

Lección 1: Introducción a la Programación de Sistemas Concurrentes y Distribuidos

- De la programación secuencial a la concurrente y distribuida
- Programas secuenciales, concurrentes y distribuidos
- La asignatura de PSCD
 - organización
 - bibliografía recomendada
 - contenidos del curso
 - notación algorítmica

De la programación secuencial a la concurrente y distribuida

- Caso 1:

```
integer x := 0  
--x = 0  
x := 1  
--x = 1
```

```
integer x := 0  
--x = 0  
x := 2  
--x = 2
```



```
integer x := 0  
--x = 0  
x := 1 || x := 2  
--x = ??????
```

De la programación secuencial a la concurrente y distribuida

- Caso 2:

```
integer x := 0  
--x = 0  
x := x + 1  
--x = 1
```

```
integer x := 0  
--x = 0  
x := x + 1  
--x = 1
```



```
integer x := 0  
--x = 0  
x := x + 1 || x := x + 1  
--x = ??????
```

Programas secuenciales, concurrentes y distribuidos

- Un programa concurrente se compone de *procesos* y *objetos compartidos*
- Un proceso es un *programa secuencial* ejecutado en algún procesador
- Los objetos compartidos se construyen o bien mediante *memoria compartida* o bien mediante una *red de comunicación*

Programas secuenciales, concurrentes y distribuidos

- Para cooperar, los procesos deben comunicar
 - la comunicación permite a un proceso influir en la ejecución de otro
- La comunicación puede tener retrasos
 - lo que implica que la información obtenida por un proceso respecto a otro puede estar desfasada
- En consecuencia: **desarrollar programas concurrentes/distribuidos correctos es más difícil que diseñar un conjunto de procesos correctos**

La asignatura de PSCD

- Dónde encaja en nuestro plan de estudios

PLAN DE ESTUDIOS - GRADO EN INGENIERÍA		
Cuatrimestre 1º	Tipo	Créds.
INTRODUCCIÓN A LOS COMPUTADORES	FB	6
FUNDAMENTOS DE ADMINISTRACIÓN DE EMPRESAS	FB	6
MATEMÁTICAS 1	FB	6
MATEMÁTICAS 2	FB	6
PROGRAMACIÓN 1	FB	6
Cuatrimestre 3º	Tipo	Créds.
TEORÍA DE LA COMPUTACIÓN	FB	6
SISTEMAS OPERATIVOS	OB	6
REDES DE COMPUTADORES	OB	6
PROGRAMACIÓN DE SISTEMAS CONCURRENTES Y DISTRIBUIDOS	OB	6
ESTRUCTURA DE DATOS Y ALGORITMOS	OB	6



La asignatura de PSCD. Resultados del aprendizaje

- El estudiante terminará con un **conocimiento** profundo de cuáles son las características específicas de los **sistemas concurrentes y distribuidos**
- Conocerá los **problemas** generados por el acceso concurrente a datos y recursos, así como las **soluciones conceptuales y tecnológicas** que se han dado a los mismos
- Tendrá nociones de qué son los sistemas **tiempo real**, y sistemas **basados en eventos**
- Conocerá **herramientas** para el diseño y programación de programas con características concurrentes y/o distribuidas

La asignatura de PSCD. Trabajo del estudiante

- 60 horas de actividades presenciales
 - sesiones de teoría, problemas y prácticas de laboratorio
- 85 horas de trabajo y estudio individual efectivo
 - estudio de textos, resolución de problemas, preparación de clases y prácticas, desarrollo de programas, etc.
- 5 horas dedicadas a distintas pruebas de evaluación

La asignatura de PSCD. Sesiones de problemas

- No hay horario específico
 - 15 sesiones con ejercicios programados
 - Planificación en Moodle (ambos grupos)
- Metodología de trabajo:
 - Trabajo previo en casa
 - Un rato de trabajo en grupo en el aula
 - Un rato de desarrollo de la solución

La asignatura de PSCD. Evaluación

- **Actividades de evaluación**
 - Prueba escrita (100%): Se plantearán cuestiones y/o problemas relacionados con el programa impartido en la asignatura, tanto relativos a los contenidos de clases de teoría y problemas como a las prácticas de laboratorio.
- **Calificación final**
 - En ambas convocatorias, para superar la asignatura, habrá que obtener una puntuación mayor o igual que 5.0.

La asignatura de PSCD. Cuestiones y fechas de interés

- Todos los materiales del curso estarán disponibles en **Moodle**
 - Transparencias de clase (**NO son apuntes, solo apoyo para la clase**)
 - Materiales de apoyo, enunciados de prácticas, etc.
- **Si no estás matriculado** en la asignatura, hay que auto-matricularse en el curso **antes del 4 de septiembre a las 17:00**
 - **Importante:** siempre con la cuenta **@unizar.es**
 - Clave = **Pscd_25_26!**

La asignatura de PSCD. Cuestiones y fechas de interés

- Creación de **grupos de prácticas**
 - Vía Moodle a partir del **4 de septiembre a las 17:00h hasta el 8 de septiembre a las 12:00h**
 - **Importante:** los estudiantes del turno de mañana deberán apuntarse en grupos de mañana y los del turno de tarde en grupos de tarde
 - Los **casos (muy) excepcionales** contactar con el profesor **Pedro Álvarez** vía correo electrónico (alvaper@unizar.es) **antes del 8 de septiembre a las 20:00h**, indicando claramente quién eres y justificando el motivo de la solicitud

La asignatura de PSCD. Prácticas de laboratorio

- Fechas de **inicio de las prácticas**:
 - Grupos de martes A: 16 de septiembre
 - Grupos de martes B: 23 de septiembre
 - Grupos de miércoles A: 10 de septiembre
 - Grupos de miércoles B: 17 de septiembre
 - Grupo de jueves A: 11 de septiembre
 - Grupo de jueves B: 18 de septiembre
- **Laboratorios** (en el Ada Byron):
 - L2.11: Grupos de martes A y B de 18:00 a 20:00
 - L0.03: Grupos de miércoles A de 8:00 a 10:00, de 17:00 a 19:00
 - L1.02: Grupos de miércoles B de 17:00 a 19:00
 - L0.01: todos los demás grupos

La asignatura de PSCD. Profesorado

- **Pedro Álvarez**

-alvaper@unizar.es, despacho 2.16, ext. 5541

- **Joaquín Ezpeleta**

-ezpeleta@unizar.es, despacho 1.17, ext. 1955

- **Simona Bernardi**

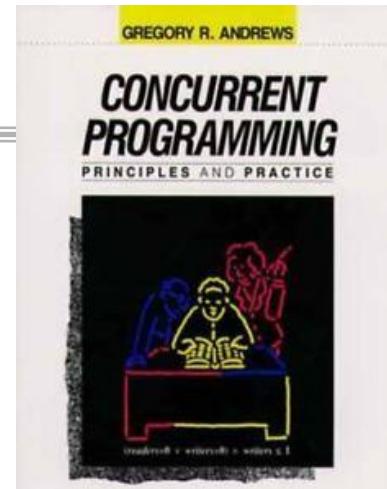
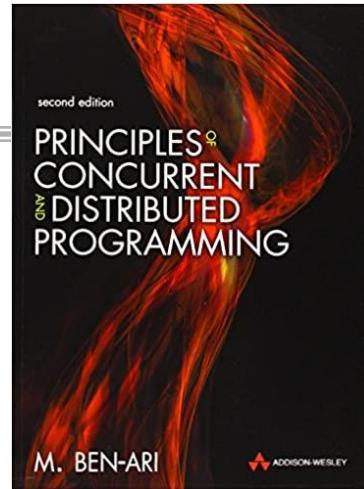
- simonab@unizar.es, despacho 2.12, ext. 5531

- **Pendiente de contratar**

- Las tutorías se gestionarán vía Google Calendar del profesor (disponible en Moodle)

Bibliografía recomendada

- M. Ben-Ari
Principles of Concurrent and Distributed Programming
Addison-Wesley, 2006
- G.R. Andrews
Concurrent Programming. Principles and practice
Addison-Wesley, 2006
- A. Williams
C++. Concurrency in Action
Manning, 2012
- M.L. Liu
Computación distribuida. Fundamentos y aplicaciones
Ed. Pearson- Addison Wesley, 2004.



Contenidos del curso

Lección 1: Introducción a la programación concurrente

Lección 2: La programación concurrente

Lección 3: Sincronización de procesos. El problema de la sección crítica

Lección 4: Breve introducción a la lógica temporal y al “model checking”

Lección 5: Diseño de programas concurrentes

Lección 6: Sincronización de procesos mediante semáforos

Lección 7: Sincronización de procesos mediante monitores

Contenidos del curso

Lección 8: Introducción a la programación distribuida

Lección 9: Programación mediante paso síncrono de mensajes

Lección 10: Coordinación mediante espacios de tuplas

Lección 11: Algoritmos distribuidos

Lección 12: Algoritmos de consenso

Lección 13: Introducción a la programación de sistemas de tiempo real

Lección 14: Programación dirigida por eventos

Contenidos del curso. Trabajo de laboratorio

1. La programación concurrente. Threads y datos compartidos.
Problemas de interferencias
2. Sincronización mediante esperas activas
3. Programación con semáforos
4. Programación con monitores
5. Programación de sistemas distribuidos-I
6. Programación de sistemas distribuidos-II

Notación algorítmica

```
constant integer i := 27
integer v := 27
real r
constant real PI := 3.1415926535 --el de siempre
boolean ha_ido_bien

integer array[1..100] mis_datos := (1..100, 0)

type integer array[1..n,1..n] mat_cuad
type integer array[1..n] vect
mat_cuad A
vect b,x
```

```
process multiplica
    mat_cuad A
    vect b, x
-----
operation obtener_mat (REF mat_cuad m)
    --Pre: TRUE
    --Post: m se ha leído de la entrada estándar
-----
operation obtener_vect (REF vect v)
    --Pre: TRUE
    --Post: v se ha leído de la entrada estándar
-----
    ...
end
```

.n
 $x[i] + A[i, j] * b[j]$

end

end

```
process multiplica
```

```
mat_cuad A
```

```
vect b, x
```

```
operation obtener_mat (REF mat_cuad m)
```

```
--Pre: TRUE
```

```
--Post: m se ha leido d
```

```
operation obtener_vec
```

```
--Pre: TRUE
```

```
--Post: v se ha leido d
```

```
....
```

```
end
```

```
process multiplica
```

```
....
```

```
for i:= 1..n
```

```
x[i] := 0
```

```
for j := 1..n
```

```
x[i] := x[i]+A[i,j]*b[j]
```

```
end
```

```
end
```

```
for i:= 1..n
```

```
write(x[i])
```

```
end
```

```
end
```

```
operation obtener_mat(REF mat_cuad m)
    --Pre: TRUE
    --Post: m se ha leído de la entrada estándar

        for i := 1..n
            for j := 1..n
                read(m[i, j])
            end
        end
    end
```

```
operation obtener_vect(REF vect v)
    --Pre: TRUE
    --Post: v se ha leído de la entrada estándar

        integer i := 1
        while i <= n
            read(v[i])
            i := i+1
        end
    end
```

Lección 2

La Programación Concurrente

- Definición de programa concurrente
- Ejecución de un programa concurrente
- Representación de la ejecución de un programa concurrente
- “Entrelazado” arbitrario de acciones atómicas
- Corrección de un programa concurrente
- Propiedades de corrección
- Equidad de un programa concurrente
- Una manera de modelar sistemas concurrentes

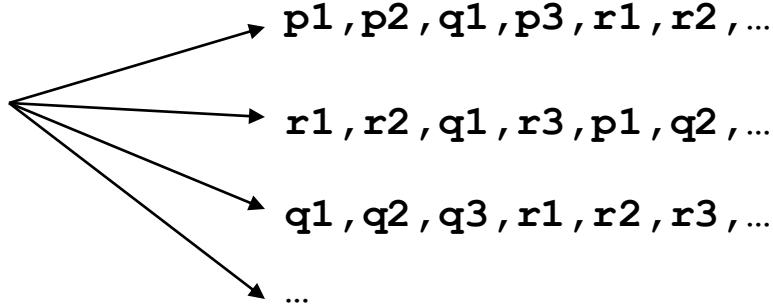
Definición de programa concurrente

- **Proceso:** programa que ejecuta una secuencia de acciones
 - programa secuencial
- **Programa concurrente:** programa en el que intervienen dos o más procesos secuenciales que cooperan en la realización de una tarea
 - procesos
 - objetos compartidos
- **Cooperar implica comunicar**
 - mediante memoria compartida
 - mediante paso de mensajes

Ejecución de un programa concurrente

- La ejecución de un programa concurrente: entrelazado de las acciones atómicas de sus procesos
 - Cada secuencia de ejecución define una historia
 - El número de historias puede ser muy elevado
 - La sincronización de procesos restringe el número de posibles historias de ejecución y (debe) evita(r) las no deseadas

P	Q	R
p1	q1	r1
p2	q2	r2
p3	q3	r3
...



Definición de programa concurrente

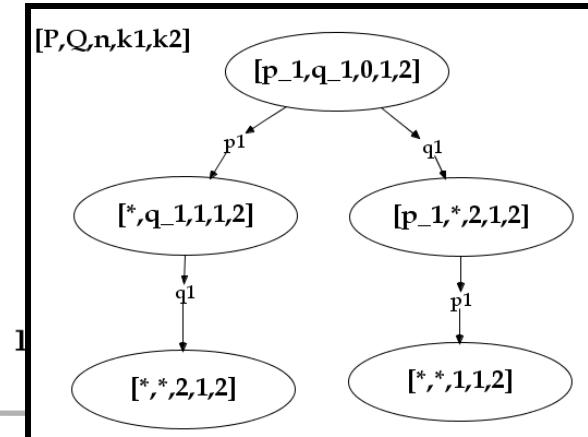
- Ejemplo de programa concurrente:

boolean encont	
process Posit	process Negat
integer p := 0	integer n := 1
encont := false	encont := false
while not encont	while not encont
p := p+1	n := n-1
encont := (f(p)=0)	encont := (f(n)=0)

Representación de la ejecución de un programa concurrente

- **Estado de un programa secuencial:** tupla de valores y contador de programa
- **Estado de un programa concurrente:** tupla de los estados de los procesos que lo componen
- **Transición** entre dos estados: representa la ejecución de la “siguiente instrucción” de alguno de los procesos
- **Diagrama de estados:** grafo que representa el conjunto de posibles estados e historias de ejecución

<pre>integer n:=0</pre>	
<i>process P</i>	<i>process Q</i>
integer k1 := 1	integer k2 := 2
n := k1	n := k2

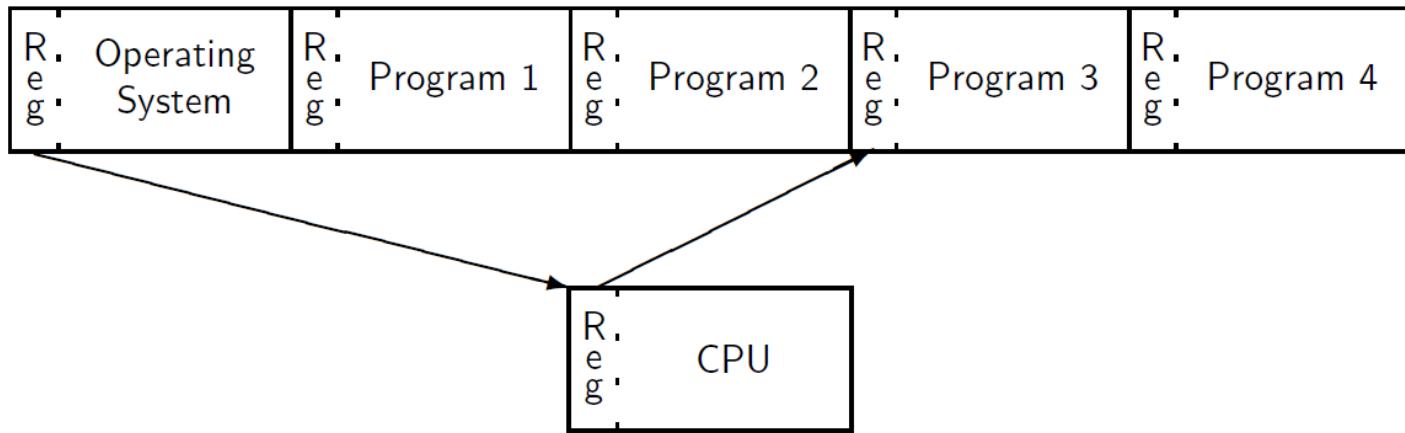


“Entrelazado” de acciones atómicas

- **Acción atómica:** su ejecución es completada sin posibilidad de entrelazado de otras acciones
 - Influencia de la atomicidad en la corrección
- **“Entrelazado” (*interleaving*) de acciones atómicas:** finalizada la ejecución de una instrucción, la “siguiente” instrucción de cualquiera de los procesos es candidata a ser ejecutada
 - Toda secuencia debe ser considerada para el análisis de la corrección de un programa concurrente
 - El tiempo de ejecución es ignorado en el análisis
- ¿Es correcta esta abstracción?

“Entrelazado” de acciones atómicas

- arquitectura “multitasking” (multi-tarea)



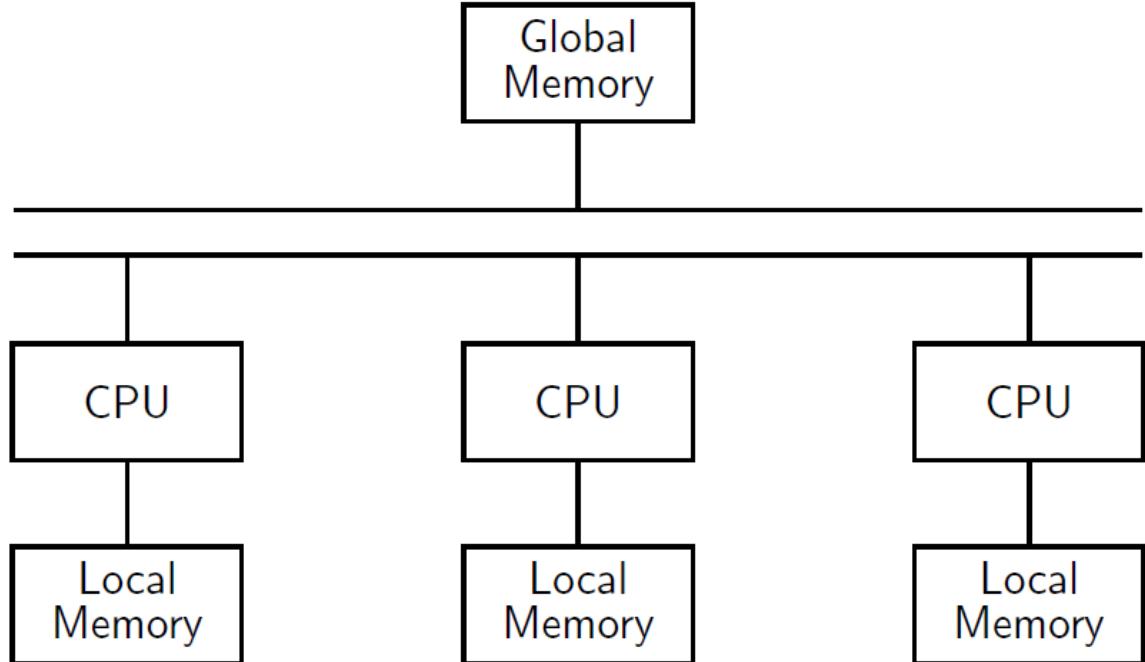
M. Ben-Ari

Principles of Concurrent and Distributed Programming

Addison-Wesley, 2006

“Entrelazado” de acciones atómicas

- arquitectura multi-procesador

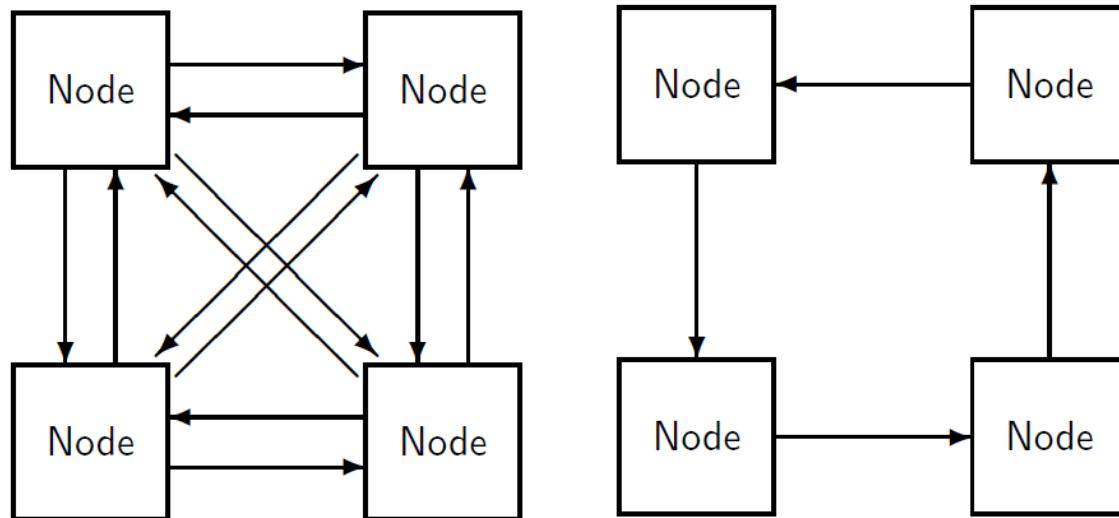


M. Ben-Ari

Principles of Concurrent and Distributed Programming
Addison-Wesley, 2006

“Entrelazado” de acciones atómicas

- arquitectura distribuida



M. Ben-Ari

Principles of Concurrent and Distributed Programming

Addison-Wesley, 2006

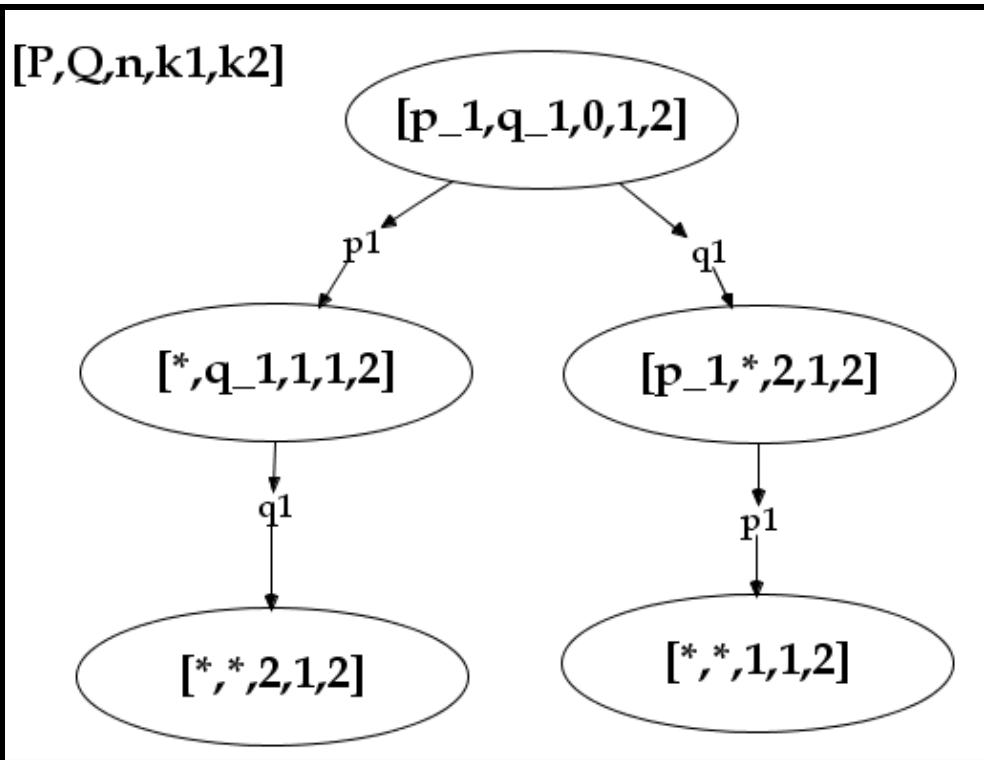
“Entrelazado” de acciones atómicas

- Tres buenas razones para usar la abstracción del entrelazado:
 - permitir razonar formalmente sobre el comportamiento del programa
 - discretización de la ejecución
 - refinamientos sucesivos en el grano de las instrucciones
 - permite diseñar sistemas robustos al cambio de “hard” y “soft”
 - es (casi) imposible repetir la historia de un programa concurrente
 - recordatorio: los “cout” no sirven

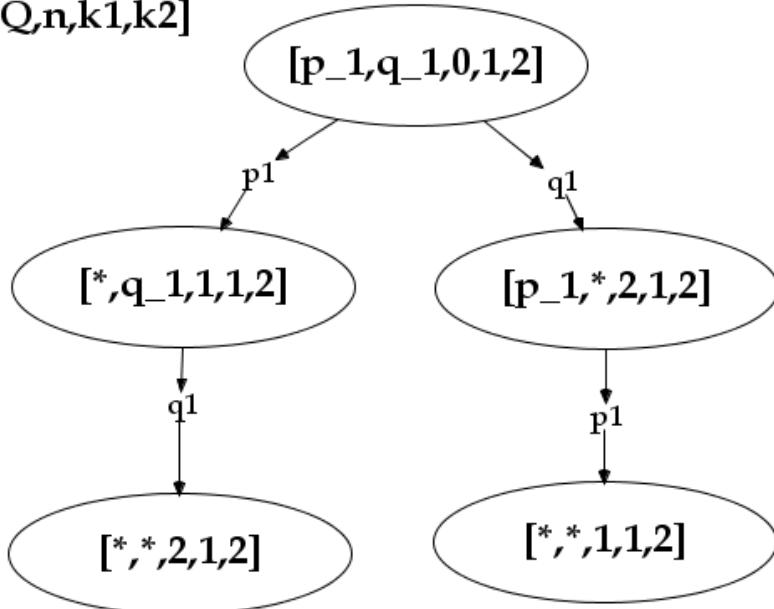
“Entrelazado” arbitrario

- Diferentes historias de ejecución pueden implicar diferentes resultados finales:

<pre>integer n:=0</pre>	
<i>process P</i>	<i>process Q</i>
integer k1 := 1	integer k2 := 2
n := k1	n := k2

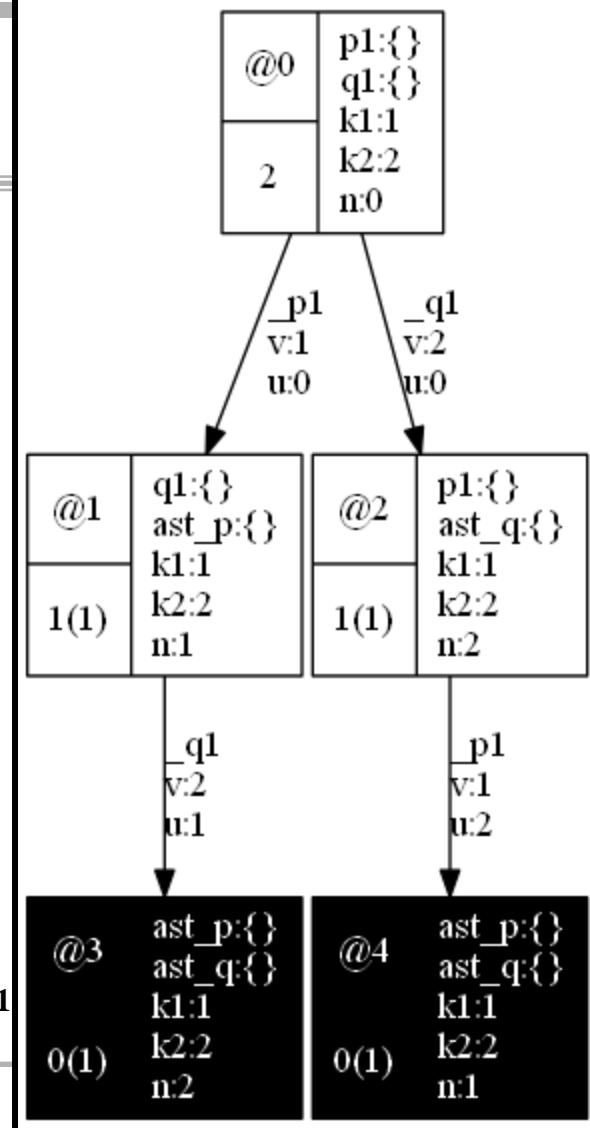


[P,Q,n,k1,k2]



ciones

integer n:=0	
process P	process Q
integer k1 := 1	integer k2 := 2
n := k1	n := k2

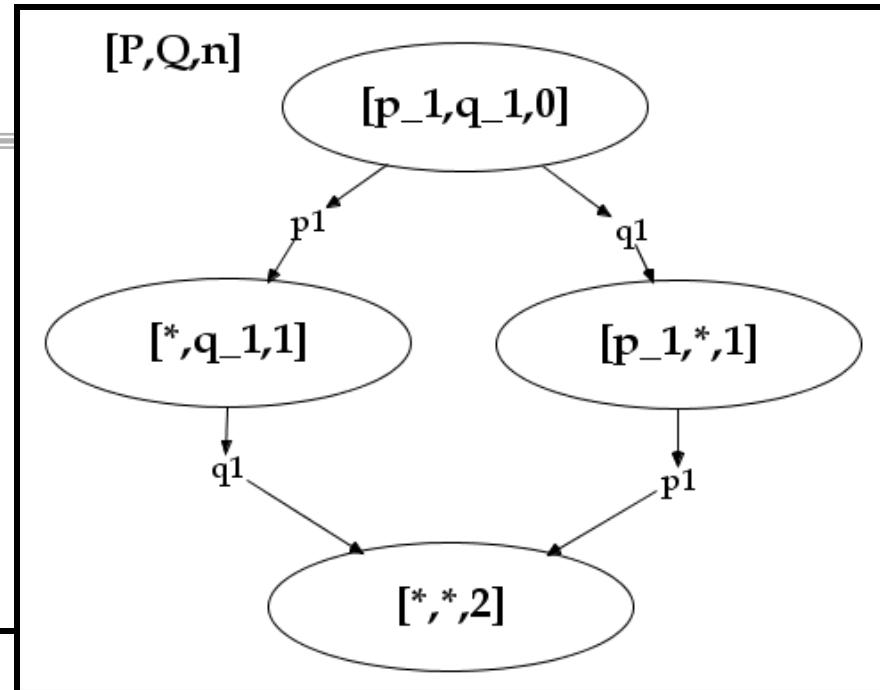


“Entrelazado” arbitrario

- Diferentes historias de ejecución pueden implicar resultados finales iguales:

```
integer n := 0
```

process P	process Q	
1 n := n + 1	n := n + 1	1



Corrección de un programa concurrente

- **Corrección** de un programa concurrente frente a depuración de un programa secuencial
 - Imposible demostrar la corrección “probando”
- **Técnicas de análisis** requieren considerar las posibles historias de ejecución y atributos que expresen el comportamiento deseado
 - Demuestran, de manera formal, la corrección
- El comportamiento deseado se define en términos de **propiedades de corrección**

Propiedades de un programa concurrente

- **Propiedad de un programa:** atributo cierto para cualquier posible historia del programa
- Básicamente, dos clases de propiedades:
 - **Propiedades de seguridad:** el programa nunca alcanza un "mal" estado
 - alternativamente: algo debe cumplirse siempre
 - corrección parcial, exclusión mutua y ausencia de bloqueos
 - **Propiedades de vivacidad:** algo "bueno" ocurrirá
 - alternativamente: algo terminará por cumplirse
 - terminación, equidad
 - dependen en gran medida de la política de "scheduling"

Equidad de un programa concurrente

- Supongamos como propiedad de vivacidad: todos los procesos activos terminan
 - ¿Posibles causas de que no se cumpla?
- Las propiedades de vivacidad vienen condicionadas por las **políticas de “scheduling”**
 - determinan, en cada instante, qué acciones elegibles han de ejecutarse a continuación
 - viene condicionada por la disponibilidad de recursos en el sistema
- La **equidad débil** (“weak fairness”) es la garantía de que en toda ejecución una acción continuamente elegible, tarde o temprano se ejecutará

Equidad de un programa concurrente

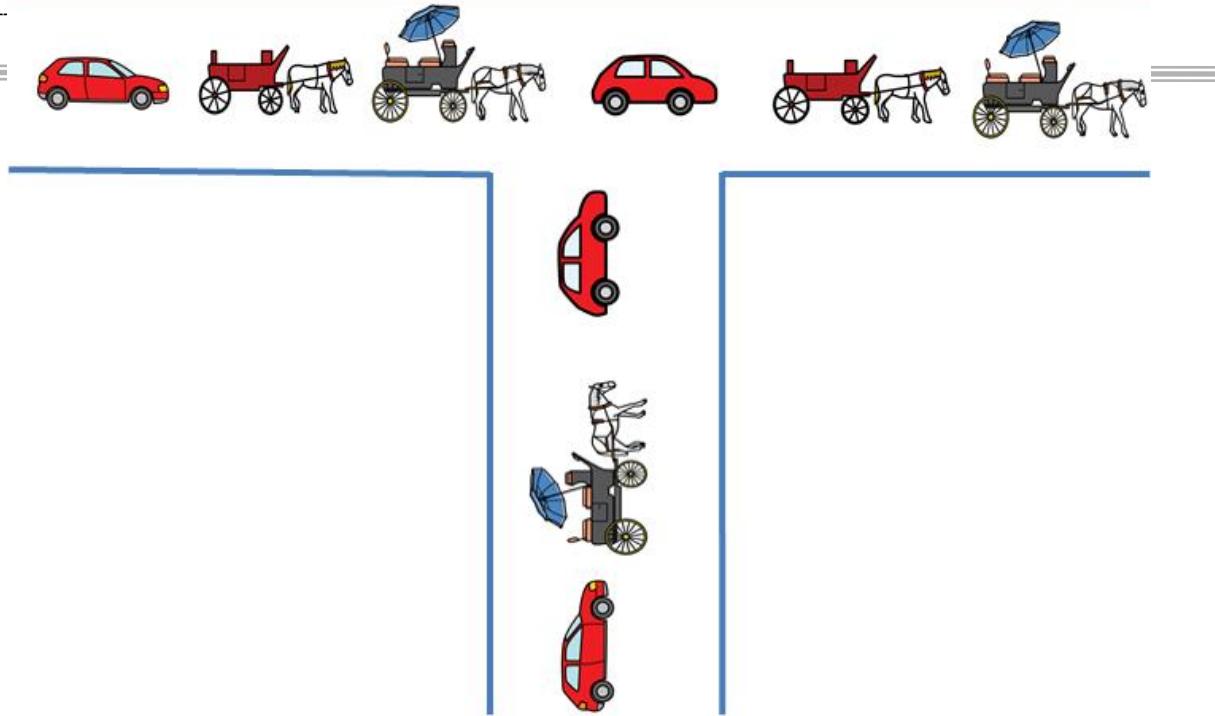
- ¿Terminan?

