Lección 7: Sincronización de procesos
El problema de la sección crítica

- Metodología de resolución de problemas de sincronización
- El problema de la sección crítica
- Especificaciones de diseño
- Esquema algorítmico preliminar para dos procesos
- Sincronización entre procesos
- Esquema algorítmico simplificado
- Implementación del “**await**”
- Algoritmos para la sección crítica
  - algoritmo de Peterson (del “tie-breaker”)
  - algoritmo por turno de espera
  - algoritmo de la panadería

Metodología de resolución de problemas de sincronización

- Objetivo de la sincronización: evitar interferencias
- Herramienta conceptual: instrucción ***await***
  - muy general
  - costosa de implementar
- Objetivos:
  - encontrar implementaciones eficientes del ***await***
    - usando primitivas de bajo nivel (hard/soft)
  - desarrollar una metodología para atacar, de una manera sistemática, el diseño de programas concurrentes
- Idea: usaremos “profusamente” invariantes globales
  - veremos los procesos como “mantenedores” del invariante

Metodología de resolución de problemas de sincronización

- Una metodología para sincronizar procesos:
  - Definir precisamente el problema:
    - identificar los procesos que intervienen
    - definir las variables compartidas utilizadas para sincronizar procesos
    - escribir un predicado invariante que, relacionando variables, exprese las restricciones de sincronización
  - Proponer un esquema de solución:
    - esquema algorítmico preliminar
  - Garantizar la satisfacción del invariante
    - protegiendo con guardas y sincronizando las acciones atómicas
  - Implementar las acciones atómicas
    - mediante instrucciones secuenciales y primitivas de sincronización

El problema de la sección crítica

- Uno de los “clásicos” (planteado por Dijstra, 1976)
- Enunciado
  - "n" procesos ejecutan repetidamente una sección crítica, SC, seguida de una sección no crítica, SNC. La **sección crítica**de un proceso es un secuencia de acciones que **debe ser ejecutada** en **exclusión mutua** con las secciones críticas del resto de procesos
    - cuando un proceso entra en SC ó SNC, sale
- Esquema del problema:

```
P(i:1..n)::
Mq TRUE
   protocoloDeEntrada
   secciónCrítica
   protocoloDeSalida
   secciónNoCrítica
FMq
```

Notar que servirá para implementar ***<await ...>***

El problema de la sección crítica

- **Objetivo**: Diseñar los protocolos de entrada y salida a la sección crítica de forma que se satisfagan los siguientes requisitos:
  - R1) **Exclusión mutua**: como máximo un proceso puede estar ejecutando su sección crítica
  - R2) **Ausencia de bloqueos**: si dos o más procesos tratan de acceder a su sección crítica, al menos uno lo logrará
  - R3) **Ausencia de retrasos innecesarios**: si ningún proceso está en la SC y uno solicita entrar, puede entrar sin esperar
  - R4) **Comportamiento equitativo**: todo proceso que desee entrar, tarde o temprano entrará

Una solución de grano grueso

- Esquema para dos procesos:

```
Vars in1:Bool:=FALSE
     in2:Bool:=FALSE
--MUTEX:¬(in1∧in2)
P1::                      P2::
Mq TRUE                   Mq TRUE
   in1:=TRUE                 in2:=TRUE
   secCrit1                  secCrit2
   in1:=FALSE                in2:=FALSE
   secNoCrit1                secNoCrit2
FMq                       FMq
```

__NO__ es correcta

Una solución de grano grueso

- Una solución:
  - atomizar
- ¿B1,B1′?
- ¿B2,B2′?

```
Vars in1:Bool:=FALSE
     in2:Bool:=FALSE
--MUTEX:¬(in1∧in2)
```

```
P1::
Mq TRUE
   --MUTEX∧¬in1
   <await (B1)
     in1:=TRUE >
   --MUTEX∧in1
   secCrit1
   <await (B1')
     in1:=FALSE >
   --MUTEX∧¬in1
   secNoCrit1
FMq
```

```
P2::
Mq TRUE
   --MUTEX∧¬in2
   <await (B2)
     in2:=TRUE >
   --MUTEX∧in2
   secCrit2
   <await (B2')
     in2:=FALSE >
   --MUTEX∧¬in2
   secNoCrit2
FMq
```

