Sistemas Operativos Avanzados

Tema 2. **Sincronización y Concurrencia**

Profesor:

Dr. J. Octavio Gutiérrez García

octavio.gutierrez@itam.mx

Hilos ejecutando dos tareas independientes

```
package testrace;

class MyThread extends Thread {
    public void loop() {
        for(int i = 0; i < 10; i++) {
            System.out.println("In thread " +
                               Thread.currentThread().getName() +
                               " iteration " + i);
        }
    }
    public void run() {
        this.loop();
    }
    public static void main(String args[]) throws Exception {
        MyThread mt1 = new MyThread();
        mt1.start();
        MyThread mt2 = new MyThread();
        mt2.start();
    }
}
```

No determinismo en programas multi-hilo

```
run:
In thread Thread-1 iteration 0
In thread Thread-1 iteration 1
In thread Thread-1 iteration 2
In thread Thread-1 iteration 3
In thread Thread-1 iteration 4
In thread Thread-1 iteration 5
In thread Thread-1 iteration 6
In thread Thread-1 iteration 7
In thread Thread-1 iteration 8
In thread Thread-1 iteration 9
In thread Thread-0 iteration 0
In thread Thread-0 iteration 1
In thread Thread-0 iteration 2
In thread Thread-0 iteration 3
In thread Thread-0 iteration 4
In thread Thread-0 iteration 5
In thread Thread-0 iteration 6
In thread Thread-0 iteration 7
In thread Thread-0 iteration 8
In thread Thread-0 iteration 9
```

```
run:
In thread Thread-0 iteration 0
In thread Thread-0 iteration 1
In thread Thread-0 iteration 2
In thread Thread-0 iteration 3
In thread Thread-0 iteration 4
In thread Thread-0 iteration 5
In thread Thread-0 iteration 6
In thread Thread-0 iteration 7
In thread Thread-0 iteration 8
In thread Thread-0 iteration 9
In thread Thread-1 iteration 0
In thread Thread-1 iteration 1
In thread Thread-1 iteration 2
In thread Thread-1 iteration 3
In thread Thread-1 iteration 4
In thread Thread-1 iteration 5
In thread Thread-1 iteration 6
In thread Thread-1 iteration 7
In thread Thread-1 iteration 8
In thread Thread-1 iteration 9
```

No determinismo en programas multi-hilo

```
In thread Thread-0 iteration 0
In thread Thread-1 iteration 0
In thread Thread-1 iteration 1
In thread Thread-1 iteration 2
In thread Thread-1 iteration 3
In thread Thread-1 iteration 4
In thread Thread-1 iteration 5
In thread Thread-1 iteration 6
In thread Thread-1 iteration 7
In thread Thread-1 iteration 8
In thread Thread-1 iteration 9
In thread Thread-0 iteration 1
In thread Thread-0 iteration 2
In thread Thread-0 iteration 3