boolean seguir := true	
<i>process Sigo</i>	<i>process Acabo</i>
while seguir	seguir := false
null	

boolean seguir := true	
hecho := false	
<i>process Sigo</i>	<i>process Acabo</i>
while seguir	while not hecho
hecho := false	null
hecho := true	seguir := false

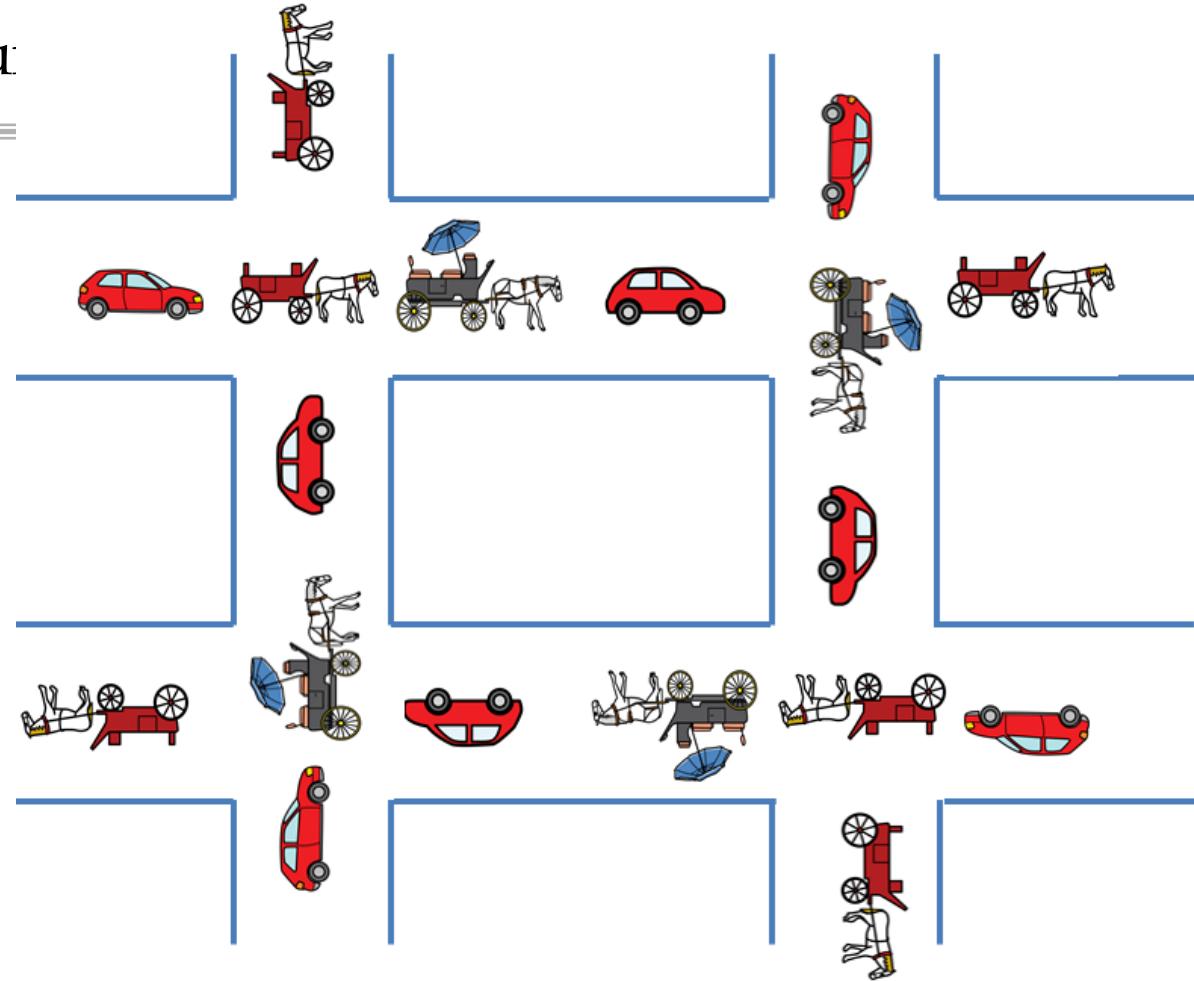
Equidad de uso

- Una situación de inanición
- starvation



Equidad de un deadlock

- Una situación de bloqueo
- deadlock



Influencia

- Programar concurrente en C++
- vers

```
#include <iostream>
#include <thread>

using namespace std;

bool seguir = true; //variable global

//-----
void sigo() {
    while(seguir) {
    }
}

//-----
void acabo() {
    seguir = false;
}

//-----
int main() {
    thread tSigo(sigo),
           tAcabo(acabo);

    tSigo.join();
    tAcabo.join();
    return 0;
}
```

boolean seguir := true	
process Sigo	process Acabo
while seguir	seguir := false
null	

Influencia

- Programación concurrente en C++
 - versión

```
1 1 . . 1  
#include <iostream>  
#include <thread>  
  
using namespace std;  
//--  
void sigo(bool* adelante) {  
    while(*adelante) {  
    }  
}  
//--  
void acabo(bool* adelante) {  
    *adelante = false;  
}  
//--  
int main(){  
    bool seguir = true;  
    thread tSigo(sigo,&seguir),  
              tAcabo(acabo,&seguir);  
  
    tSigo.join();  
    tAcabo.join();  
    return 0;  
}
```

boolean seguir := true		
process Sigo	process Acabo	
while seguir	seguir := false	
null		

Influenci

- Programa concurrente en C++
 - versión

```
#include <iostream>
#include <thread>

using namespace std;

//-
void sigo(bool& adelante) {
    while(adelante) {
    }
}

//-
void acabo(bool& adelante) {
    adelante = false;
}

//-
int main(){
    bool seguir = true;
    thread tSigo(sigo,std::ref(seguir)),
          tAcabo(acabo,std::ref(seguir));

    tSigo.join();
    tAcabo.join();
    return 0;
}
```

boolean seguir := true	
process Sigo	process Acabo
while seguir	seguir := false
null	

Influencia de la atomización

- Programa concurrente en Ada

```
procedure prueba_fairness is
    seguir: boolean := TRUE;

    task type sigo;
    task type acabo;

    task body sigo is
        begin
            while seguir loop
                --put_line("Sigo");
            end loop;
        end sigo;

    task body acabo is
        begin
            seguir := false;
        end acabo;
    end;

    p: sigo;
    q: acabo;

begin
    null;
end prueba_fairness;
```

```
class sigo extends Thread{  
    datos_comunes dC;  
  
    sigo(datos_comunes d){  
        dC = d;  
    }  
  
    public void run(){  
        while(dC.seguir){  
            ...  
        }  
    }  
}
```

```
class acabo extends Thread{  
    datos_comunes dC;  
  
    acabo(datos_comunes d){  
        dC = d;  
    }  
  
    public void run(){  
        dC.seguir = false;  
    }  
}
```

dad en la corrección

```
class datos_comunes{  
    public boolean seguir = true;  
}
```

```
class prueba_fairness {  
  
    public static void main(String args[]){  
        datos_comunes dC;  
        sigo p;  
        acabo q;  
  
        dC = new datos_comunes();  
        p = new sigo(dC);  
        q = new acabo(dC);  
  
        p.start();  
        q.start();  
    }  
}
```

Influencia de la atomicidad en la corrección

- Programa en código máquina para una arquitectura de registros:

integer x := 0	
process P	process Q
1 x := x + 1	1 x := x + 1

integer x := 0	
load R1,x	load R1,x
add R1,#1	add R1,#1
store R1,x	store R1,x

— Condiciones de carrera ("race conditions")

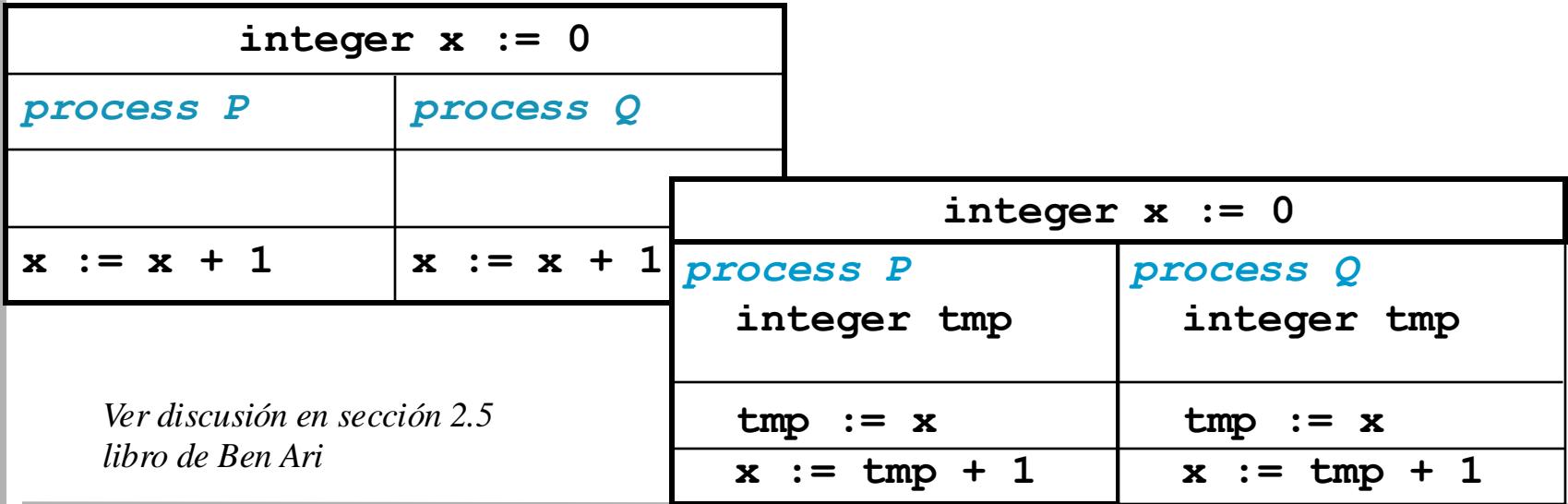
Influencia de la atomicidad en la corrección

- Programa en código máquina para una arquitectura de pila:

integer x := 0	
push x	push x
push #1	push #1
add	add
pop x	pop x

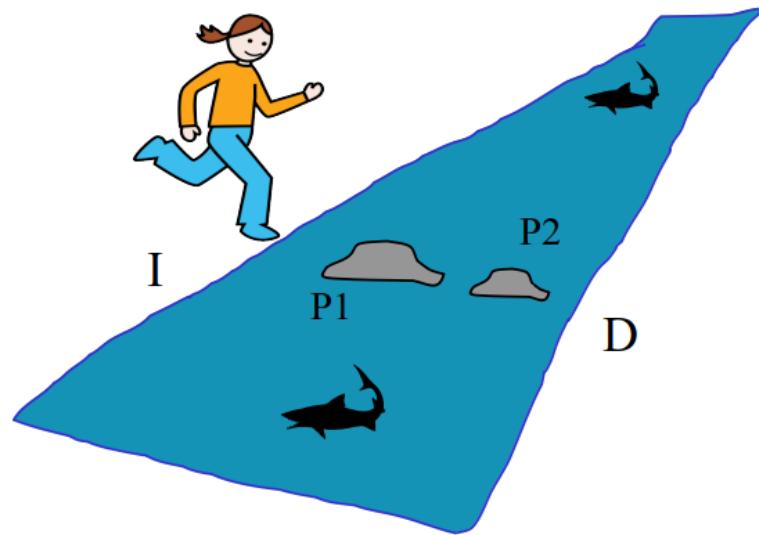
Influencia de la atomicidad en la corrección

- Entonces ¿qué vamos a considerar instrucciones atómicas?
 - asignaciones
 - evaluación de guardas en estructuras de control
- ¿Seguro?

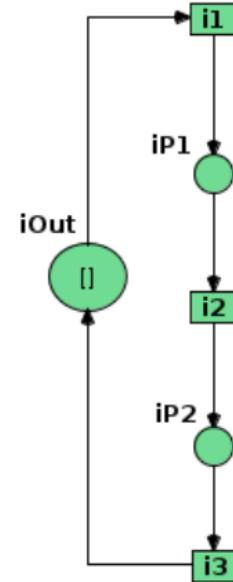


Ver discusión en sección 2.5
libro de Ben Ari

Una manera de modelar sistemas concurrentes



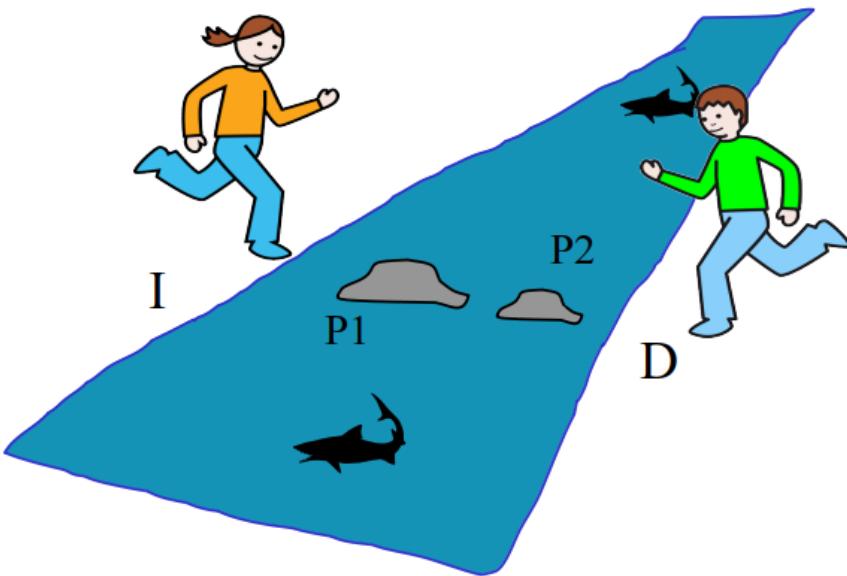
- no retroceso



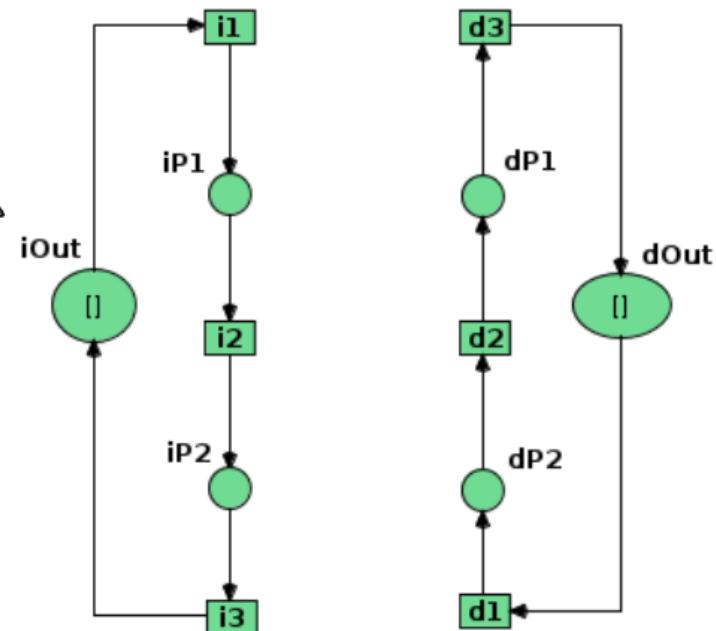
Una manera de modelar sistemas concurrentes

- **Modelo:** abstracción de la realidad
 - visión simplificada
 - centrada en determinados aspectos
- Un modelo se debe validar
- Un modelo debe servir para:
 - entender el sistema real
 - inferir propiedades del sistema a partir de propiedades del modelo

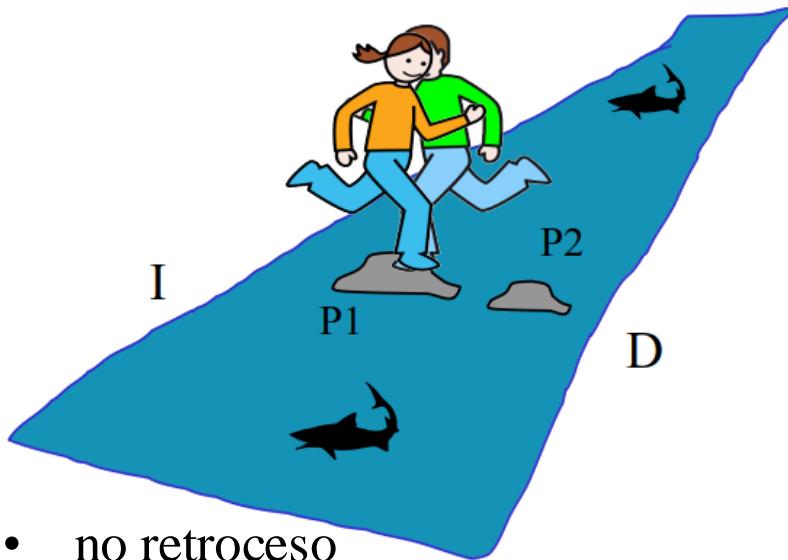
Una manera de modelar sistemas concurrentes



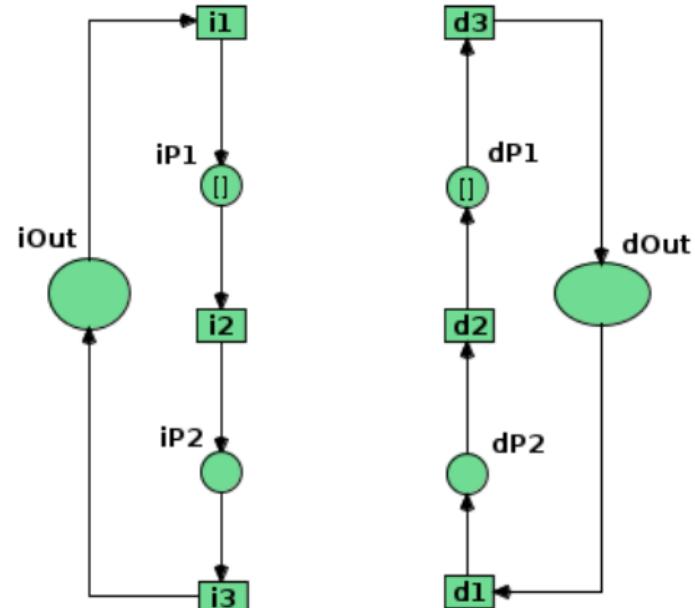
- no retroceso
- no dos en la misma piedra



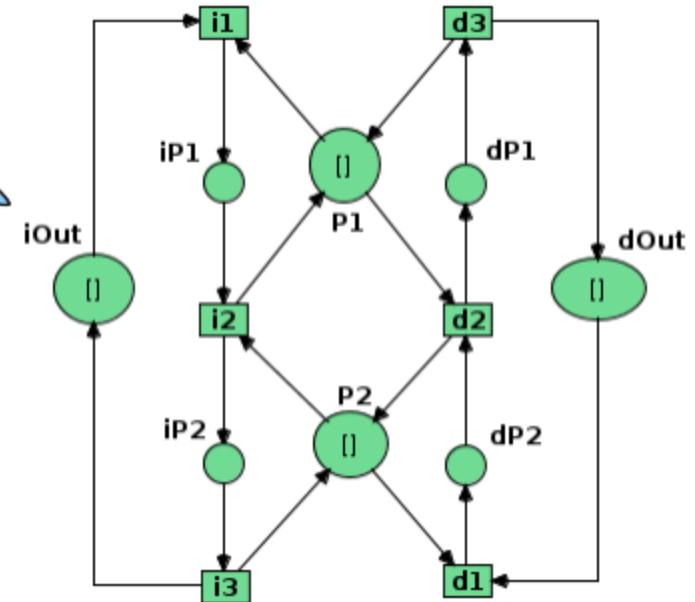
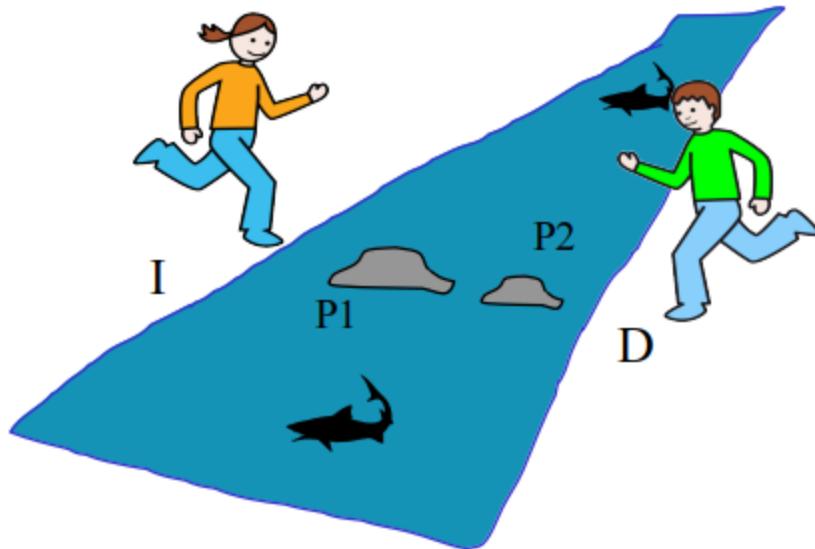
Una manera de modelar sistemas concurrentes



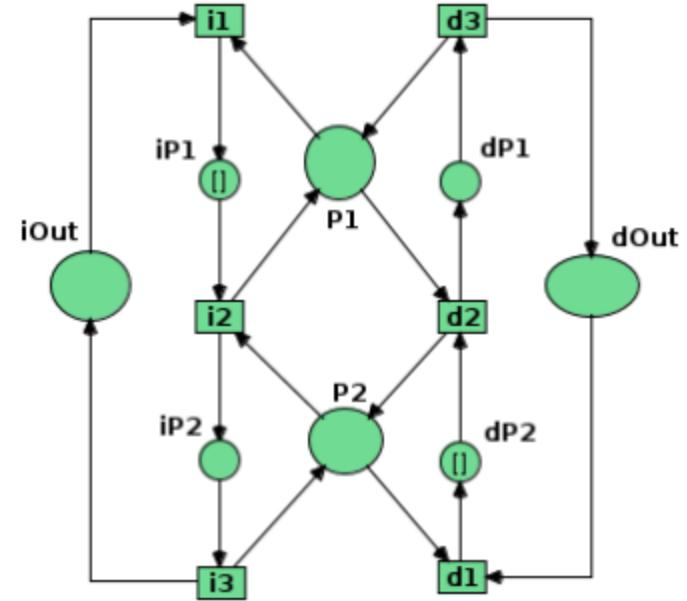
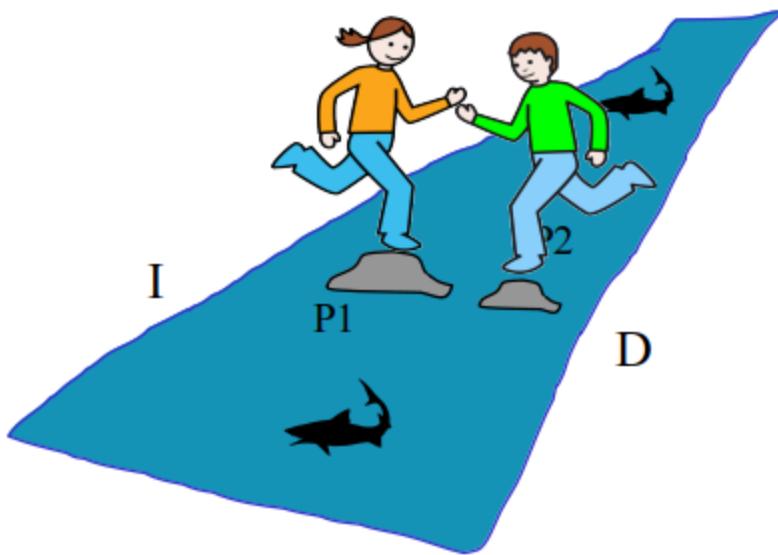
- no retroceso
- no dos en la misma piedra



Una manera de modelar sistemas concurrentes

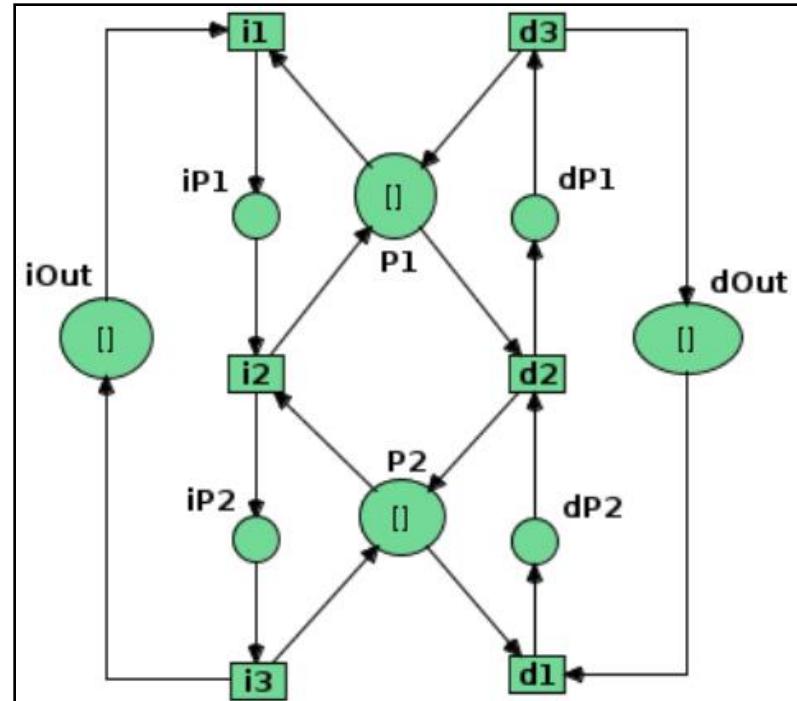


Una manera de modelar sistemas concurrentes



Una manera de modelar sistemas concurrentes

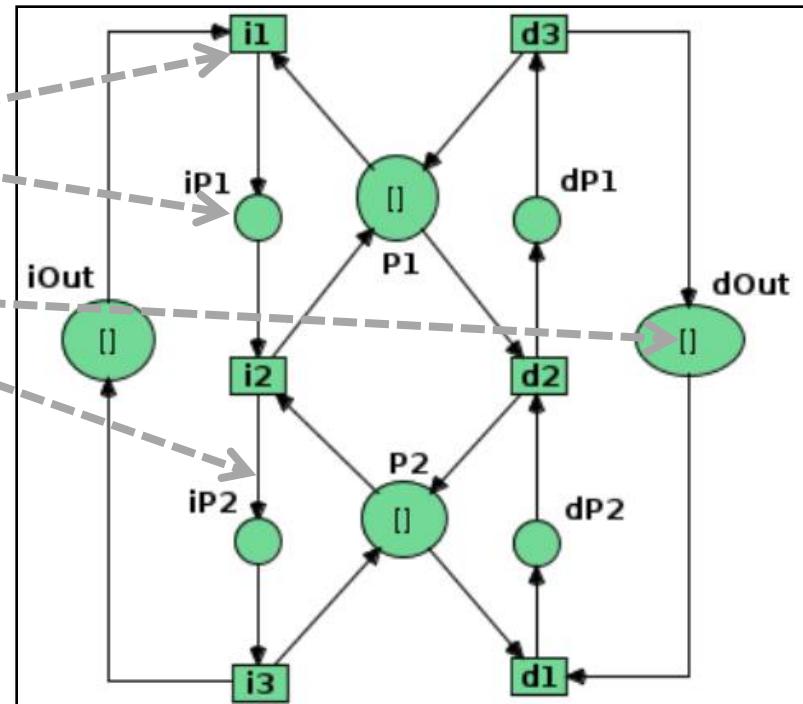
- Recordatorio de conceptos básicos y mínimos sobre redes de Petri:
 - lugar
 - transición
 - arco
 - marcado
 - transición sensibilizada
 - disparo de transición
 - redes ordinarias
 - redes coloreadas



Una manera de modelar sistemas concurrentes

- Recordatorio de conceptos básicos y mínimos sobre redes de Petri:

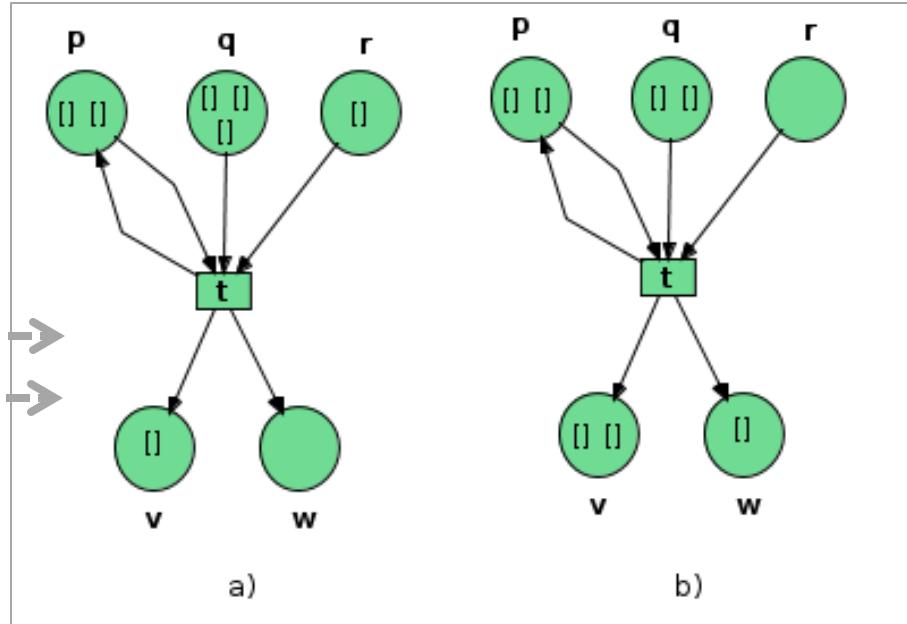
- lugar
- transición
- arco
- marcado
- transición sensibilizada
- disparo de transición
- redes ordinarias
- redes coloreadas



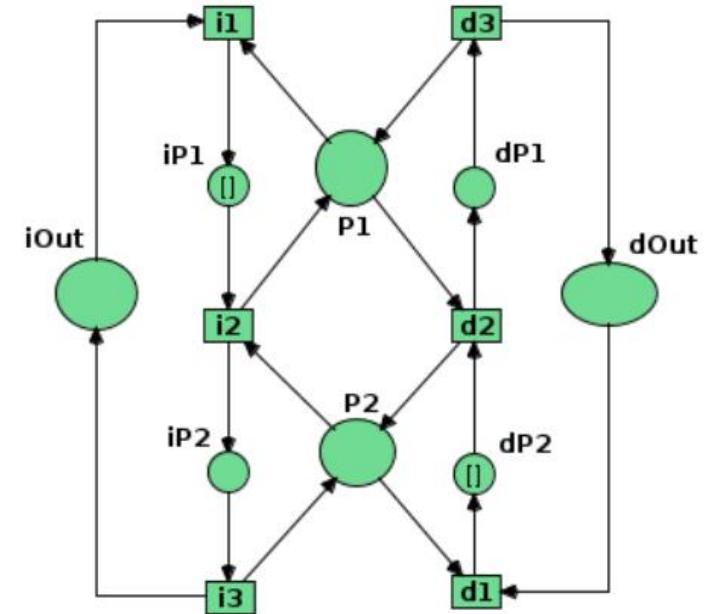
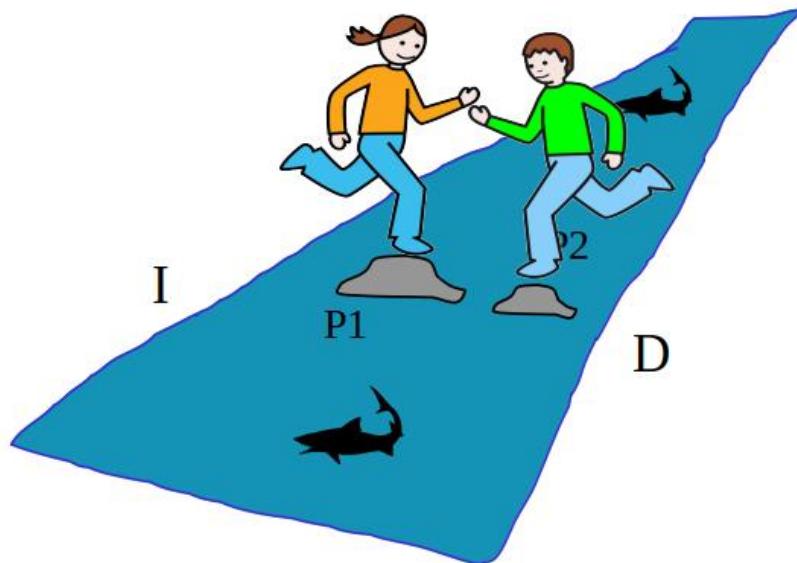
Una manera de modelar sistemas concurrentes

- Recordatorio de conceptos básicos y mínimos sobre redes de Petri:

- lugar
- transición
- arco
- marcado
- transición sensibilizada →
- disparo de transición →
- redes ordinarias
- redes coloreadas

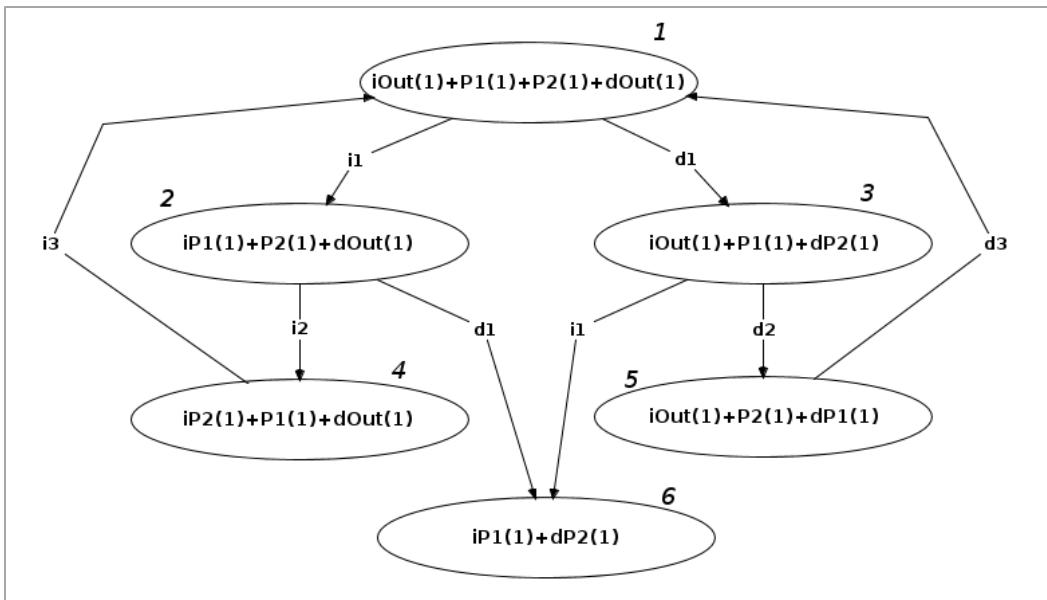
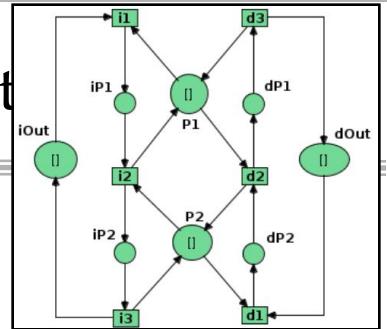