Una solución de grano grueso

- Una solución buena
- Notar que aún es de grano grueso

```
Vars in1:Bool:=FALSE
     in2:Bool:=FALSE
--MUTEX:¬(in1∧in2)
P1::
Mq TRUE
   --MUTEX∧¬in1
   <await (¬in2)
     in1:=TRUE >
   --MUTEX∧in1
   secCrit1
   in1:=FALSE
   --MUTEX∧¬in1
   secNoCrit1
FMq
```

```
P2::
Mq TRUE
   --MUTEX∧¬in2
   <await (¬in1)
     in2:=TRUE >
   --MUTEX∧in2
   secCrit2
   in2:=FALSE
   --MUTEX∧¬in2
   secNoCrit2
FMq
```

- *Cumple R1,R2,R3*
- *¿Qué es necesario para R4?*

Una solución de grano fino

- El estado que realmente me interesa

$$\neg(in1 \wedge in2) \leftrightarrow \neg in1 \vee \neg in2)$$

- Solución alternativa:

```
Vars ocup:Bool:=FALSE
--MUTEX:(ocup=¬(in1∧in2))
P1::                          P2::
Mq TRUE                       Mq TRUE
    <await (¬ocup)                <await (¬ocup)
      ocup:=TRUE >                  ocup:=TRUE >
    secCrit1                      secCrit2
    ocup:=FALSE                   ocup:=FALSE
    secNoCrit1                    secNoCrit2
FMq                           FMq
```

•fácilmente generalizable para n procesos
•fácilmente implementable como grano fino

Una solución de grano fino

- Muchos procesadores cuentan con instrucciones del tipo "test-and-set"

```
TS(cer,val):<val:=cer;cer:=TRUE>
```

```
Vars ocup:Bool:=FALSE
P1::
Vars val:Bool
Mq TRUE
   TS(ocup,val)
   Mq val
    TS(ocup,val)
   FMq
   secCrit1
   ocup:=FALSE
   secNoCrit1
FMq
```

```
P2::
Vars val:Bool
Mq TRUE
   TS(ocup,val)
   Mq val
    TS(ocup,val)
   FMq
   secCrit2
   ocup:=FALSE
   secNoCrit2
FMq
```

Solución tipo "spin-lock" en torno a la var. "ocup"

