo sumo un proceso está en su SC

Ausencia de Deadlock**:** Si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.

Ausencia de Demora Innecesaria: Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

Eventual Entrada: Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Las 3 primeras son propiedades de seguridad, y la 4° de vida.

Solución trivial ⟨**SC**⟩. Pero, cómo se implementan los ⟨ ⟩ ??

El problema de la Sección Crítica. Solución hardware: deshabilitar interrupciones

```
process SC[i=1 to n] {
    while (true) {
        deshabilitar interrupciones;
        sección crítica;                    /*  …
        habilitar interrupciones;
        sección no crítica;
    }
}
```

Solución correcta para una máquina monoprocesador.

Durante la SC no se usa la multiprogramación ⇒ penalización de performance

La solución no es correcta en un multiprocesador.

El problema de la Sección Crítica. Primer intento de solución

```
bool in1=false, in2=false
{MUTEX: ¬(in1 ∧ in2)      debiera mantenerse invariante}
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 ⇒ **fortalecerlo**

El problema de la Sección Crítica. Una solución de “grano grueso”

```
bool in1=false, in2=false
{MUTEX: ¬(in1 ∧ in2)}
process SC1 {
   while (true) {   ⟨await (not in2) in1 = true; ⟩  #protocolo de entrada
                    sección crítica;
                    in1 = false;                     #protocolo de salida
                    sección no crítica;
   }
}
process SC2 {
   while (true) {   ⟨await (not in1) in2 = true; ⟩  #protocolo de entrada
                    sección crítica;
                    in2 = false;                     #protocolo de salida
                    sección no crítica;
   }
}
```

Satisface las 4 condiciones??

El problema de la Sección Crítica. Cumple las condiciones?

Exclusión mutua: por construcción, P1 y P2 se excluyen en el acceso a la SC

```
bool in1=false, in2=false

process SC1 {                          process SC2 {
  while (true) { ⟨await (not in2) in1 = true; ⟩   while (true) { ⟨await (not in1) in2 = true; ⟩
        sección crítica;                       sección crítica;
        in1 = false;                           in2 = false;
        sección no crítica; }                  sección no crítica; }
}                                      }
```

Ausencia de deadlock: si hay deadlock, P1 y P2 están bloqueados en su protocolo de entrada $\Rightarrow$ 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 P1 está fuera de su SC o terminó, in1 es false.
- Si P2 está tratando de entrar a SC y no puede, la guarda en su protocolo de entrada debe ser falsa (esto es, in es true):

$\neg$ in1 $\wedge$ in1 = FALSE $\Rightarrow$ no hay demora innecesaria

El problema de la Sección Crítica. Cumple las condiciones?

Eventual entrada:

bool in1=false, in2=false	
process SC1 {	process SC2 {
while (true) { ⟨await (not in2) *in1 = true*; ⟩	while (true) { ⟨await (not in1) *in2 = true*; ⟩
sección crítica;	sección crítica;
in1 = false;	*in2 = false;*
sección no crítica; }	sección no crítica; }
}	}

Si P1 está tratando de entrar a su SC y no puede, P2 está en SC e in2 es TRUE.
Un proceso que está en SC eventualmente sale ⇒ in2 será FALSE y la guarda de P1 verdadera.

Análogamente para P2

Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito, las guardas son TRUE con infinita frecuencia.

Luego, se garantiza la eventual entrada con una política de scheduling fuertemente fair (aunque estas políticas son imprácticas en la mayor parte de los casos)

Sección Crítica. Cambio de variables

bool in1=false, in2=false	
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; } }

SEA lock = in1 v in2 ⇒

```
bool lock=false;
process SC1 {
   while (true) {   ⟨await (not lock) lock= true; ⟩ #protocolo de entrada
                    sección crítica;
                    lock = false;                    #protocolo de salida
                    sección no crítica;
   }
}
process SC2 {
 while (true) {     ⟨await (not lock) lock= true; ⟩ #protocolo de entrada
                    sección crítica;
                    lock = false;                    #protocolo de salida
                    sección no crítica;
   }
}
```

Procesos idénticos ⇒ Puede extenderse a n procesos

Sección Crítica. Solución de “grano fino”

Objetivo ⇒ hacer “atómico” el await de grano grueso.

Idea: usar instrucciones como Test & Set (TS), Fetch & Add (FA) o Compare & Swap, disponibles en la mayoría de los procesadores

TS toma una variable compartida **lock** como argumento y devuelve un resultado booleano:

```
bool TS(bool lock) {
    ⟨bool initial = lock;  # guarda valor inicial
     lock = true;          # setea lock
     return initial; ⟩     # devuelve valor inicial
}
```

Sección Crítica. Solución de “grano fino”: *spin locks*

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

```
bool lock=false;          # lock compartido
process SC[i=1 to n] {
   while (true) {
         while (TS(lock)) skip ;  # protocolo de entrada
         sección crítica;
         lock = false;            # protocolo de salida
         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.
Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC)

Sección Crítica. Solución de “grano fino”: *spin locks*

Baja performance en multiprocesadores si varios procesos compiten por el acceso.

lock es una vble compartida y su acceso contínuo es muy costoso (“*memory contention*”).