Una manera de modelar sistemas concurrentes

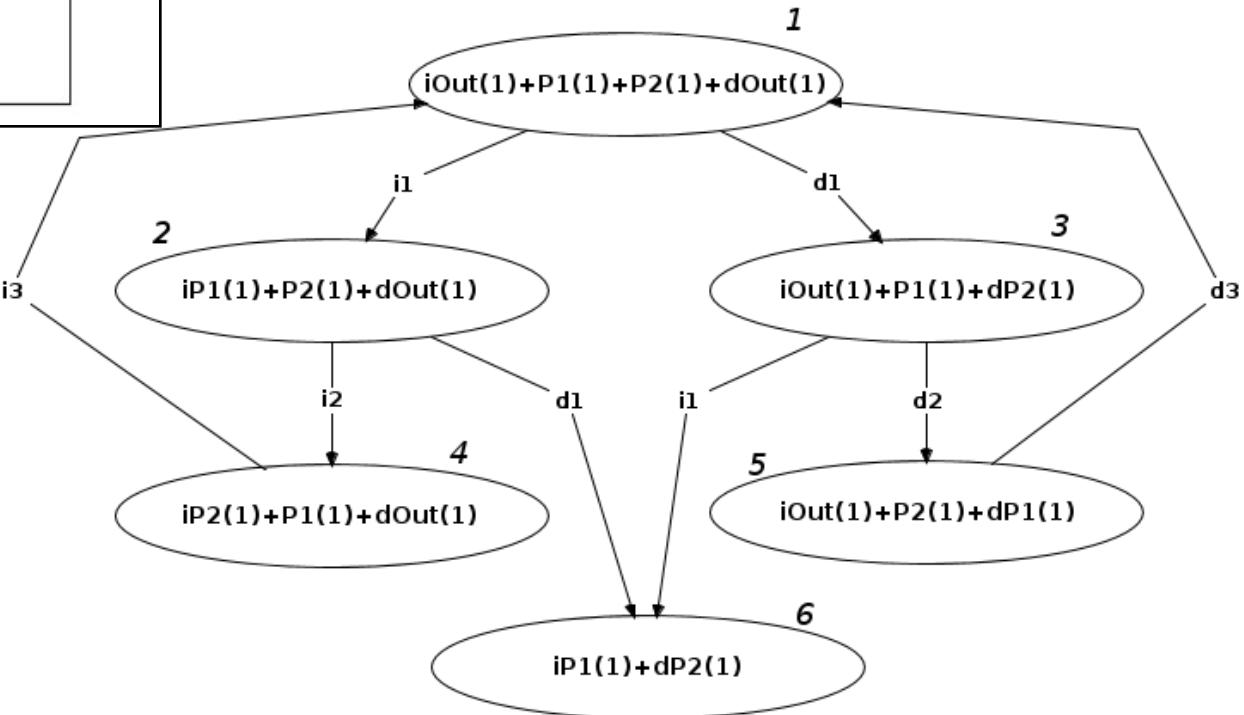
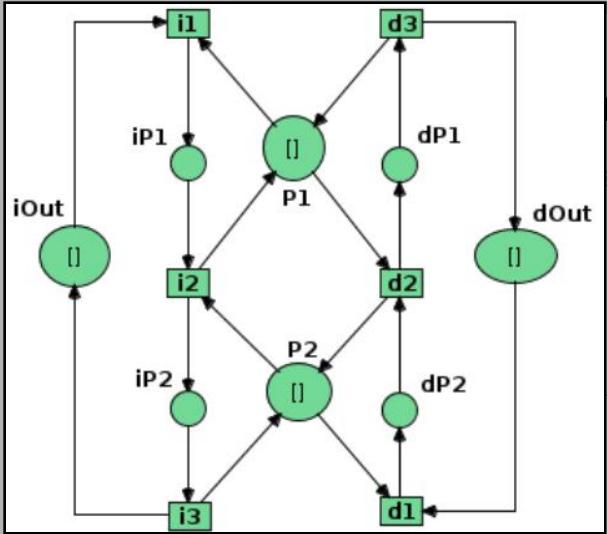


Una manera de modelar sistemas concurrentes

- Grafo de **estados alcanzables** del ejemplo anterior
- Estudio de propiedades:
 - repetitividad
 - ausencia de bloqueos totales
 - ausencia de bloqueos parciales
 - vivacidad
 - equidad/inanición



Modelar sistemas concurrentes



Una manera de modelar sistemas concurrentes

- En los programas concurrentes vamos considerar **dos tipos distintos de lugares**
 - los que representan posiciones del "contador de programa" de cada proceso ("ordinarios")
 - puede tener o no marca
 - los que representan las variables (coloreados)
 - que siempre tienen un valor
- Da lugar a una subclase de redes
 - la estructura de las redes cambia
 - aparecen variables en los arcos de lugares coloreados
 - las reglas de sensibilización/disparo son un poco distintas
 - los valores de las variables intervienen
 - el grafo de estados alcanzables necesita considerar esa información

Una manera de modelar sistemas concurrentes

- Consideremos el siguiente programa concurrente:

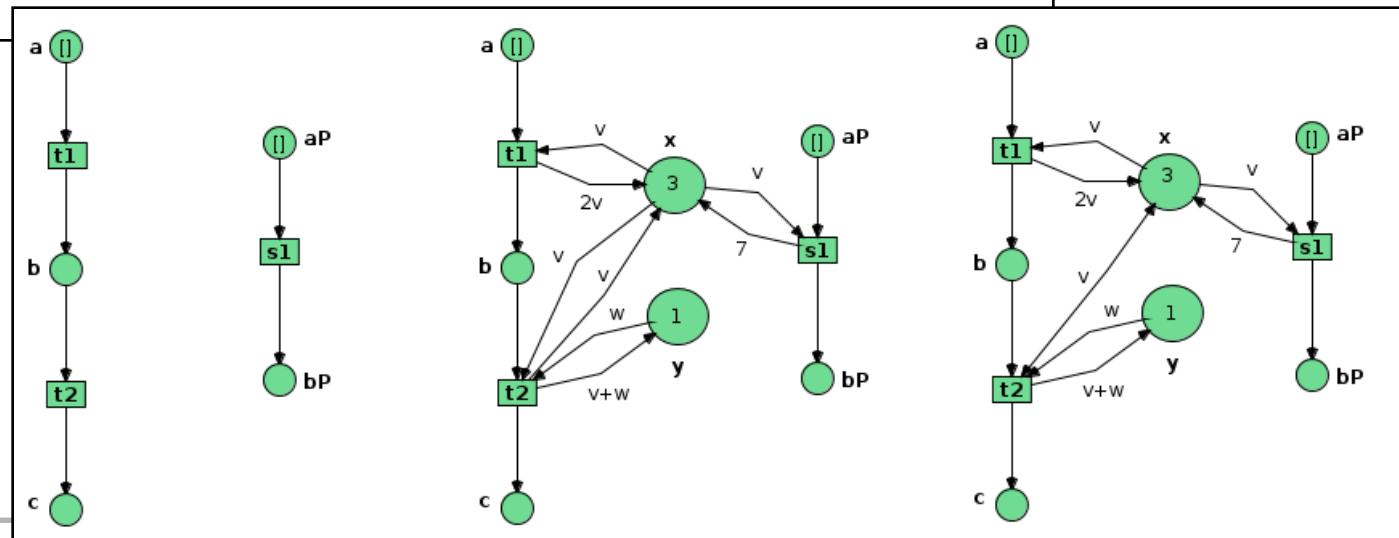
```
integer x := 3, y := 1
Process P::          Process Q::
    <x := 2x>          <x := 7>
    <y := x+y>
end
```

```
integer x := 3, y := 1
Process P::          Process Q::
    (a) <x := 2x>      (aP) <x := 7>
    (b) <y := x+y>     (bP)
    (c)
end
```

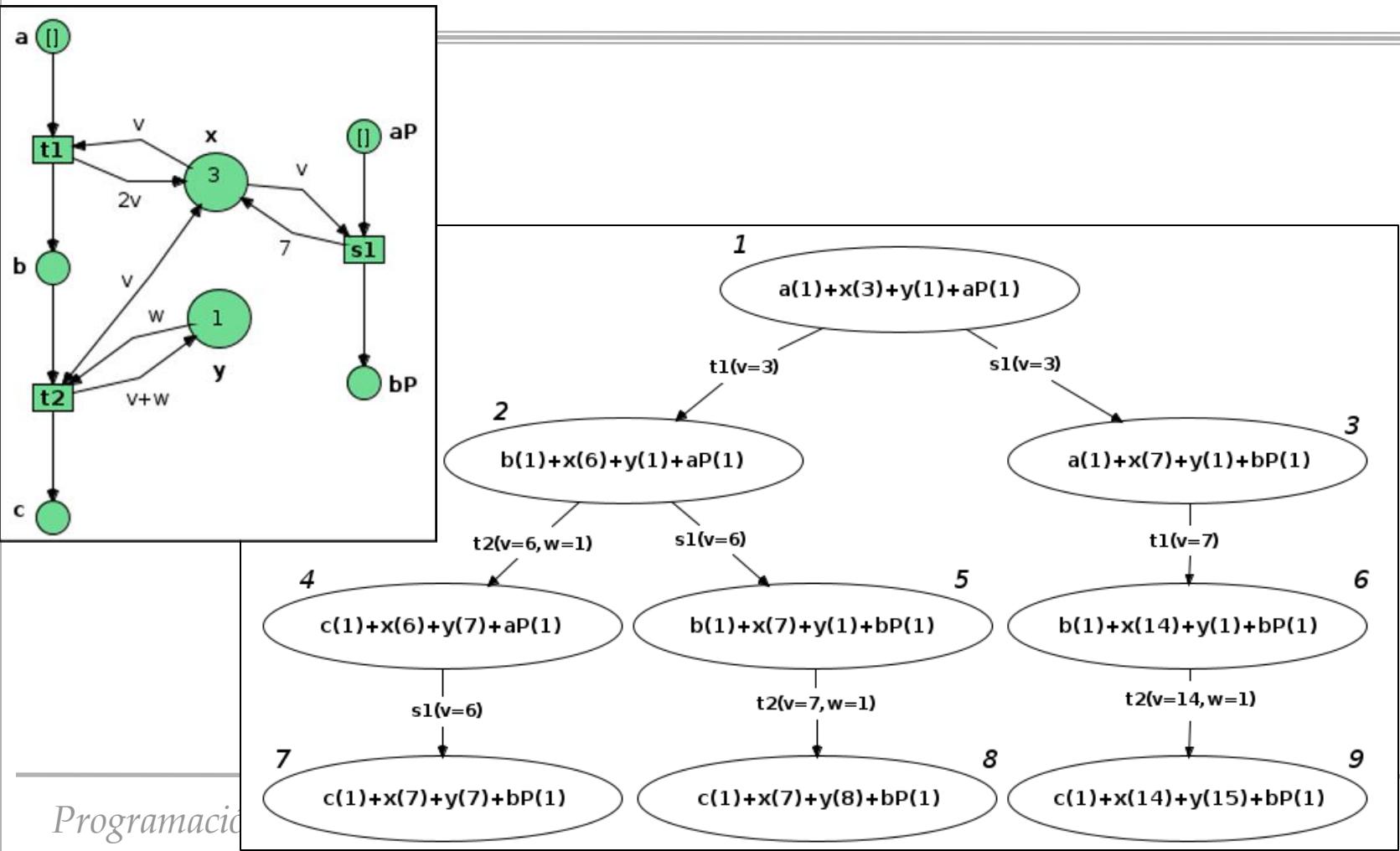
Una manera de modelar sistemas concurrentes

- Un ejemplo de modelo para un programa concurrente

```
integer x := 3, y := 1
Process P:::
  (a) <x := 2x>
  (b) <y := x+y>
  (c)
end
Process Q:::
  (aP) <x := 7>
  (bP)
end
```



Una manera de modelar sistemas concurrentes



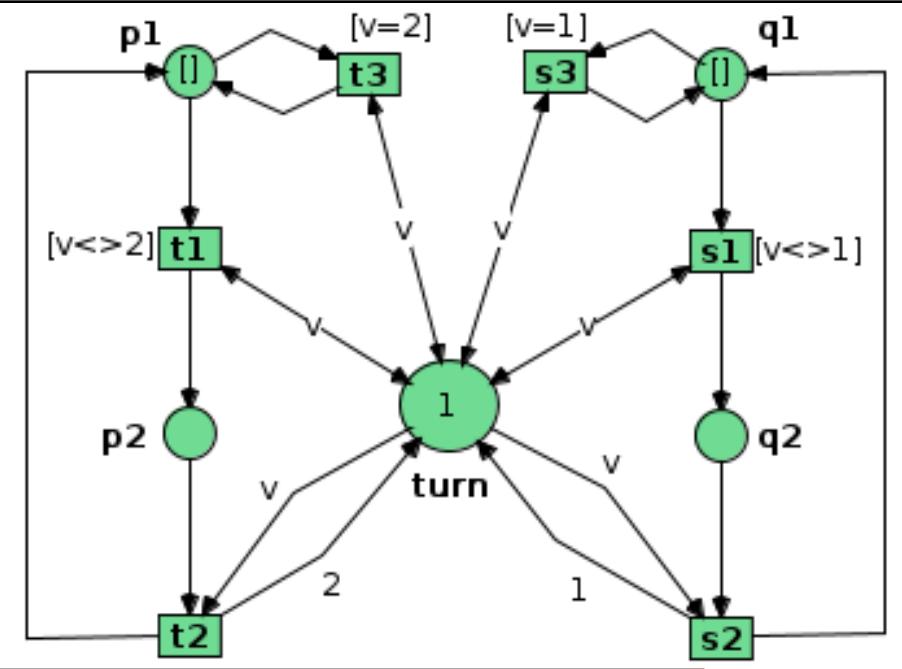
Una manera de modelar sistemas concurrentes

- Ejercicio: Dar un modelo para el siguiente programa:

```
integer x := 0
_____
Process P:
  while true
    (p1) x := 1
    (p2) x := 2
  end
end
_____
Process Q:
  while true
    (q1) x := 3
  end
```

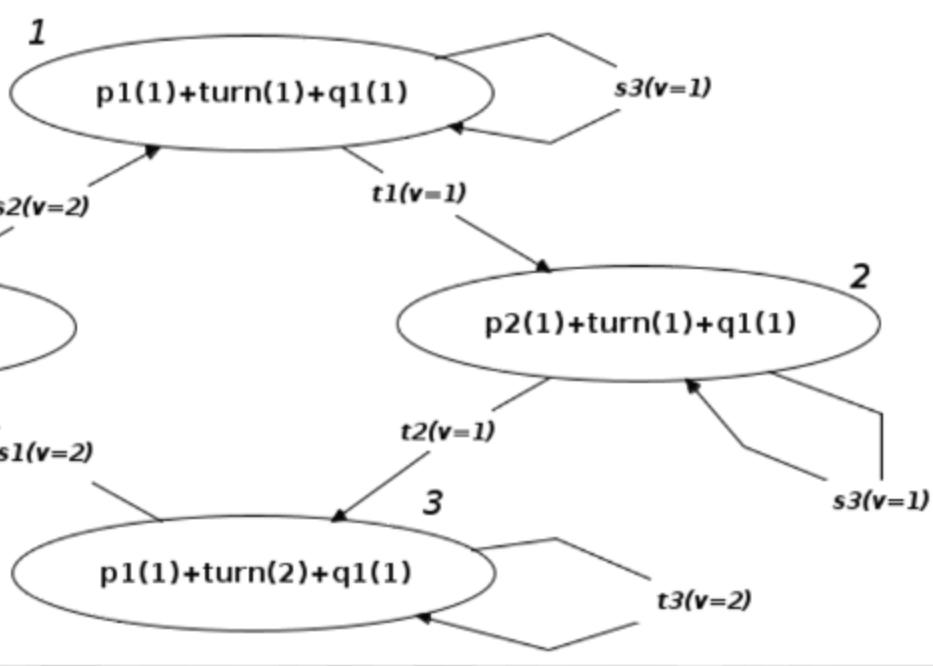
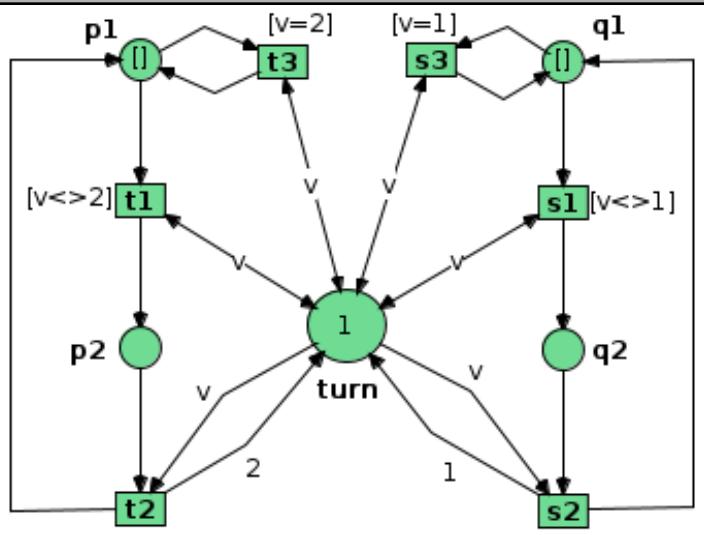
Una manera de modelar

- En ocasiones, no cualquier valor de una variable nos interesa
 - guardas en las transiciones

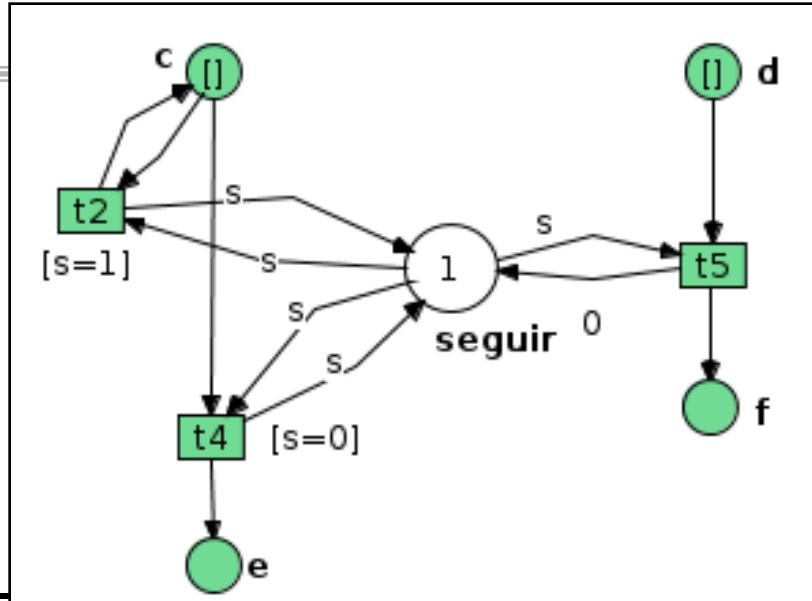


```
integer turn := 1
Process P:
  while true
    (p1) while turn=2;
           //SC1
    (p2) turn := 2
  end
end
Process Q:
  while true
    (q1) while turn=1;
           //SC2
    (q2) turn := 1
  end
```

Paralelizar sistemas concurrentes



Un primer programa



```
boolean seguir := true
```

```
process Sigo
```

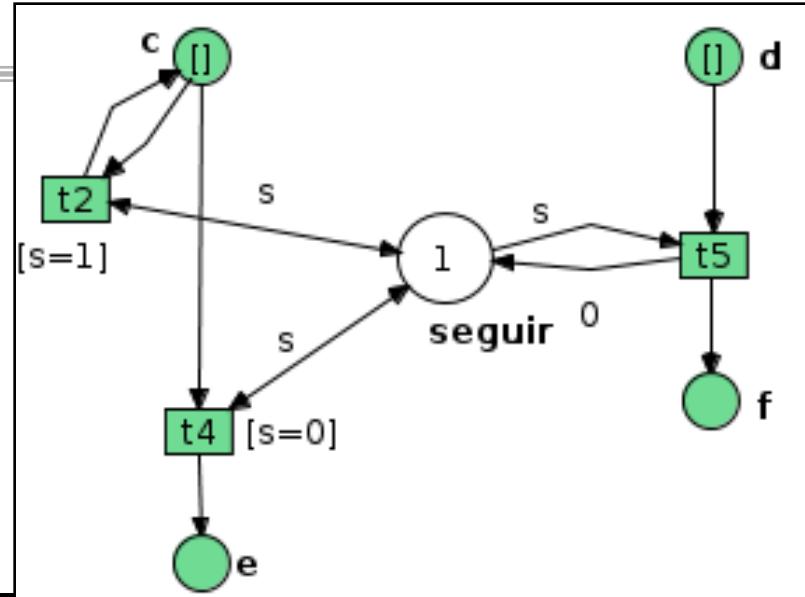
```
process Acabo
```

```
while seguir
```

```
seguir := false
```

```
null
```

Un primer programa



```
boolean seguir := true
```

```
process Sigo
```

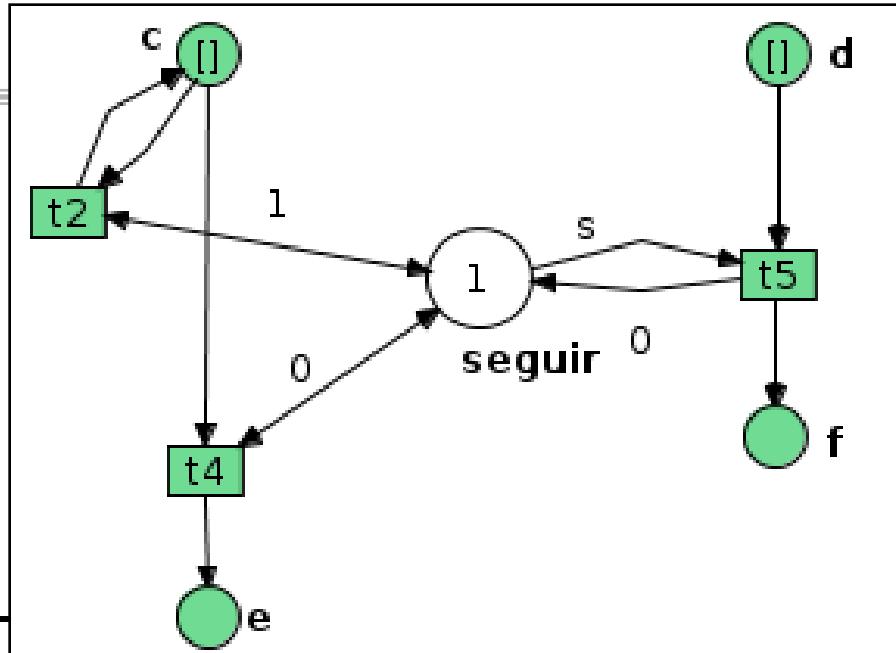
```
process Acabo
```

```
while seguir
```

```
seguir := false
```

```
null
```

Un primer programa



```
boolean seguir := true
```

```
process Sigo
```

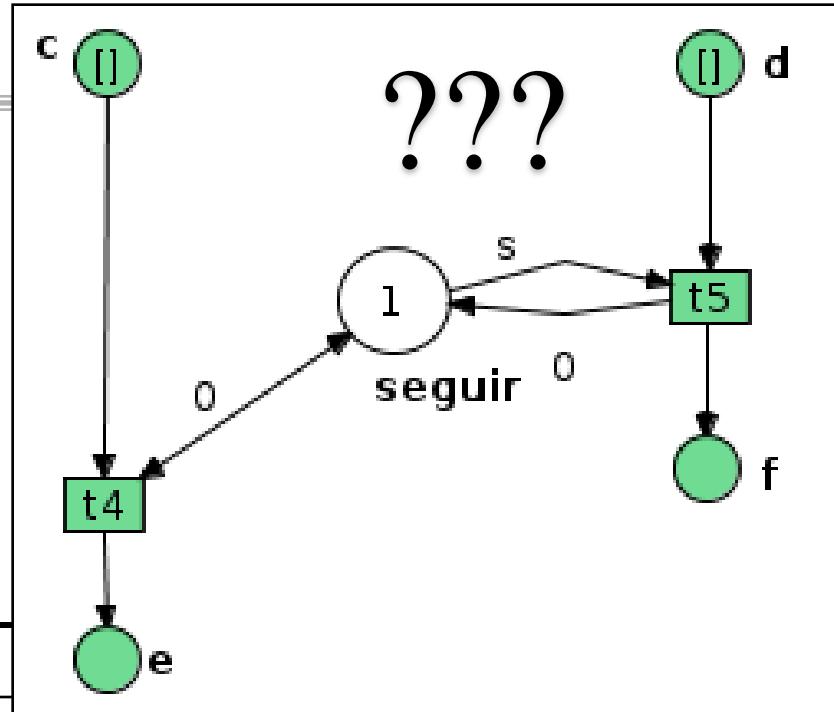
```
while seguir
```

```
null
```

```
process Acabo
```

```
seguir := false
```

Un primer programa



Lección 3: Sincronización de procesos. El problema de la sección crítica

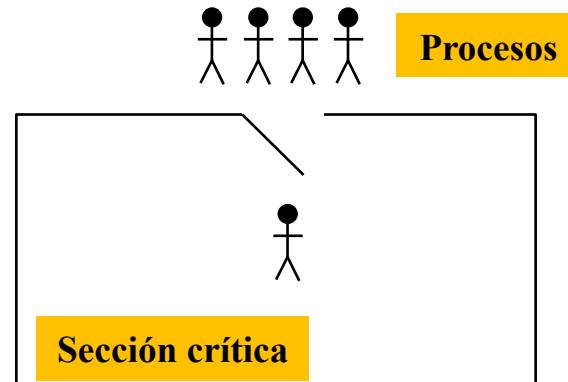
- Sincronización de procesos
- El problema de la sección crítica
- Primeros intentos de solución: propuesta de algoritmos y sus correspondientes modelos
- Algoritmos para la sección crítica:
 - Algoritmo de Dekker (2 procesos)
 - Algoritmo de la panadería
 - Algoritmo por turno de espera

Sincronización de procesos

- Objetivo de la sincronización: realización coordinada de tareas entre procesos
 - Una tarea antes que otra o a la vez
- Herramienta conceptual: instrucción *await*
 - Muy general
 - Costosa de implementar
- Variedad de mecanismos de sincronización de procesos
 - semáforos, monitores, memoria compartida
 - Lenguajes de programación también ofrecen instrucciones de alto nivel para la sincronización (por ejemplo, “**synchronized**” de Java)

El problema de la sección crítica

- Enunciado del problema (Dijkstra, 1976):
 - "n" procesos ejecutan repetidamente una sección no crítica, SNC, seguida de una **sección crítica**, SC
 - La sección crítica de un proceso es un secuencia de acciones que debe ser ejecutadas en **exclusión mutua** con las secciones críticas del resto de procesos.
 - Siempre que un proceso “entra” en la SC, “sale”



El problema de la sección crítica

- Esquema del algoritmo:

variables globales	
<i>Process P</i>	<i>Process Q</i>
variables locales	variables locales
while true	while true
SNC	SNC
Protocolo de entrada	Protocolo de entrada
SC	SC
Protocolo de salida	Protocolo de salida

El problema de la sección crítica

- **Objetivo:** Diseñar los protocolos de entrada y salida (mecanismos de sincronización) a la sección crítica de forma que se satisfagan los siguientes requisitos:
 - **Exclusión mutua:** como máximo un proceso puede estar ejecutando su sección crítica
 - **Ausencia de bloqueos:** si dos o más procesos tratan de acceder a su sección crítica, al menos uno lo logrará
 - **Ausencia de situaciones de inanición (individual):** todo proceso que desee entrar en su SC, tarde o temprano entrará

Varios intentos (con solo dos procesos!) ...

Intento	Mutex	Ausencia bloqueos	Ausencia inanición
1º. "primero uno y después otro"	??	??	??
2º. "una variable para cada proceso"	??	??	??
3º. "quiero entrar!"	??	??	??
4º. "quiero entrar pero te doy una oportunidad"	??	??	??

Primer intento: primero uno después otro

- Algoritmo:

integer turn := 1		
	<i>Process P</i>	<i>Process Q</i>
p1	while true	while true
p2	SNC	SNC
p3	while turn = 2;	while turn = 1;
p4	SC	SC
	turn := 2	turn := 1

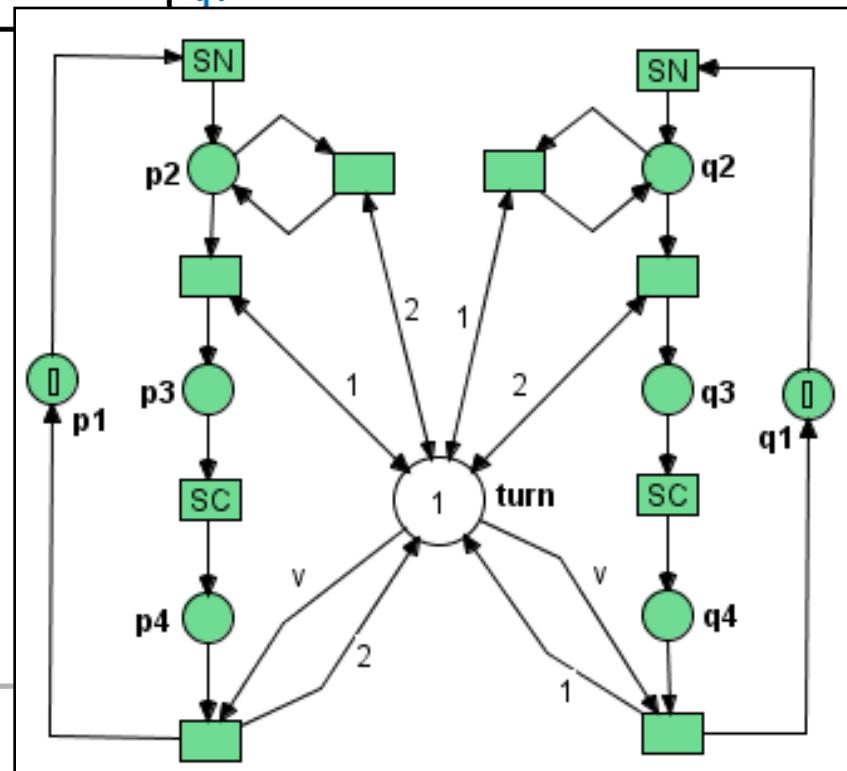
q1
q2
q3
q4

Primer intento: modelo

integer turn := 1

<i>Process P</i>	<i>Process Q</i>
<i>while true</i>	<i>while true</i>
p1 SNC	SNC
p2 while turn = 2;	while turn = 1;
p3 SC	SC
p4 turn := 2	turn := 1

q1
q2
q3
q4

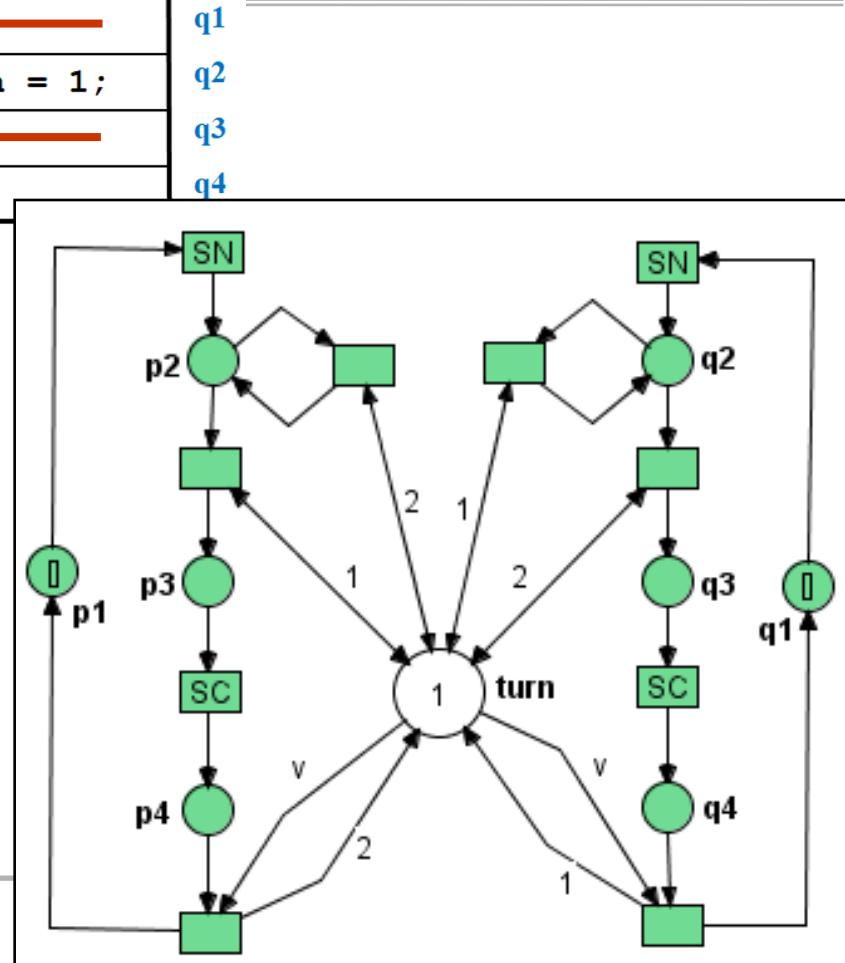


Primer intento: modelo

integer turn := 1

<i>Process P</i>	<i>Process Q</i>
<u>while true</u>	<u>while true</u>
p1 -SNC	-SNC
p2 while turn = 2;	while turn = 1;
p3 -SC	-SC
p4 turn := 2	turn := 1