- Cumple las propiedades
- Válido para cualquier número de procesos

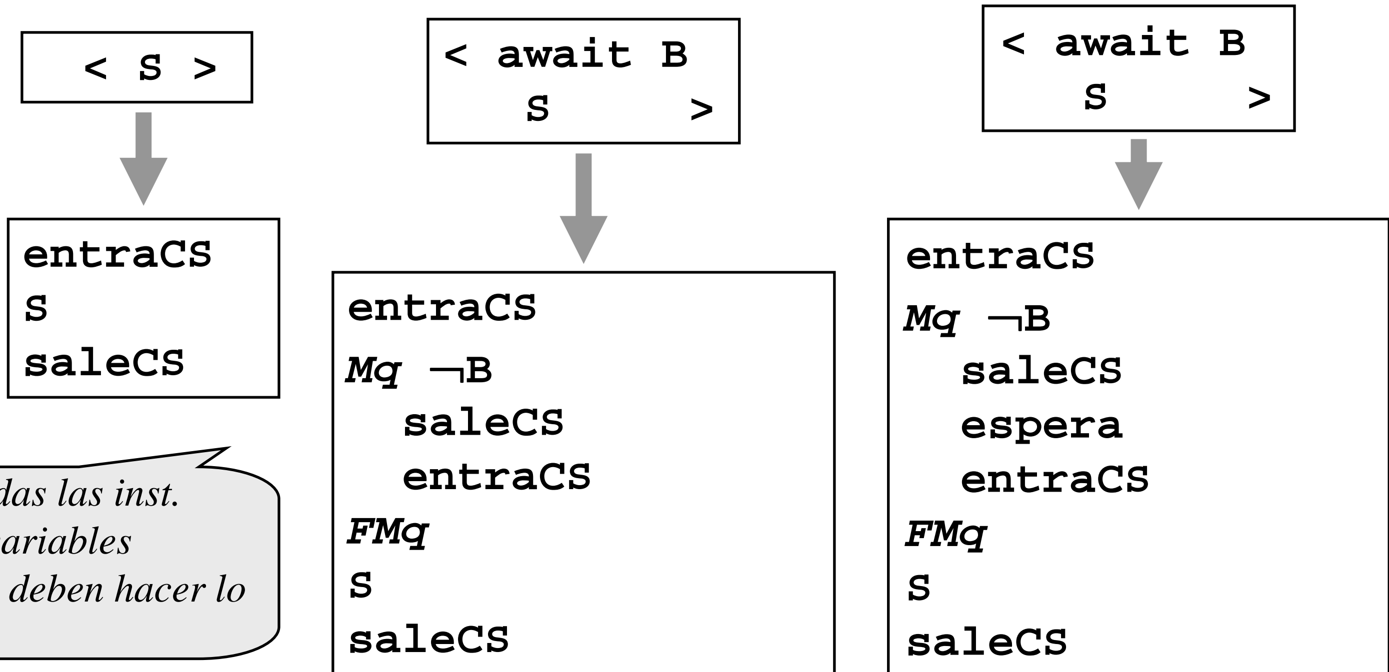
Una solución de grano fino

- "kit" (parcial) de supervivencia (J.L. Briz):

```
▪Tipo test&set
    •IBM370, M68040, VAX, [SPARC]
▪Tipo compare & swap
    •IBM 370, Pentium
▪Tipo swap-atomic
    •SPARC, MC88100
▪Tipo load-locked/store-conditional
    •Alpha, MIPS (>= R4000)
▪Tipo Fetch & add
    •CONVEX
```

Una implementación general del "await"

- Cualquier solución al problema de la sección crítica se puede usar para implementar la instrucción "**await**"



El algoritmo de Peterson

- soluciones basadas en “test-and-set” requieren
  - que tales instrucciones existan en el procesador
  - schedulers fuertemente equitativos
- el algoritmo de Peterson [1981] (“tie-breaker”)
  - supera esas restricciones
  - es bastante más complejo
  - será suficiente con un scheduler **débilmente equitativo**

El algoritmo de Peterson

- El problema consiste en implementar:

```
--MUTEX∧¬in1
<await (¬in2)
   in1:=TRUE >
--MUTEX∧in1
```

```
--MUTEX∧¬in1
Mq in2
   seguir
MFq
in1:=TRUE
--MUTEX∧in1
```

¿?

```
--MUTEX∧¬in1
in1:=TRUE
Mq in2
   seguir
MFq
--MUTEX∧in1
```

¿?

El algoritmo de Peterson

- Solución de grano grueso (para dos procesos):

```
Vars en1:Bool:=FALSE
     en2:Bool:=FALSE
     ult:Ent:=1
```

```
P1::
Mq TRUE
   en1:=TRUE;ult:=1
   <await ¬en2 ∨ ult=2>
   secCrit1
   en1:=FALSE
   secNoCrit1
FMq
```

```
P2::
Mq TRUE
   en2:=TRUE;ult:=2
   <await ¬en1 ∨ ult=1>
   secCrit2
   en2:=FALSE
   secNoCrit2
FMq
```

El algoritmo de Peterson

- Solución de grano fino:

```
Vars en1:Bool:=FALSE
     en2:Bool:=FALSE
     ult:Ent:=1
```

```
P1::
Mq TRUE
   en1:=TRUE
   ult:=1
   Mq en2 ∧ ult=1
      seguir
   FMq
   secCrit1
   en1:=FALSE
   secNoCrit1
FMq
```

```
P2::
Mq TRUE
   en2:=TRUE
   ult:=2
   Mq en1 ∧ ult=2
      seguir
   FMq
   secCrit2
   en2:=FALSE
   secNoCrit2
FMq
```