In thread Thread-0 iteration 4
In thread Thread-0 iteration 5
In thread Thread-0 iteration 6
In thread Thread-0 iteration 7
In thread Thread-0 iteration 8
In thread Thread-0 iteration 9
```

Hilos modificando datos concurrentemente

```
class Counter {
    public static long count = 0;
}
class UseCounter implements Runnable {
    public void run() {
        for (int i = 0; i < 3; i++) {
            Counter.count++;
            System.out.print(Counter.count + " ");
        }
    }
}
public class DataRace {
    public static void main(String args[]) throws InterruptedException {
        UseCounter c = new UseCounter();
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        Thread t3 = new Thread(c);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Condición de Carrera (Data race)

```
run:
1  2  3  4  5  6  7  8  9
```

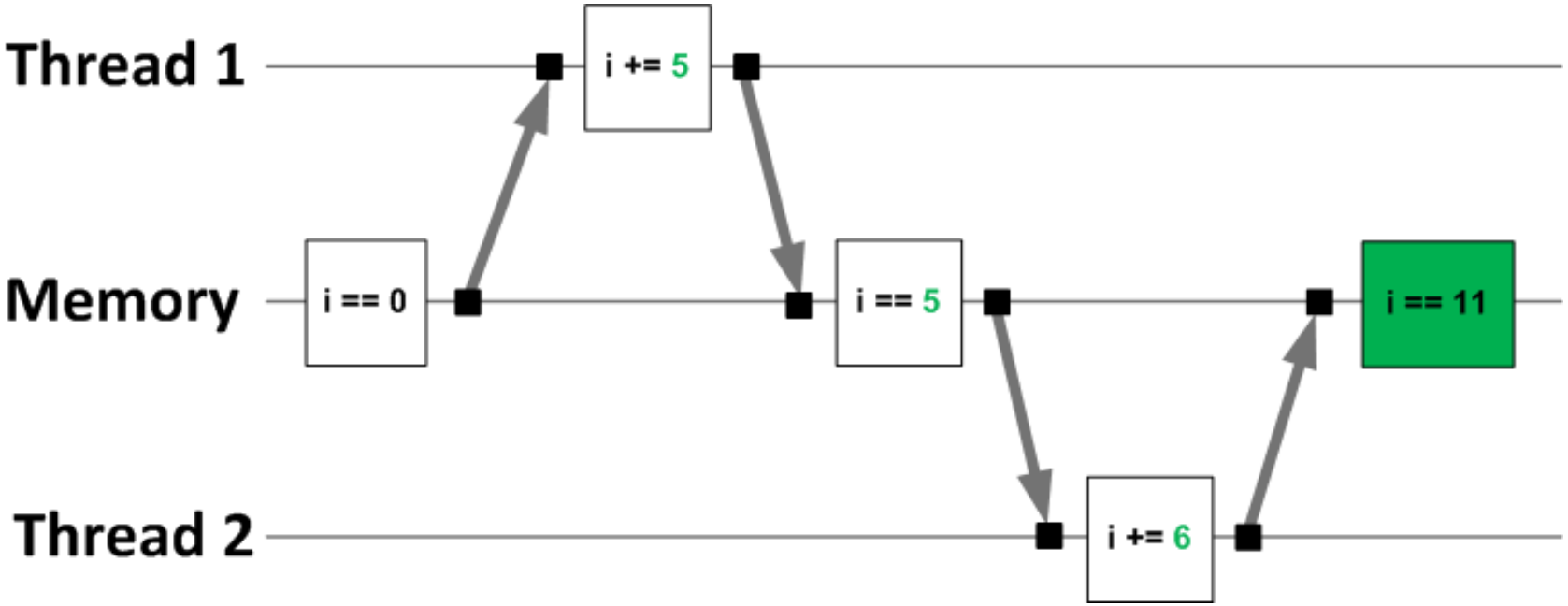
Con sincronización

```
run:
2  3  3  5  6  7  8  9  4
```

Sin sincronización

Ejemplo a detalle

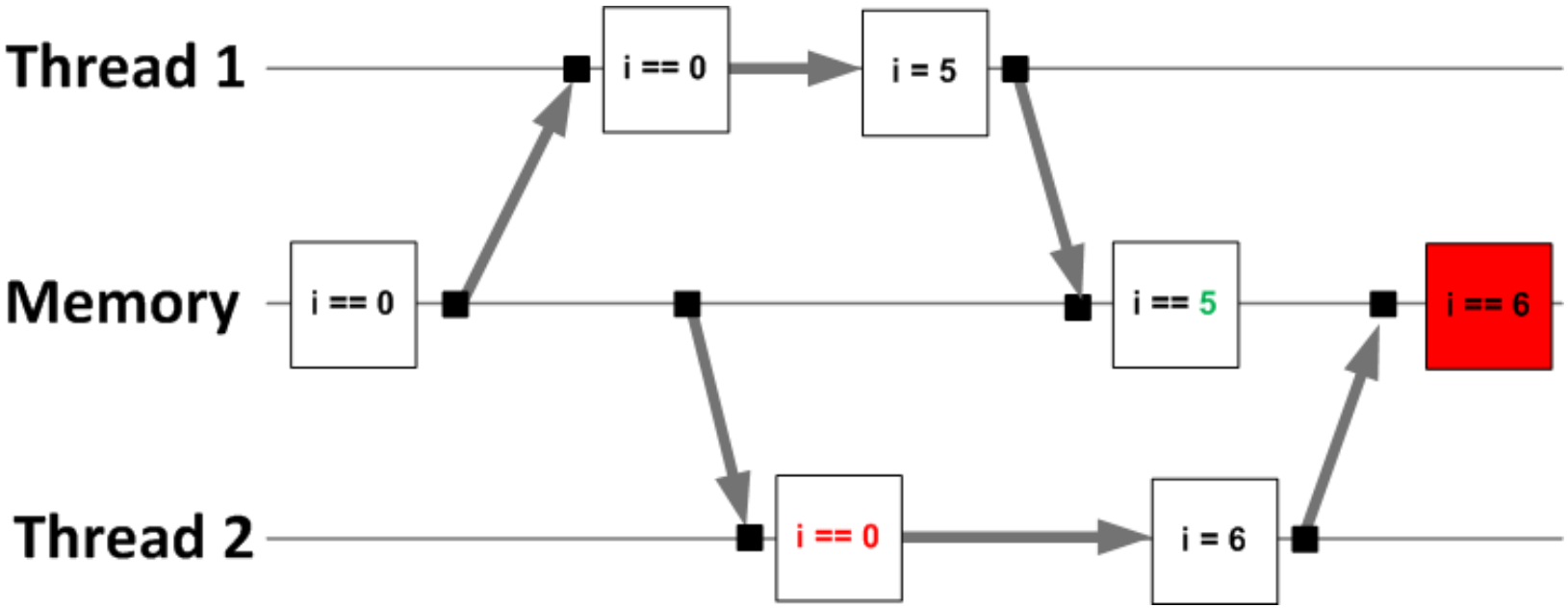
Hilo 1 suma 5 a *i*



Hilo 2 suma 6 a *i*

Ejemplo a detalle

Hilo 1 suma 5 a *i*



Hilo 2 suma 6 a *i*

Data races

- En resumen, “data races” ocurren cuando múltiples hilos acceden a los mismos datos y al menos uno los modifica

- Normalmente es un bug

Data races

- Difícil de detectar si ocurrieron
  - No tiene efectos inmediatos
  - El programa continua normalmente
  - Daña los datos globales
- Difícil de detectar manualmente
  - No son reproducibles, dependen del “timing” de los hilos
  - Herramientas de desarrollo no proveen mucho soporte

Detección automática de Data races

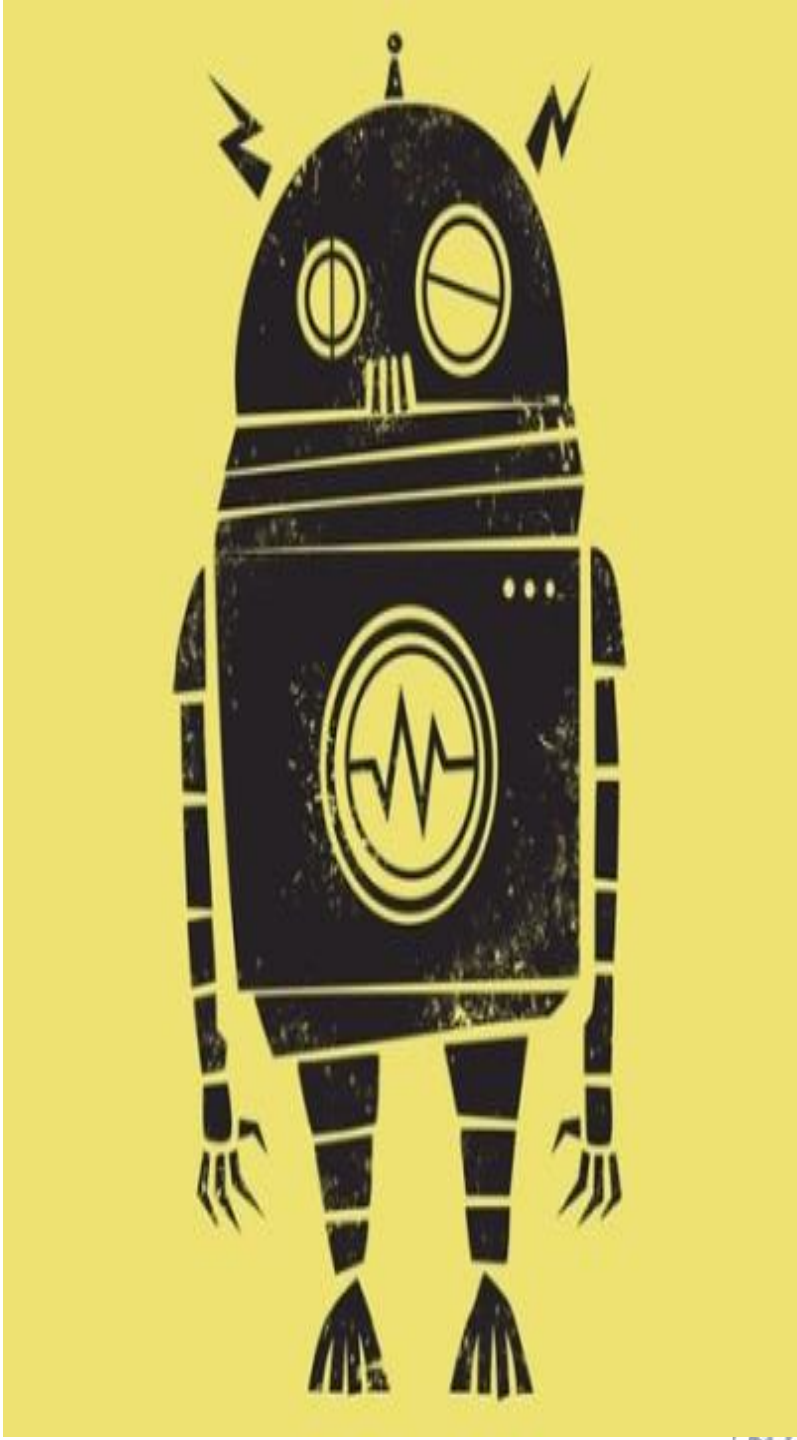
- Detección estática
  - Analiza el código fuente
  - Data races son prevenidas por el programador
- Detección dinámica
  - Analiza la ejecución del programa en tiempo en tiempo real
  - Post-mortem


Dinámicos
VS
Estáticos

Enfoque estático

- Ventajas
  - No requiere que el programa esté en ejecución
  - Analiza todo el código
  - No depende de los datos de entrada del programa
  - Existen muchas herramientas (e.g., FindBugs)

- Desventajas
  - Sin solución para casos generales
  - Se tiene que reducir la profundidad del análisis



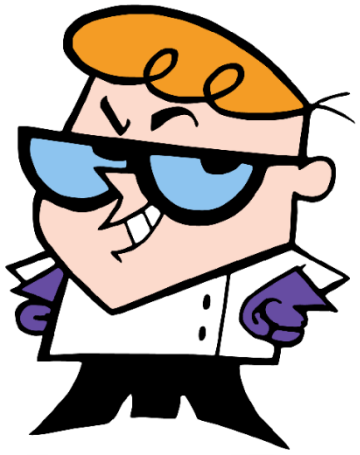
Enfoque dinámico

- Ventajas
  - Información completa del flujo del programa
  - Menores niveles de falsas alarmas
- Desventajas
  - Costo computacional muy alto
- No existen detectores dinámicos estables (al menos para Java) según Vitaly Trifanov de Devexperts

¿Detectores dinámicos o estáticos?