Abstracción



Primer intento: historia de ejecución

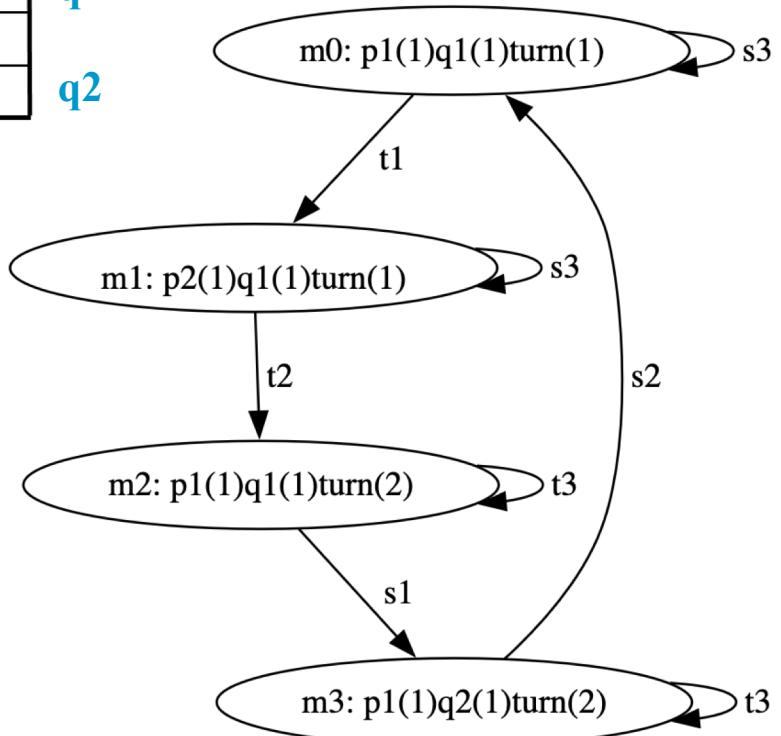
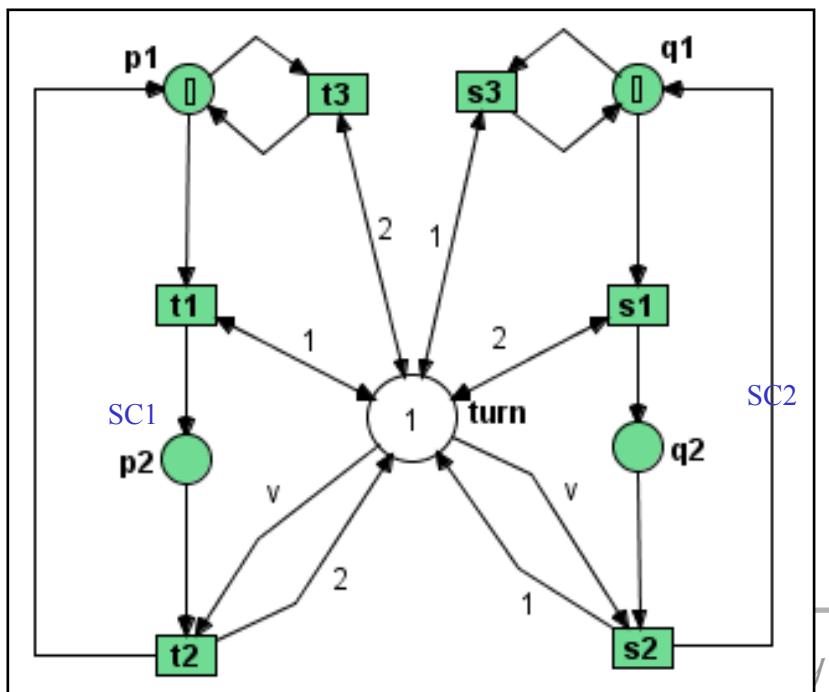
integer turn := 1	
Process P	Process Q
<u>while true</u>	<u>while true</u>
SNC	SNC
while turn = 2;	while turn = 1;
SC	SC
turn := 2	turn := 1

p1

p2

q1

q2



Primer intento: historia de ejecución

integer turn := 1	
Process P	Process Q
<u>while true</u>	<u>while true</u>
<u>SNC</u>	<u>SNC</u>
<u>while turn = 2;</u>	<u>while turn = 1;</u>
<u>SC</u>	<u>SC</u>
<u>turn := 2</u>	<u>turn := 1</u>

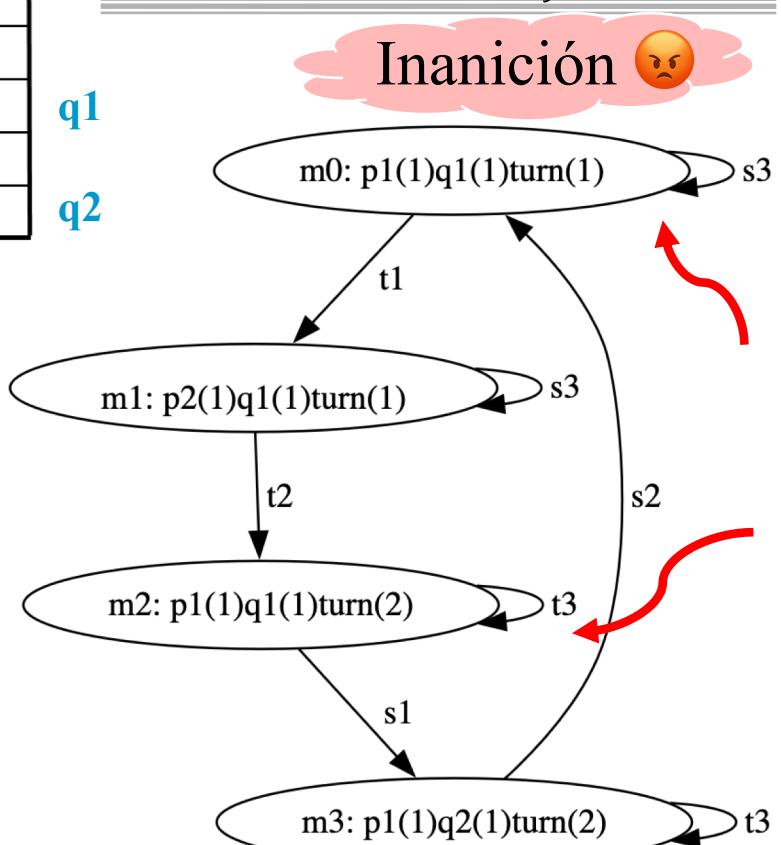
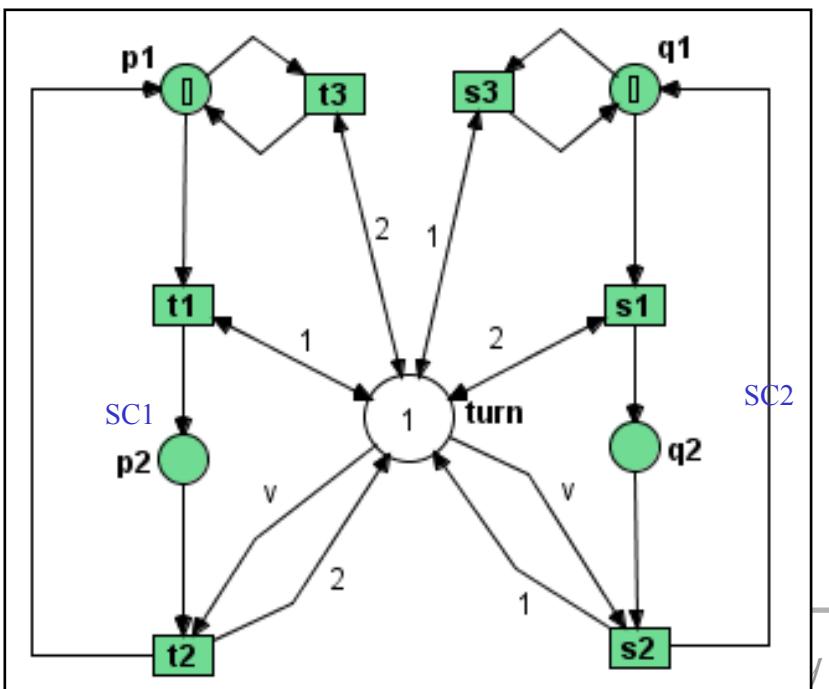
p1

p2

q1

q2

Inanición 😠



Segundo intento: una variable para cada proceso

- Cada proceso controla con una variable de sincronización propia el acceso a la SC
- Algoritmo:

<code>boolean wantP := false, wantQ := false</code>	
<i>Process P</i>	<i>Process Q</i>
<code>while true</code>	<code>while true</code>
SNC	SNC
<code>while wantQ = true;</code>	<code>while wantP = true;</code>
<code>wantP := true</code>	<code>wantQ := true</code>
SC	SC
<code>wantP := false</code>	<code>wantQ := false</code>

p1

p2

p3

q1

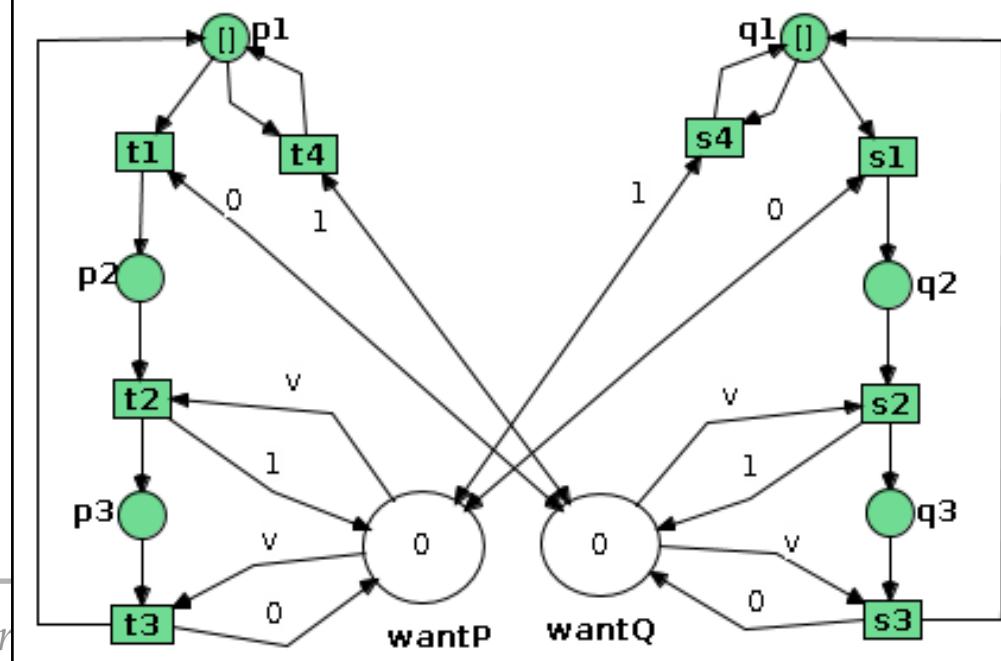
q2

q3

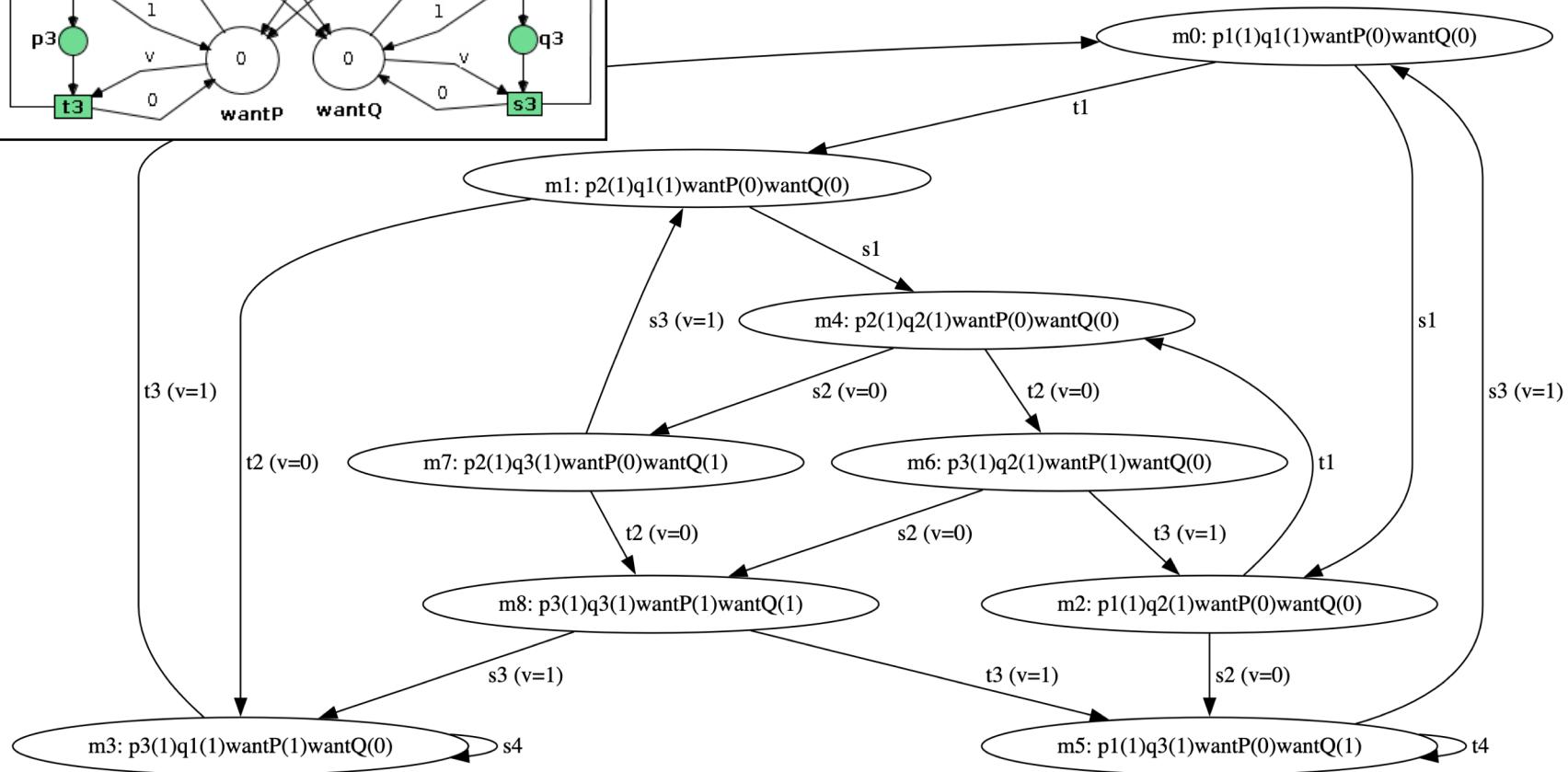
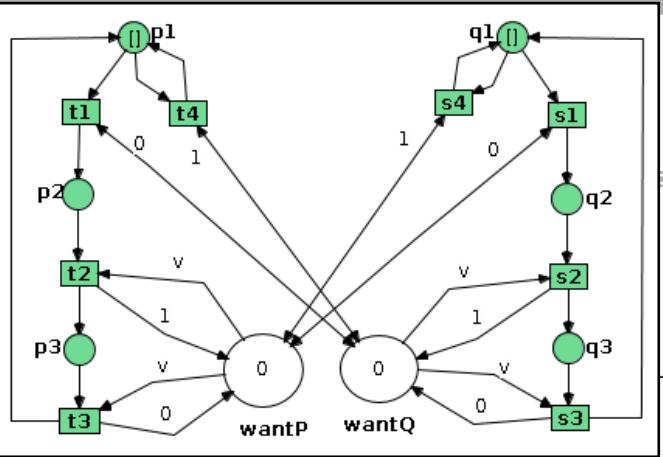
Segundo intento: modelo

boolean wantP := false, wantQ := false	
Process P	Process Q
while true	while true
SNC	SNC
while wantQ = true;	while wantP = true;
wantP := true	wantQ := true
SC	SC
wantP := false	wantQ := false

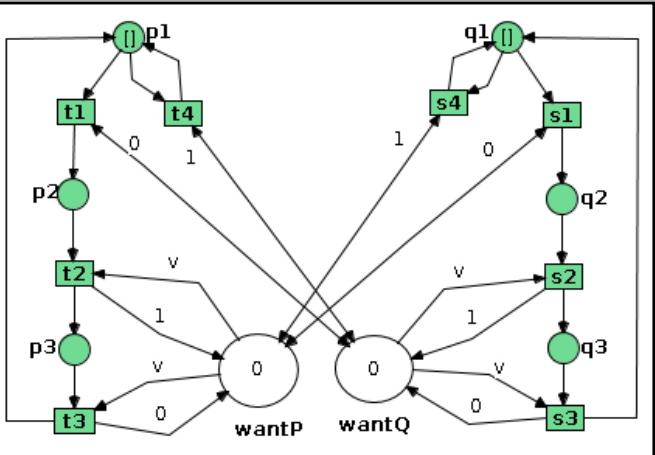
q1
q2
q3



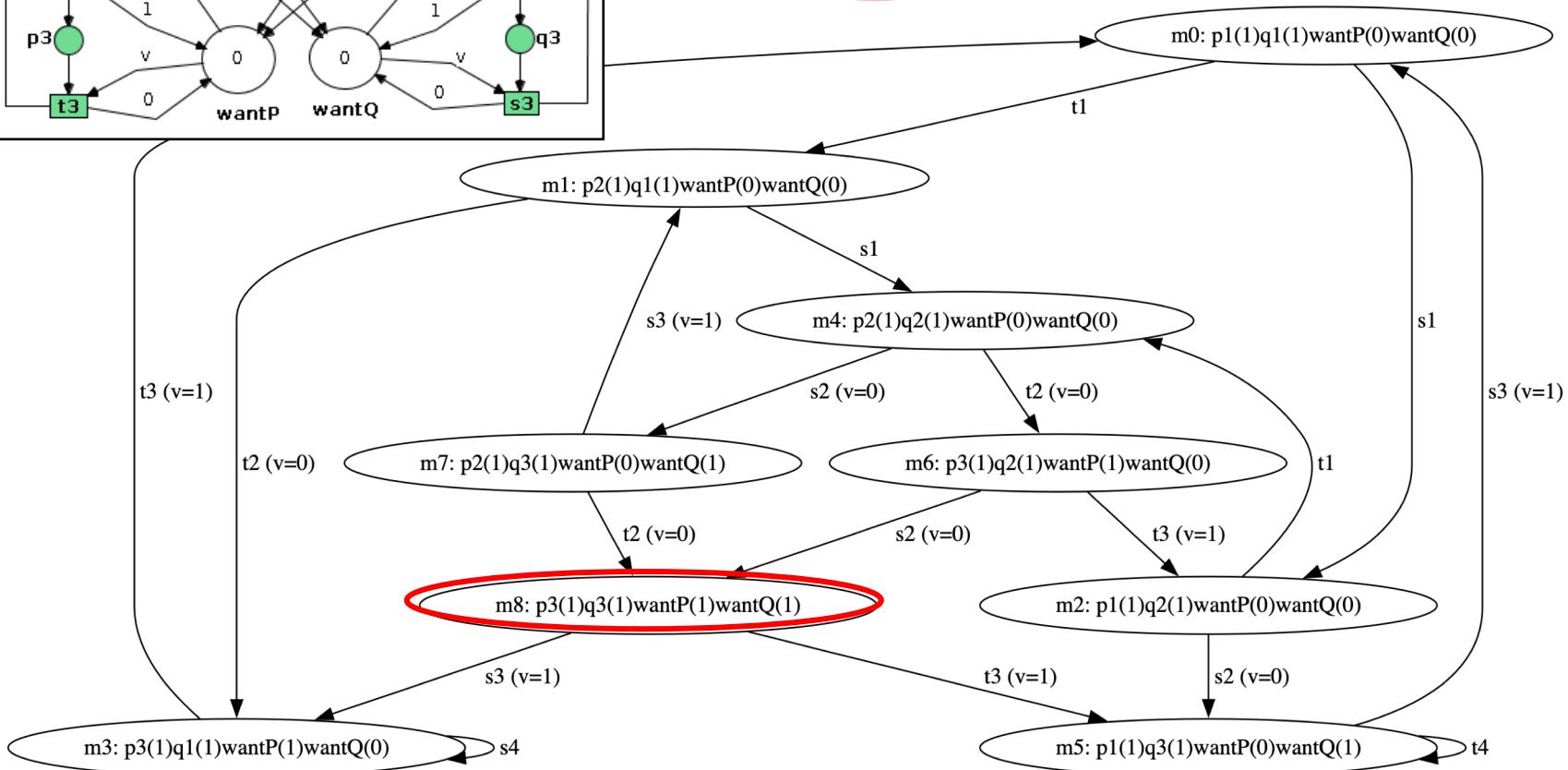
Segundo intento: historia de ejecución



Segundo intento: historia de ejecución



No mutex 😠



Tercer intento: quiero entrar!

- Instrucción de sincronización forma parte de la SC
- Algoritmo:

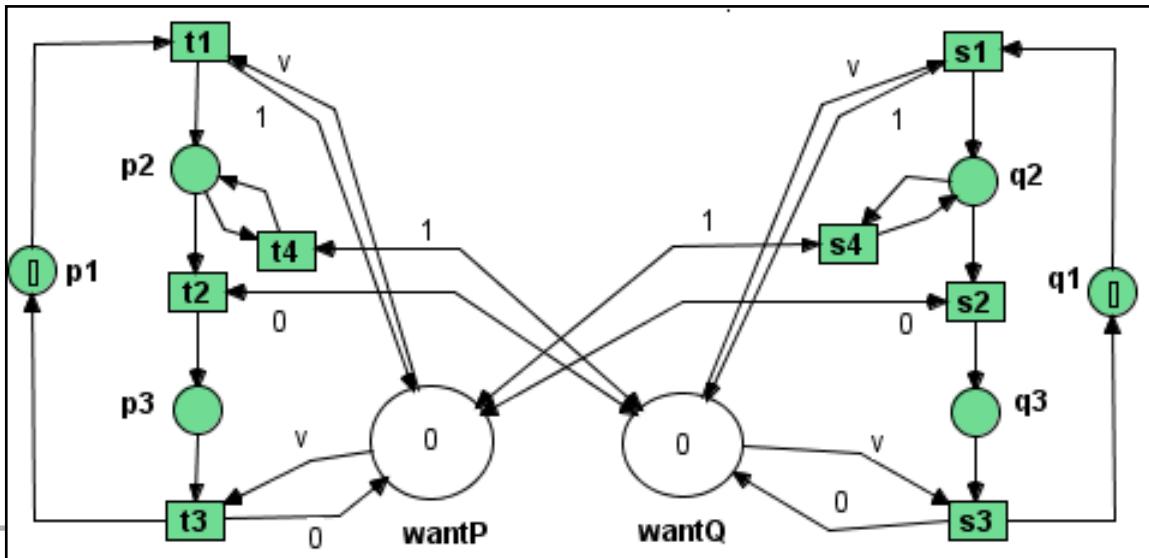
boolean wantP := false, wantQ := false		
	<i>Process P</i>	<i>Process Q</i>
	while true	while true
	SNC	SNC
p1	wantP := true	wantQ := true
p2	while wantQ = true;	while wantP = true;
	SC	SC
p3	wantP := false	wantQ := false

q1
q2
q3

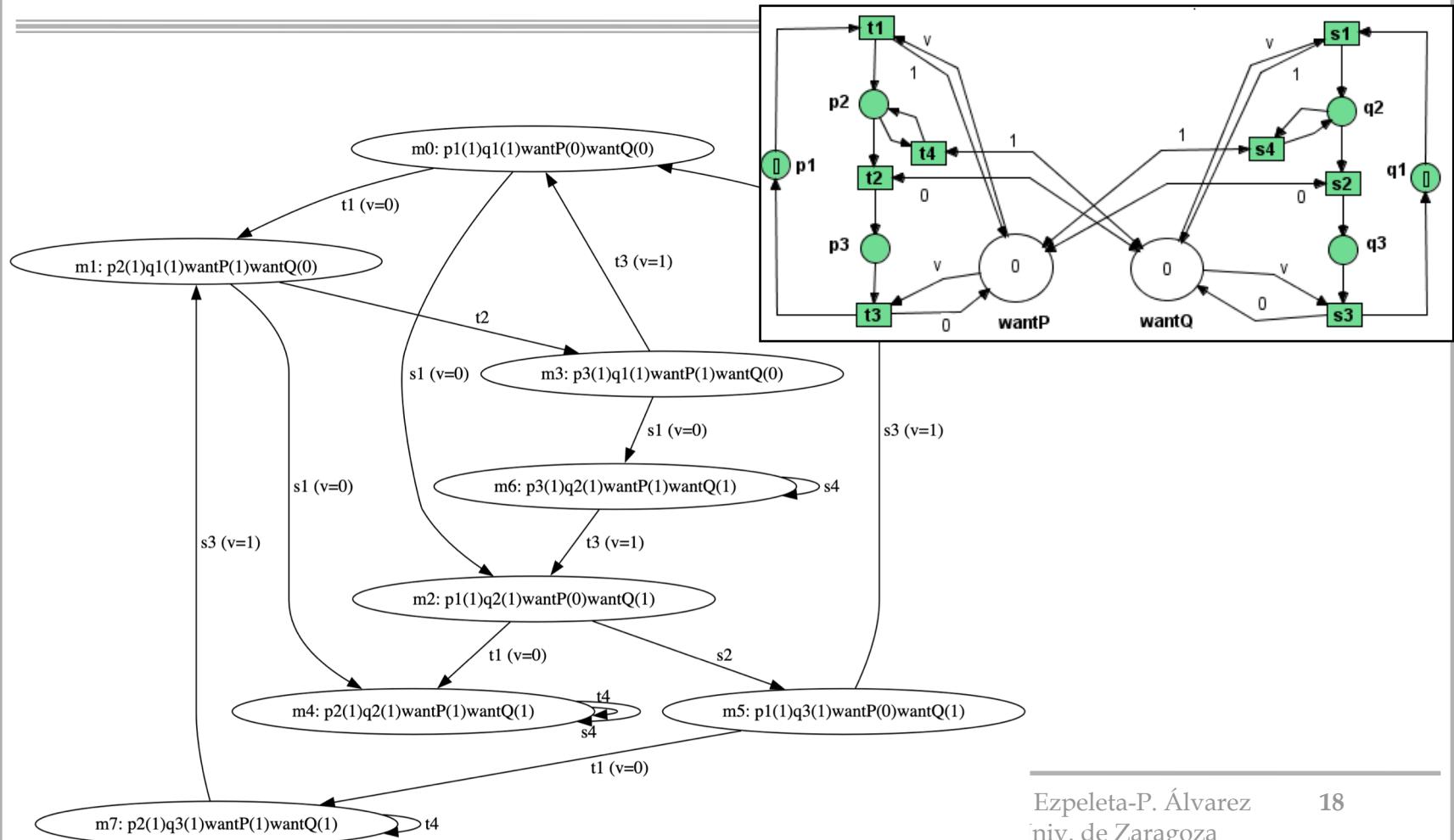
Tercer intento: modelo

boolean wantP := false, wantQ := false	
Process P	Process Q
while true	while true
SNC	SNC
p1 wantP := true	wantQ := true
p2 while wantQ = true;	while wantP = true;
SC	SC
p3 wantP := false	wantQ := false

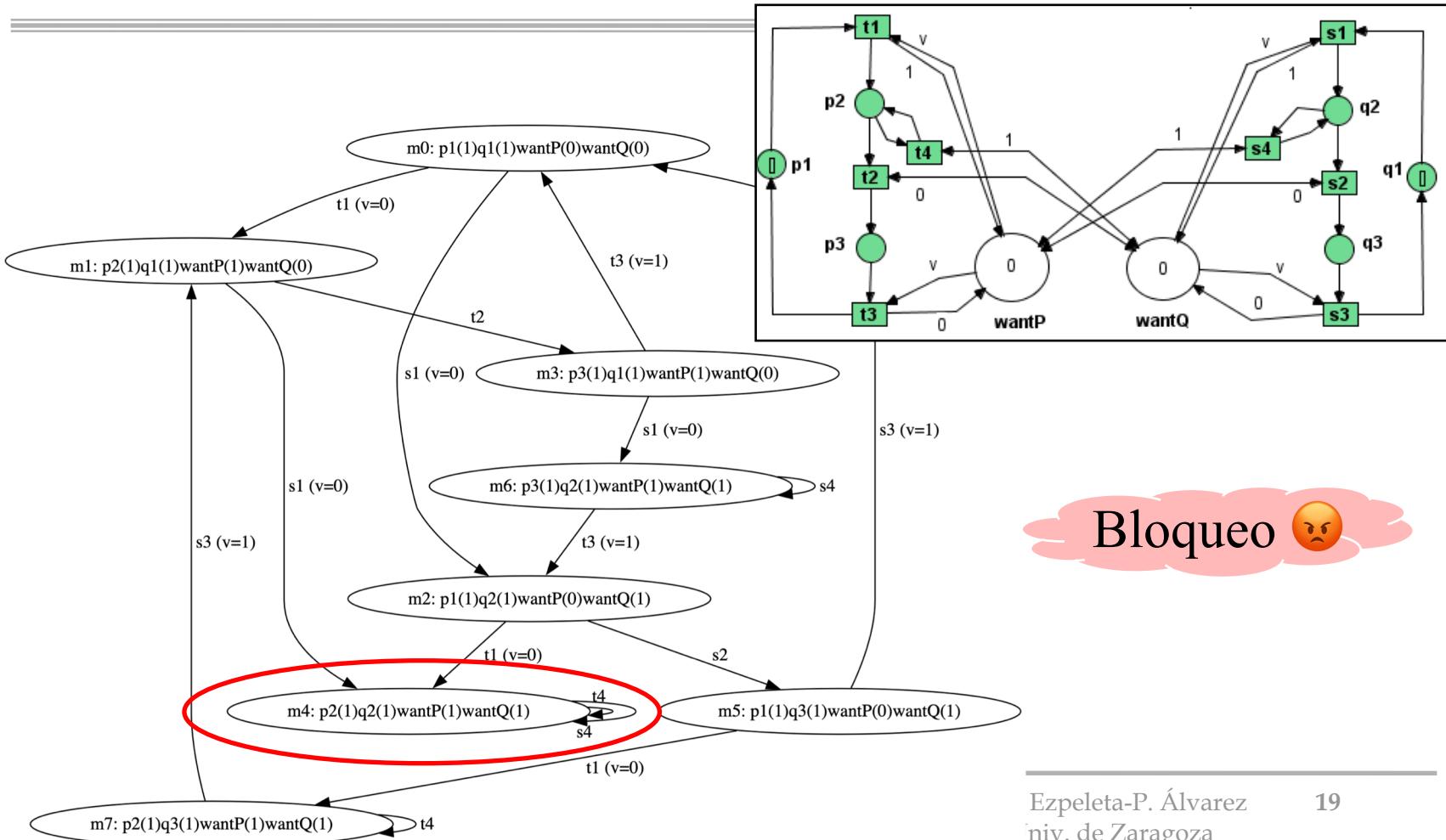
q1
q2
q3



Tercer intento: historia de ejecución



Tercer intento: historia de ejecución



Cuarto intento: quiero entrar pero te doy una oportunidad

- Un proceso no debe empecinarse en entrar a su SC
- Algoritmo:

boolean wantP := false, wantQ := false	
<i>Process P</i>	<i>Process Q</i>
while true	while true
SNC	SNC
p1 wantP := true	wantQ := true
p2 while wantQ = true	while wantP = true
p3 wantP := false	wantQ := false
p4 wantP := true	wantQ := true
p5 SC	SC
wantP := false	wantQ := false