Además, podría producirse un alto overhead por cache inválida

TS escribe siempre en lock aunque el valor no cambie ⇒ > tiempo.
Mejora ⇒ *Test-and-Test-and-Set*

```
while (true) {
        while (TS(lock)) skip ;
        sección crítica;
        lock = false;
        sección no crítica;    }
```

```
while (lock) skip ;          # spin mientras lock es true
while (TS(lock)) { ;         # intenta tomar el lock
  while (lock) skip; }       # si falla, spin nuevamente
```

Memory contention se reduce, pero no desaparece. En particular, cuando lock pasa a *false* posiblemente todos intenten hacer TS

Sección Crítica. Implementación de sentencias Await

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

```
SCEnter              #protocolo de entrada a la SC
S
SCExit               #protocolo de salida de la SC
```

Para una acción atómica condicional ⟨ await (B) S; ⟩

```
SCEnter
while (not B) { SCExit;  SCEnter; }
S
SCExit
```

Correcto, pero ineficiente: un proceso está *spinning* contínuamente saliendo y entrando a la SC hasta que otro altere una variable referenciada en B

Sección Crítica. Implementación de sentencias Await

Para reducir la contención de memoria, los procesos pueden demorarse antes de volver a intentar ganar el acceso a SC:

```
SCEnter
while (not B) { SCExit;  Delay;  SCEnter; }
S
SCExit
```

Delay podría ser un loop que itera un n° al azar de veces.
Este clase de protocolo podría usarse incluso dentro de SCEnter (en lugar del *skip* en el protocolo de test-and-set)

Si **S** es ***skip*****,** y **B** cumple ASV, ⟨ await (B); ⟩ puede implementarse

while (not B) skip;

Sección Crítica. Soluciones Fair

Spin locks $\Rightarrow$ no controla el orden de los procesos demorados $\Rightarrow$ es posible que alguno no entre nunca si el scheduling no es fuertemente fair (*race conditions*)

Algoritmo Tie-Breaker **(o algoritmo de Peterson)** $\rightarrow$ protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales, pero es más complejo

Usa una variable adicional para romper empates, indicando qué proceso fue el último en comenzar a ejecutar su protocolo de entrada a la SC $\Rightarrow$ ***ultimo*** variable compartida de acceso protegido

Se demora (quita prioridad) al último en comenzar su protocolo de entrada

Sección Crítica. Algoritmo Tie-Breaker (grano grueso)

```
bool in1 = false, in2 = false;
Int ultimo = 1;
process SC1 {
    while (true) {   in1 = true; ultimo = 1;
                     ⟨await (not in2 or ultimo==2);⟩
                     sección crítica;
                     in1 = false;
                     sección no crítica;
    }
}
process SC2 {
    while (true) {   in2 = true; ultimo = 2;
                     ⟨await (not in1 or ultimo==1);⟩
                     sección crítica;
                     in2 = false;
                     sección no crítica;
    }
}
```

Sección Crítica. Algoritmo Tie-Breaker (grano grueso)

```
bool in1 = false, in2 = false;
Int ultimo = 1;
process SC1 {
   while (true) { in1 = true; ultimo = 1;
                  ⟨await (not in2 or ultimo==2);⟩
                  sección crítica;
                  in1 = false;
                  sección no crítica;  }
}
process SC2 {
   while (true) { in2 = true; ultimo = 2;
                  ⟨await (not in1 or ultimo==1);⟩
                  sección crítica;
                  in2 = false;
                  sección no crítica;  }
}
```

Esta solución es "casi" de grano fino:

Las expresiones en los await no cumplen ASV, Pero no es necesario evaluar atómicamente las condiciones de demora ya que:

Si SC1 evalúa y obtiene true entonces:
- Si in2 es falso, in2 podría volverse true pero en ese caso SC2 debe haber puesto ultimo en 2 y la condición sigue siendo true aunque haya cambiado in2

- Si ultimo = 2, la condición se mantiene porque ultimo cambiará sólo después que SC1 ejecute su sección crítica.

Análogamente para SC2.

⇒ Puedo implementar por busy waiting

Sección Crítica. Algoritmo Tie-Breaker (grano fino)

```
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;
    }
}
```

```
bool in1 = false, in2 = false;
Int ultimo = 1;
process SC1 {
   while (true) { in1 = true; ultimo = 1;
                  ⟨await (not in2 or ultimo==2);⟩
                  sección crítica;
                  in1 = false;
                  sección no crítica;   }
}
process SC2 {
   while (true) { in2 = true; ultimo = 2;
                  ⟨await (not in1 or ultimo==1);⟩
                  sección crítica;
                  in2 = false;
                  sección no crítica;   }
}
```

Sección Crítica. Tie-Breaker n- proceso

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 $\Rightarrow$ a lo sumo uno a la vez puede estar en la SC.

Algoritmo $\Rightarrow$

Sección Crítica. Tie-Breaker n- proceso

```
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
            ultimo[j] = i; in[i] = j;
            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;
    }
}
```

Sección Crítica. Algoritmo Ticket

Tie-Breaker n-proceso ⇒ complejo y costoso en tiempo

Algoritmo Ticket ⇒ se reparten números y se espera turno

Los clientes toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los clientes con un número más chico sean atendidos.

```
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} }
```

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

Sección Crítica. Algoritmo Ticket

Potencial problema: los valores de *proximo* y *turno* son ilimitados.
En la práctica, podrían resetearse a un valor chico (por ej, 1).