- Se podrían usar ambos

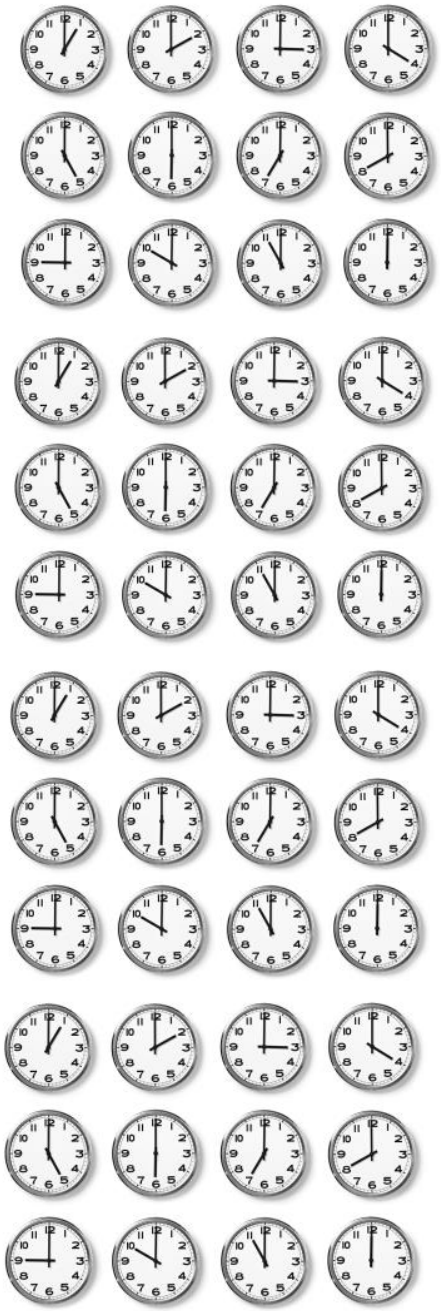
- Detectores estáticos podrían eliminar inconsistencia de sincronización en etapas tempranas de implementación y detectores dinámicos podrían monitorear solo partes del programa en ejecución (e.g., no monitorear clases thread-safe)



Relojes lógicos

- Asignan una secuencia numérica a los mensajes intercambiados entre los procesos

- Se asume que cada proceso tiene su propio reloj local.

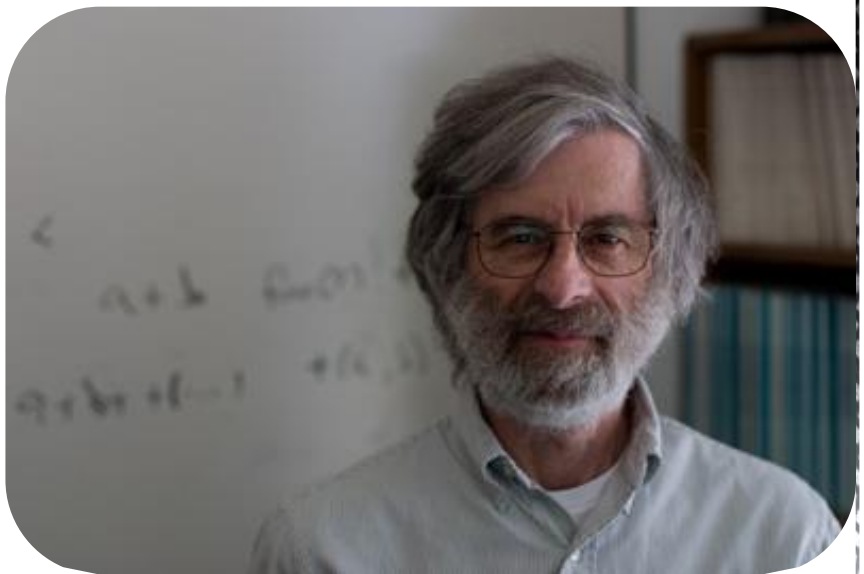


Relación Happens-before de Lamport

- a -> b evento *a* sucedió antes que evento *b*

- Transitiva:

  si *a* -> *b* and *b* -> *c* then *a* -> *c*



Relación Happens-before de Lamport

Asigna el valor del “reloj” a cada evento

- Si ***a->b*** entonces **reloj(*a*) < reloj(*b*)**

Si *a* y *b* ocurren en diferentes procesos que no intercambian mensajes, entonces ni ***a->b*** o ***b->a*** son verdad. En ese caso se dice que los eventos son concurrentes.