q1
q2
q3
q4
q5

Cuarto intento: modelo

```
boolean wantP := false, wantQ := false
```

Process P

```
while true
```

SNC

p1 wantP := true

p2 while wantQ = true

p3 wantP := false

p4 wantP := true

SC

p5 wantP := false

Process Q

```
while true
```

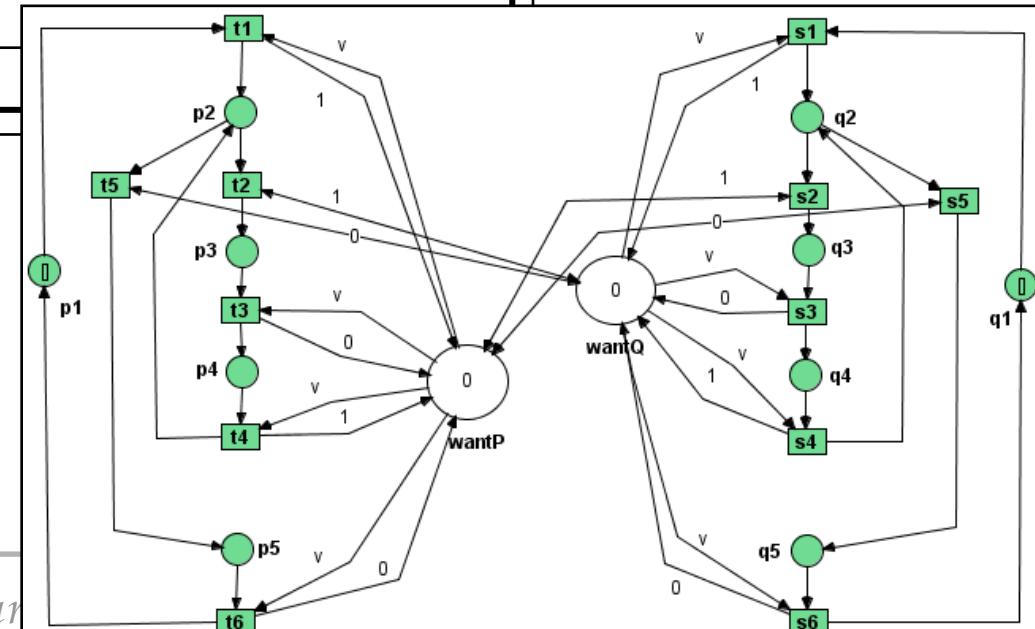
SNC

wantQ := true q1

while wantP = true q2

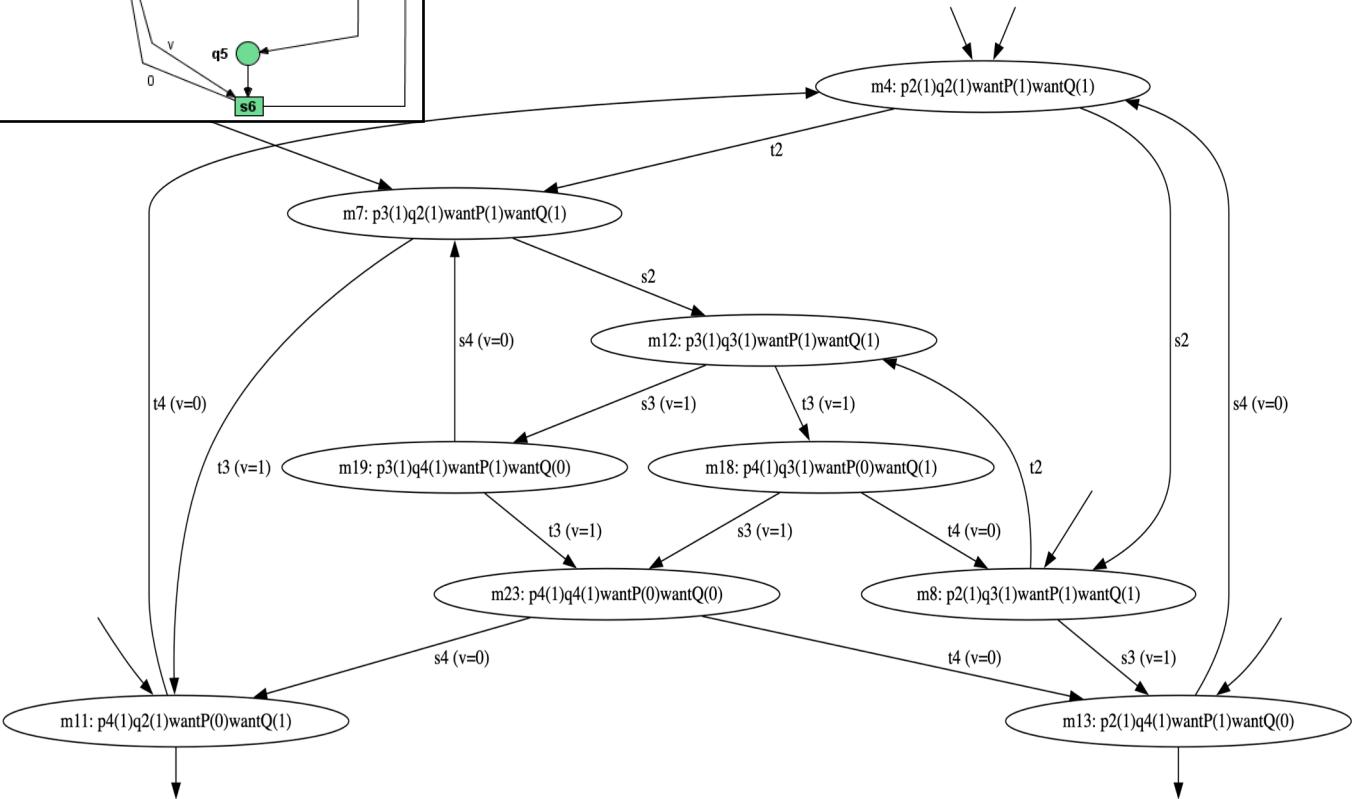
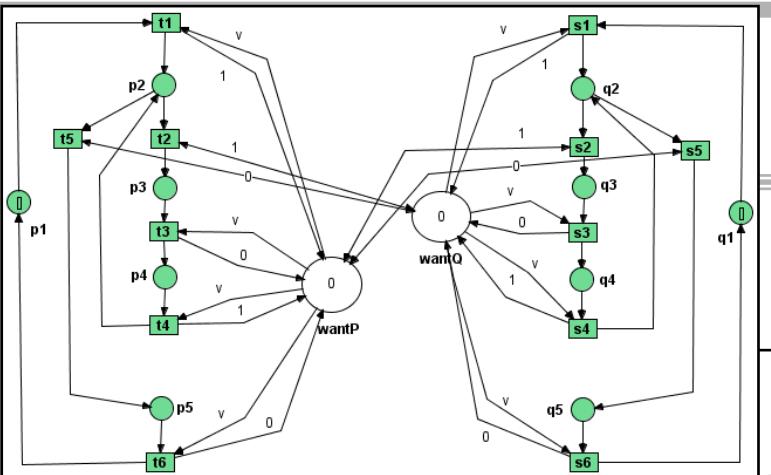
wantQ := false q3

wantQ := true q4



Cuarto intento: historia de ejecución

24 estados

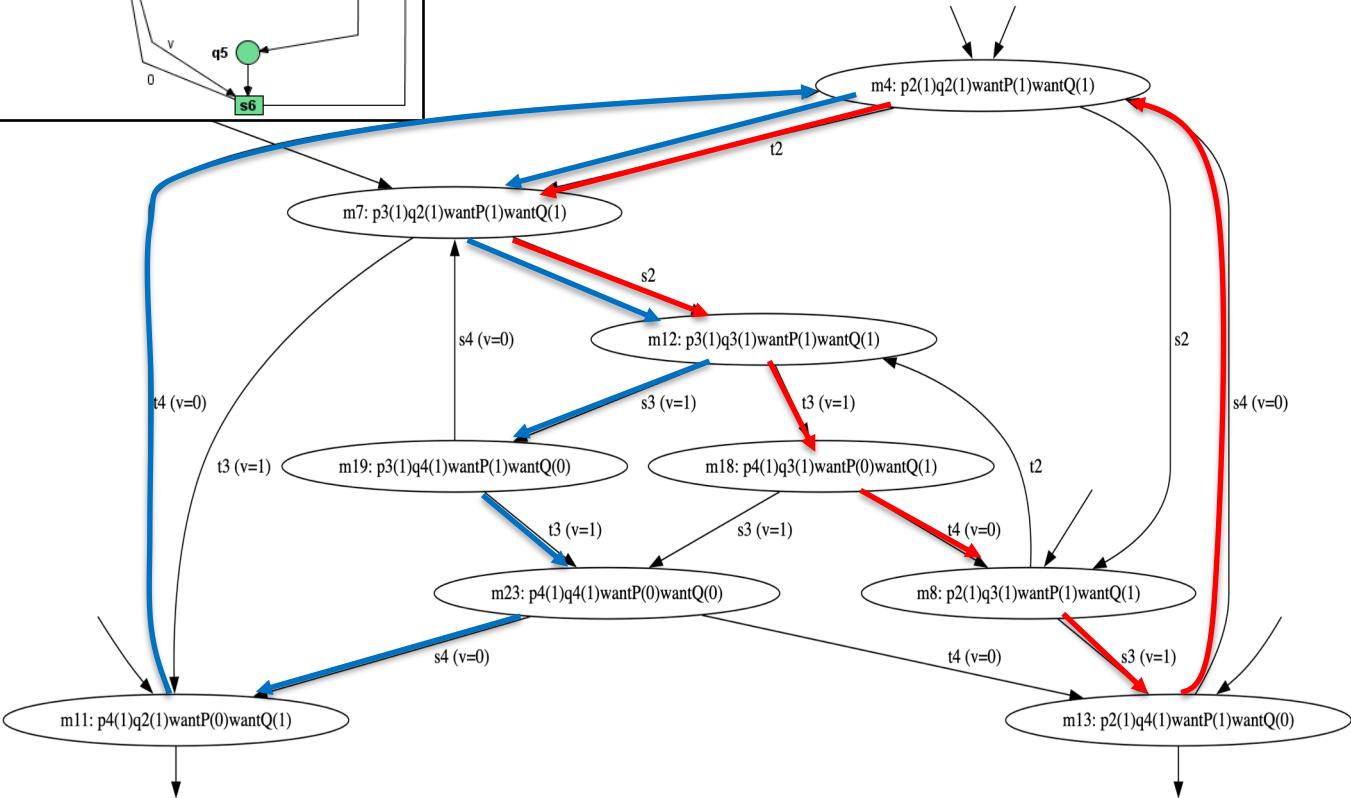
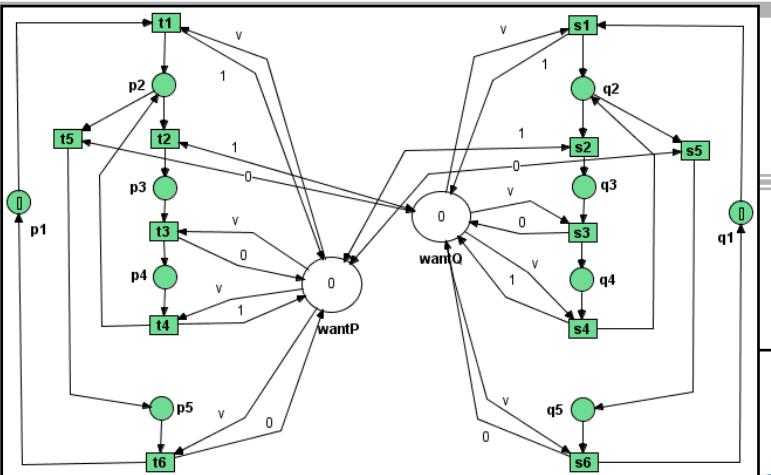


Programación

Cuarto intento: historia de ejecución

Livelocks 😠

24 estados



Programación

Varios intentos (con solo dos procesos!) ...

Intento	Var.	Mutex	Ausencia bloqueos	Ausencia inanición
1º. "primero uno y después otro"	turn	✓	✓	✗
2º. "una variable para cada proceso"	wantP, wantQ	✗	✓	✓
3º. "quiero entrar!"	wantP, wantQ	✓	✗	✓
4º. "quiero entrar pero te doy una oportunidad"	wantP, wantQ	✓	✗ livelocks	✓

Solución al problema: Algoritmo de Dekker

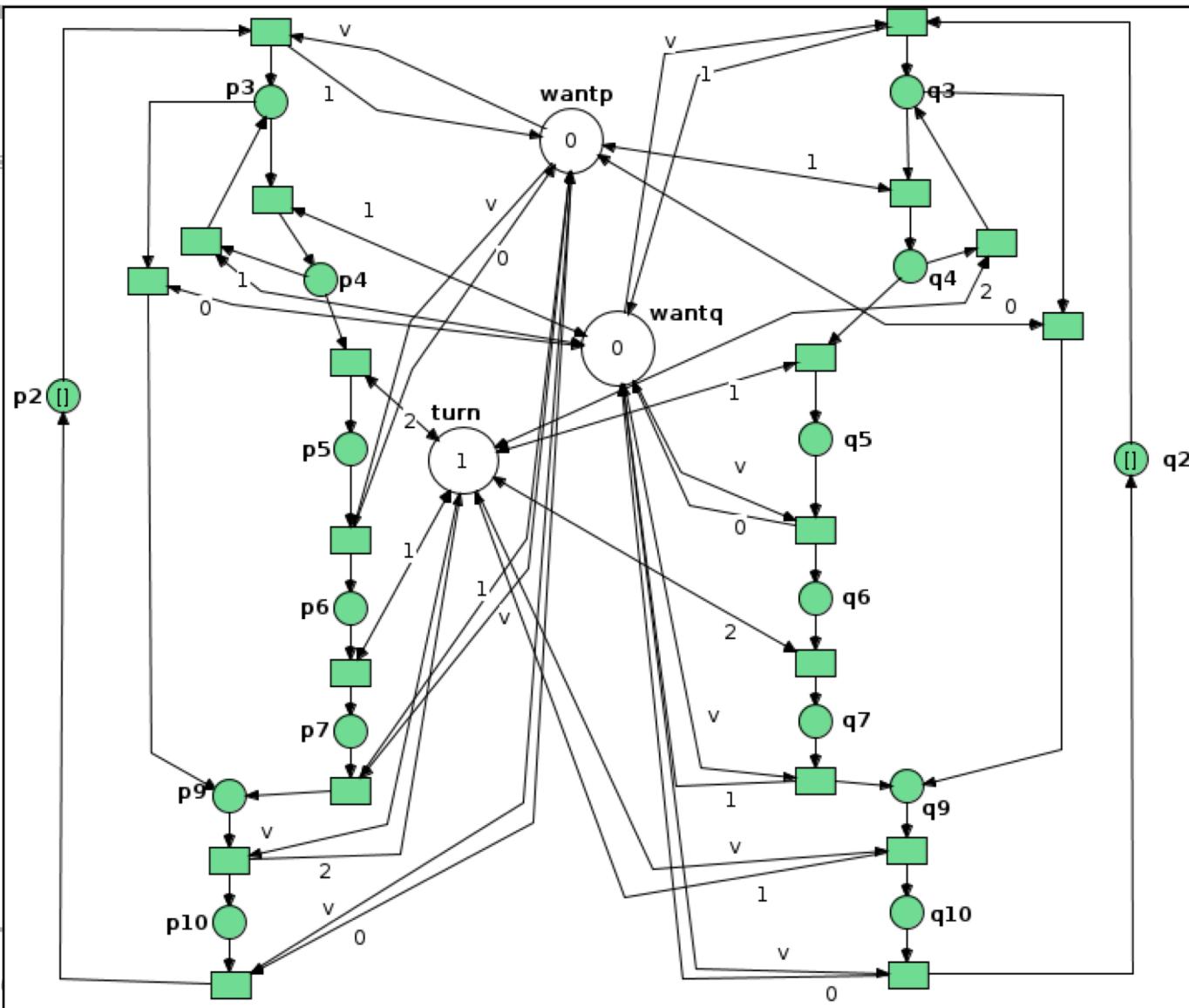
- Algoritmo:

boolean wantP := false, wantQ := false	
integer turn := 1	
Process P	Process Q
while true	while true
SNC	SNC
p2 wantP := true	q2 wantQ := true
p3 while wantQ = true	q3 while wantP = true
p4 if turn = 2	q4 if turn = 1
p5 wantP := false	q5 wantQ := false
p6 await turn = 1	q6 await turn = 2
p7 wantP := true	q7 wantQ := true
SC	SC
p9 turn := 2	q9 turn := 1
p10 wantP := false	q10 wantQ := false

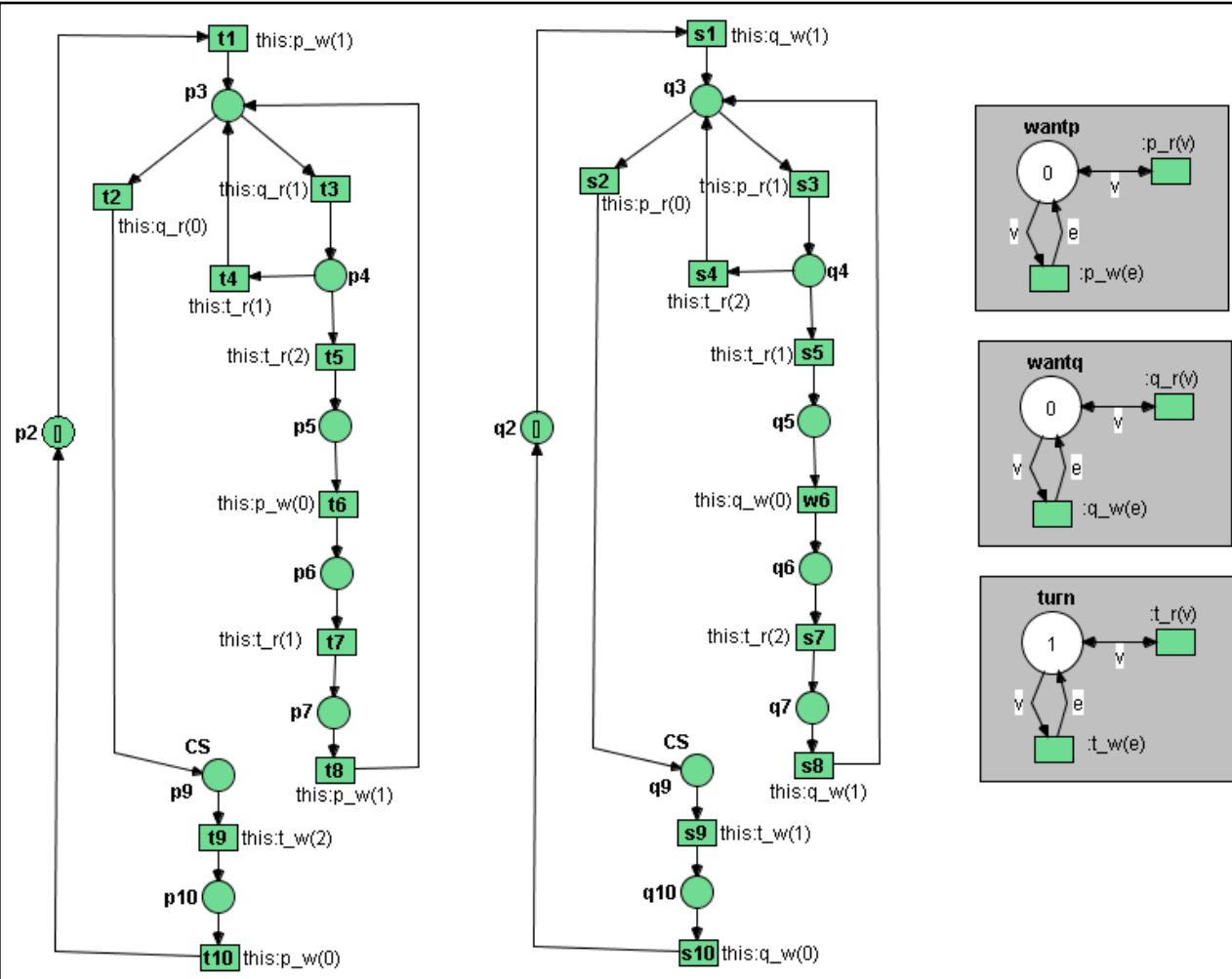
Dekker: modelo



Programación



Dekker: modelo en detalle I

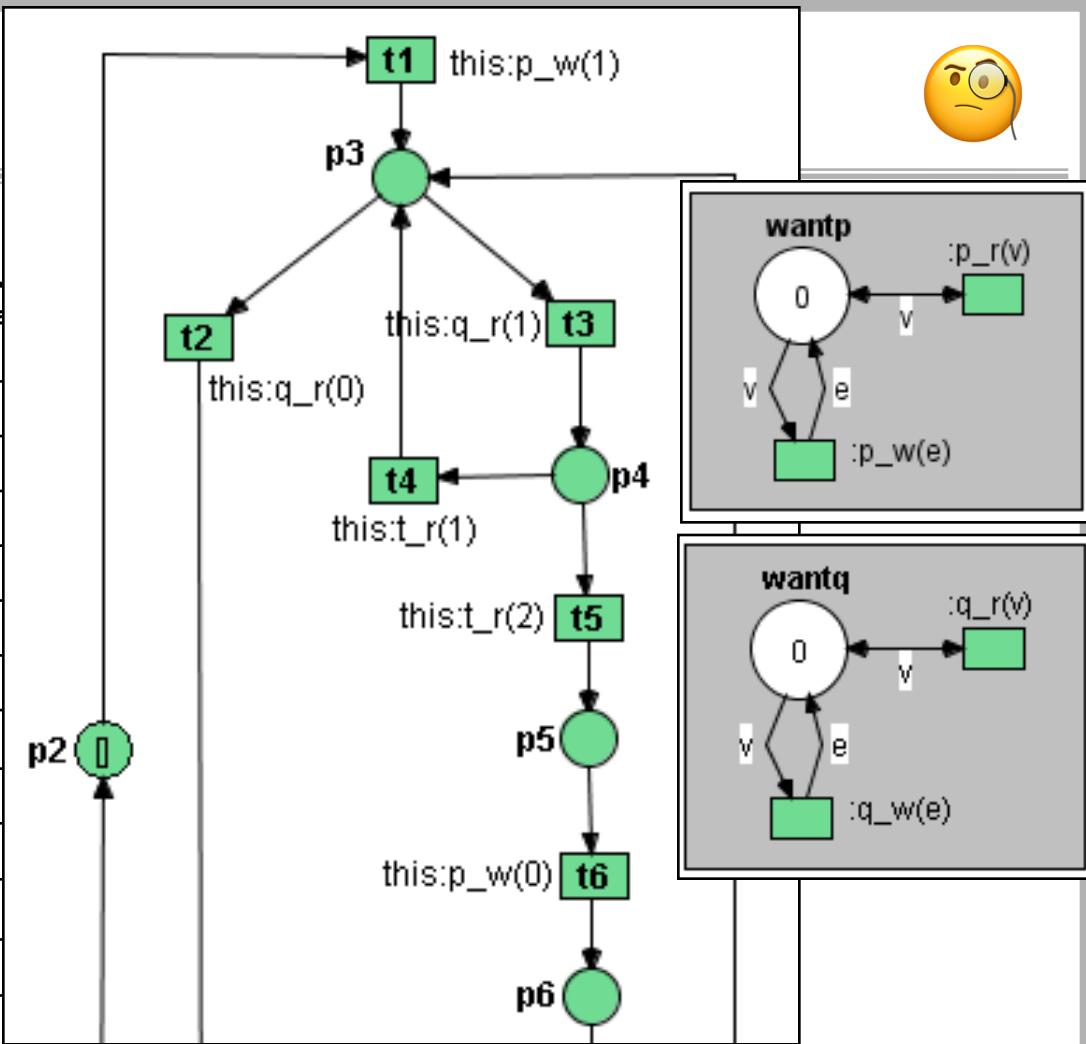


Dekker: modelo en detalle II

- Algoritmo:

```
boolean wantP := false
integer
Process P
while true
    SNC
    wantP := true
    while wantQ = true
        if turn = 2
            wantP := false
            await turn = 1
            wantP := true
        SC
        turn := 2
        wantP := false
```

p2
p3
p4
p5
p6
p7
p9
p10



Dekker: modelo en detalle III

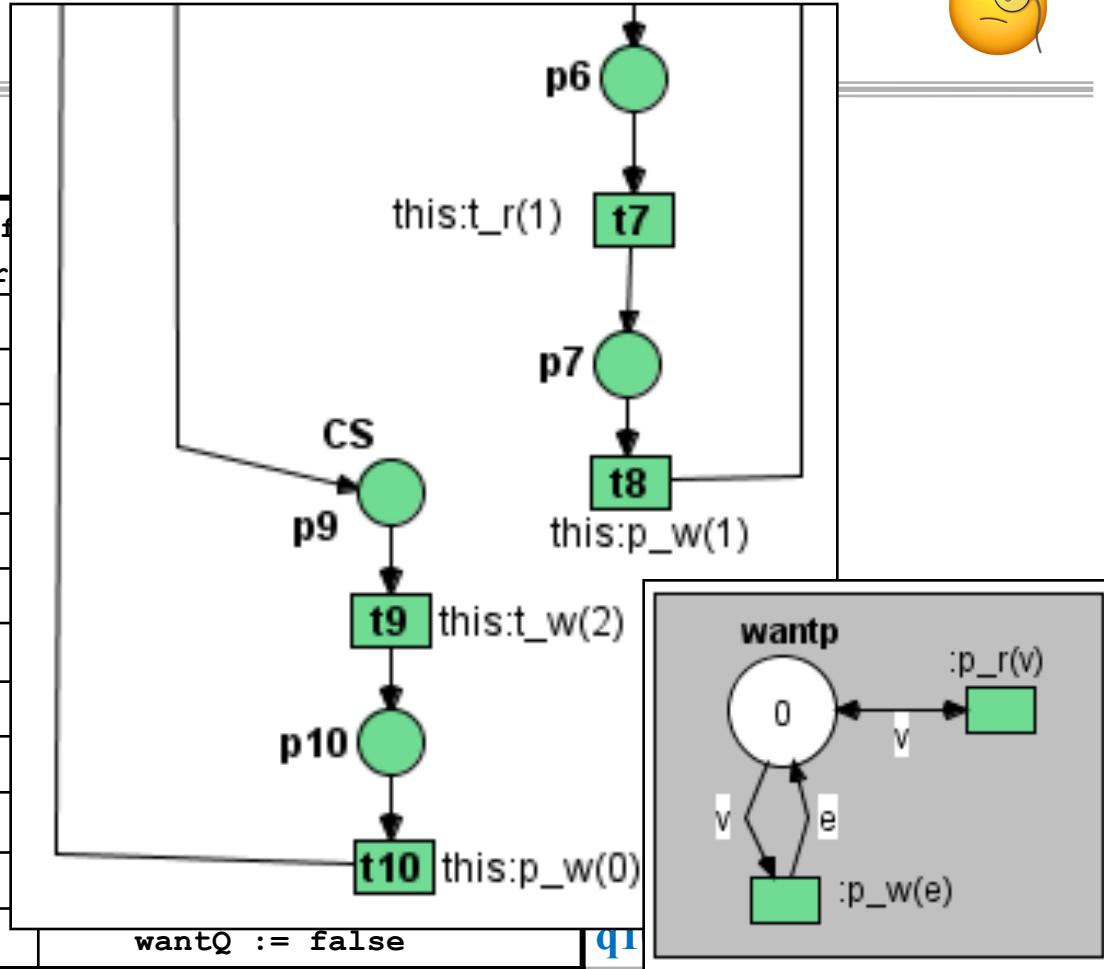


- Algoritmo:

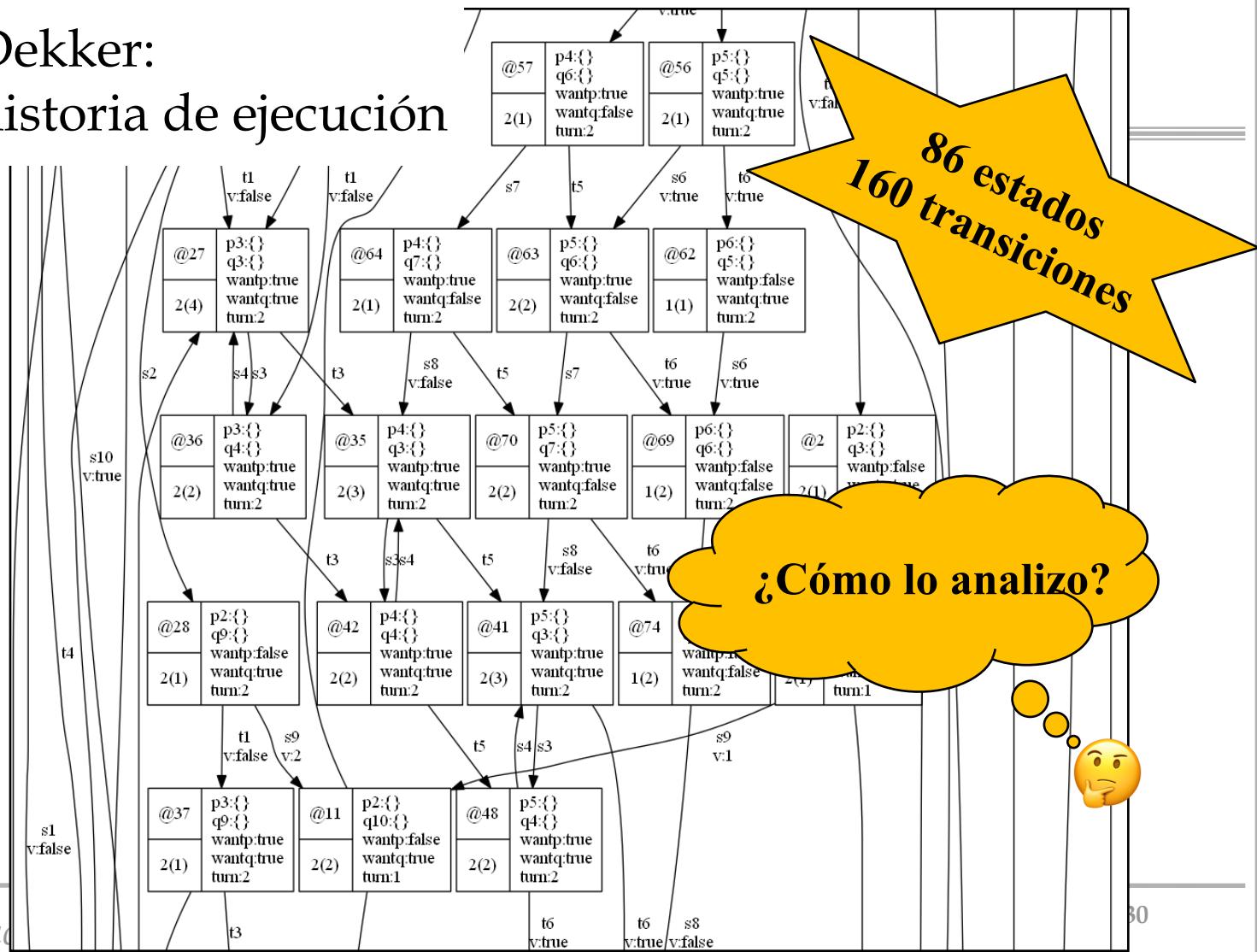
```

        boolean wantP := false
        integer turn := 0
Process P
while true
    SNC
    wantP := true
    while wantQ = true
        if turn = 2
            wantP := false
        await turn = 1
        wantP := true
    SC
    turn := 2
    wantP := false

```



Dekker: historia de ejecución



Solución con herramientas más potentes

```
TS(comun,local) :<local:=comun;comun:=true>
```

- Muchos procesadores cuentan con instrucciones del tipo “test-and-set” atómico
- Y vale para n procesos

boolean ocup := false	
Process P	Process Q
boolean no_ent	boolean no_ent
while true	while true
SNC	SNC
TS(ocup,no_ent)	TS(ocup,no_ent)
while no_ent = true	while no_ent = true
TS(ocup,no_ent)	TS(ocup,no_ent)
SC	SC
ocup := false	ocup := false

Solución con herramientas más potentes

- Pero ¿existe eso?

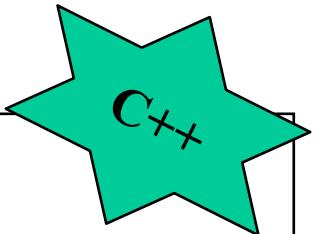
- **Tipo test&set**
 - IBM370, M68040, VAX, [SPARC]
- **Tipo compare&swap**
 - IBM 370, Pentium, ARM (LDREX, STREX)
- **Tipo swap-atomic**
 - SPARC, MC88100
- **Tipo load-locked/store-conditional**
 - Alpha, MIPS (>= R4000)
- **Tipo Fetch&add**
 - CONVEX

Solución con herramientas más potentes

- También nos lo suministran a alto nivel

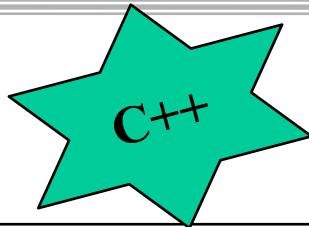
```
int x = 0;
atomic_flag tas = ATOMIC_FLAG_INIT; //false

void inc() {
    for (int i=0; i<N_ITER; i++) {
        while (tas.test_and_set()) { //prot. entrada
            this_thread::yield();
        }
        x++;
        tas.clear(); //protocolo salida
    }
}
```



Manual en línea:

https://cplusplus.com/reference/atomic/atomic_flag/test_and_set/



std::atomic_flag::clear

```
void clear (memory_order sync = memory_order_seq_cst) volatile;
```

Clear flag

Clears the [atomic_flag](#) (i.e., sets it to false).

Clearing the [atomic_flag](#) makes the next call to member [atomic_flag::test_and_set](#) on this object return false.

std::atomic_flag::test_and_set

```
bool test_and_set (memory_order sync = memory_order_seq_cst) volatile noexcept; bool test_and_set (memory_order sync = memory_order_seq_cst) volatile noexcept;
```

Test and set flag

Sets the [atomic_flag](#) and returns whether it was already set immediately before the call.

The entire operation is atomic (an **atomic read-modify-write** operation): the value is not affected by other threads between the instant its value is read (to be returned) and the moment it is modified by this function.

class

std::atomic_flag

```
struct atomic_flag;
```

Atomic flag

Atomic flags are boolean atomic objects that support two operations: [test-and-set](#) and [clear](#).

Atomic flags are *lock-free* (this is the only type guaranteed to be *lock-free* on all library implementations).

fx Member functions

(constructor)	Construct atomic flag (public member function)
test_and_set	Test and set flag (public member function)
clear	Clear flag (public member function)
operator= is deleted (non-copyable/moveable).	