El predicado TICKET es un invariante global, pues ***numero*** es leído e incrementado en una acción atómica y ***proximo*** es incrementado en una acción atómica $\Rightarrow$ hay a lo sumo un proceso en la SC.

La ausencia de deadlock y de demora innecesaria resultan de que los valores de ***turno*** son únicos.
Con scheduling débilmente fair se asegura eventual entrada

El **await** puede implementarse con busy waiting (la expresión booleana referencia una sola variable compartida).

El incremento de ***proximo*** puede ser un load/store normal (a lo sumo un proceso puede estar ejecutando su protocolo de salida)

Sección Crítica. Algoritmo Ticket

Cómo se implementa la primera acción atómica??

⟨ **turno[i] = numero; numero = numero + 1;** ⟩

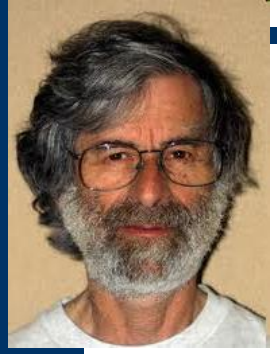
Sea Fetch-and-Add una instrucción con el siguiente efecto:

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;
    }
}
```

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

Si no existe FA: *SCenter; turno[i]=numero; numero=numero+1; SCexit*



Sección Crítica. Algoritmo Bakery (Lamport)

En Ticket, si no existe FA la solución puede no ser fair.
El algoritmo **Bakery** es más complejo, pero es fair y no requiere instrucciones especiales

No requiere un contador global ***proximo*** que se “entrega” a cada proceso al llegar a la SC.

Cada proceso que trata de ingresar recorre los números de los demás y se autoasigna uno mayor.

Luego espera a que su número sea el menor de los que esperan.

Los procesos se chequean entre ellos y no contra un global

Sección Crítica. Algoritmo Bakery