El algoritmo de Peterson

P1::	P2::
Mq TRUE	*Mq* TRUE
en1:=TRUE	en2:=TRUE
ult:=1	ult:=2
--in1	--in2
Mq en2 ∧ ult=1	*Mq* en1 ∧ ult=2
seguir	*seguir*
FMq	*FMq*
--in1∧	--in2∧
--(¬in2∨ ult=2)	--(¬in1∨ ult=1)
secCrit1	secCrit2
en1:=FALSE	en2:=FALSE
secNoCrit1	secNoCrit2
FMq	*FMq*

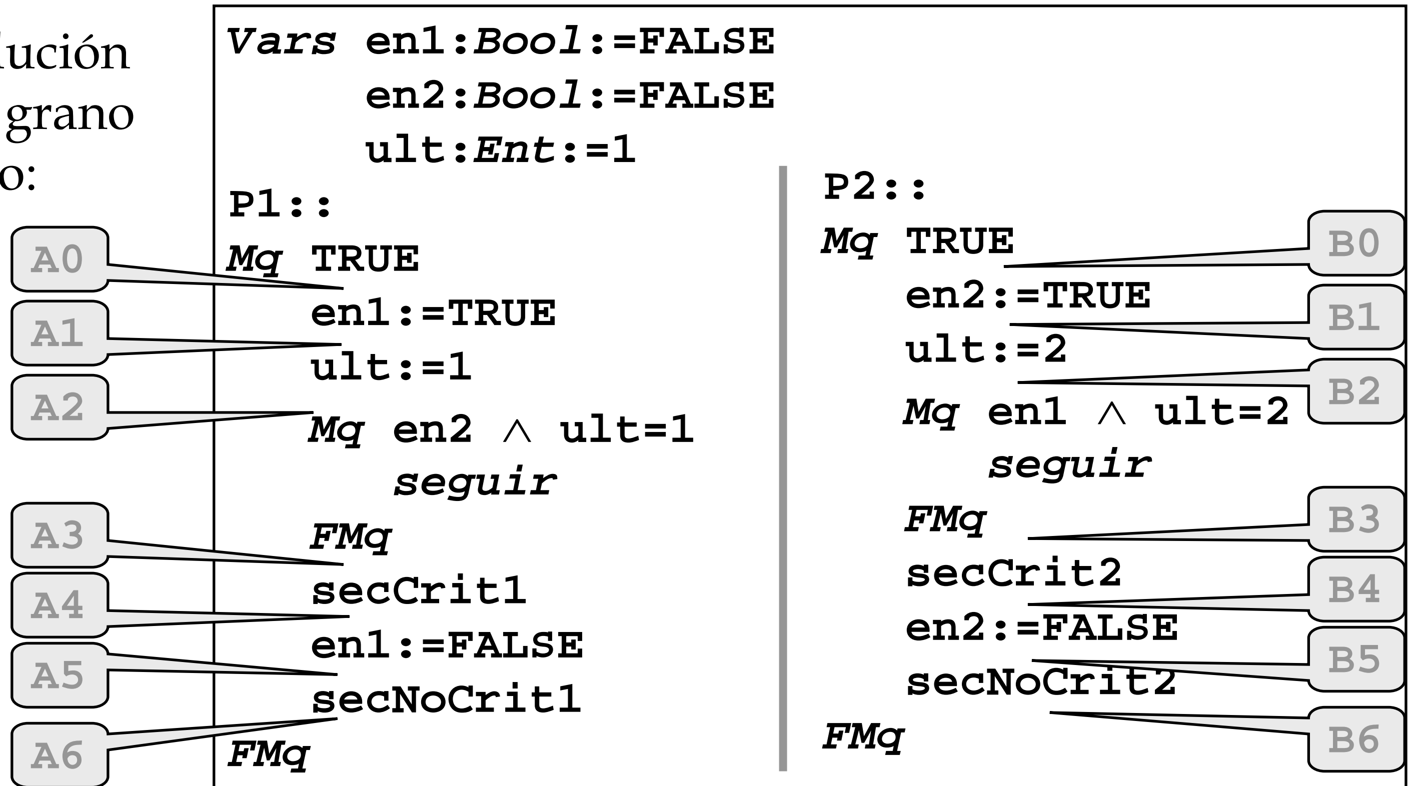
El algoritmo de Peterson

```
P1::
Mq TRUE
   <en1:=TRUE;mid1:=TRUE>
   <ult:=1;mid1:=FALSE>
   --in1∧¬mid1
   Mq en2 ∧ ult=1
      seguir
   FMq
   --in1∧¬mid1∧
   --(¬in2∨ ult=2∨ mid2)
   secCrit1
   en1:=FALSE
   secNoCrit1
FMq
```

```
P2::
Mq TRUE
   <en2:=TRUE;mid2:=TRUE>
   <ult:=2;mid2:=FALSE>
   --in2∧¬mid2
   Mq en1 ∧ ult=2
      seguir
   FMq
   --in2∧¬mid2∧
   --(¬in1∨ ult=1∨ mid1)
   secCrit2
   en2:=FALSE
   secNoCrit2
FMq
```