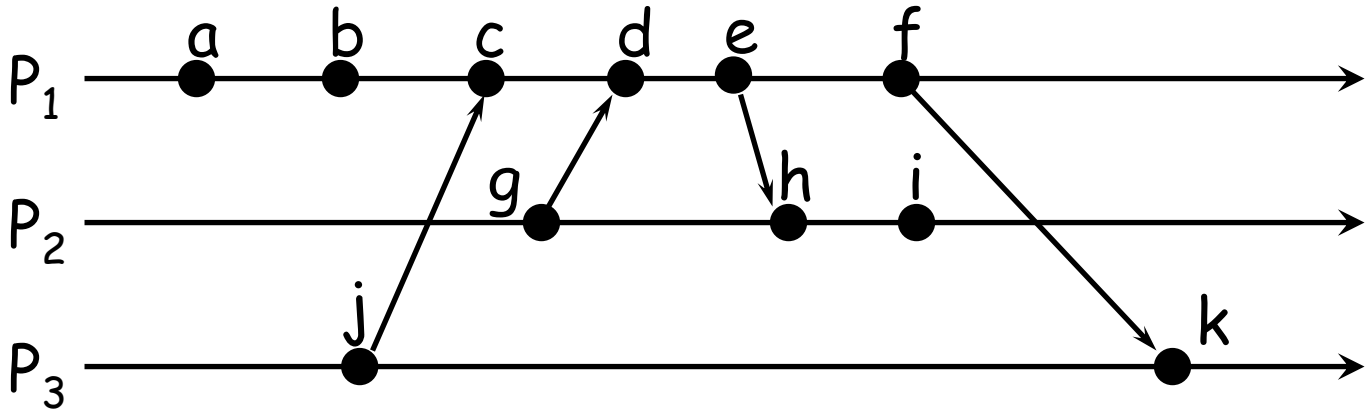
Algoritmo Happens-before de Lamport

- Cada mensaje contiene una “estampa de tiempo del reloj lógico del emisor”

- Cuando el mensaje llega:
  - Si el reloj del receptor < la estampa del mensaje
    - Actualiza el reloj a -> ***estampa del mensaje + 1***

- El reloj debe de ser avanzado, cada vez que ocurre un evento en un proceso

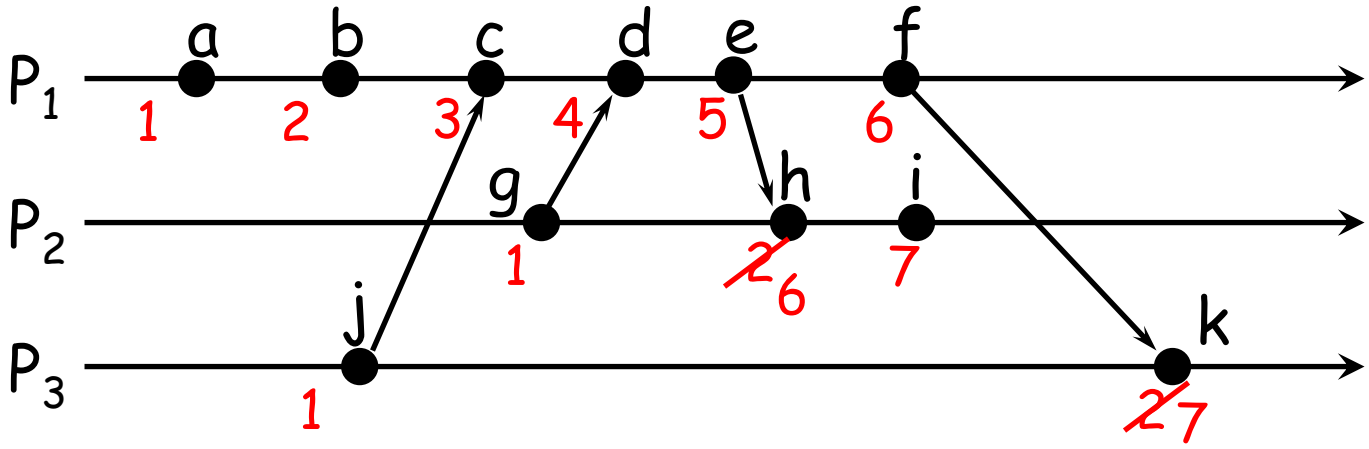
Algoritmo de ordenamiento Happens-before



Permite obtener un orden parcial de los eventos

Tanto detectores dinámicos como estáticos (de data races) han sido desarrollados utilizando como base la relación happens-before

Algoritmo de ordenamiento Happens-before



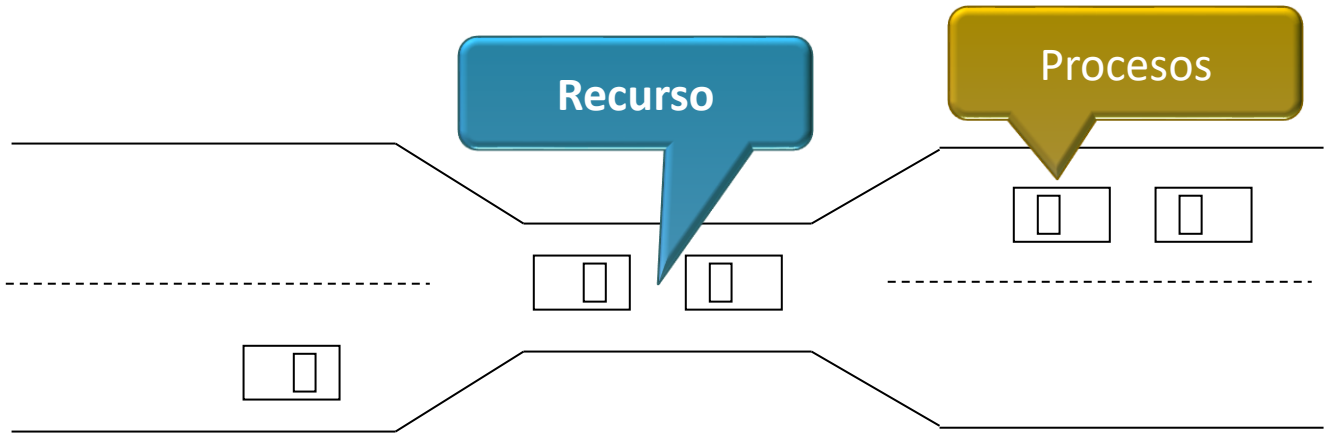
Permite obtener un orden parcial de los eventos

Tanto detectores dinámicos como estáticos (de data races) han sido desarrollados utilizando como base la relación happens-before

Deadlock

Se requiere dinero para hacer dinero

No puedes obtener un trabajo sin experiencia,