```
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  }  }
```

Esta solución de grano grueso no es implementable directamente:
- la asignación a turno[i] exige calcular el máximo de n valores
- el await referencia una variable compartida dos veces

TAREA: analizar detenidamente el algoritmo bakery (descripto en el libro de Andrews)

Sincronización Barrier

Problemas resueltos por algoritmos iterativos que computan sucesivas mejores aproximaciones a una rta, y terminan al encontrarla o al converger.

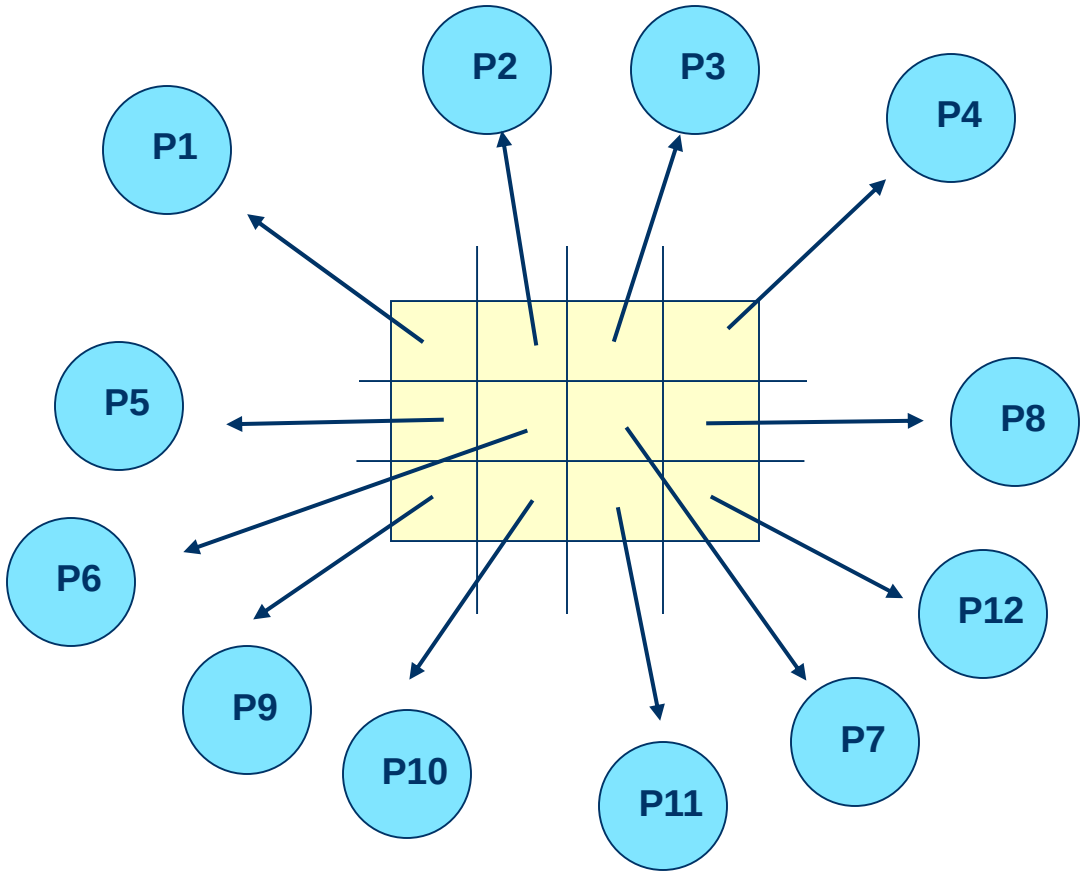
En general manipulan un arreglo, y cada iteración realiza la misma computación sobre todos los elementos del arreglo.