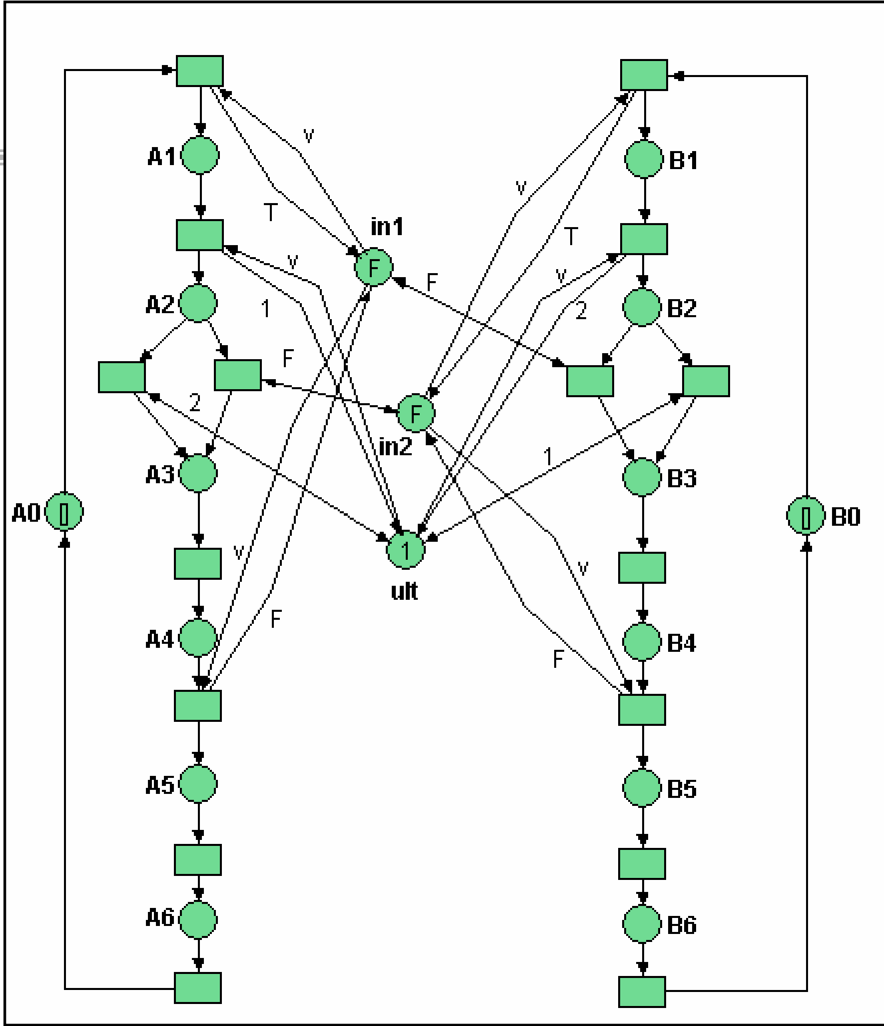
El algoritmo de Peterson

- Solución de grano fino:



El algoritmo de Peterson

- ¿Qué deben cumplir los estados alcanzables para asegurar la exclusión mutua?