No puedes obtener experiencia sin un trabajo



Caracterización de deadlocks

4 condiciones necesarias

1. **Exclusión mutua**

   Uno o más de un recurso debe de estar en posesión de un proceso en un modo exclusivo

2. **Mantiene y Espera**

   Un proceso reserva y conserva un recurso mientras espera por otro

3. **No hay derecho preferente**

   Nadie puede hacer que el proceso libere un recurso

4. **Espera circular**

   Proceso A espera -> Proceso B espera -> Proceso C espera -> Proceso A

Prevención de deadlocks

Evita cualquiera de las **4** condiciones necesarias

1. **Exclusión mutua**
   Establece un sistema de turnos para los recursos
   Comparte los recursos concurrente cuando sea posible

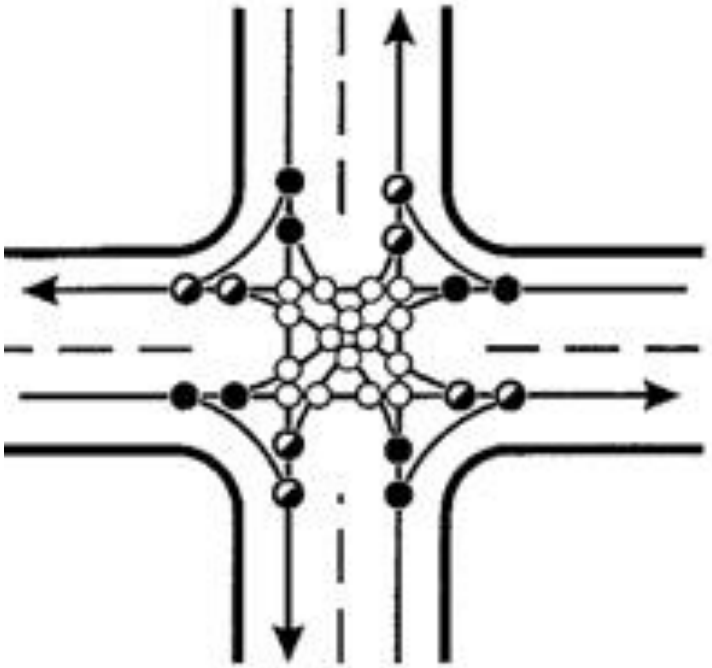
2. **Mantiene y Espera**
   Reserva todos los recursos requeridos al mismo tiempo

3. **No hay derecho preferente**
   Introduce derecho preferente -> prioridades.
   Introduce Timeouts

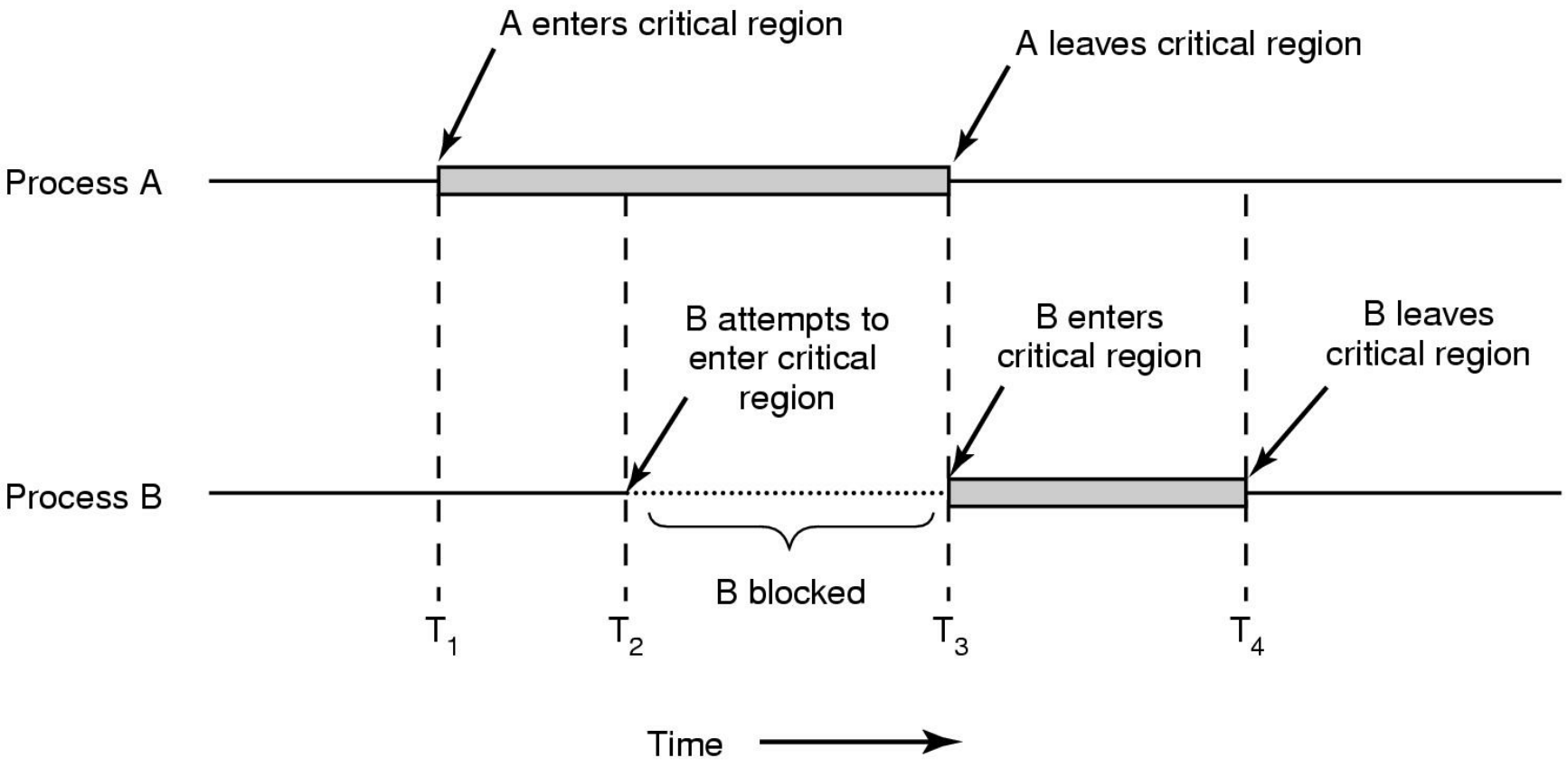