$\Rightarrow$ Múltiples procesos para computar partes disjuntas de la solución en paralelo

En la mayoría de los algoritmos iterativos paralelos cada iteración depende de los resultados de la iteración previa.

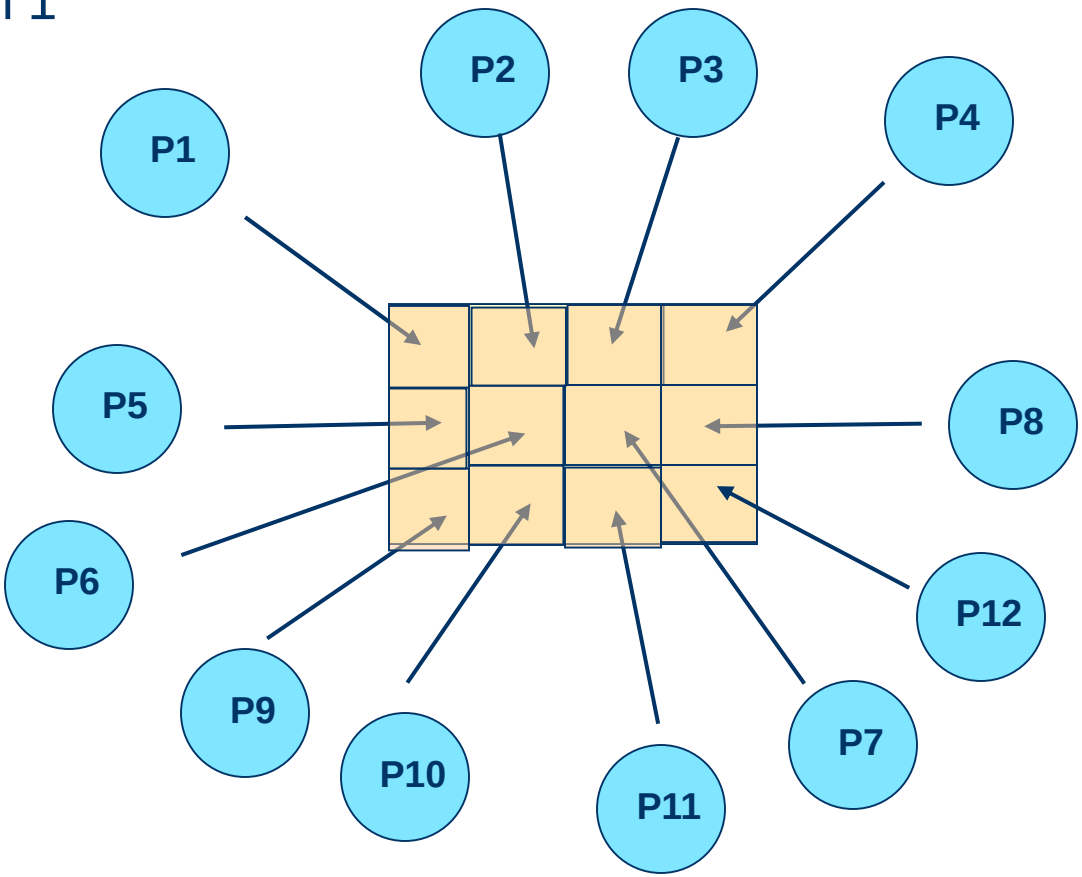
Sincronización Barrier

Inicial



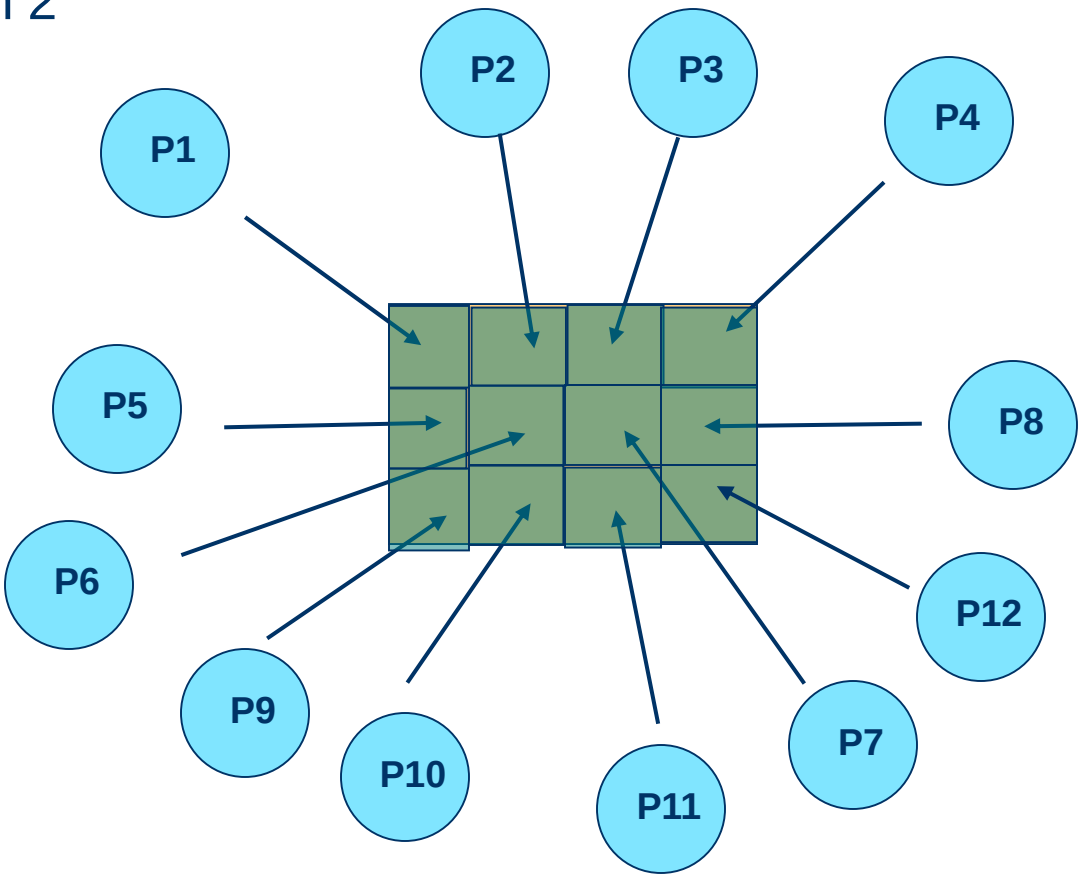
Sincronización Barrier

Iteración 1



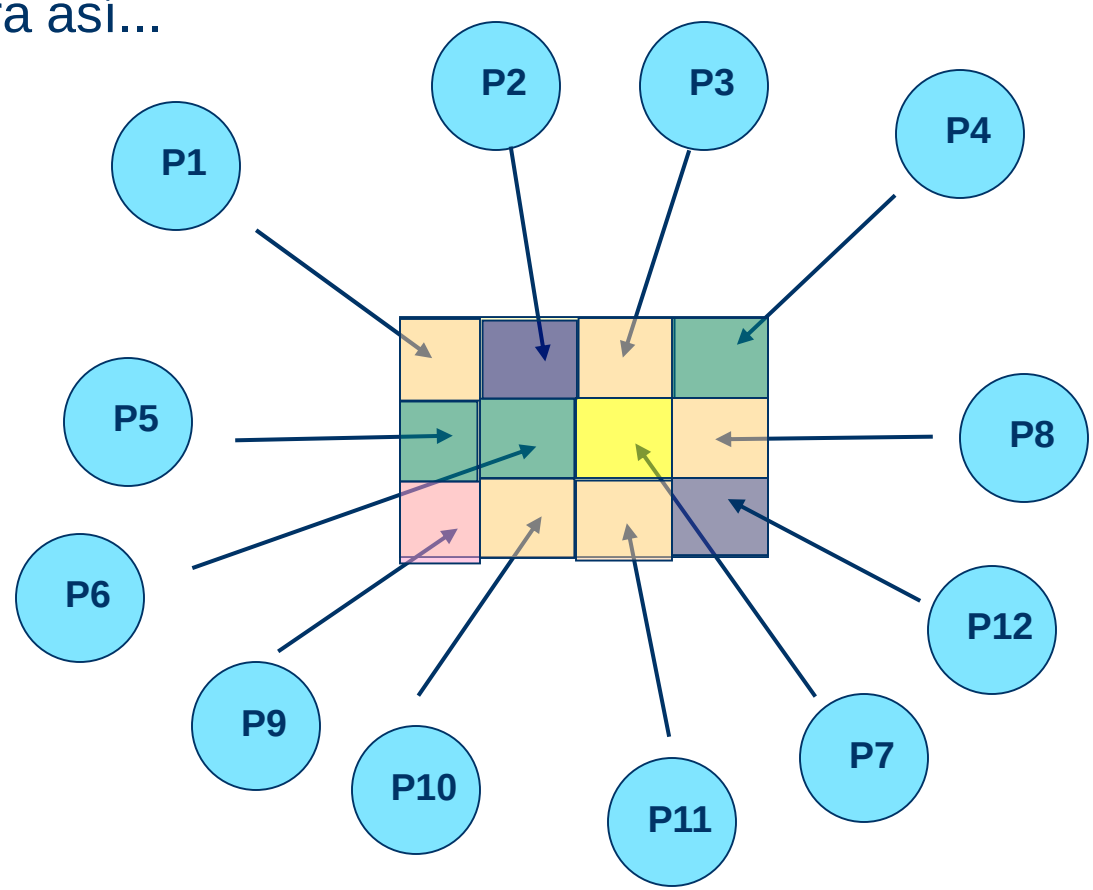
Sincronización Barrier

Iteración 2



Sincronización Barrier

Si no fuera así...



Sincronización Barrier

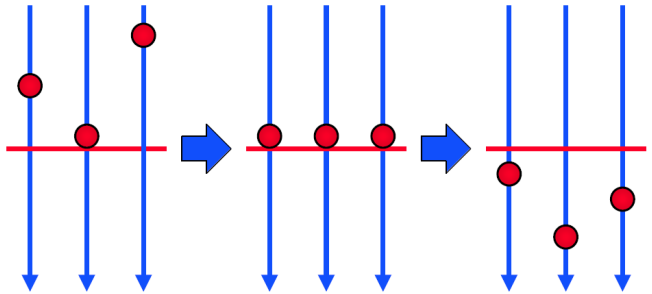
Ignorando terminación, y asumiendo *n* tareas paralelas en cada iteración, se tiene la forma general:

```
while (true) {
    co [i=1 to n]
        código para implementar la tarea i;
    oc   }
```

Ineficiente, ya que produce *n* procesos en cada iteración.

⇒ crear procesos al comienzo y sincronizarlos al final de c/ iteración

```
process Worker[i=1 to n] {
  while (true) {
    código para implementar la tarea i;
    esperar a que se completen las n tareas; }
}
```



Sincronización barrier: el punto de demora al final de c/ iteración es una barrera a la que deben llegar todos antes de permitirles pasar

Sincronización Barrier: Contador Compartido

n workers necesitan encontrarse en una barrera.

⇒ *cantidad* incrementado por c/ worker al llegar

⇒ cuando *cantidad* es *n*, se les permite pasar

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

Se puede implementar con:

```
FA(cantidad,1);
while (cantidad <> n) skip;
```

Problemas: *cantidad* necesita ser 0 en cada iteración, puede haber contención de memoria, coherencia de cache, ...

Sincronización Barrier: Flags y Coordinadores

Puede “distribuirse” *cantidad* usando *n* variables (arreglo *arribo*[1..n])

El await pasaría a ser ⟨ **await (arribo[1] + … + arribo[n] == n);** ⟩

Reintroduce memory contention y es ineficiente

Puede usarse un conjunto de valores adicionales y un proceso más.
⇒ ***Cada Worker espera por un único valor***

Sincronización Barrier: Flags y Coordinadores

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
# arribo y continuar son "flags"
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;
    }
}
```

Sincronización Barrier: Arboles

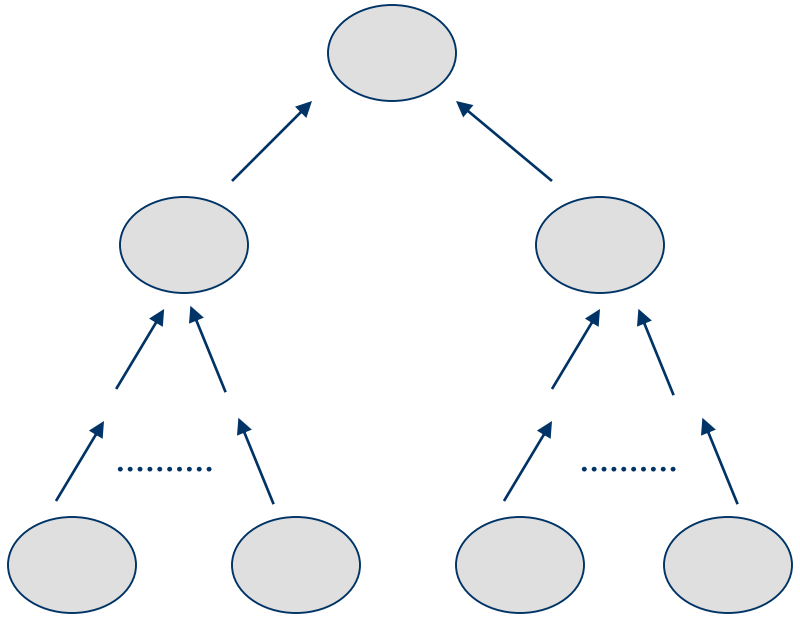
***Problemas*:**
Requiere un proceso (*y procesador*) extra.

El tiempo de ejecución del coordinador es proporcional a *n*

Posible solución: combinar las acciones de workers y coordinador, haciendo que cada worker sea también coordinador.

Por ejemplo, workers en forma de árbol: las señales de *arribo* van hacia arriba en el árbol, y las de *continuar* hacia abajo

⇒ ***combining tree barrier*** (más eficiente para *n* grande)



Sincronización Barrier: Barreras Simétricas

En *combining tree barrier* los procesos juegan diferentes roles.