•*62 nodos*
•*116 transiciones*

El algoritmo de Peterson

- Versión para **n** procesos

```
Vars en:vector(1..n) de Ent:=(1..n,0)
     ult:vector(1..n) de Ent:=(1..n,0)
P(i:1..n)::
Mq TRUE
   Pt j:=1..n-1
      en(i):=j
      ult(j):=i
      Pt k:=1..n tq i<>k
         Mq en(k)>=en(i) ∧ ult(j)=i
            seguir
         FMq
      FPt
   FPt
   secCrit
   en(i):=0
   secNoCrit
FMq
```

- *en(i): etapa que ejecuta Pi*
- *ult(j): ult. proceso que entró end etapa j*

El algoritmo por turno de espera

- Solución de grano grueso (para n procesos):

```
Vars num:Ent:=1  --el dispensador
     sig:Ent:=1  --el panel
     turno:vector(1..n) de Ent:=(1..n,0)
--TICK: ∀ α∈{1..n}.[ (P(α) en secCrit → (turno(α)=sig)) ∧
--   (∀ β∈{1..n}\{α}.(turno(α)=0 ∨ turno(α)<>turno(β))) ∧
P(i:1..n)::                        -- turno(α) < num ∧...]
Mq TRUE
   <turno(i):=num;num:=num+1>
   <await (turno(i)=sig)>
   secCrit
   <sig:=sig+1>
   secNoCrit
FMq
```

Objetivo: asegurar que cada proceso tenga un único turno

El algoritmo por turno de espera

```
Vars num:Ent:=1
     sig:Ent:=1
     turno:vector(1..n) de Ent:=(1..n,0)
--TICK: ∀ α∈{1..n}.[ (P(α) en secCrit → (turno(α)=sig)) ∧
--   (∀ β∈{1..n}\{α}.(turno(α)=0 ∨ turno(α)<>turno(β))) ∧
P(i:1..n)::                                  -- turno(α) < num ∧... ]
Mq TRUE
   turno(i):=FA(num,1)
   Mq (turno(i)<>sig)
      seguir
   FMq
   secCrit
   sig:=sig+1
   secNoCrit
FMq
```

```
FA(var,inc):<
   tmp:=var;var:=var+inc;Dev(tmp)
>
```

El algoritmo de la panadería

- Una solución de grano grueso

L. Lamport, 1974

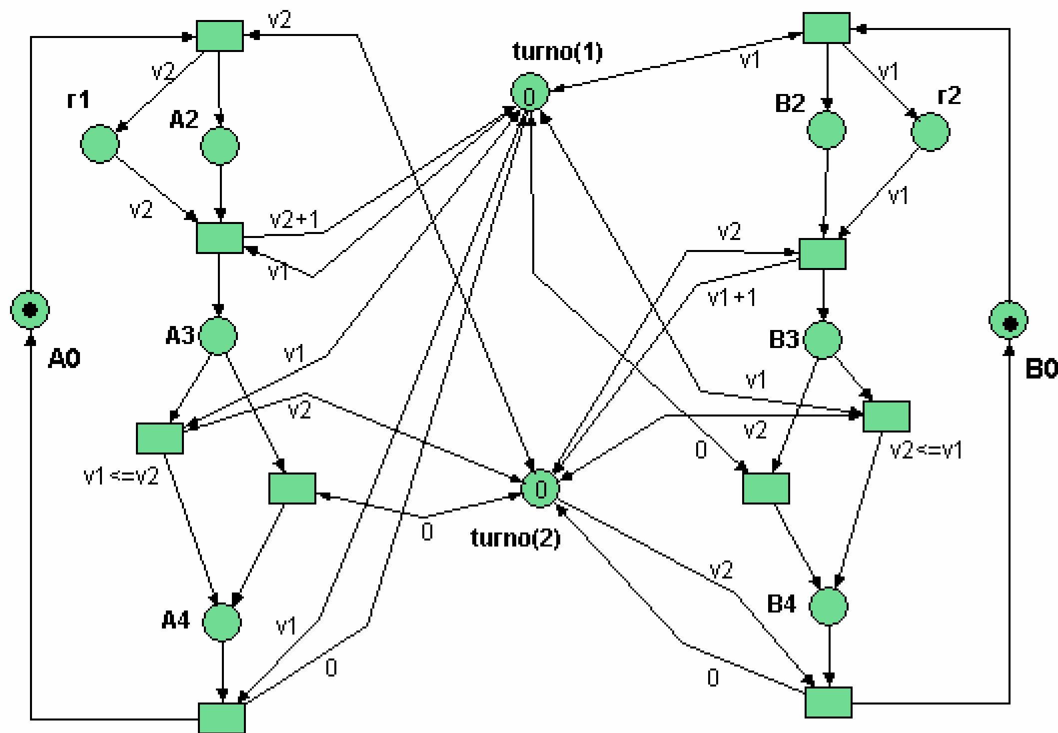
```
Vars turno:vector(1..n) de Ent:=(1..n,0)
--TUR: (P(i) en secCrit → (turno(i)<>0 ∧
--     ∀j∈{1..n},i<>j.(turno(j)=0 ∨ turno(i)<turno(j)))
P(i:1..n)::
Mq TRUE
   <turno(i):= max{turno(1),...,turno(n)}+1>
   Pt j:=1..n tq j<>i
     <await turno(j)=0 ∨ turno(i)<turno(j) >
   FPt
   secCrit
   <turno(i):=0>
   secNoCrit
FMq
```

