

Concurrencia y Paralelismo

Clase 2



Facultad de Informática
UNLP

Links a los archivos con audio (formato MP4)

Los archivos con las clases con audio están en formato MP4. En los link de abajo están los videos comprimidos en archivos RAR.

- ◆ Sincronización por Variables Compartidas (Locks – Barreras)
https://drive.google.com/uc?id=1B_pFlBswRc19QUIz8srb9LJ2MHcDeJgc&export=download

Sincronización por Variables Compartidas

Locks - Barreras



Herramientas para la concurrencia

➤ Memoria Compartida

- Variables compartidas
- Semáforos
- Monitores

➤ Memoria distribuida (pasaje de mensajes)

- Mensajes asincrónicos
- Mensajes sincrónicos
- Remote Procedure Call (RPC)
- Rendezvous

Locks y barreras

Problema de la Sección Crítica: 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 de implementarse con instrucciones de cualquier procesador.
- Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es intercalada).
- 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;      ⇔ SC
    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

Propiedades a cumplir

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

Ausencia de Deadlock (Livelock): 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).

- Solución trivial $\langle SC \rangle$. Pero, ¿cómo se implementan los $\langle \rangle$?

El problema de la Sección Crítica

Implementación de sentencias *await*

- Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional $\langle S; \rangle \Rightarrow \text{SCEnter ; } S; \text{SCExit}$
- Para una acción atómica condicional $\langle \text{await } (B) S; \rangle \Rightarrow$
SCEnter ; while (not B) {SCExit; S; SCExit;}
- Si S es *skip*, y B cumple ASV, $\langle \text{await } (B); \rangle$ puede implementarse por medio de \Rightarrow **while (not B) skip;**

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

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

El problema de la Sección Crítica.

Solución 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;  
  }  
}
```

- 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.

Solución de “grano grueso”

bool in1=false, in2=false # MUTEX: $\neg(in1 \wedge in2)$ #

```
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 \Rightarrow 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;
  }
}
```

- ¿Satisface las 4 propiedades?

El problema de la Sección Crítica.

Solución de “grano grueso” - ¿Cumple las condiciones?

Exclusión mutua: por construcción, SC1 y SC2 se excluyen en el acceso a la SC.

bool in1=false, in2=false # MUTEX: $\neg(\text{in1} \wedge \text{in2})$ #

```
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;
  }
}
```

Ausencia de deadlock: si hay deadlock, SC1 y SC2 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 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*; $(\neg \text{in1} \wedge \text{in1} = \text{false}) \Rightarrow$ *no hay demora innecesaria*.

El problema de la Sección Crítica.

Solución de “grano grueso” - ¿Cumple las condiciones?

bool in1=false, in2=false # MUTEX: $\neg(\text{in1} \wedge \text{in2})$ #

```
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;
  }
}
```

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 \rightarrow **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.

El problema de la Sección Crítica.

Solución de “grano grueso”

bool in1=false, in2=false # MUTEX: $\neg(in1 \wedge in2)$ #

```
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;
  }
}
```

- ¿Si hay n procesos? → Cambio de variables.

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;
  }
}
```

El problema de la Sección Crítica.

Solución de “grano grueso”

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;
  }
}
```

- Generalizar la solución a n procesos

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

El problema de la Sección Crítica.

Solución de “grano fino”: *Spin Locks*

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.

¿Como funciona *Test & Set*?

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

El problema de la Sección Crítica.

Solución de “grano fino”: *Spin Locks*

```
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.

Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

El problema de la Sección Crítica.

Solución de “grano fino”: *Spin Locks*

TS escribe siempre en *lock* aunque el valor no cambie \Rightarrow 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;
```

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

Problema de la Sección Crítica.

Solución Fair: algoritmo *Tie-Breaker*

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* (2 procesos): protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales \Rightarrow más complejo.

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada \Rightarrow esta última variable es compartida y de acceso protegido.

Demora (quita prioridad) al último en comenzar su *entry protocol*.

Problema de la Sección Crítica.

Solución Fair: algoritmo *Tie-Breaker*

Solución de “*Grano Grueso*” al Algoritmo *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;
    }
}
```

Problema de la Sección Crítica.