Algoritmo de la Panadería (N procesos)

- Algoritmo:

```
integer array[1..N] number := (1..N,0)

Process P(i:1..N)

while true

SNC

number[i] := 1 + max(number)

for all other processes j

    await (number[j] = 0) or (number[i] << number[j])

SC

number[i] := 0
```

(number[i] < number[j]) or
((number[i] = number[j]) and (i < j))

Algoritmo de turno de espera (N procesos)

- Algoritmo:

```
integer number := 0, next:= 0
integer array[1..N] turn := (1..N,0)
```

Process P(i:1..N)

while true

turn[i] := assignTurn(number, 1)

while turn[i] <> next

follow

SC

next := next + 1

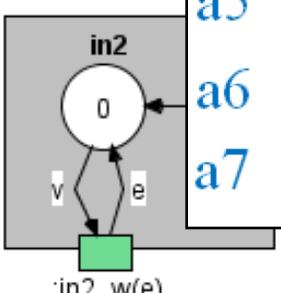
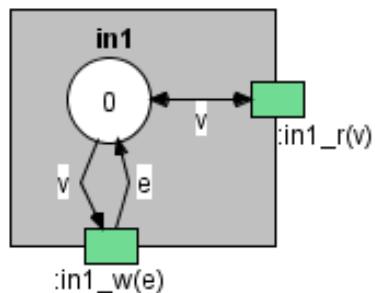
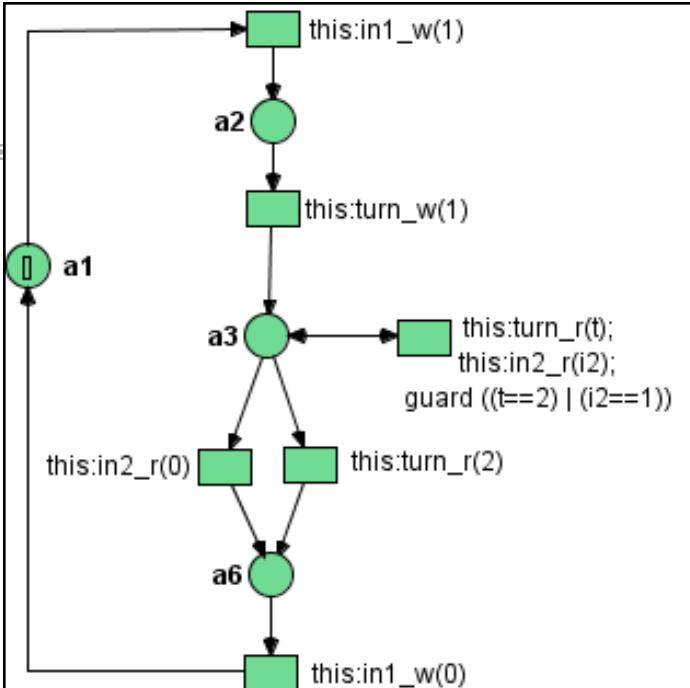
SNC

Cada proceso turno único

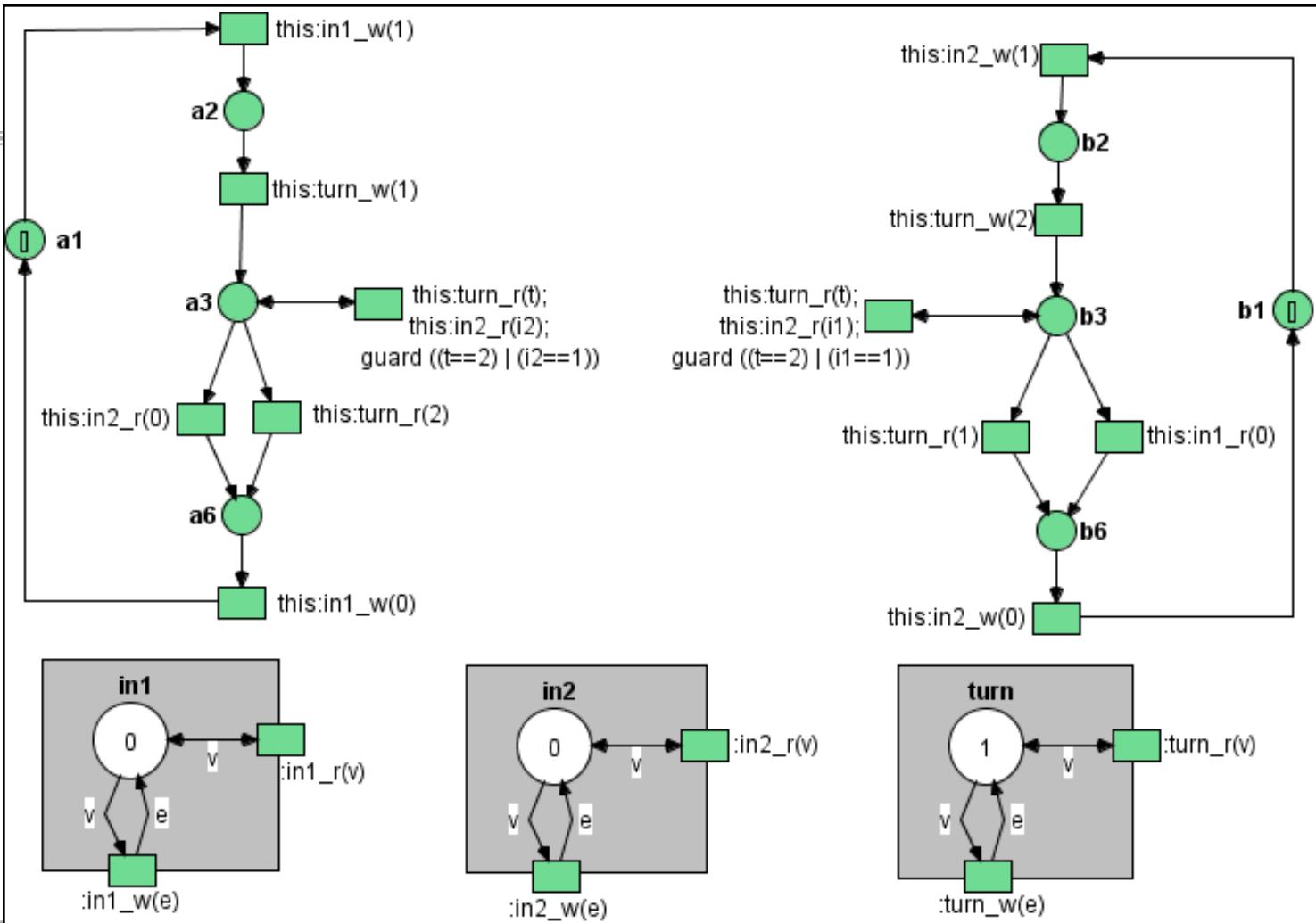
```
assignTurn (value, increment) {
    temp:= value
    value := value + increment
    return (temp) }
```

El algoritmo de Peterson

	boolean en1 := false, en2 := false int ult := 1	
	<i>Process P</i>	<i>Process Q</i>
a1	while true	while true
a2	en1 := true	en2 := true
a3	ult := 1	ult := 2
a4	while en2 and (ult = 1) seguir	while en1 and (ult = 2) seguir
a5	SC	SC
a6	en1 := false	en2 := false
a7	SNC	SNC



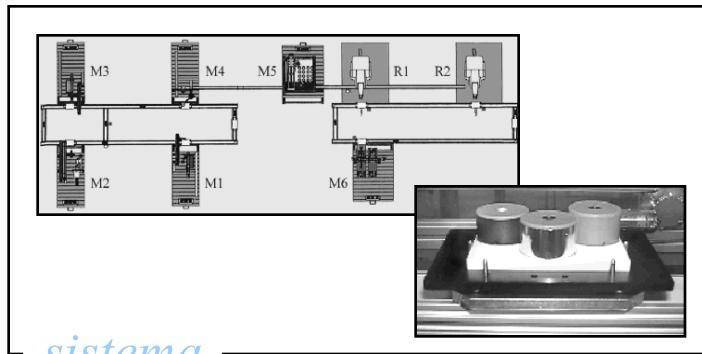
	boolean en1 := false
	int ult := 0
<i>Process P</i>	
while true	
a1	en1 := true
a2	ult := 1
a3	while en2 and (ult = 1)
a4	seguir
a5	SC
a6	en1 := false
a7	SNC



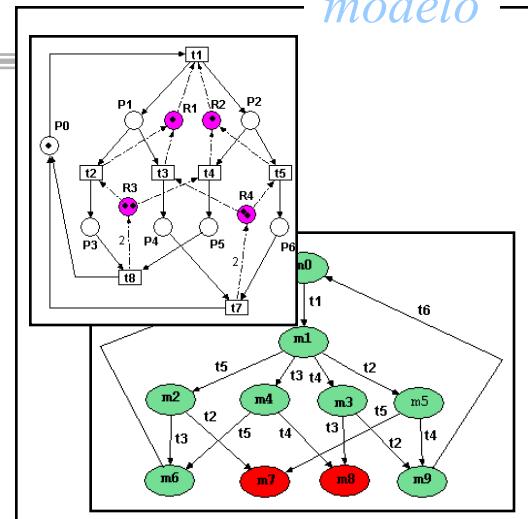
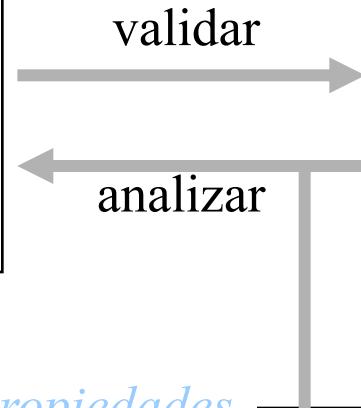
Lección 4: Breve introducción a la lógica temporal y el “model checking”

- Sistemas y modelos, otra vez
- ¿Qué es la lógica temporal (lineal)?
- ¿Qué es el “model checking”?
- La herramienta María
- ¿Qué pasa con el algoritmo de Dekker?

Sistemas y modelos



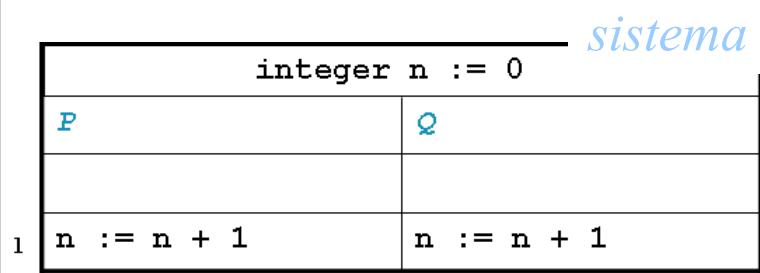
sistema



requisitos/propiedades

1- 1000 piezas al mes
2- piezas correctas
3- producción continua
...

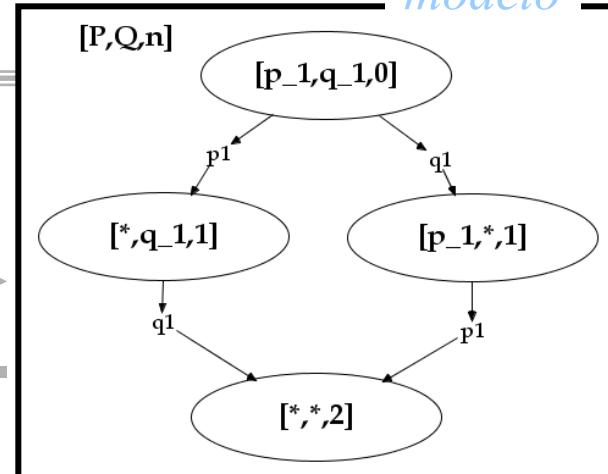
Sistemas y modelos



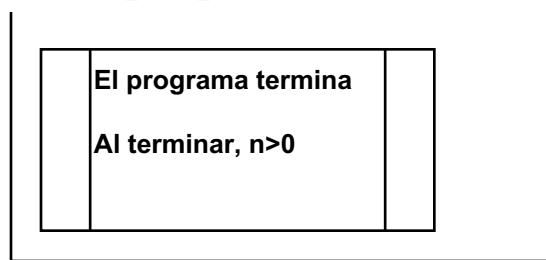
sistema

validar

analizar



requisitos/propiedades



¿Qué es la lógica temporal (lineal)?

- Desarrollada por **Clarke y Emerson** a principios de los 80
 - “*Automatic verification of finite-state concurrent systems using temporal logic specifications*”
 - E. M. Clarke , E. A. Emerson , A. P. Sistla
 - ACM Transactions on Programming Languages and Systems
- Se obtiene añadiendo operadores “temporales” a una lógica proposicional/de primer orden
 - tiempo como cambio de estado

¿Qué es la lógica temporal (lineal)?

- Aserciones
 - "Tengo hambre"
 - "María es alta"
 - "Siempre tengo hambre"
 - "Mañana tendré hambre"
 - "Tendré hambre hasta que coma algo"
- Dos tipos habituales de lógica temporal:
 - lineal (LTL): se razona sobre una línea temporal
 - arborescente (CTL): se razona sobre todas las posibles líneas temporales

¿Qué es la lógica temporal (lineal)?

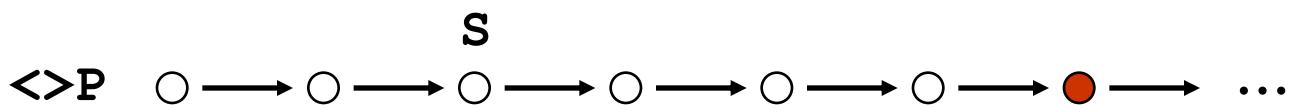
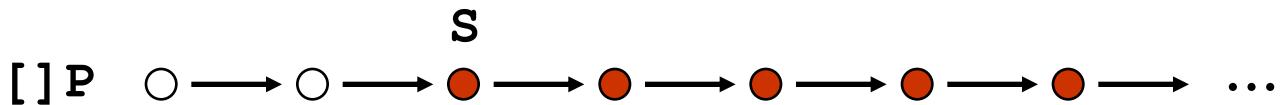
- Elementos básicos:
 - proposiciones atómicas: afirmaciones sobre los estados del sistema
 - operadores booleanos
 - negación (\neg), conjunción (\wedge), disyunción (\vee), implicación (\Rightarrow)
- Ejemplo: $(x > 22) \wedge (y \geq x) \Rightarrow y > 22$
 - “ $x > 22$ ”, “ $y \geq x$ ”, “ $y > 22$ ” son proposiciones atómicas
 - \wedge, \Rightarrow son operadores booleanos
- En términos de programas, la LTL razona sobre las posibles ejecuciones, viéndolas como secuencias infinitas de estados
 - la ejecución de una instrucción cambia el estado

¿Qué es la lógica temporal (lineal)?

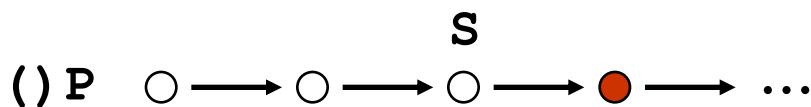
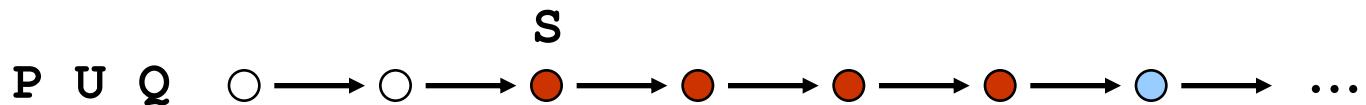
- Añade dos operadores para trabajar con una ejecución:
 - **always**
 - se denota como “[]” (también G)
 - la fórmula “[]P” se cumple en un estado “S” de una ejecución si
 - S satisface P
 - todos los estados posteriores a S en la ejecución satisfacen P
 - **eventually**
 - se denota como “<>” (también F)
 - la fórmula “<>P” se cumple en un estado “S” de una ejecución si
 - o bien S o bien un estado posterior a S en la ejecución satisface P
 - nótese que ambos operadores incluyen al estado S

¿Qué es la lógica temporal (lineal)?

- Significado intuitivo:

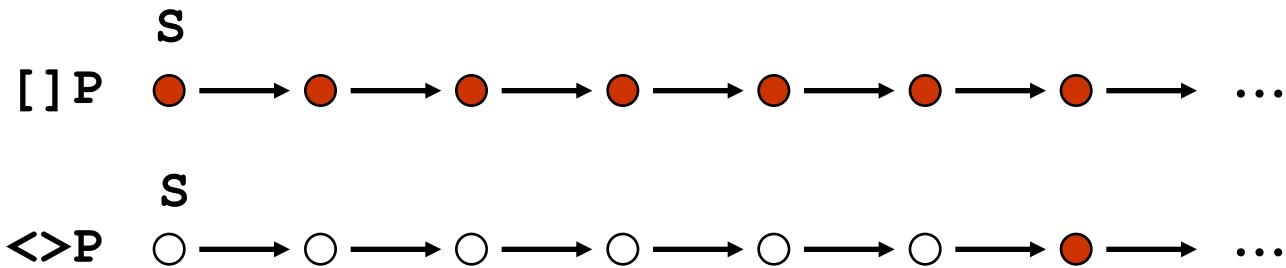


- Por flexibilidad se suelen completar con



¿Qué es la lógica temporal (lineal)?

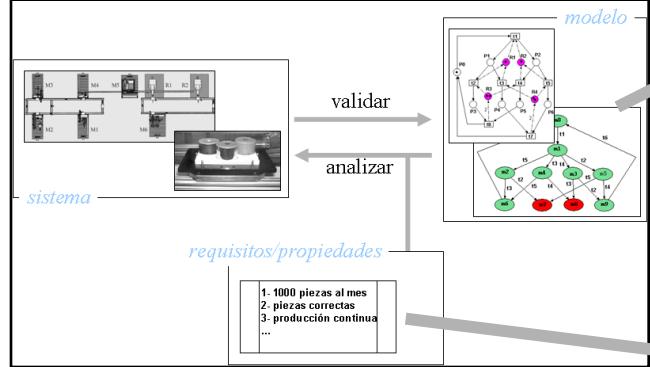
- Normalmente, **s** será el estado inicial del sistema



¿Qué es la lógica temporal (lineal)?

- “ $[]$ ” se usa para propiedades de seguridad
 - $[]\neg P$, siendo P lo malo que no queremos que ocurra
 - “*Siempre ha de ocurrir que dos programas no modifiquen a la vez el mismo registro de la bbdd*”
- “ $\langle \rangle$ ” se usa para propiedades de vivacidad
 - $\langle \rangle P$, siendo P lo bueno que queremos que ocurra
 - “*Todas las transacciones enviadas a la base de datos terminan*”

¿Qué es “model checking”?



estructura
de Kripke

fórmula
LTL

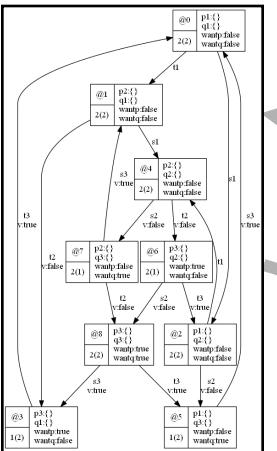
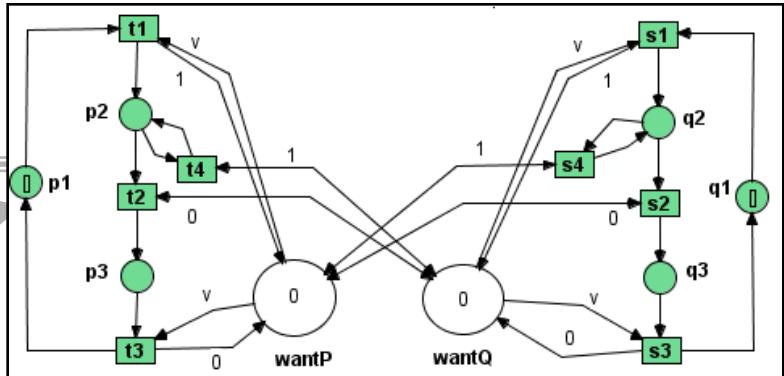


¡OK!

¡MAL!
mira...

boolean wantP := false, wantQ := false	
Process P	Process Q
while true	while true
SNC	SNC
while wantQ = true	while wantP = true
wantP := true	wantQ := true
SC	SC
wantP := false	wantQ := false

ng''?



maria

```

place p1 black_token: {};
place p2 black_token;
place p3 black_token;

place q1 black_token: {};
place q2 black_token;
place q3 black_token;

place wantp bool: false ;
place wantq bool: false ;

trans t1
in { place p1:{}; place wantq: false; }
out { place p2:{}; place wantq: false; }
;
```

“property holds”

```
[ () 
  (place p3 equals empty)
  ||
  (place q3 equals empty)
]
```

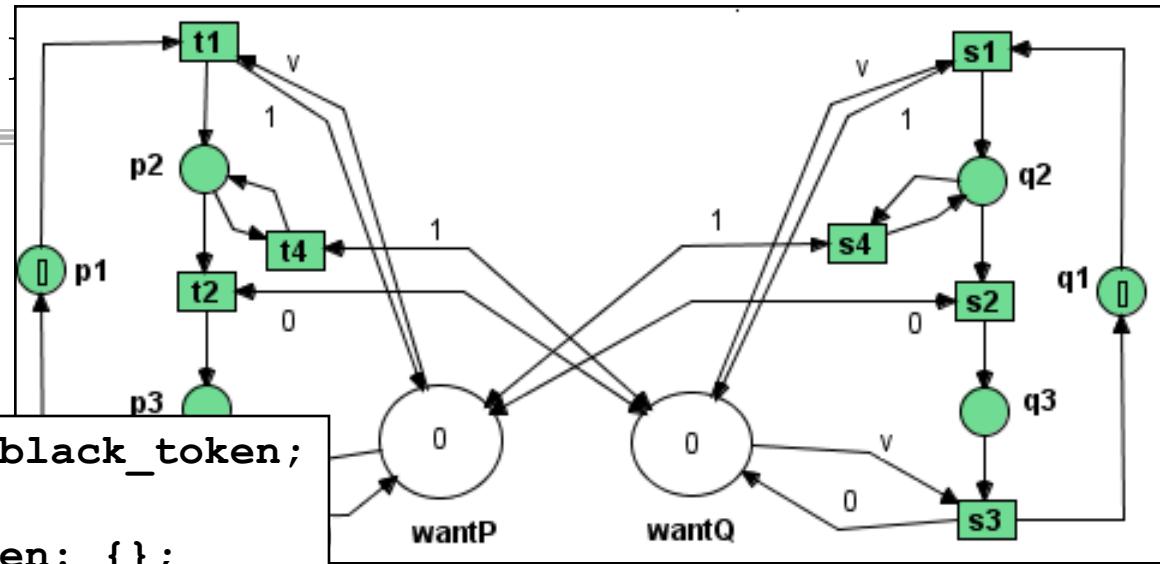


maria

La herramienta María

- Maria
 - <http://www.tcs.hut.fi/Software/maria/index.en.html>
 - “*Maria is a reachability analyzer for concurrent systems that uses Algebraic System Nets (a high-level variant of Petri nets) as its modelling formalism*”
- Marko Mäkelä
 - Laboratory for Theoretical Computer Science (TCS)
 - Helsinki University of Technology (TKK)

La herramienta



```
typedef struct {} black_token;
```

```
place p1 black_token: {};
```

```
place p2 black_token;
```

```
place p3 black_token;
```

```
place q1 black_token: {};
```

```
place q2 black_token;
```

```
place q3 black_token;
```

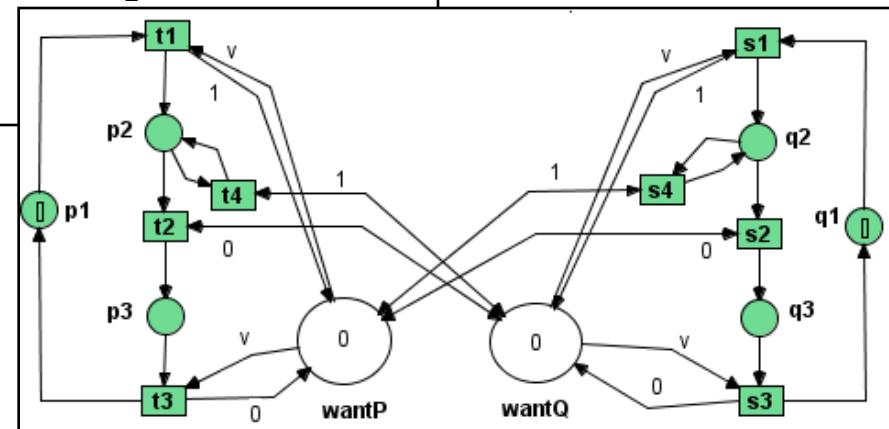
```
place wantp bool: false ;
```

```
place wantq bool: false ;
```

```

trans t1
  in { place p1:{}; place wantp: v; }
  out { place p2:{}; place wantp: true; }
;
trans t2
  in { place p2:{}; place wantq: false; }
  out { place p3:{}; place wantq: false; }
;
trans t3
  in { place p3:{}; place wantp: v; }
  out { place p1:{}; place wantp: false; }
;
. . .

```



La herramienta María

```
/cygdrive/y/datos/cosasDeClase/progConcurrente/maria

director@direccion ~
$ cd "Y:\datos\cosasDeClase\progConcurrente\maria"

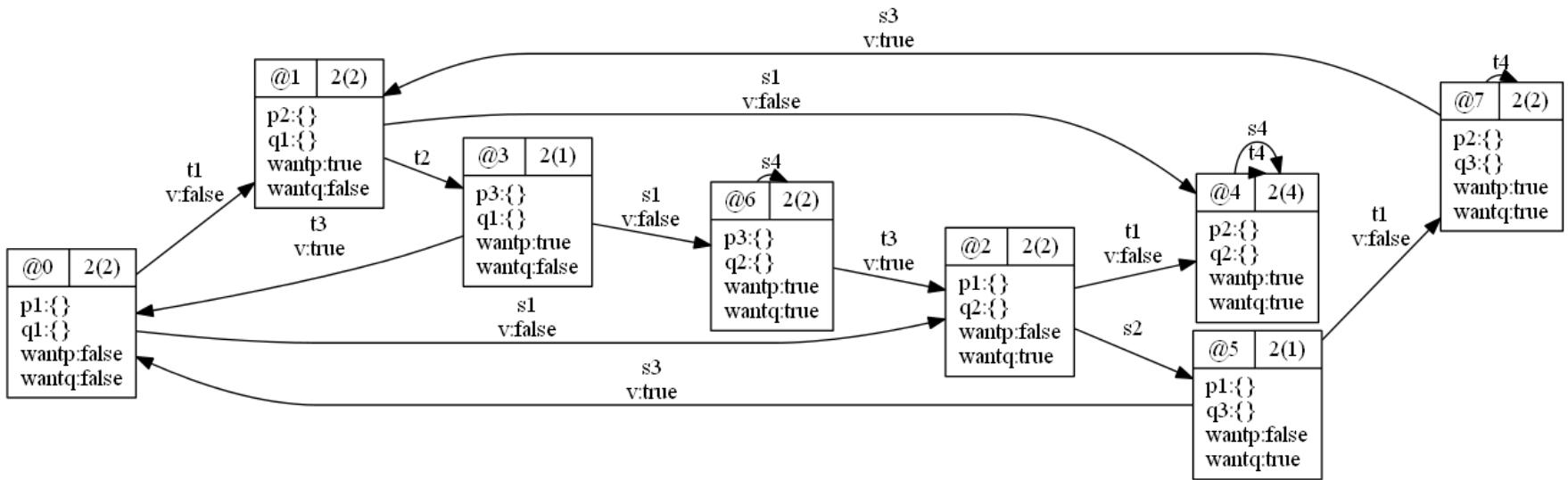
director@direccion /cygdrive/y/datos/cosasDeClase/progConcurrente/maria
$ maria -b L03_tercer_intento.bn
" L03_tercer_intento.bn": 8 states (4 bytes), 12 arcs
@0$
@0$[]((place p3 equals empty) || (place q3 equals empty))
(command line):2:property holds
" L03_tercer_intento.bn": 8 states (4 bytes), 12 arcs
@0$
@0$show @3
@3:state (
  p3:
  {}
  q1:
  {}
  wantp:
  true
  wantq:
  false
)
1 predecessor
2 successors
@0$
@0$visual dumpgraph
@0$
```

La herramienta María

```
@0$exit
```

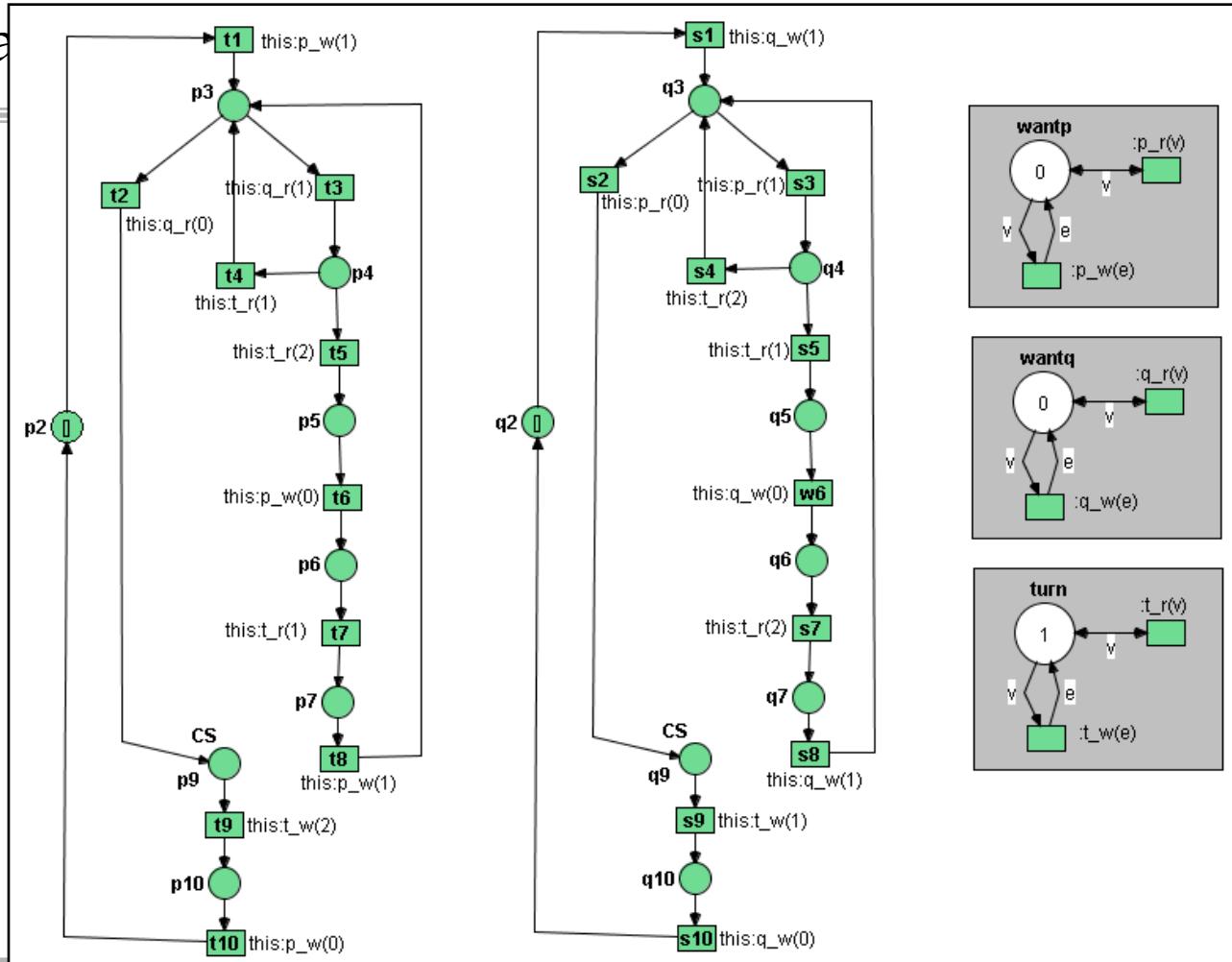
```
director@direccion /cygdrive/y/datos/cosasDeClase/progConcurrente/maria
$ dot -Tpng maria-vis.out -o L03_tercer_intento.png
```

```
director@direccion /cygdrive/y/datos/cosasDeClase/progConcurrente/maria
$ -
```



¿Qué pasa

- ¿Se comporta adecuadamente?



¿Qué pasa con el algoritmo?

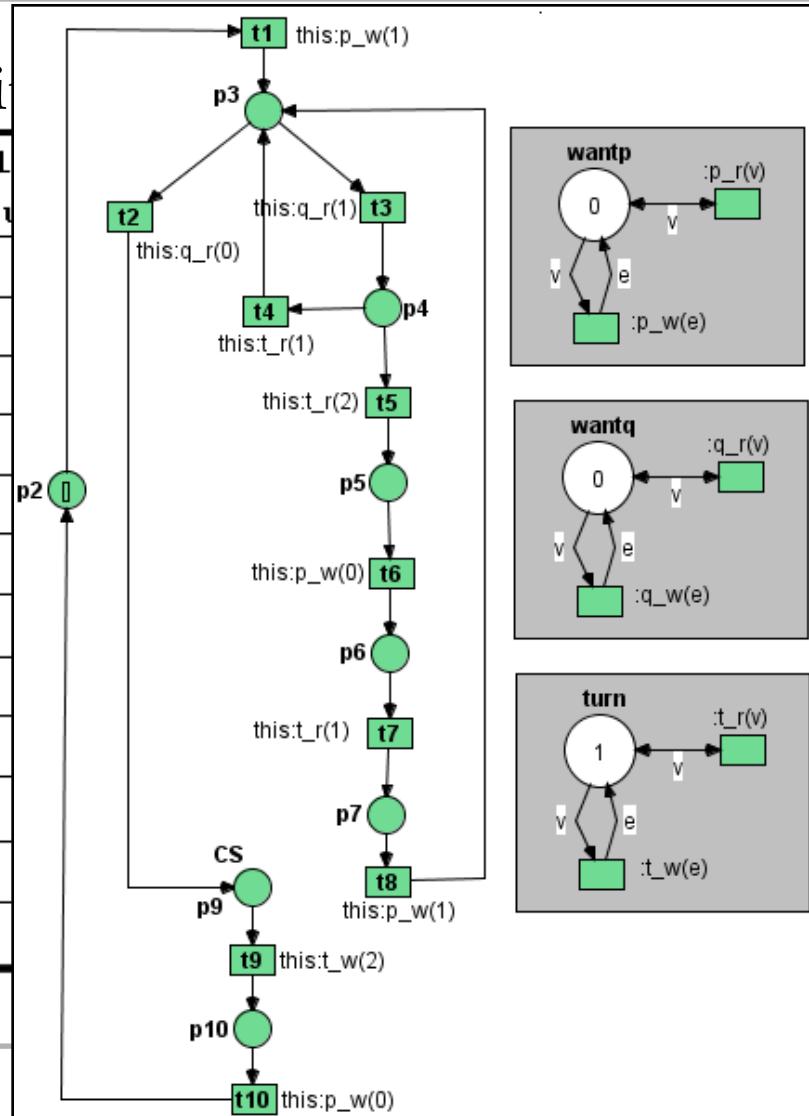
p2
p3
p4
p5
p6
p7
p9
p10

```

boolean wantP := false
integer turn

Process P
loop forever
    SNC
        wantP := true
    while wantQ
        if turn = 2
            wantP := false
        await turn = 1
        wantP := true
    SC
        turn := 2
    wantP := false

```

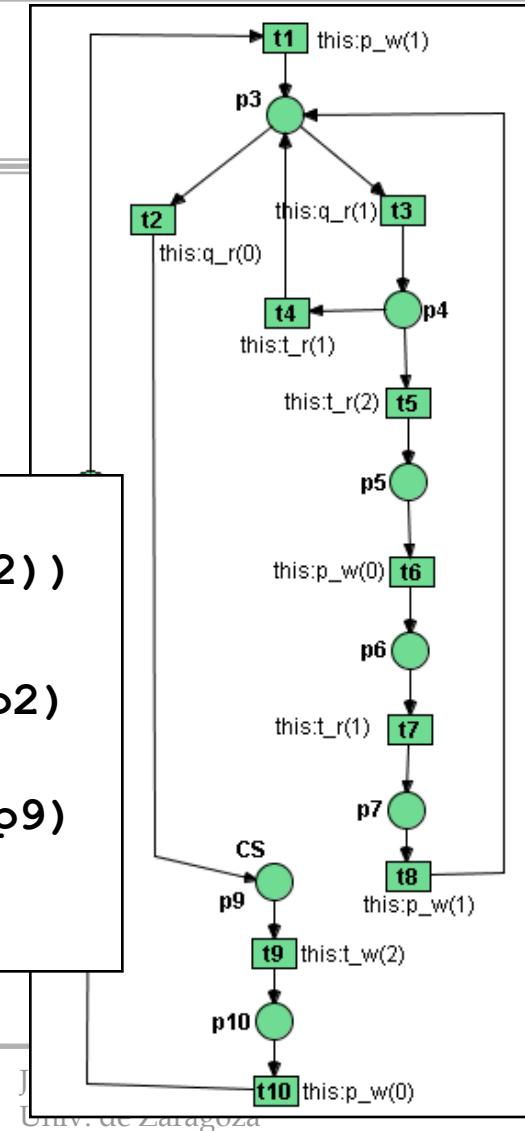


¿Qué pasa con el algoritmo de Dekker?