Una ***barrera simétrica*** para *n* procesos se construye a partir de pares de barreras simples para dos procesos:

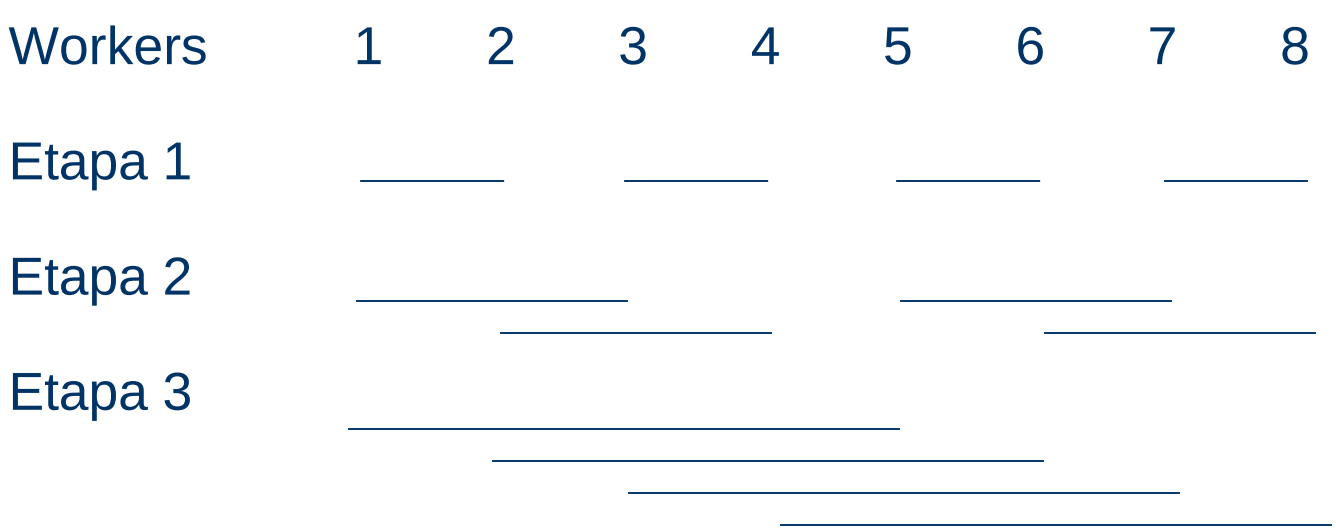
W[i]::⟨ await (arribo[i] == 0); ⟩
arribo[i] = 1;
⟨ await (arribo[j] == 1); ⟩
arribo[j] = 0;

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

Cómo se combinan para construir una barrera *n* proceso???

Worker[1:n] arreglo de procesos. Si *n* es potencia de 2 ⇒ *butterfly barrier*

Sincronización Barrier: Butterfly Barrier



$\log_2 n$ etapas: cada Worker sincroniza con uno distinto en c/ etapa

En la etapa *s*, un Worker sincroniza con otro a distancia 2^{s-1}

Cuando cada Worker pasó $\log_2 n$ etapas, todos deben haber llegado a la barrera y por lo tanto todos pueden seguir

Computaciones de Prefijo Paralelo

Algoritmos *data parallel* → varios procesos ejecutan el mismo código y trabajan en distintas partes de datos compartidos.

Ejemplo: computar en paralelo las sumas de los prefijos de un arreglo a[n], para obtener sum[n], donde sum[i] es la suma de los primeros i elementos de a.

Secuencialmente:

```
sum[0] = a[0];
for [i=1 to n-1]  sum[i] = sum[i-1] + a[i];
```

Cómo se puede paralelizar?

Computaciones de Prefijo Paralelo

Antes: Cómo se podría obtener en paralelo la suma de todos los elementos??
Idea: Sumar en paralelo pares contiguos, luego pares a distancia 2, luego a distancia 4, etc. $\Rightarrow$ ($\log_2 n$) pasos

1) Setear sum[i]=a[i]
2) En paralelo, sumar sum[i-1] a sum[i], $\forall i > 1$ (suma a distancia 1)
3) Luego, doblar la distancia, sumando sum[i-2] a sum[i], $\forall i > 2$
4) Luego de ($\log_2 n$) rondas se tienen todas las sumas parciales

valores iniciales de a[1:6]	1	2	3	4	5	6
sum después de distancia 1	1	3	5	7	9	11
sum después de distancia 2	1	3	6	10	14	18
sum después de distancia 4	1	3	6	10	15	21

Suma paralela de prefijos

```
int a[n], sum[n], viejo[n];
process Sum[i=0 to n-1] {
      int d = 1;
      sum[i] = a[i];
      barrier(i);
      while (d<n) {
            viejo[i] = sum[i];
            barrier(i);
            if ( (i-d) >= 0 ) sum[i] = viejo[i-d] + sum[i];
            barrier(i);
            d = 2 * d;
       }
}
```

Cuál sería el efecto de usar un multiprocesador sincrónico ??

Defectos de la sincronización por busy waiting

***Protocolos "busy-waiting"*: complejos y sin clara separación e*l* vbles de sincronización y las usadas para computar resultados.**

Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.

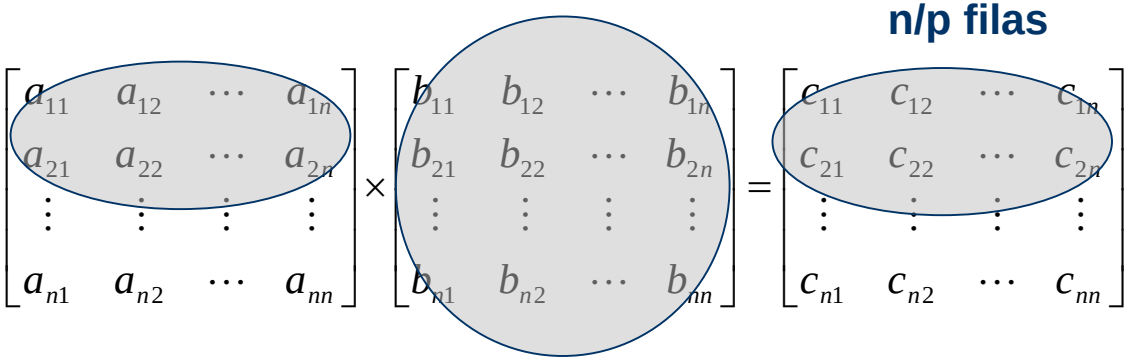
Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso *spinning* puede ser usado de manera más productiva por otro proceso.