Solución Fair: algoritmo *Tie-Breaker*

Solución de “*Grano Fino*” al 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;
    }
}
```

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 \Rightarrow 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;
  }
}
```



Problema de la Sección Crítica.

Solución Fair: algoritmo *Ticket*

Tie-Breaker n-proceso \Rightarrow complejo y costoso en tiempo.

Algoritmo *Ticket*: se reparten números y se espera a que sea el turno.

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

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

```
{ TICKET: proximo > 0 ^ ( $\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su } SC) \Rightarrow (\text{turno}[i] == \text{proximo}) \wedge$   
  ( $\text{turno}[i] > 0 \Rightarrow (\forall j: 1 \leq j \leq n, j \neq i: \text{turno}[i] \neq \text{turno}[j])$  ) ) }
```

```
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 de la Sección Crítica.

Solución Fair: algoritmo *Ticket*

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

Cumplimiento de las propiedades:

- El predicado **TICKET** es un invariante global, pues **número** es leído e incrementado en una acción atómica y **próximo** 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)

Problema de la Sección Crítica.

Solución Fair: algoritmo *Ticket*

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

¿Cómo se implementa la primera acción atómica donde se asigna el número?

- 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;  
  }  
}
```


El problema de la Sección Crítica.

Solución Fair: algoritmo *Bakery*

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

Algoritmo *Bakery*: Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna 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.

- 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.

El problema de la Sección Crítica.

Solución Fair: algoritmo *Bakery*

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

```
{BAKERY: ( $\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su SC}) \Rightarrow (\text{turno}[i] > 0) \wedge (\forall j : 1 \leq j \leq n, j \neq i: \text{turno}[j] = 0 \vee \text{turno}[i] < \text{turno}[j])$ ) ) }
```

```
process SC[i = 1 to n]
```

```
{ while (true)
```

```
    {  $\langle \text{turno}[i] = \max(\text{turno}[1:n] + 1; )$ 
```

```
      for [j = 1 to n st j <> i]  $\langle \text{await } (\text{turno}[j] == 0 \text{ or } \text{turno}[i] < \text{turno}[j]); \rangle$ 
```

```
      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.

El problema de la Sección Crítica.

Solución Fair: algoritmo *Bakery*

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

```
{BAKERY: ( $\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su } SC) \Rightarrow (\text{turno}[i] > 0) \wedge ( \forall j : 1 \leq j \leq n, j \neq i: \text{turno}[j] = 0 \vee \text{turno}[i] < \text{turno}[j] ) )$  ) }
```

```
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) )  $\rightarrow$  skip;
```

```
      sección crítica
```

```
      turno[i] = 0;
```

```
      sección no crítica
```

```
    }
```

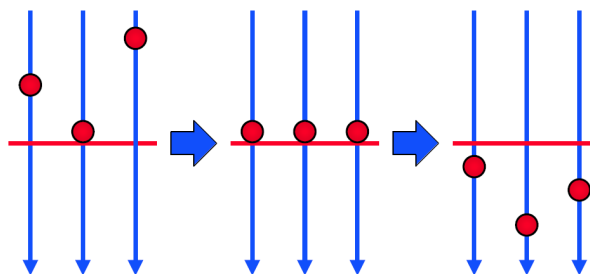
```
}
```



Sincronización *Barrier*

Sincronización barrier: una barrera es un punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución.

Dependiendo de la aplicación las barreras pueden necesitar reutilizarse más de una vez (por ejemplo en algoritmos iterativos).



Sincronización *Barrier*

Contador Compartido

n procesos necesitan encontrarse en una barrera:

- Cada proceso incrementa una variable *Cantidad* al llegar.
- Cuando *Cantidad* es n los procesos pueden 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;
```

Sincronización *Barrier*

Contador Compartido

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

¿Cuándo se reinicia Cantidad en 0 para la siguiente iteración?

Sincronización *Barrier*

Flags y Coordinadores

- Si no existe FA → Puede distribuirse *Cantidad* usando n variables (arreglo *arribo*[1.. n]).
- El *await* pasaría a ser:
 $\langle \text{await } (\text{arribo}[1] + \dots + \text{arribo}[n] == n); \rangle$
- Reintroduce contención de memoria y es ineficiente.