- Propiedades: un proceso solito puede ejecutarse

```
[] ([] q2) => (<>p2 AND <>p9))
```

```
[] (
  ([](is black_token {} subset place q2))
=>
  ( <>(is black_token {} subset place p2)
    &&
    <>(is black_token {} subset place p9)
  )
)
```

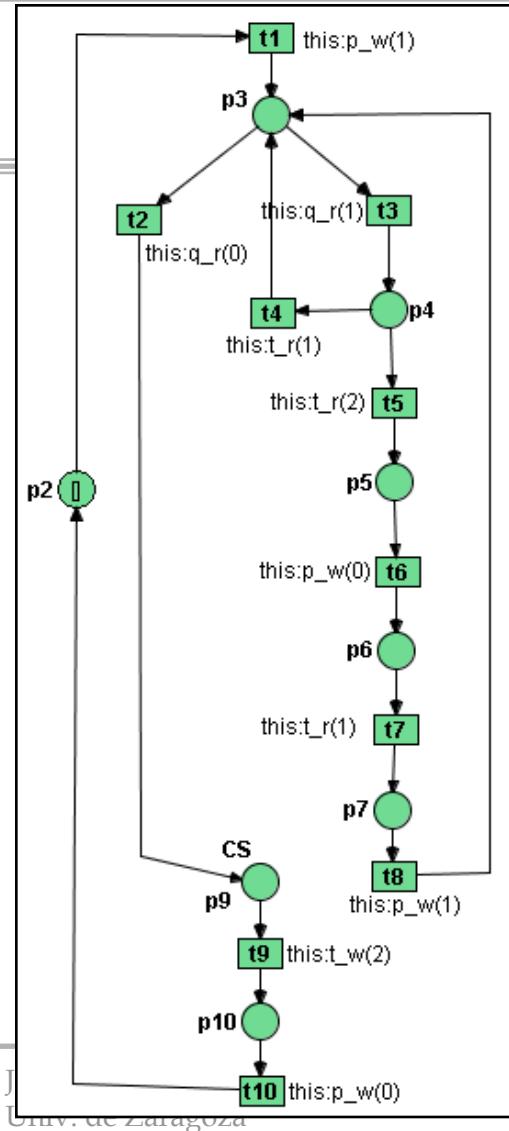


¿Qué pasa con el algoritmo de Dekker?

- Propiedades: el acceso a las SC es en exclusión mutua

```
[] (!p9 OR !q9)
```

```
[] (
    (place p9 equals empty)
    ||
    (place q9 equals empty)
)
```



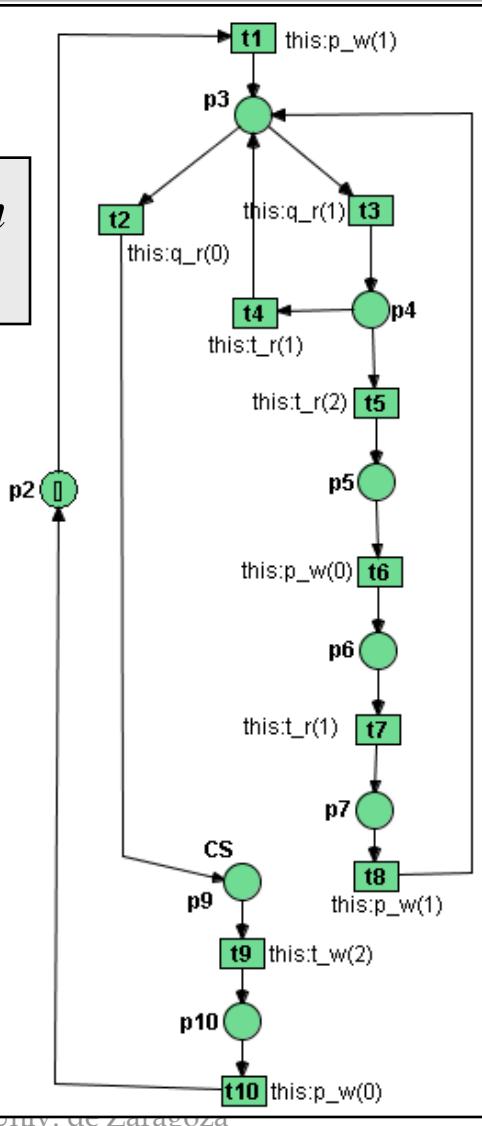
¿Qué pasa con el algoritmo de Dekker?

- Propiedades: no hay bloqueos

```
[] ( p2 OR  
    (p3 AND wantq=false) OR  
    (p3 AND wantq=true) OR  
    (p4 AND turn=1) OR  
    (p4 AND turn=2) OR  
    p5 OR  
    (p6 AND turn=1) OR  
    p7 OR  
    p9 OR  
    p10 OR  
    q2 OR  
    ...  
)
```

deadlock fatal;

alguna transición sensibilizada

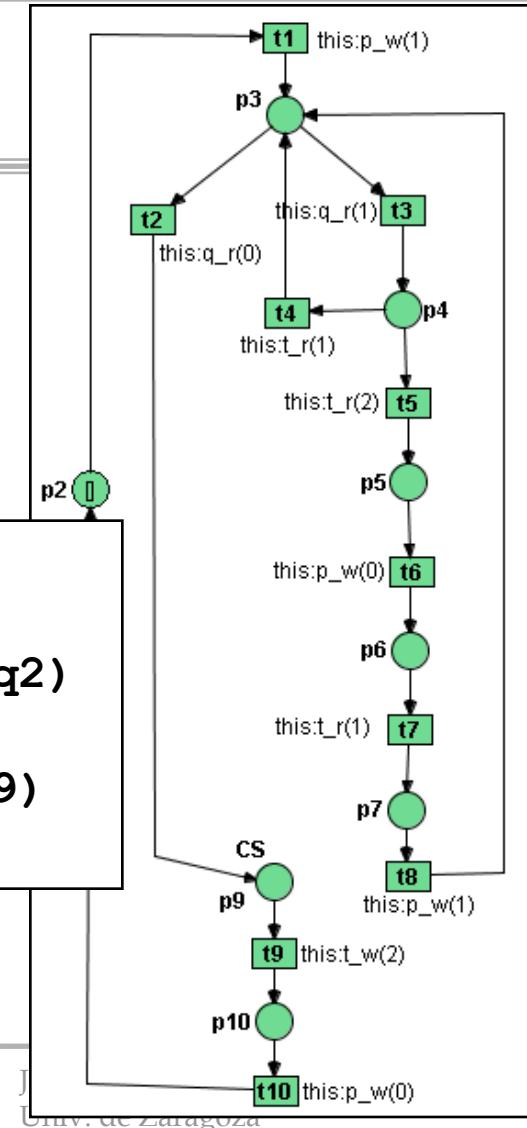


¿Qué pasa con el algoritmo de Dekker?

- Propiedades: no hay esperas innecesarias

```
[] ((p2 AND []) q2) => () () p9)
```

```
[]
  (is black_token {} subset place p2)
  && [] (is black_token {} subset place q2)
=>
  () () (is black_token {} subset place p9)
)
```



Lección 5: Diseño de programas concurrentes

- Introducción
- Pautas para el diseño de programas concurrentes
- La instrucción **await**
- Ejemplos de diseño de programas con la instrucción **await**

Introducción

- El diseño de programas concurrentes (correctos) es una tarea compleja
 - A la dificultad del diseño de programas secuenciales hay que añadir las posibles interferencias entre los procesos
 - Los posibles entrelazados pueden dar resultados distintos en distintas ejecuciones
 - Lo que hace que la depuración sea complicada
- Es indispensable poder "asegurar" la corrección por el diseño
- Podemos plantear una solución de grano grueso y refinamientos posteriores
 - Hasta llegar a una solución fácilmente implementable

Pautas para el diseño de programas concurrentes

- Algunas pautas
 - Identificar muy claramente los procesos y los datos compartidos
 - Diseñar los procesos, determinando claramente la separación entre los datos compartidos y los que vamos a añadir para gestionar los aspectos de sincronización
 - Plantear un diseño en base a instrucciones atómicas de tipo **await**
 - Llevar a cabo la traducción del diseño en base a las instrucciones permitidas en nuestro lenguaje (semáforos, monitores, etc.)

La instrucción `await`

- Se trata de una instrucción de alto nivel ("grano grueso")
 - Potente, pero costosa de implementar
- Sintaxis
 - `< await B`
 - `S`
 - `>`
- - **B**: es una expresión booleana
 - **S**: es un conjunto de instrucciones
- Nos evitará el uso de esperas activas

< await B
S

>

La instrucción **await**

- Semántica: cuando se pasa el control a un proceso para ejecutar una instrucción **await**
 - Se evalúa la guarda
 - Si es cierta, se ejecuta S, y se pasa a la instrucción que sigue al **await**
 - Si es falsa, no se hace nada, y no se pasa a la siguiente instrucción
 - Por lo que el **await** seguirá siendo la instrucción elegible para este proceso
 - Y todo esto es **atómico**

< await B
S

>

La instrucción **await**

- Las formas habituales de sincronización son casos particulares de la instrucción:

- Exclusión mutua

< await true
S
>

< S >

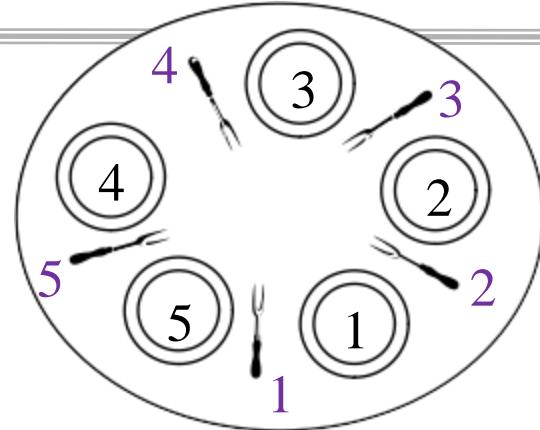
- Sincronización por condición

< await B
// nada
>

< await B >

Ejemplos de diseño: el problema de los filósofos

- Dijkstra 65, Hoare 85
 - Prototipo de sistema en que los procesos comparten recursos conservativos
- Hay 5 filósofos sentados alrededor de una mesa para comer espaguetis
- Hay 5 tenedores, e infinita pasta
- Hacen falta dos tenedores para comer
 - Cada filósofo puede usar los que tiene más cerca
 - Cada filósofo coge primero el de su izda., y luego el de su dcha.
- Se pide un programa que simule el comportamiento del sistema



```
Process filosofo(i:1..5):
    while true
        //piensa
        //coge tenedores
        //come
        //deja tenedores
    end
end
```

Ejemplos de diseño: el problema de los filósofos

- ¿Cómo modelar el estado de los tenedores?

```
constant integer N := 5
boolean array[1..N] libre := (1..N,true)

Process filosofo(i:1..N):
    while true
        //piensa
        //espera a que tenedor izda. libre y lo coge
        //espera a que tenedor dcha. libre y lo coge
        //come
        //deja tenedor a la izda.
        //deja tenedor a la dcha.
    end
end
```

Ejemplos de diseño: el problema de los filósofos

- Versión 1: primero un tenedor, y luego el otro

```
const integer N := 5
boolean array[1..N] libre := (1..N,true)

Process filosofo(i:1..N):
    while true
        //piensa
        //espera a que tenedor izda. libre y lo coge
        //espera a que tenedor dcha. Libre y lo coge
        //come
        //deja tenedor a la izda.
        //deja tenedor a la dcha.
    end
end
```

< await libre[i]
libre[i] := false
>

< await libre[1 + i mod N]
libre[1 + i mod N] := false
>

Ejemplos de diseño: el problema de los filósofos

- Si estudiamos el comportamiento de la solución, vemos que se puede alcanzar un bloqueo total, en el que todos los filósofos mueren de hambre
 - Encontrar la situación de bloqueo
- Plantear una solución alternativa, **V2**, en que los tenedores se cojan siempre por orden.
 - No se bloqueará porque no se puede cerrar ciclo de espera por recursos
- Plantear tercera solución alternativa, **V3**, en que cada filósofo coge ambos tenedores a la vez (o no los coge)
 - No se bloqueará porque los estados no son *hold-and-wait*

Ejemplos de diseño: máximo de un vector

- Considérese el código que se muestra a continuación, en el que se lanzan M procesos concurrentes con el objetivo de que entre todos encuentren el valor máximo de un vector
 - cada proceso se encarga de un trozo del vector.
- Una vez todos han acabado, el proceso informador muestra el valor máximo del vector
- Se pide completar el diseño de acuerdo a la especificación.

```
constant integer N := ... //lo que sea, >= 1
                      M := ... //lo que sea, >= 1
                      NM := N*M
integer array[1..NM] val := ... //lo que sea, ya inicializado
integer max
// código a completar
```

```
Process P(i: 1..M):: 
    // código a completar
    // al terminar todos los procesos, "max" contiene el
    // valor máximo del vector "val"
end
```

```
Process informador:: 
    // código a completar
    write('Max= ', max)
end
```

```
//Pre: 1 <= i1 <= i2 <= NM
//Post: maxTrozo() = el valor máximo de v[i1],...,v[i2]
operation maxTrozo(integer array[1..NM] v, integer i1,i2): integer
```

Ejemplos de diseño: máximo de un vector

Considérese un programa concurrente compuesto por 10 procesos: 5 procesos **lectores** y 5 procesos **calculadores**, que comparten una matriz 5x50 de números enteros. El ejercicio pide completar el esquema de programa que se muestra a continuación, de manera que:

- Cada proceso lector lee de la entrada estándar una secuencia de enteros compuesta por 50 números y rellena una fila completa de la matriz
- Cada proceso calculador busca el máximo parcial de una fila de la matriz
- Una vez que todos los procesos calculadores han hecho su trabajo, el proceso con identificador 1 muestra por la salida estándar el máximo global de la matriz

Ejemplos de diseño: máximo de un vector

Considérese un programa concurrente compuesto por 10 procesos: 5 procesos **lectores** y 5 procesos **calculadores**, que comparten una matriz 5x50 de números enteros. El ejercicio pide completar el esquema de programa que se muestra a continuación, de manera que:

- Cada proceso lector lee de la entrada estándar una secuencia de enteros compuesta por 50 números y rellena una fila completa de la matriz
- Cada proceso calculador busca el máximo parcial de una fila de la matriz
- Una vez que todos los procesos calculadores han hecho su trabajo, el proceso con identificador 1 muestra por la salida estándar el máximo global de la matriz

```
integer array[1..5,1..50] D
```

```
...
```

```
Process Lector(i:1..5)::
```

```
...
```

```
end
```

```
Process Calculador(i:1..5)::
```

```
...
```

```
end
```

Lección 6: Sincronización de procesos mediante semáforos

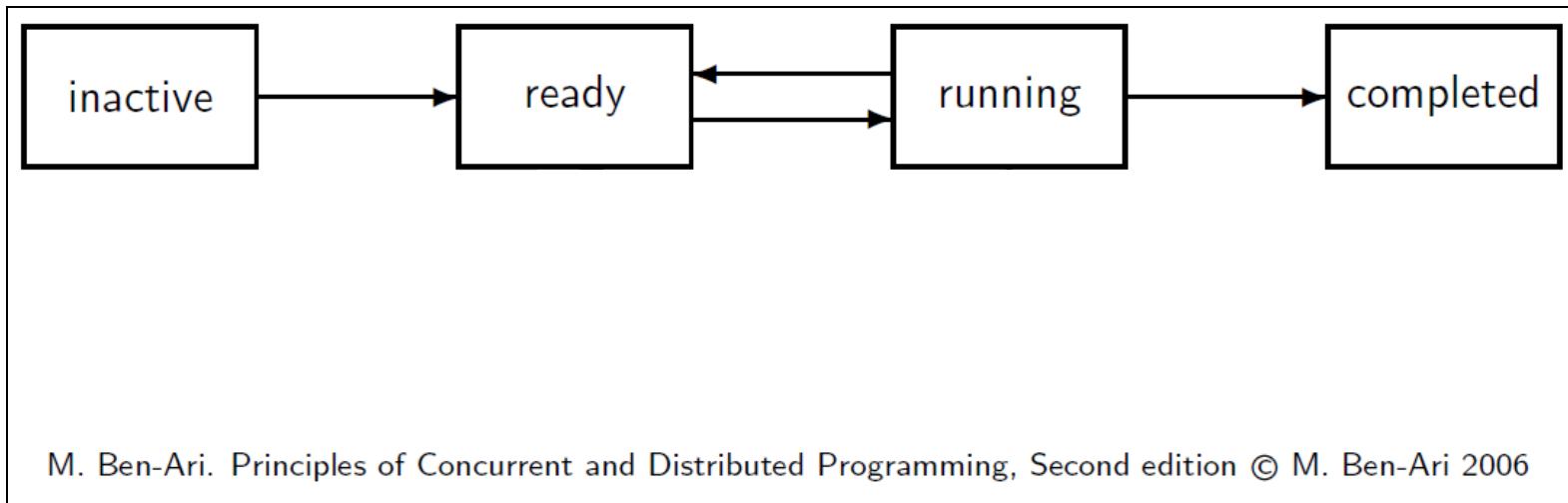
- Introducción
- Estados de un proceso
- Semáforos: sintaxis y semántica
- Técnicas de programación con semáforos:
 - el problema de la sección crítica
 - el problema de la cena de los filósofos
- La técnica del paso del testigo
 - El problema de la cena de los filósofos

Introducción

- La sincronización por espera activa tiene inconvenientes:
 - los algoritmos son difíciles de diseñar y verificar
 - se mezclan variables del problema a resolver y del método de resolución
 - en los casos de multi-programación, se ocupa innecesariamente el procesador “haciendo nada”
- Los semáforos son una de las primeras soluciones
 - son una solución conceptual, que puede implementarse de diferentes formas
 - propuestos por Dijkstra en 1968

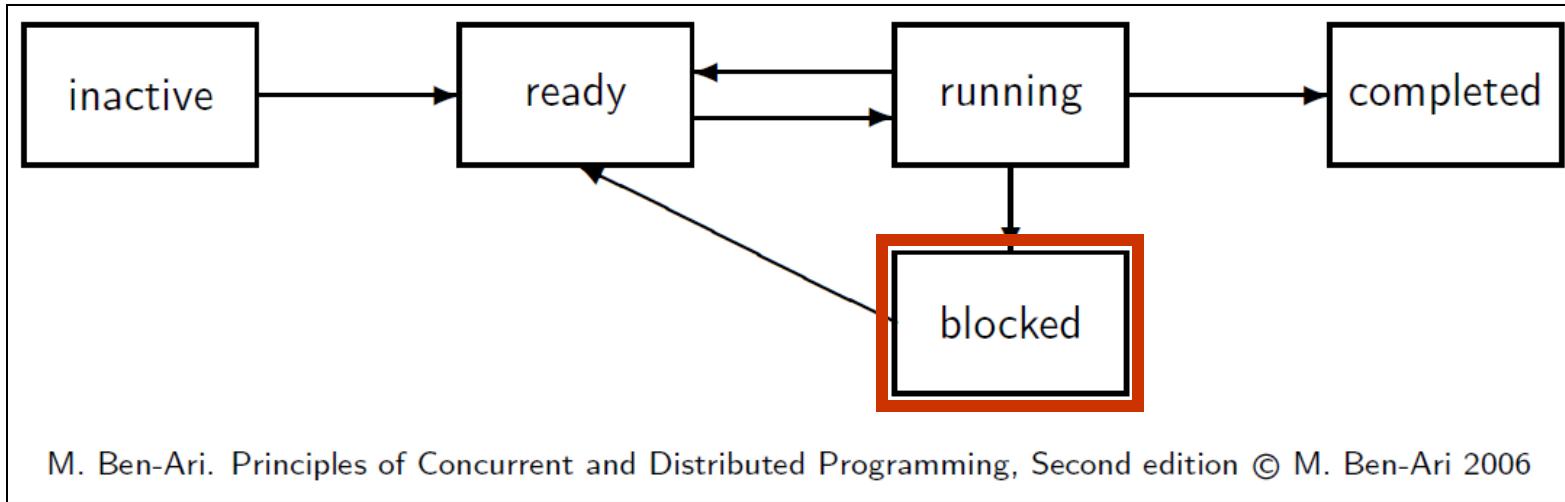
Estados de un proceso

- Para multitarea, y un proceso P , $P.state$ es su estado



Estados de un proceso

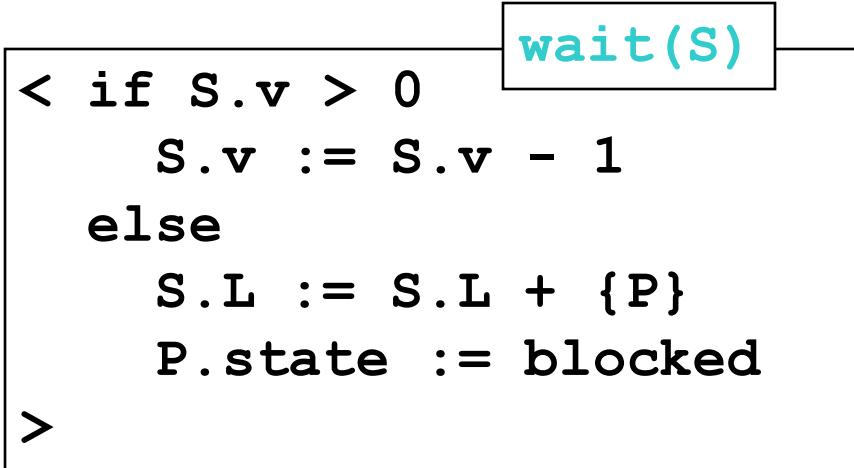
- Para multitarea, y un proceso P, **P.state** es su estado



Semáforos: sintaxis y semántica

- **Semáforo:** instancia de un tipo abstracto de dato
 - $S = (v, L)$
- tal que ...
 - v es un natural
 - L es un conjunto de procesos
- con dos instrucciones atómicas...
 - **wait**
 - **signal**

Semáforos: sintaxis y semántica

- Declaración **semaphore S := (S₀, {})**
- Semántica de las operaciones (atómicas) cuando la invoca P:


```
< if S.v > 0
    S.v := S.v - 1
else
    S.L := S.L + {P}
    P.state := blocked
>
```

Semáforos: sintaxis y semántica

- Semántica de las operaciones (atómicas) cuando la invoca P:

signal (S)

```
< if S.L = {}  
    S.v := S.v + 1  
else  
    S.L := S.L - {Q}  
    Q.state := ready  
>
```

- Propiedad: un semáforo S cumple el siguiente invariante

$S.v = S_0 + \#signal(S) - \#wait(S)$

- Existen también los "strong semaphores": "L" es una cola

El problema de la sección crítica

- Ejemplo: la sección crítica

semaphore mutex := (1, {})	
<i>Process P</i>	<i>Process Q</i>
while true	while true
SNC	SNC
p1 wait(mutex)	q1 wait(mutex)
SC	SC
p2 signal(mutex)	q2 signal(mutex)

Semáforos: sintaxis y semántica

- Semántica alternativa para semáforos: versión “*busy-wait*”

```
semaphore S := (S0, {})
```

```
< await S.v >    wait(S)
                    S.v := S.v - 1
>
```

```
                                signal(S)
< S.v := S.v + 1 >
```

Semáforos: sintaxis y semántica

- En realidad, S.L no hace falta, y podemos identificar S con S.v

```
semaphore S := S0
```

```
< await S > 0  
    S := S - 1  
>
```

wait(S)

```
< S := S + 1 >
```

signal (S)

Semáforos: sintaxis y semántica

- Existe un caso especial: los semáforos binarios
- Declaración
 - con s_0 : 0 .. 1
- Semántica de las operaciones (atómicas) cuando la invoca P:

```
binary semaphore S := S0
```

- con s_0 : 0 .. 1

- Semántica de las operaciones (atómicas) cuando la invoca P:

```
< await S = 1  
    S := 0 >
```

wait(S)

```
< await S = 0  
    S := 1 >
```

signal(S)

Semáforos: sintaxis y semántica

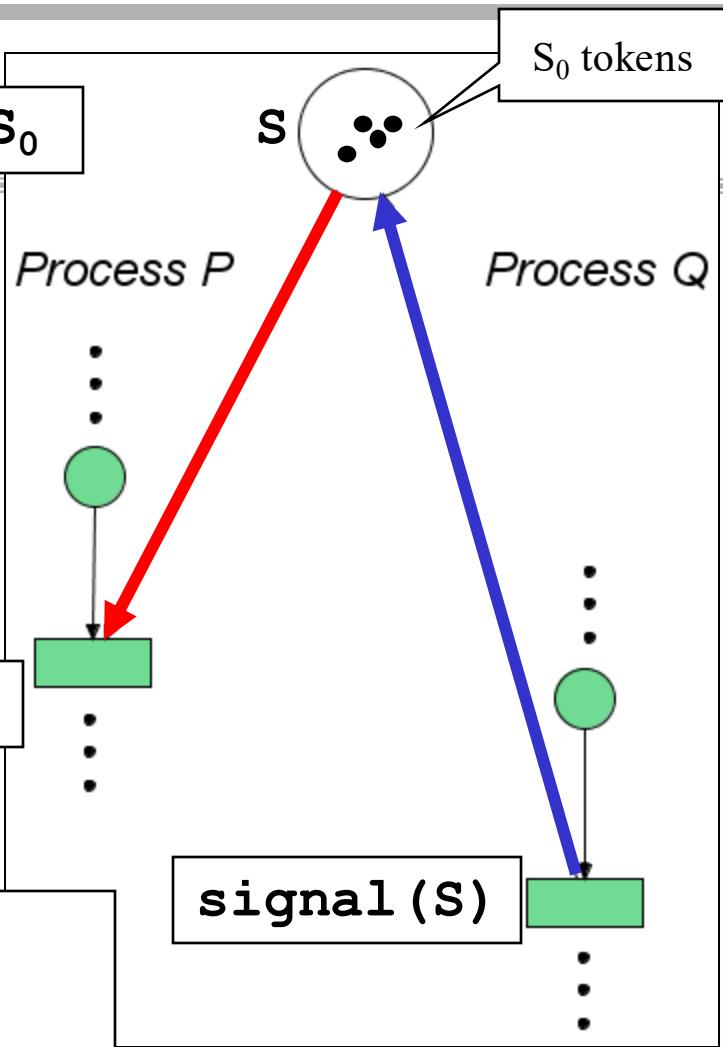
- Para binarios, existe una versión con una semántica alternativa (mutex):

```
< if S.v = 1  
    ??????????  
else if S.L = {}  
    S.v := 1  
else  
    S.L := S.L - {Ω}  
    Q.state := ready  
>
```

signal (S)

El problema

```
semaphore S := S0
```

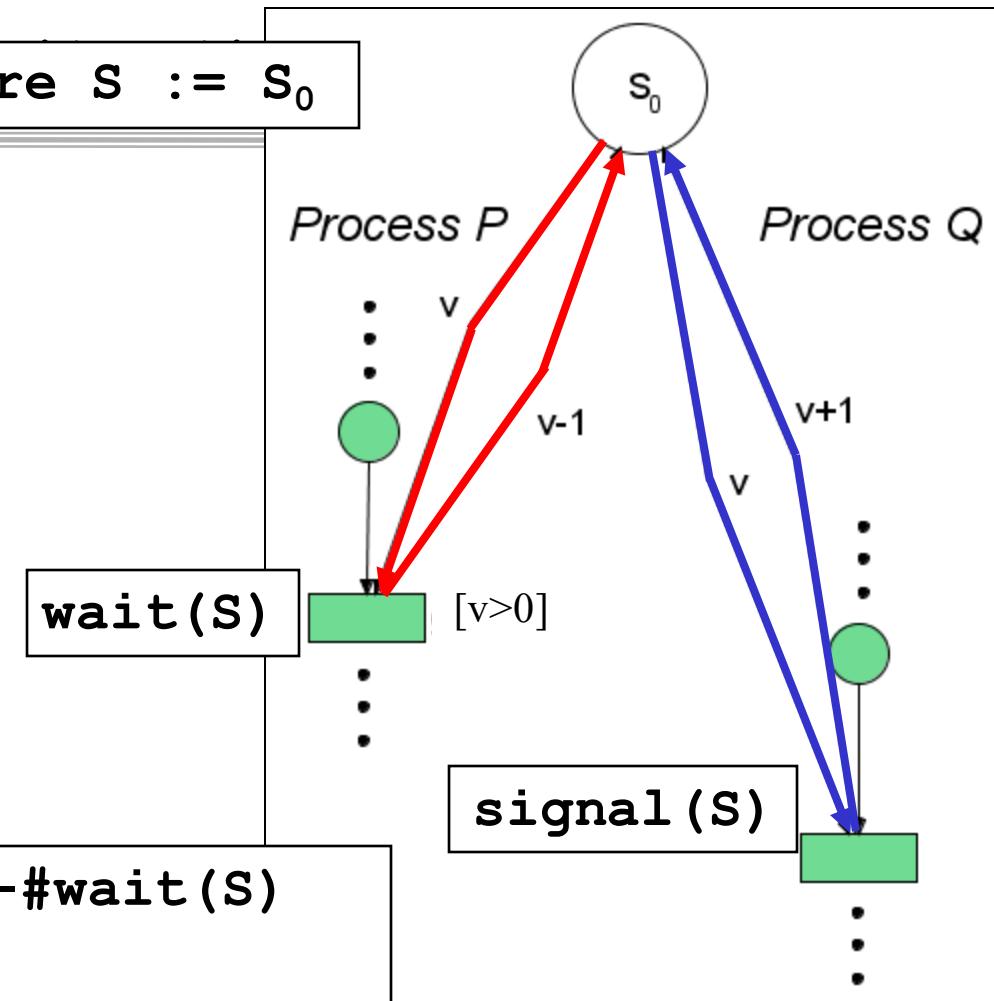


$$M(S) = S_0 + \#signal(S) - \#wait(S)$$

$$M(S) \geq 0$$

El problema

```
semaphore S := S0
```



$$M(S) = S_0 + \#signal(S) - \#wait(S)$$

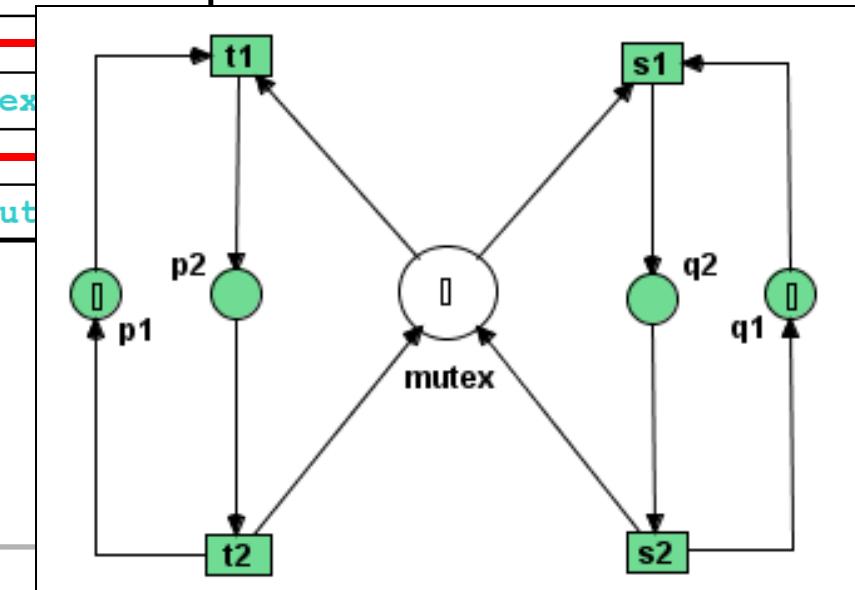
$$M(S) \geq 0$$

El problema de la sección crítica

- Ejemplo: la sección crítica para dos procesos

semaphore mutex := (1, {})	
Process P	Process Q
<u>while true</u>	<u>while true</u>
-SNC	-SNC
wait(mutex)	wait(mutex)
-SC	-SC
signal(mutex)	signal(mutex)

- ¿Y para n procesos?