4. **Espera circular**
   Obliga a los procesos a reservar los recursos en un orden específico

Mecanismos de sincronización

- Candados
- Semáforos
- Barreras
- Monitores



Regiones críticas



Exclusión mutua con soporte de hardware

- Deshabilitar interrupciones

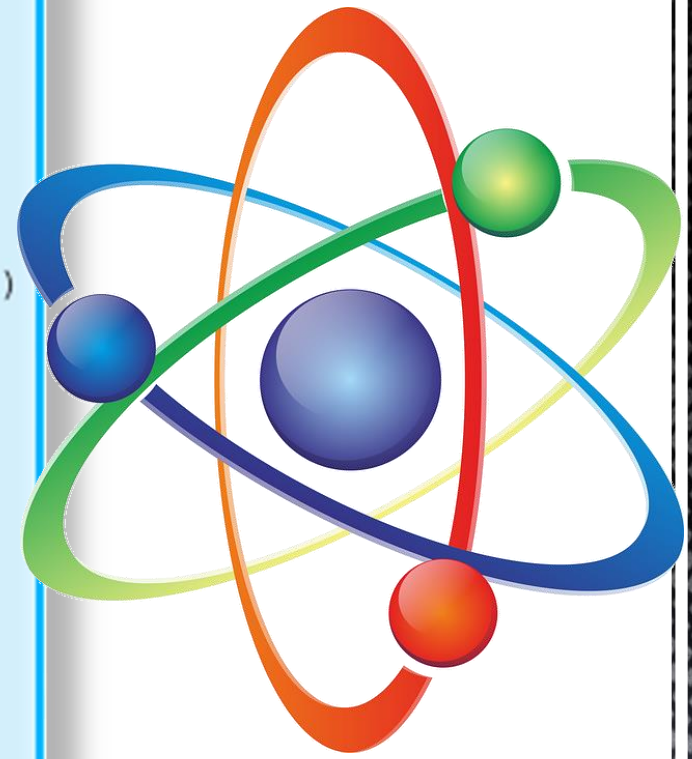
```
while (true) {
  /* deshabilita interrupciones */;
  /* región critica */;
  /* habilita interrupciones */;
  /* demás instrucciones  */;
}
```

Exclusión mutua con soporte de hardware

Instrucción compare_and_swap

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```



Exclusión mutua con soporte de hardware

- Desventajas
  - Esperando pero “trabajando”
  - Se podría ocasionar “Hambruna” -> Starvation
  - Podrían existir deadlocks

Estructura de un semáforo

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Límite

Entrada

Salida

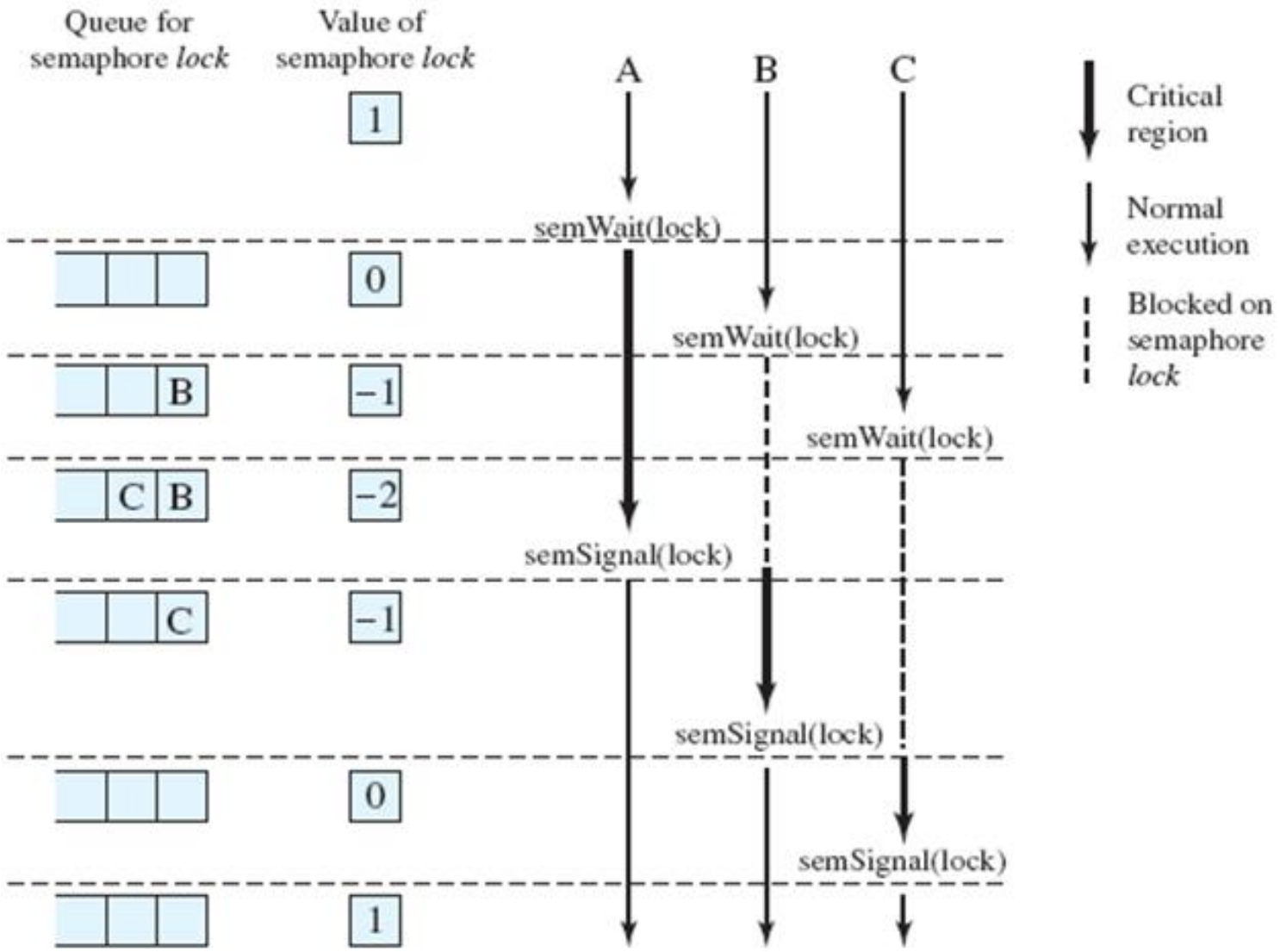
Exclusión mutua con semáforos

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