El algoritmo de la panadería

- Una solución de grano fino, mala (para n=2)

```
     Vars turno1:vector(1..2) de Ent:=(1..2,0)
P1::                                P2::
Mq TRUE                             Mq TRUE
   turno(1):= turno(2)+1               turno(2):= turno(1)+1
   Mq turno(2)<>0 ∧                    Mq turno(1)<>0 ∧
       turno(1)>turno(2)                   turno(2)>turno(1)
      seguir                              seguir
   FMq                                 FMq
   secCrit                             secCrit
   turno(1) := 0                       turno(2) := 0
   secNoCrit                           secNoCrit
FMq                                 FMq
```

El algoritmo de la panadería




```
prod panaderia.init
panaderia
../../bin/probe panaderia    --ojo al path
#statistics
   Number of nodes: 52
   Number of arrows: 86
#query node ( (A_4==<..>) && (B_4==<..>) )
      27,      44,      47
     3 paths
#look 27
Node 27
  A_4: <..>
  B_4: <..>
  turno_1: <.1.>
  turno_2: <.1.>
------------------------------------------------
#look 44
Node 44
  A_4: <..>
  B_4: <..>
  turno_1: <.1.>
  turno_2: <.2.>
------------------------------------------------
#look 47
Node 47
  A_4: <..>
  B_4: <..>
  turno_1: <.2.>
  turno_2: <.1.>
```

El algoritmo de la panadería

```
Vars turno:vector(1..2) de Ent:=(1..2,0)
```

```
P1::
Mq TRUE
   turno(1):=1
   turno(1):=turno(2)+1
   Mq turno(2)<>0 ∧
       turno(1)>turno(2)
      seguir
   FMq
   secCrit
   turno(1):=0
   secNoCrit
FMq
```

```
P2::
Mq TRUE
   turno(2):=1
   turno(2):=turno(1)+1
   Mq turno(1)<>0 ∧
       turno(2)>=turno(1)
      seguir
   FMq
   secCrit
   turno(2):=0
   secNoCrit
FMq
```