Ejemplos de diseño: máximo de un vector

- Considérese el código que se muestra a continuación, en el que se lanzan M procesos concurrentes con el objetivo de que entre todos encuentren el valor máximo de un vector
 - cada proceso se encarga de un trozo del vector.
- Una vez todos han acabado, el proceso informador muestra el valor máximo del vector
- Se pide completar el diseño de acuerdo a la especificación.

```
constant integer N := ... //lo que sea, >= 1
                      M := ... //lo que sea, >= 1
                      NM := N*M
integer array[1..NM] val := ... //lo que sea, ya inicializado
integer max
// código a completar
```

```
Process P(i: 1..M):::
    // código a completar
    // al terminar todos los procesos, "max" contiene el
    // valor máximo del vector "val"
end
```

```
Process informador:::
    // código a completar
    write('Max= ', max)
end
```

```
//Pre: 1 <= i1 <= i2 <= NM
//Post: maxTrozo() = el valor máximo de v[i1],...,v[i2]
operation maxTrozo(integer array[1..NM] v, integer i1,i2): integer
```

```

constant integer N := ... //lo que sea, >= 1
    M := ... //lo que sea, >= 1
    NM := N*M

integer array[1..NM] val := ... //lo que sea, ya inicializado
integer max
integer numHT := 0
Semaphore mutHT := 1
Semaphore todosHT := 0

Process P(i: 1..M):::
    integer maxLocal := maxTrozo(val,(i))
    wait(mutHT)
    signal(muHT)
end

```



```

numHT++
if numHT=1
    max := maxLocal
else
    if maxLocal > max
        max := maxLocal
    end
end
if numHT=M
    signal(todosHT)
end

```

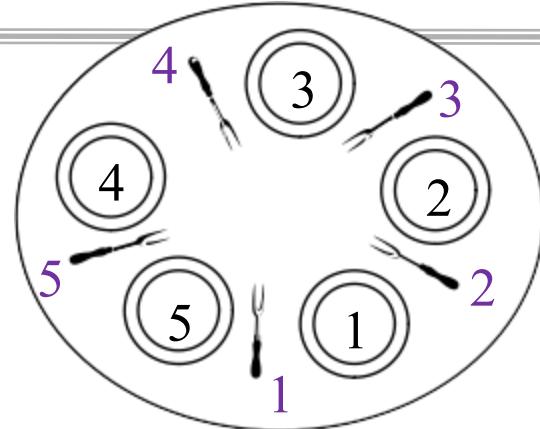
```

Process informador():::
    wait(todosHT)
    write("Max=",max)
end

```

Ejemplos de diseño: el problema de los filósofos

- Dijkstra 65, Hoare 85
 - Prototipo de sistema en que los procesos comparten recursos conservativos
- Hay 5 filósofos sentados alrededor de una mesa para comer espaguetis
- Hay 5 tenedores, e infinita pasta
- Hacen falta dos tenedores para comer
 - Cada filósofo puede usar los que tiene más cerca
 - Cada filósofo coge primero el de su izda., y luego el de su dcha.
- Se pide un programa que simule el comportamiento del sistema



```
Process filosofo(i:1..5):
  while true
    //piensa
    //coge tenedores
    //come
    //deja tenedores
  end
end
```

El problema de la cena de los filósofos

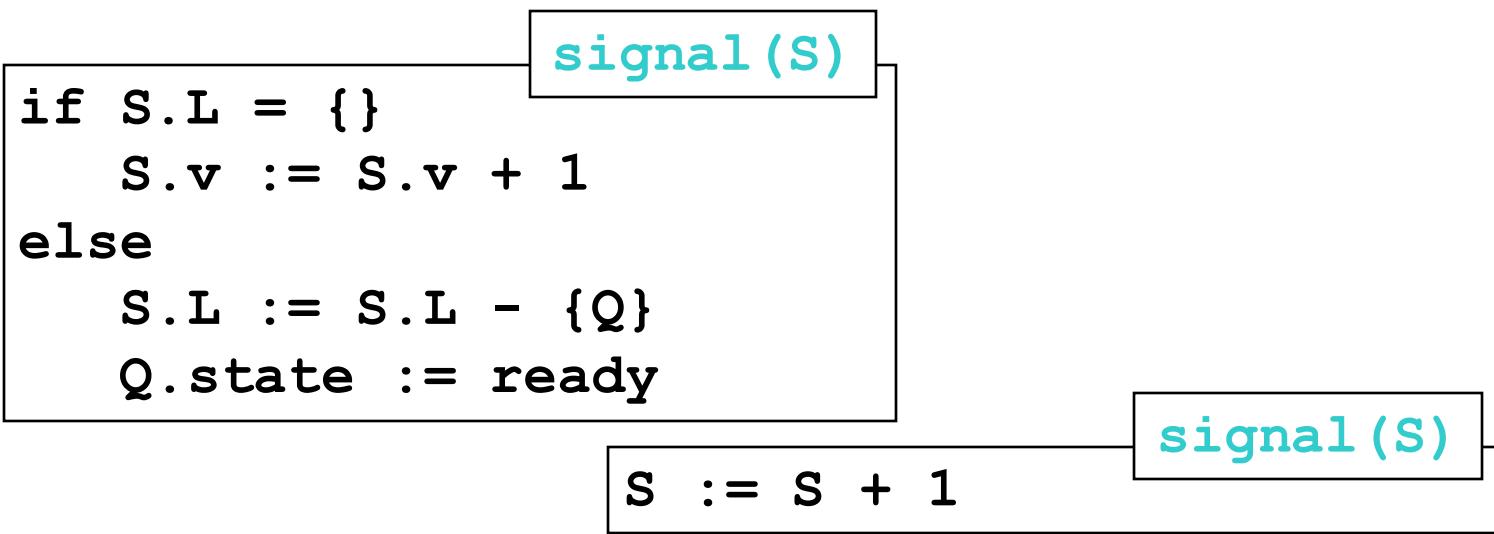
- Solución con semáforos:
 - Los recursos compartidos (tenedores) los representamos por medio de semáforos
 - El uso que hacen los procesos de esos recursos (coger/dejar) se “mapea” a las operaciones ofrecidas por los semáforos correspondientes (wait/signal)

```
Semaphore array[1..numFil] tDisponible := (1..numFil, 1)

Process filosofo(i:1..numFil)
    while true
        // Pensando
        wait(tDisponible[i])
        wait(tDisponible[1 + i mod numFil])
        // Comiendo
        signal(tDisponible[1 + i mod numFil])
        signal(tDisponible[i])
    end
end loop
```

Semáforos: sintaxis y semántica

- ¿Cuál es el matiz entre las distintas versiones de semáforos?
 - "normal", "strong", "busy-wait"
 - Pista: estudiar el comportamiento desde el punto de vista de la equidad en el problema de la sección crítica



Un ejercicio sencillo

- Consideremos un esquema productor/consumidor en el que:
 - el productor produce un mensaje, que debe depositar en un buffer
 - el consumidor lee el mensaje del buffer
- Requisitos:
 - no se sobrescribe un mensaje
 - no se lee dos veces el mismo mensaje
 - el buffer tiene capacidad para un mensaje

```
        string buffer

Process Productor::           Process Consumidor:: 
    string paraEnviar           string mensaje
    while true                 while true
        paraEnviar := produceMensaje()   mensaje := buffer
        buffer := paraEnviar            write(mensaje)
    end                         end
end                           end
```

El paso del testigo

- Resolver problemas de sincronización complejos con semáforos es una tarea complicada
 - difícil obtener una solución
 - difícil asegurar la corrección
- La técnica del “paso del testigo” permite desarrollar, a partir de un diseño mediante instrucciones “**await**”, una solución mediante semáforos
 - Diseñar con “**await**” (mucho más fácil y seguro)
 - Implementar con semáforos

El paso del testigo

- Veamos una forma sistemática de implementar exclusiones mutuas y sincronización por condición con semáforos (binarios)
- Necesitamos gestionar un conjunto de instrucciones que deban sincronizar, del siguiente tipo:

$I_i : <S_i>$

$I_j : <\text{await } B_j \quad S_j>$

- Para ello:
 - Un semáforo (binario) para asegurar la ejecución en exclusión mutua de las instrucciones:

Semaphore testigo := 1

- Para cada : $\langle\text{await } B_j \quad S_j\rangle$

Semaphore b_j := 0

integer d_j := 0

El paso del testigo

- Cambiar $\langle S_i \rangle$ por

```
Ii: wait(testigo)  
      Si  
      pasarTestigo(...)
```

- Cambiar $\langle \text{await } B_j \quad S_j \rangle$ por

```
Ij: wait(testigo)  
      if no Bj  
          dj := dj+1  
          signal(testigo)  
          wait(bj)  
      end  
      Sj  
      pasarTestigo(...)
```

El paso del testigo

- Donde

```
Operation pasarTestigo( . . . )
  switch
    B1 AND d1>0 : d1:=d1-1; signal (b1)
    .
    .
    Bn AND dn>0 : dn:=dn-1; signal (bn)
    otherwise: signal (testigo)
  end
end
```

El paso del testigo

- Ejercicio:
implementar
usando
semáforos

```
constant integer N := ... //lo que sea
                      M := ... //lo que sea
                      NM := N*M
integer array[1..NM] val := ... //lo que sea
integer max

integer numHanAcabado := 0 //# procs han acabado

Process P(i: 1..M):::
    integer maxLocal

    maxLocal := maxTrozo(val,(i-1)*N+1,i*N)
    //proteger numHanAcabado y max
    (
        numHanAcabado := numHanAcabado+1
        if numHanAcabado=1
            max := maxLocal
        else
            if maxLocal > max
                max := maxLocal
            end
        end
    )
end

Process informador:::
    await numHanAcabado = M
        write('Max= ', max)
    }
end
```

El paso del testigo

- Ejercicio:
implementar
usando
semáforos

```
integer s := 2

Process P1
    while true
        <await s >= 1
            s := s-1
        >
        //en la zona critica
        < s := s+1 >
    end
end

Process P2
    while true
        <await s >= 1
            s := s-1
        >
        //en la zona critica
        < s := s+1 >
    end
end
```

```
Process P3
    while true
        <await s >= 2
            s := s-2
        >
        //en la zona critica
        < s := s+2 >
    end
end

Process P4
    while true
        <await s >= 2
            s := s-2
        >
    end
end
```

```
bool array[1..n] libre := (1..n,TRUE)
```

```
Process filosofo(i:1..n)::
```

```
  while true
```

```
    //pensando
```

```
    <await libre[i] AND  
      libre[1+i mod n]  
      libre[i] := FALSE  
      libre[1+i mod n] := FALSE  
>
```

```
    //comiendo
```

```
    < libre[i] := TRUE  
      libre[1+i mod n] := TRUE  
>
```

```
  end
```

```
end
```

```
bool array[1..n] libre := (1..n,TRUE)  
int array[1..n] d := (1..n,0)  
Semaphore array[1..n] b := (1..n,0)  
Semaphore testigo := 1  
bool array[1..n] libre := (1..n,TRUE)
```

```
wait(testigo)
```

```
if NOT (libre[i] AND libre[1+i mod n])  
  d[i]++
```

```
  signal(testigo)  
  wait(b[i])
```

```
end
```

```
libre[i] := FALSE
```

```
libre[1+i mod n] := FALSE  
pasarTestigo(i)
```

```
wait(testigo)
```

```
libre[i] := TRUE
```

```
libre[1+i mod n] := TRUE
```

```
pasarTestigo(i)
```

```

bool array[1..n] libre := (1..n,TRUE)
int array[1..n] d := (1..n,0)
Semaphore array[1..n] b := (1..n,0)
Semaphore testigo := 1
bool array[1..n] libre := (1..n,TRUE)

```

```

wait(testigo)
if NOT (libre[i] AND libre[1+i mod n])
    d[i]++
    signal(testigo)
    wait(b[i])
end
libre[i] := FALSE
libre[1+i mod n] := FALSE
pasarTestigo(i)

```

```

operation pasarTestigo(int i)
    bool testigoPasado = false
    int j := i+1
    while j <> i AND NOT testigoPasado
        if libre[j]
            AND libre[1+j mod n]
            AND d[j]>0
                d[j]--
                testigoPasado := TRUE
                signal(b[j])
        else
            j++
        end
    end
    if NOT testigoPasado
        signal(testigo)
    end
end

```

```

wait(testigo)
libre[i] := TRUE
libre[1+i mod n] := TRUE
pasarTestigo(i)

```

Lección 7: Sincronización de procesos mediante monitores

- Introducción
- ¿Qué es un monitor?
- Características y funcionamiento de un monitor
- Implementación de un monitor en C++
- Algunos ejemplos de aplicación:
 - El caso de los productores/consumidores
 - El problema de la cena de los filósofos
 - El caso de los lectores/escritores

Introducción

- Los semáforos tienen algunas características que pueden generar inconvenientes:
 - las variables compartidas son globales a todos los procesos
 - las acciones que acceden y modifican dichas variables están diseminadas por los procesos
 - para poder decir algo del estado de las variables compartidas, es necesario mirar todo el código
 - la adición de un nuevo proceso puede requerir verificar que el uso de las variables compartidas es el adecuado

se necesita encapsulación

¿Qué es un monitor?

- **E. Dijkstra** [1972]: propuesta de una unidad de programación denominada *secretary* para encapsular datos compartidos, junto con los procedimientos para acceder a ellos.
- **Brinch Hansen** [1973]: propuesta de las *clases compartidas* ("shared class"), una construcción análoga a la anterior.
- El nombre de *monitor* fue acuñado por **C.A.R. Hoare** [1973].
- Posteriormente, **Brinch Hansen** incorpora los monitores al lenguaje Pascal Concurrente [1975]

¿Qué es un monitor?

- componente *pasivo*
 - frente a un proceso, que es activo
- constituye un *módulo* de un programa concurrente
 - proporcionan un **mecanismo de abstracción**
 - encapsulan la representación de recursos abstractos junto a sus operaciones
 - con las ventajas inherentes a la encapsulación
 - las operaciones de un monitor se ejecutan, *por definición*, en **exclusión mutua**
 - dispone de mecanismos específicos para la sincronización: *variables “condición”*

Un sencillo ejemplo

```
monitor CS
```

```
    integer x := 0
```

```
    operation increment()
```

```
        x := x + 1
```

```
    end
```

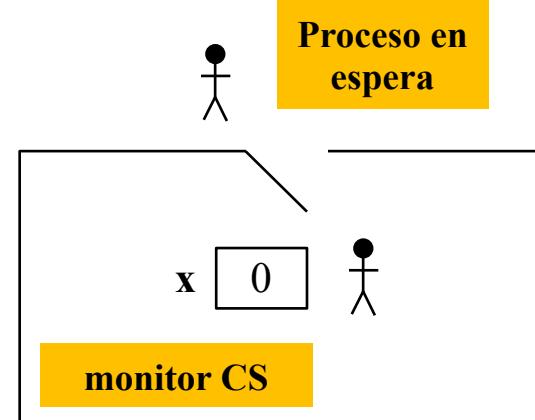
```
end
```

```
Process P
```

```
Process Q
```

```
CS.increment()
```

```
CS.increment()
```



Características de un monitor

- Variables permanentes
 - “permanentes” porque existen y mantienen su valor mientras existe el monitor
 - describen el estado del monitor
 - han de ser inicializadas antes de usarse
- Las acciones:
 - son parte de la interfaz, por lo que pueden ser usadas por los procesos para cambiar su estado
 - sólo pueden acceder a las variables permanentes y sus parámetros y variables locales
 - son la única manera posible de cambiar el estado del monitor
- Invocación por un proceso: **nombreMonitor.operación(listaParámetros)**

```
monitor CS
```

```
integer x := 0
```

```
operation increment()
```

```
    x := x + 1
```

```
end
```

```
end
```

Process P	Process Q
CS.increment()	CS.increment()

Funcionamiento de un monitor

- Respecto a la sincronización:
 - la **exclusión mutua** se asegura por definición
 - por lo tanto, sólo un proceso puede estar ejecutando acciones de un monitor en un momento dado
 - aunque varios procesos pueden en ese momento ejecutar acciones que nada tengan que ver con el monitor
 - la **sincronización condicionada**
 - con frecuencia es necesaria una sincronización explícita entre procesos
 - para ello, se usarán las variables “condición”
 - se usan para hacer esperar a un proceso hasta que determinada condición sobre el estado del monitor se “anuncie”
 - también para despertar a un proceso que estaba esperando por su causa

Sobre las variables “condición”

- Representa una *condición* de interés para los procesos que se sincronizan por medio del monitor
 - cada variable tiene asociada una *cola FIFO para los procesos que están bloqueados*
- Ofrece dos operaciones atómicas básicas:
 - *waitC(variable_condicion)*
 - el proceso es bloqueado en la cola de la variable condición”
 - *signalC(variable_condicion)*
 - “el primer proceso de la cola es desbloqueado”



**No confundirlas con las
operaciones de semáforos!**

Sobre las variables “condición”

- instrucción *waitC(c)*:
 - el proceso invocador queda “bloqueado” y pasa a la cola FIFO asociada a la variable *c*, en espera de ser despertado
 - el cerrojo que garantiza la exclusión mutua del monitor queda libre
- instrucción *signalC(c)*:
 - si la cola de la señal está vacía: no pasa nada y la operación sigue con su ejecución
 - al terminar, el monitor está disponible para otro proceso
 - si la cola no está vacía:
 - se saca el primer proceso de la cola y se “desbloquea”
 - políticas de reanudación determinan qué proceso continúa su ejecución
- instrucción *signalC_all(c)*
- instrucción *emptyC(c)*

Sobre las variables “condición”

- **Ejemplo:** diséñese un programa concurrente en el que
 - 10 procesos incrementan 1000 veces cada uno una variable **x** que inicialmente vale 0
 - Un proceso espera a que la variable llegue al valor 3000, e informa por la salida estándar del hecho

```
Process P(i:1..10):
```

```
    //incrementar la variable
```

```
end
```

```
Process info:
```

```
    //informar cuando llegue a 3000
```

```
end
```

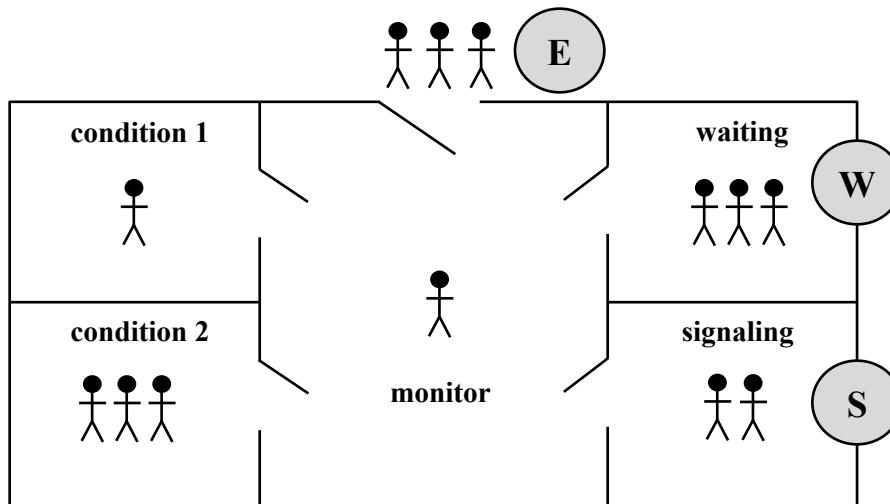
Sobre las variables “condición”

- Diferencias entre las instrucciones de un monitor y las de un semáforo con nombre similar:

Semáforos	Monitores
<i>wait</i> puede bloquearse, o no	<i>waitC</i> siempre se bloquea
<i>signal</i> siempre tiene un efecto	<i>signalC</i> no tiene efecto si la cola está vacía
<i>signal</i> puede desbloquear un proceso cualquiera de la cola (depende del tipo de semáforo)	<i>signalC</i> siempre desbloquea al primer proceso en la cola
<i>wait/signal</i> en cualquier parte del programa	<i>waitC/signalC</i> solo dentro de monitores

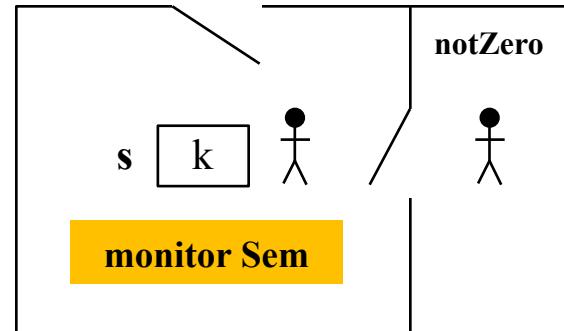
Sobre las variables “condición”

- *Políticas de reanudación:*
 - versión clásica de un monitor: $E < S < W$
 - “Immediate Resumption Requirement” (IRR)
 - versión implementada en Java: $E = W < S$



Implementando un semáforo con un monitor

```
monitor Sem
    integer s := 1
    condition notZero
    operation wait()
        if s = 0
            waitC(notZero)
        end
        s := s - 1
    end
    operation signal()
        s := s + 1
        signalC(notZero)
    end
end
```



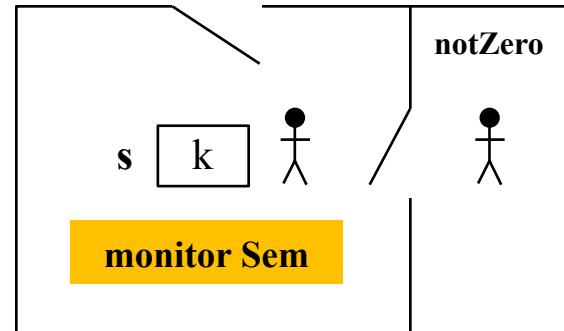
```
Process P(i:1..N) ::  
while true  
    SNC  
    Sem.wait()  
    SC  
    Sem.signal()  
end
```

Implementando un semáforo con un monitor

```
monitor Sem
    integer s := 1
    condition notZero
    operation wait()
        while s = 0
            waitC(notZero)
        end
        s := s - 1
    end
    operation signal()
        s := s + 1
        signalC(notZero)
    end
end
```

E<S<W

E=W<S



Process *P(i:1..N)* ::

```
while true
    SNC
    Sem.wait()
    SC
    Sem.signal()
end
```

El problema de los productores/consumidores

- **Ejemplo:**
 - tenemos un sistema con un proceso productor y un consumidor
 - Caso 1: buffer intermedio de **capacidad infinita**
 - Caso 2: buffer intermedio de **capacidad finita**

el_tipo queue buffer := [] ...	
<i>Process productor</i>	<i>Process consumidor</i>
el_tipo d	el_tipo d
while true	while true
produce(d)	d:=consume(buffer)
append(d,buffer)	usa(d)

El problema de los productores/consumidores

```
monitor almacen_limitado
  ...
  operation append(el_tipo d)
    ...
  operation consume() return el_tipo
  ...
```

```
process productor
  el_tipo d
  while true
    preparar d
    almacen_limitado.append(d)
  end
```

```
process consumidor
  el_tipo d
  while true
    d:= almacen_limitado.consume()
    usa(d)
  end
```

```

monitor almacen_limitado
    integer n := ... --capacidad, >=1
    ...
    condition no_lleno,no_vacio

operation append(el_tipo d)
    ...
    while "esta lleno"
        waitC(no_lleno)
    end
    ...

operation consume() return el_tipo
    el_tipo d
    ...
    while "esta vacío"
        waitC(no_vacio)
    end
    ...
    return d

```

```
monitor almacen_limitado
    integer n := ... --capacidad, >=1
    ...
    condition no_lleno,no_vacio

operation append(el_tipo d)
    ...
    while "esta lleno"
        waitC(no_lleno)
    end
    ...
    signalC(no_vacio)

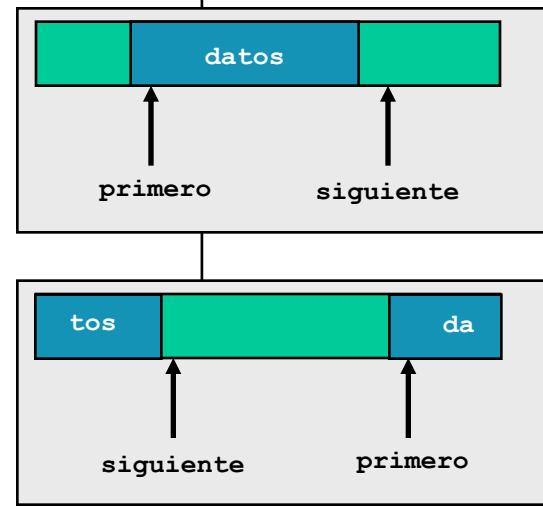
operation consume() return el_tipo
    el_tipo d
    ...
    while "esta vacío"
        waitC(no_vacio)
    end
    ...
    signalC(no_lleno)
    return d
```

El problema de los productores/consumidores

```
monitor almacen_limitado
    integer n := .... //n>=1
    integer primero := 1
        siguiente := 1
    el_tipo array[1..n] almacen
    natural n_datos := 0

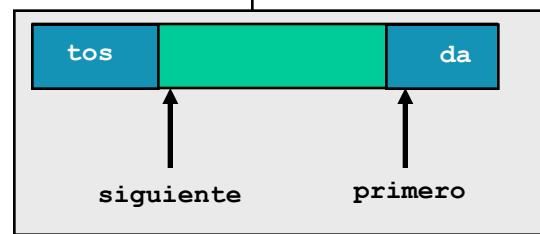
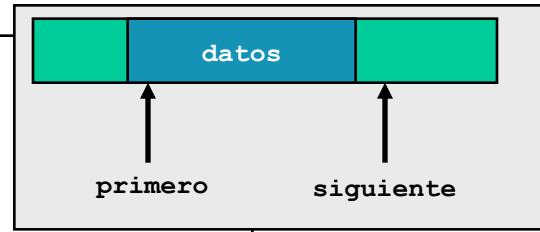
    condition no_lleno,no_vacio

    operation append(el_tipo d)
        while n_datos=n
            waitC(no_lleno)
        end
        almacen[siguiente] := d
        siguiente := (siguiente mod n) + 1
        n_datos := n_datos+1
        signalC(no_vacio)
```



El problema de los productores/consumidores

```
monitor almacen_limitado
    ...
    operation consume() return el_tipo
        el_tipo d
        while n_datos=0
            waitC(no_vacio)
        end
        d := almacen[primero]
        primero := (primero mod n) + 1
        n_datos := n_datos-1
        signalC(no_lleno)
    return d
```



Implementación de un monitor en C++

- Un monitor como instancia de una clase que implemente el comportamiento deseado
- Usando variables condición y mutex
- El acceso en exclusión mutua lo gestionaremos explícitamente
 - declarar un mutex dentro del objeto
 - bloquearlo al iniciar cada función
- Declarar todas las variables del monitor como atributos privados

```
#include <mutex>
#include <condition_variable>
```

Ejemplo

```
Process P(i: 1..10)::  
    for j:=1..1000  
        var_x.inc()  
    end  
end  
  
Process informador::  
    var_x.informa()  
end
```

Diseño

```
Monitor var_x  
    int x := 0  
    condition alMenos3000  
  
    operation inc()  
        x++  
        if x=3000  
            signalC(alMenos3000)  
        end  
    end  
  
    operation informa()  
        if x<3000  
            waitC(alMenos3000)  
        end  
        write("Al menos hay 3000")  
    end  
end
```

Implementación de un monitor en C++

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3000)
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000)
        end
        write("Al menos hay 3000")
    end
end
```

monitorVar_x.hpp

```
class MonitorVar_x {
public:
    ...
private:
    ...
};
```

monitorVar_x.cpp

```
...
void MonitorVar_x::inc() {
    ...
}

void MonitorVar_x::informar() {
    ...
}
...
```

Implementación de un monitor en C++

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3000)
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000)
        end
        write("Al menos hay ")
    end
end
```

```
#include <mutex>
#include <condition_variable>

class MonitorVar_x {
public:
    MonitorVar_x(); //constructor
    ~MonitorVar_x(); //destructor
    void inc();
    void informar();
private:
    std::mutex mtxMonitor;
    std::condition_variable alMenos3000;
    int x;
};
```

Implementación de un monitor

```
Monitor var_x
    int x := 0
    condition alMenos3000

    operation inc()
        x++
        if x=3000
            signalC(alMenos3000)
        end
    end

    operation informa()
        if x<3000
            waitC(alMenos3000)
        end
        write("Al menos hay " + x)
    end
end
```

¿E<S<W?

¿E=W<S?

```
MonitorVar_x::MonitorVar_x() {
    x = 0;
}

MonitorVar_x::~MonitorVar_x() {
}

void MonitorVar_x::inc() {
    unique_lock<mutex> lck(mtxMonitor);

    x = x + 1;
    if (x == 3000) {
        alMenos3000.notify_one();
    }
} // aquí se libera mtxMonitor

void MonitorVar_x::informar() {
    unique_lock<mutex> lck(mtxMonitor);

    if (x < 3000) {
        alMenos3000.wait(lck);
    }
    cout << "Al menos hay 3000: "
        << x << endl;
} // aquí se libera mtxMonitor
```

monitorVar_x.cpp

Implementaci

```
Process P(i: 1..10):
    for j:=1..1000
        var_x.inc()
    end
end

Process informador::
    var_x.informa()
end
```

```
const int N = 10;

void informar(MonitorVar_x &mE) {
    mE.informar();
}

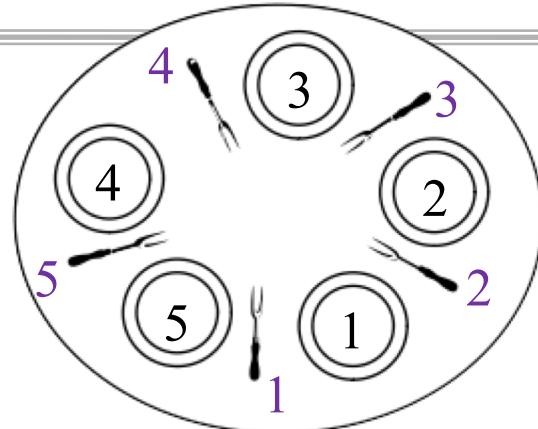
void incrementar(MonitorVar_x &mE) {
    for (int i=0; i<1000; ++i) {
        mE.inc();
    }
}

int main() {
    MonitorVar_x monX;
    thread P[N];
    thread informador = thread(informar, ref(monX));
    for (int i=0; i<N; ++i) {
        P[i] = thread(incrementar, ref(monX));
    }
    ...
    return 0;
}
```

main.cpp

El problema de los filósofos

- Dijkstra 65, Hoare 85
 - Prototipo de sistema en que los procesos comparten recursos conservativos
- Hay 5 filósofos sentados alrededor de una mesa para comer espaguetis
- Hay 5 tenedores, e infinita pasta
- Hacen falta dos tenedores para comer
 - Cada filósofo puede usar los que tiene más cerca
 - **Cada filósofo coge ambos tenedores a la vez**
- Se pide un programa que simule el comportamiento del sistema



```
Process filosofo(i:1..5):
    while true
        //piensa
        //coge tenedores
        //come
        //deja tenedores
    end
end
```

El problema de los filósofos

Diseño

```
Process filosofo(i:1..N)::  
    while true  
        //pensar  
        tenedores.coger(i)  
        //comer  
        tenedores.dejar(i)  
    end  
end
```

Monitor tenedores

```
boolean array[1..N] ocupado := (1..N, false)  
condition liberado //espero a que se libere alguno  
  
operation coger(integer i)  
    . . .  
operation dejar(integer i)  
    . . .
```

Monitor tenedores

```
boolean array[1..N] ocupado := (1..N, false)
condition liberado //espero a que se libere alguno

operation coger(integer i)
    while(ocupado[i] OR ocupado[i⊕1])
        waitC(liberado)
    end
    ocupado[i] := true
    ocupado[i⊕1] := true
end operation

. . .

end
```

Monitor tenedores

```
boolean array[1..N] ocupado := (1..N,false)
condition liberado //espero a que se libere alguno

. . .

operation dejar(integer i)
    ocupado[i] := false
    ocupado[i⊕1] := false
    signalC_all(liberado)
end operation
end monitor
```

El problema de los filósofos

Implementación C++

```
#include "monitorTenedores.hpp"
MonitorTenedores tenedores;
void filosofo(unsigned i){
    while (true) {
        //pensar
        tenedores.coger(i)
        //comer
        tenedores.dejar(i)
    }
}
int main() {
    thread filosofo[N];
    ...
}
```

```
Process filosofo(i:1..N):::
    while true
        //pensar
        tenedores.coger(i)
        //comer
        tenedores.dejar(i)
    end
end
```

El problema de los filósofos

Implementación C++

```
#include "monitorTenedores.hpp"

void filosofo(unsigned i, MonitorTenedores &tenedores) {
    while (true) {
        //pensar
        tenedores.coger(i)
        //comer
        tenedores.dejar(i)
    }
}

int main() {
    MonitorTenedores tenedores;
    thread filosofo[N];
    ...
}
```

```
Process filosofo(i:1..N)::  
    while true  
        //pensar  
        tenedores.coger(i)  
        //comer  
        tenedores.dejar(i)  
    end  
end
```

El problema de los filósofos

monitorTenedores.hpp

```
#include <mutex>
#include <condition_variable>

class MonitorTenedores {
public:
    MonitorTenedores();
    void coger(int i);
    void dejar(int i);
private:
    bool ocupado[N]; //asumir N declarado
    mutex mtxMonitor; //para la condición
    condition_variable liberado;
};
```

```
Monitor tenedores
boolean array[1..N] ocupado := (1..N, false)
condition liberado

operation coger(integer i)
. . .
operation dejar(integer i)
. . .
```

El problema de los filosofos

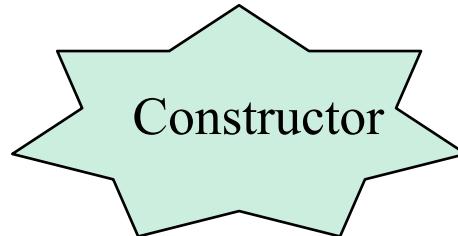
monitorTenedores.cpp

```
Monitor tenedores
boolean array[1..N] ocupado := (1..N, false)
condition liberado

operation coger(integer i)
.
.
.
operation dejar(integer i)
.
.
```

```
MonitorTenedores::MonitorTenedores () {
```

```
    for(int i=0; i<N; i++) {
        ocupado[i] = false;
    }
};
```



El problema de los filósofos

monitorTenedores.cpp

```
operation coger(integer i)
    while(ocupado[i] OR ocupado[i+1])
        waitC(liberado)
    end
    ocupado[i] := true
    ocupado[i⊕1] := true
end
```

```
void MonitorTenedores::coger(int i) {
    unique_lock<mutex> lck(mtxMonitor);
    //esperar tenedores libres
    while(ocupado[i] || ocupado[(i + 1) % N]) {
        liberado.wait(lck);
    }
    ocupado[i] = true;
    ocupado[(i + 1) % N] = true;
};
```

Se bloquea hasta poder cogerlo

Se libera automáticamente al cerrar el bloque

El problema de los filósofos

monitorTenedores.cpp

```
operation dejar(integer i)
    ocupado[i] := false
    ocupado[i⊕1] := false
    signalC_all(liberado)
end
```

```
void MonitorTenedores::dejar(int i) {

    unique_lock<mutex> lck(mtxMonitor);

    ocupado[i] = false;
    ocupado[(i + 1) % N] = false;
    liberado.notify_all();
}
```

Implementación de un monitor en C++

- ¿Qué pasa si una función del monitor quiere invocar otra función del mismo?
- ¿Puedo implementar funciones recursivas?

```
recursive_mutex mtxMonitor  
condition_variable_any liberado;
```

```
unique_lock<recursive_mutex> lck(mtxMonitor);
```

El problema de los lectores/escritores

- **Problema:**
 - dos tipos de procesos para acceder a una base de datos:
 - lectores: consultan la BBDD
 - escritores: consultan y modifican la BBDD
 - cualquier transacción aislada mantiene la consistencia de la BBDD
 - cuando un escritor accede a la BBDD, es el único proceso que la puede usar
 - varios lectores pueden acceder simultáneamente

El problema de los lectores/escritores

Diseño

Process lector

```
while true
```

```
...
```

```
controlaLyE.pideLeer()
```

```
...
```

```
controlaLyE.dejaDeLeer()
```

```
...
```

```
end
```

monitor controlaLyE

```
...
```

```
operation pideLeer()
```

```
operation dejaDeLeer()
```

```
operation pideEscribir()
```

```
operation dejaDeEscribir()
```

```
end
```

Process escritor

```
while true
```

```
...
```

```
controlaLyE.pideEscribir()
```

```
...
```

```
controlaLyE.dejaDeEscribir()
```

```
...
```

```
end
```

El problema de los lectores/escritores

Diseño

```
monitor controlaLyE
    integer nLec := 0
        nEsc :=0
    condition okLeer --señala nEsc=0
        okEscribir --señala nEsc=0 AND nLec=0
    operation pideLeer()
        while (nEsc>0)
            waitC(okLeer)
            nLec := nLec+1

    operation dejaDeLeer()
        nLec := nLec-1
        if (nLec=0)
            signalC(okEscribir) -- ¿signalC(okLeer) ?
```

El problema de los lectores/escritores

```
monitor controlaLyE
    ...
    operation pideEscribir()
        while (nLec>0) OR (nEsc>0)
            waitC(okEscribir)
            nEsc := nEsc+1

    operation dejaDeEscribir()
        nEsc := nEsc-1
        signalC(okEscribir)
        signalC_all(okLeer)
```

Un ejercicio de trabajo

- Diseñar, mediante monitores, un programa que se comporte como el siguiente

```
int x := . . . //lo que sea
y := . . . //lo que sea
z := . . . //lo que sea
w := . . . //lo que sea

process cliente(i:1..300):::
    int mix
    mix := . . . //lo que sea

    <await x+y>0
        y := y+1
        x := mix
    >
    <await z>0
        z := z+3+mix-w
    >
    <y := 2*x+y>
end
```