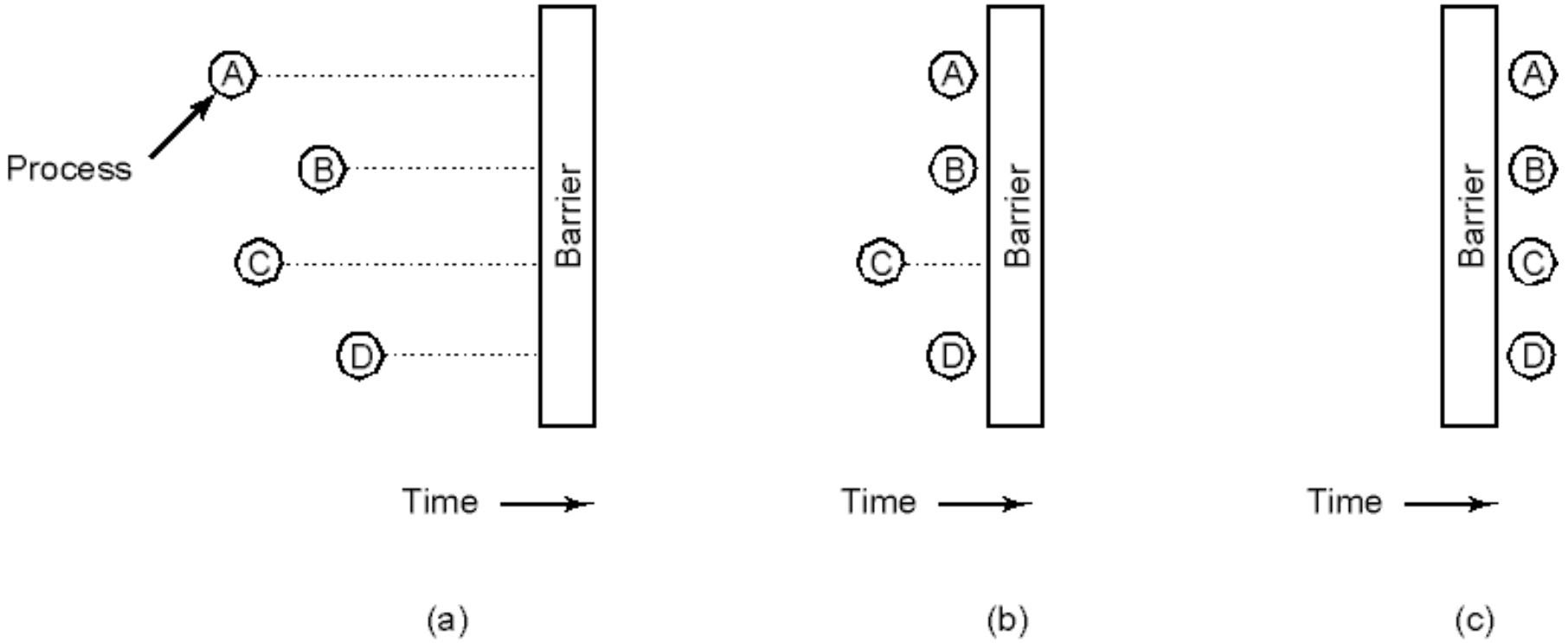
Entrada

Salida

Semáforos


Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
−1
semWait(lock)
C B
−2
semSignal(lock)
C
−1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*

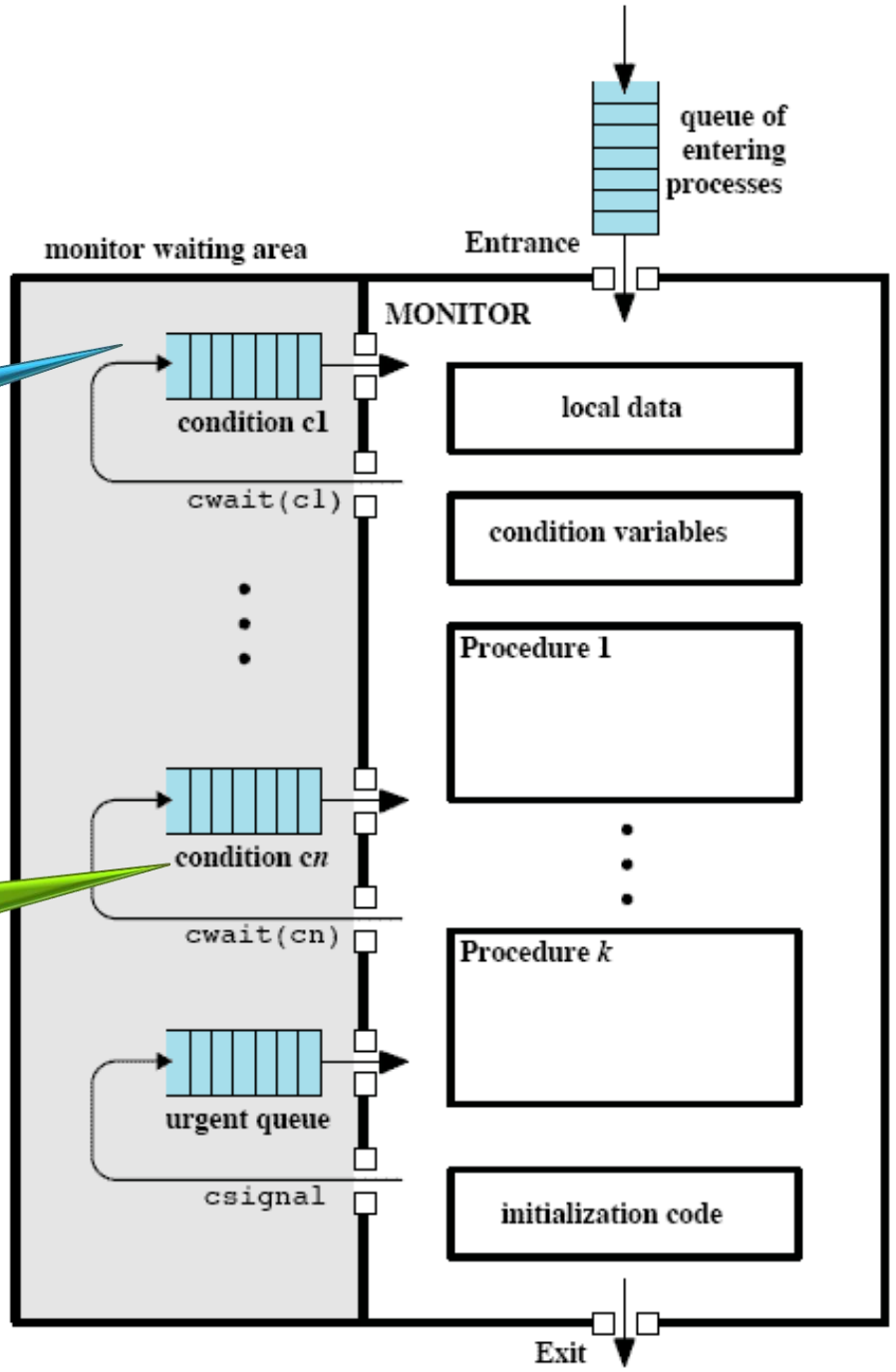
Barreras


Process
A
B
C
D
Barrier
Time
(a)
A
B
C
D
Barrier
Time
(b)
Barrier
A
B
C
D
Time
(c)

Monitores

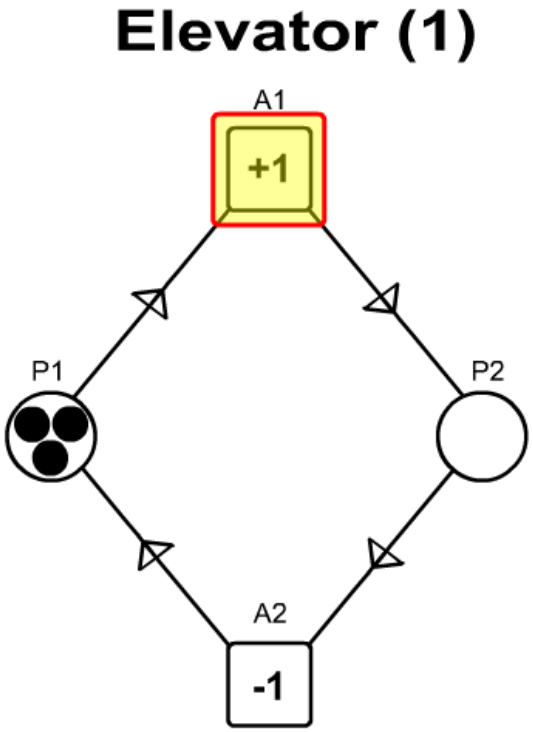
Colas de hilos (procesos)

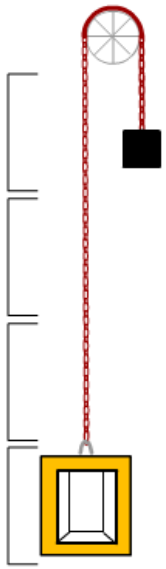
Las variables de condición controlan las colas



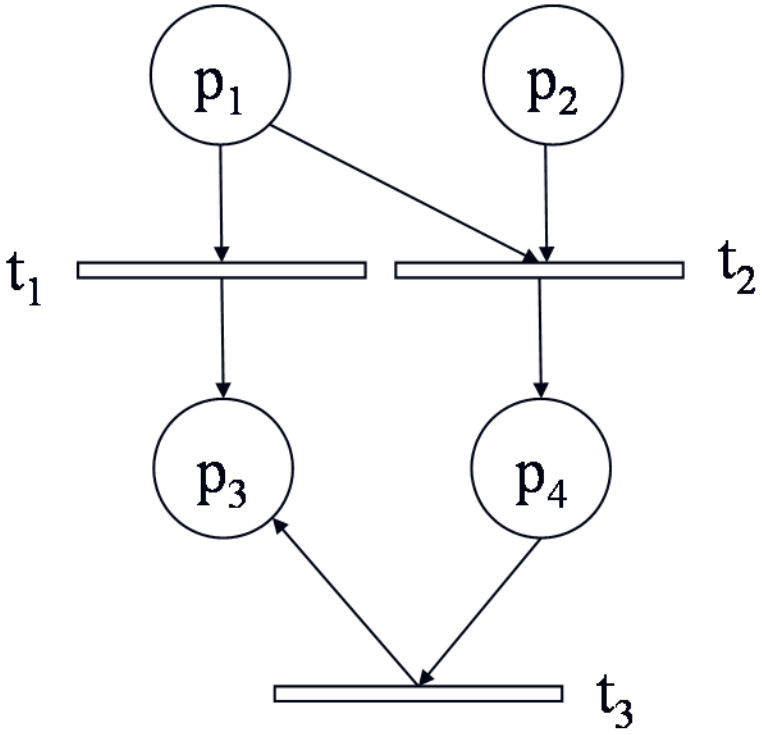
Redes de Petri

- Mecanismo formal para modelado de concurrencia, paralelismo y sincronización





Redes de Petri : Estructura



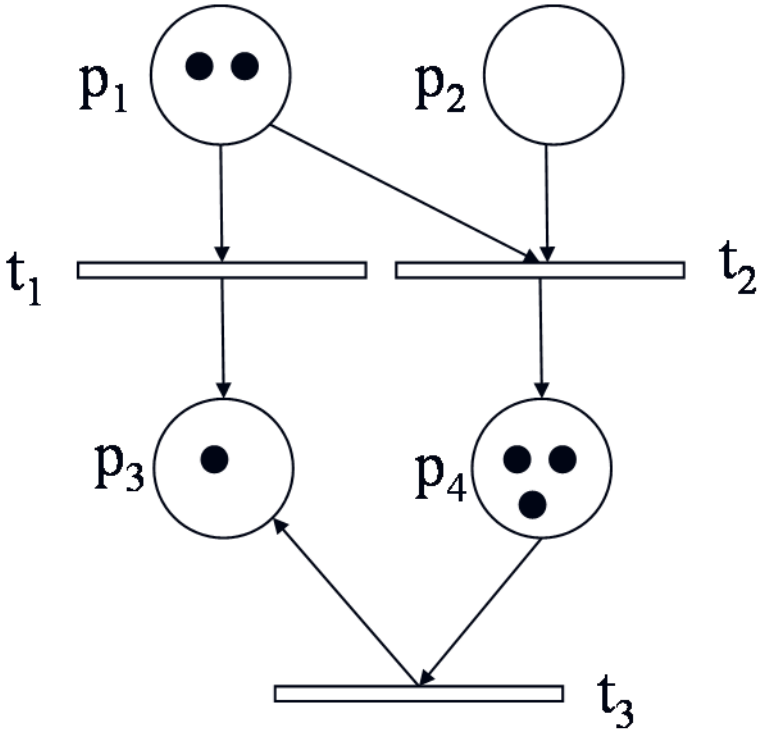
$P = \{p_1, p_2, p_3, p_4\}$

$T = \{t_1, t_2, t_3\}$

$In = \{<p_1, t_1>, <p_1, t_2>, <p_2, t_2>, <p_4, t_3>\}$

$Out = \{<t_1, p_3>, <t_2, p_4>, <t_3, p_3>\}$

Redes de Petri : Marcado (Tokens)



$P = \{p_1, p_2, p_3, p_4\}$