Puede usarse un conjunto de valores adicionales y un proceso más \Rightarrow
Cada Worker espera por un único valor

```
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;
         $\langle \text{await } (\text{continuar}[i] == 1); \rangle$ 
        continuar[i] = 0;
    }
}

process Coordinador
{ while (true)
    {   for [i = 1 to n]
        {  $\langle \text{await } (\text{arribo}[i] == 1); \rangle$ 
          arribo[i] = 0;
        }
        for [i = 1 to n] continuar[i] = 1;
    }
}
```

Sincronización *Barrier*

Flags y Coordinadores

```
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;
    }
}
```


Sincronización *Barrier*

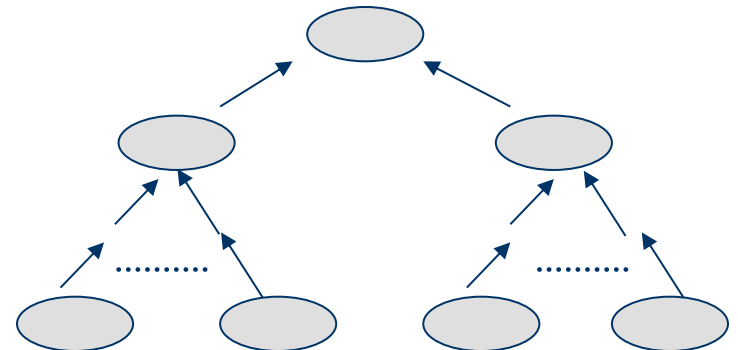
Árboles

- **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 arriba van hacia arriba en el árbol, y las de continuar hacia abajo \Rightarrow ***combining tree barrier*** (más eficiente para n grande).



Sincronización *Barrier*

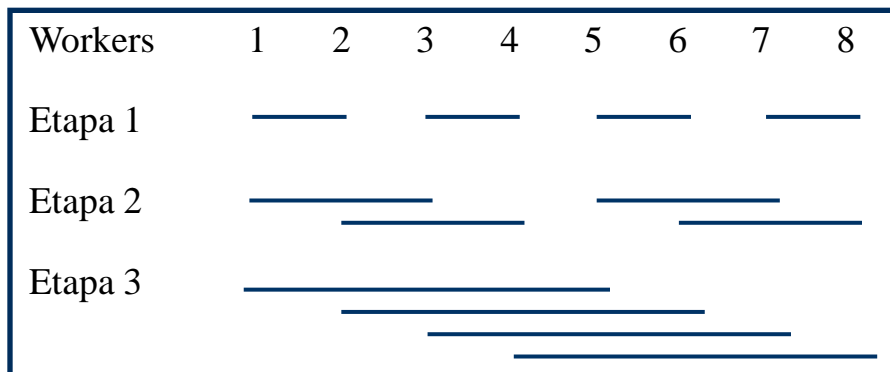
Barreras Simétrica

- 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]:: $\langle \text{await} (\text{arribo}[i] == 0); \rangle$
 $\text{arribo}[i] = 1;$
 $\langle \text{await} (\text{arribo}[j] == 1); \rangle$
 $\text{arribo}[j] = 0;$

W[j]:: $\langle \text{await} (\text{arribo}[j] == 0); \rangle$
 $\text{arribo}[j] = 1;$
 $\langle \text{await} (\text{arribo}[i] == 1); \rangle$
 $\text{arribo}[i] = 0;$

- ¿Cómo se combinan para construir una barrera n proceso? *Worker[1:n]* arreglo de procesos. Si n es potencia de 2 \Rightarrow *Butterfly Barrier*.



- $\log_2 n$ etapas: cada *worker* sincroniza con uno distinto en cada etapa.
- En la etapa s , un *worker* sincroniza con otro a distancia 2^{s-1} .
- Cuando cada *worker* pasó $\log_2 n$ etapas, todos pueden seguir.

Sincronización *Barrier*

Barreras Simétrica – *Butterfly barrier*

```
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.
  }
}
```

Defectos de la sincronización por *busy waiting*

- Protocolos “*busy-waiting*”: complejos y sin clara separación entre variables 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.