El algoritmo de la panadería

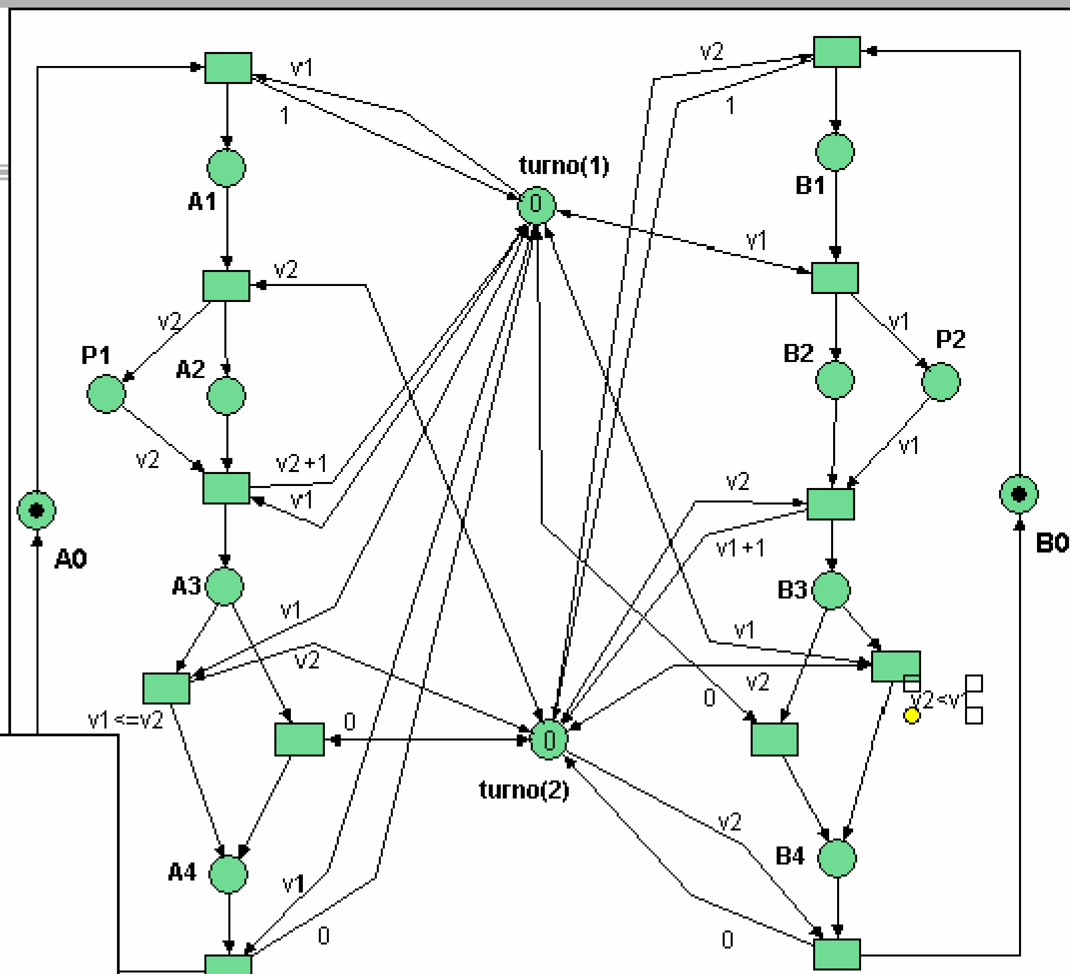
```
Vars turno:vector(1..2) de Ent:=(1..2,0)

P1::
Mq TRUE
   ...
   Mq turno(2)<>0 ∧
       turno(1)>turno(2)
      seguir
   FMq
   --t(1)>0 ∧
   --(t(2)=0 ∨ t(1)<=t(2))
   ...
FMq

P2::
Mq TRUE
   ...
   Mq turno(1)<>0 ∧
       turno(2)>=turno(1)
      seguir
   FMq
   --t(2)>0 ∧
   --(t(1)=0 ∨ t(2)<t(1))
   ...
FMq
```

¡NO!

El algoritmo de la panadería




```
prod panaderia.init
panaderia
../../bin/probe panaderia
#statistics
    Number of nodes: 55
    Number of arrows: 84
#query node ( (A_4==<..>) && (B_4==<..>) )
       0 paths
```

El algoritmo de la panadería

```
Vars turno:vector(1..n) de Ent:=(1..n,0)
--TUR: (P(i) en secCrit → (turno(i)<>0 ∧
--      ∀j∈{1..n},i<>j.(turno(j)=0 ∨ turno(i)<turno(j) ∨
        (turno(i)=turno(j) ∧ i<j) ))
P(i:1..n)::
Mq TRUE
   turno(i):=1;turno(i):=max{turno(1),...,turno(n)}+1
   Pt j:=1..n tq j<>i
     Mq turno(j)<>0 ∧ (turno(i),i)">"(turno(j),j)
        seguir
     FMq
   FPt
   secCrit
   turno(i):=0
   secNoCrit
FMq
```

$(a,b)">"(a',b')::$

$a>a' \vee (a=a' \wedge b>b')$