⇒ Necesidad de *herramientas* para diseñar protocolos de sincronización.

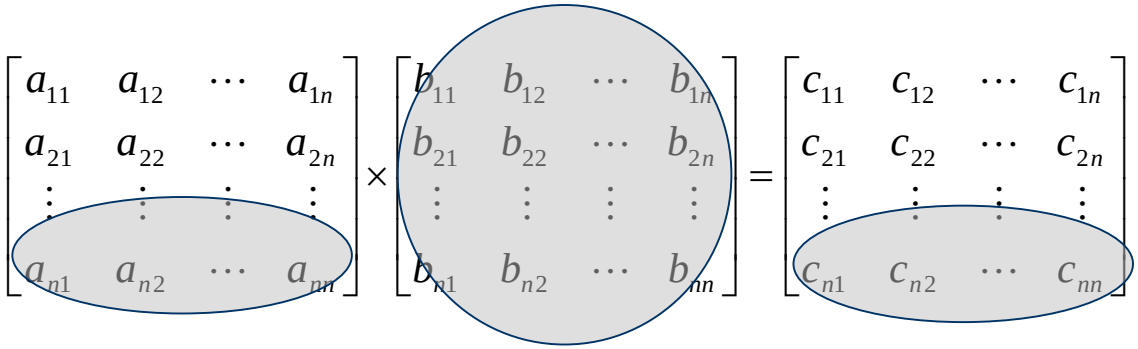
Resolución ejercicio similar al del cuestionario

En paralelo

Worker 1



Worker p



Resolución ejercicio similar al del cuestionario

P= 8, n=128. Cuántas asignaciones, sumas y productos hace cada procesador?
Si P1=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3; si P8 es 4 veces más lento, Cuánto tarda el proceso total? Qué puede hacerse para mejorar el speedup?

```
process worker [w = 1 to P] { # strips en paralelo (p strips de n/P filas)
  int first = (w-1) * n/P + 1        # Primera fila del strip
  int last = first + n/P - 1;        # Ultima fila del strip
  for [i = first to last] {
     for [j = 1 to n] {
           c[i,j] = 0.0;
           for [k = 1 to n]
              c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
     }
}
```

Resolución ejercicio similar al del cuestionario

```
process worker [w = 1 to 8] {        # 8 strips en paralelo. n/p = 16
  int first = (w-1) * 16 + 1;        # Primera fila del strip
  int last = first + 16 - 1;         # Ultima fila del strip
  # P1 = 1 a 16, P2 = 17 a 32, P3 = 33 a 48, P4 = 49 a 64
  # P5 = 65 a 80, P6 = 81 a 96, P3 = 97 a 112, P8=113 a 128
  for [i = first to last] {

     for [j = 1 to 128] {
           c[i,j] = 0.0;   #  128 asignaciones

           for [k = 1 to 128]
             c[i,j] = c[i,j] + a[i,k]*b[k,j];   #128 prods, 128 sumas, 128 asign.
        }
   }
}
```

Sin considerar los incrementos de índices:

Total = $128^2*16+128*16 = 264192$ asig,

$128^2*16 = 262144$ sumas, $128^2*16 = 262144$ prod.

Resolución ejercicio similar al del cuestionario

P1 a P8 tienen igual número de operaciones.

Total = 264192 asignaciones 262144 sumas 262144 productos

P1 =...= P7 = 264192 + 524288 + 786432 = 1574912 unid. de tiempo

Para P8 = 1574912 x 4 = 6299648 unidades de tiempo
Hay que esperar a P8 ...

Si todo fuera secuencial:
$\mathbf{128^3+128^2}$ =2113536 as. $\mathbf{128^3}$ =2097152 su $\mathbf{128^3}$ =2097152 prod

En P1 = ... = P7 = 2113536 + 4194304 + 6291456 = 12599296 unid. de tiempo

Speedup = 12599296 / 6299648 = 2

Resolución ejercicio similar al del cuestionario

Speedup = 12599296 / 6299648 = 2

Si le damos a P8 solo 2 filas y a P1 a P7 18 filas, podemos corregir los tiempos:

P1=...=P7= 297216 as. 294912 sum 294912 prod
P1=...=P7= 297216 + 589824 + 884736= 1771776 unid. de tiempo

P8 con dos filas = 787456 unid. de tiempo

Speedup = 12599296 / 1771776 = 7.1

⇒ *Mejor balance carga* ⇒ *Mejor Speedup*

⇒ *POR QUE NO el Speedup = 8 ??*

Tareas propuestas

Investigar los semáforos como herramienta de sincronización entre procesos

Buscar información sobre problemas clásicos de sincronización entre procesos y su resolución con semáforos.