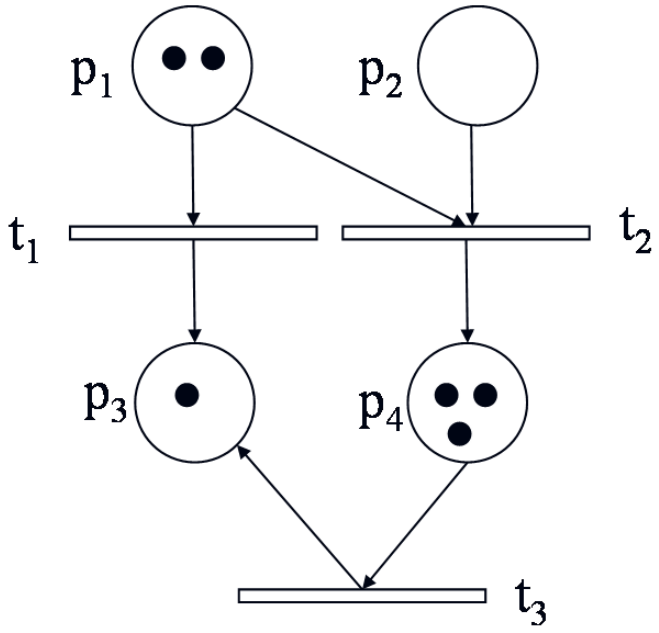
$T = \{t_1, t_2, t_3\}$

$In = \{<p_1, t_1>, <p_1, t_2>, <p_2, t_2>, <p_4, t_3>\}$

$Out = \{<t_1, p_3>, <t_2, p_4>, <t_3, p_3>\}$

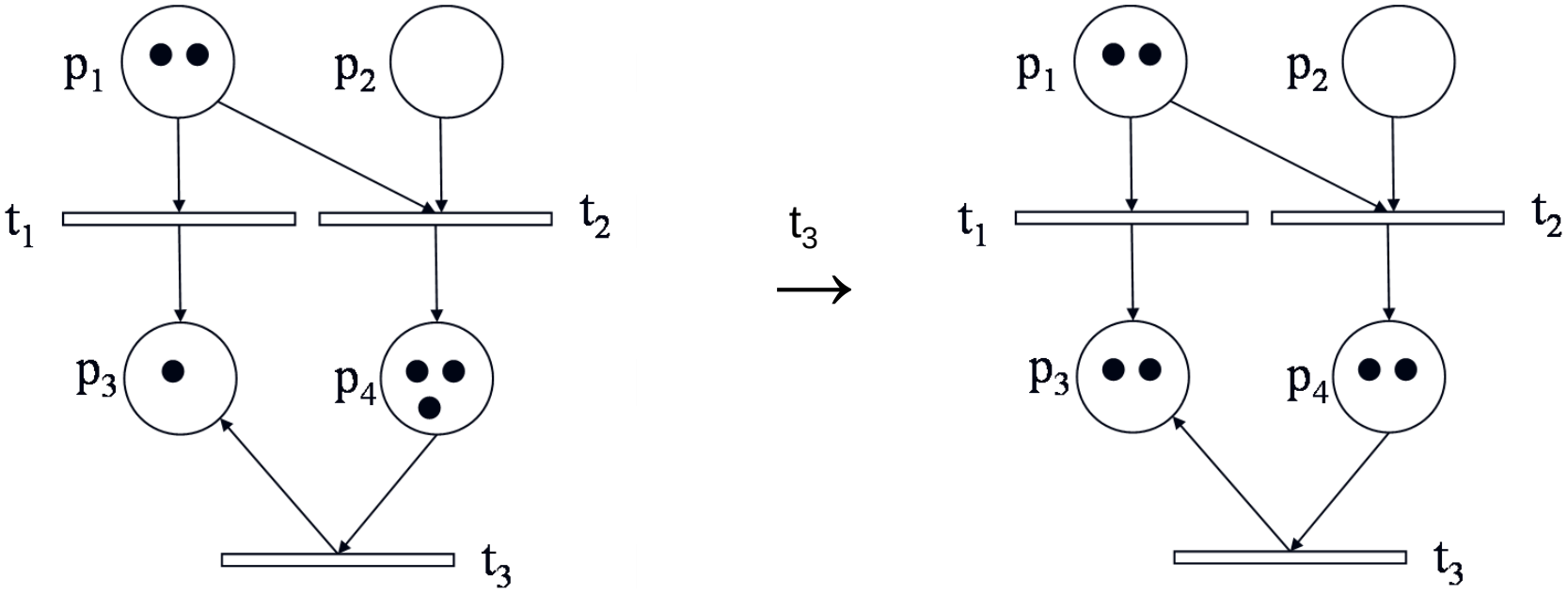
$M = \{2, 0, 1, 3\}$

Redes de Petri : Transiciones habilitadas



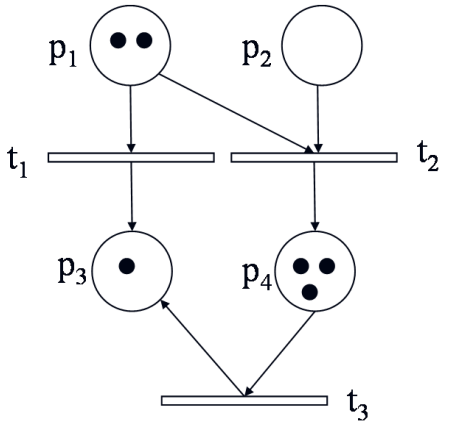
Transiciones habilitadas: t_1, t_3

Redes de Petri : Reglas de disparo



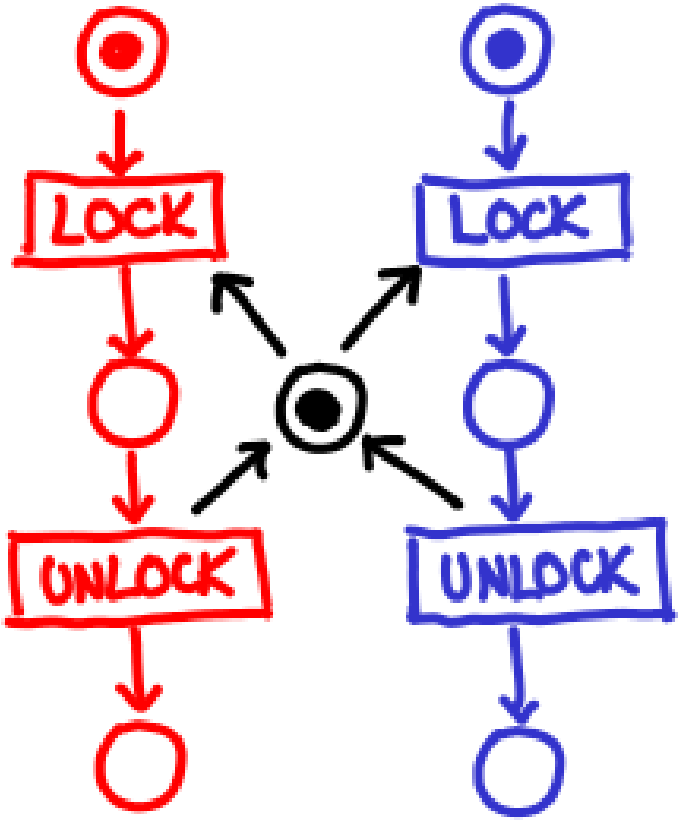
M = {<2, 0, 1, 3>, <2, 0, 2, 2>}

Redes de Petri : Tokens



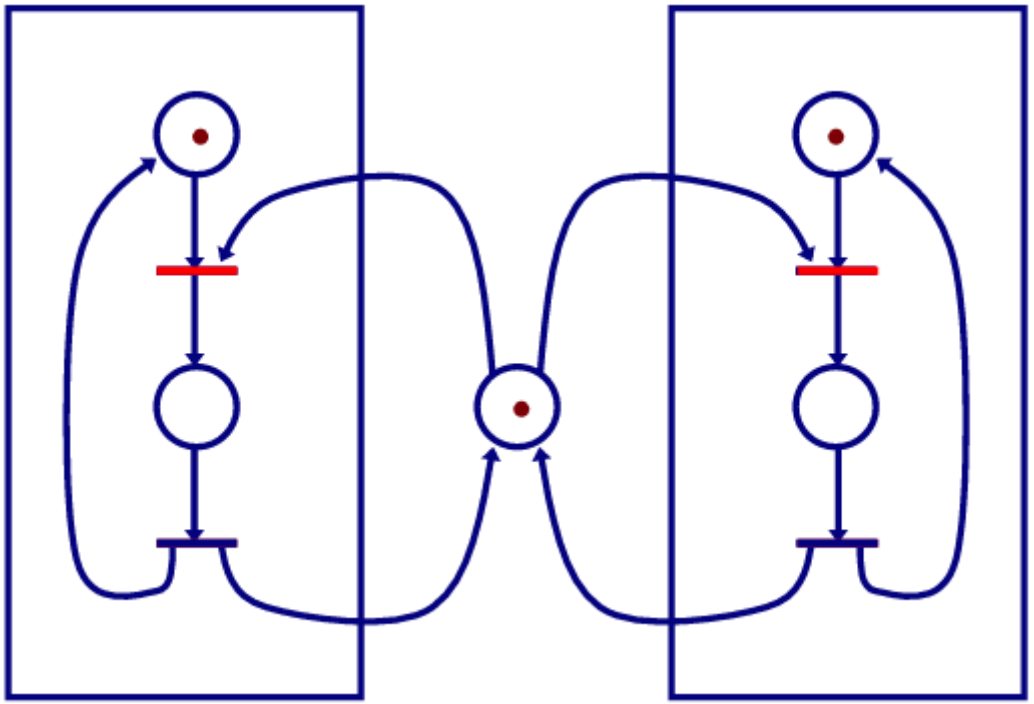
- Disponibilidad de un recurso
- Trabajos a realizar
- Control del Flujo
- Condiciones de sincronización (junto con transiciones)

Redes de Petri : Concurrencia



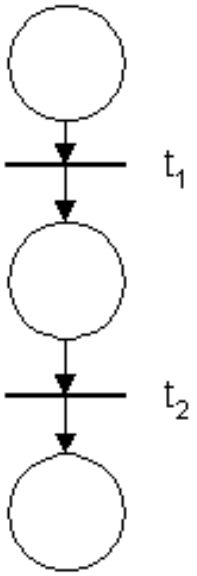
Redes de Petri : Concurrencia

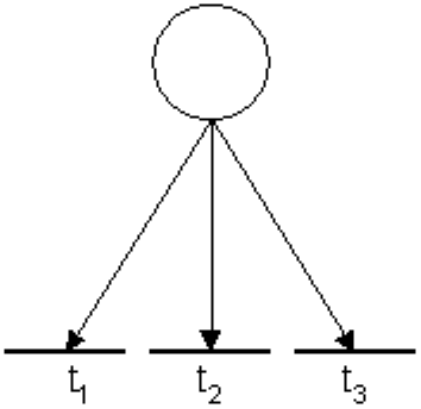
- Dos procesos son forzados a sincronizarse



Redes de Petri : Patrones básicos

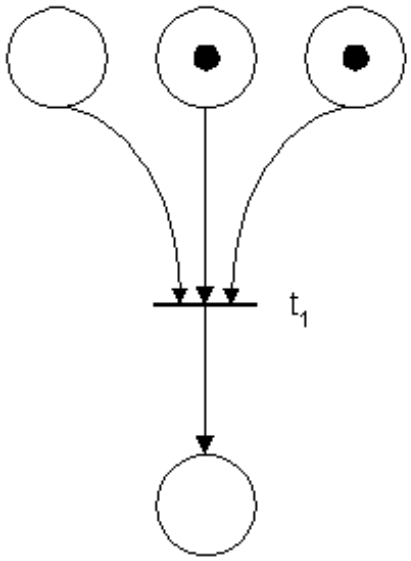
- Secuencial
- Conflicto

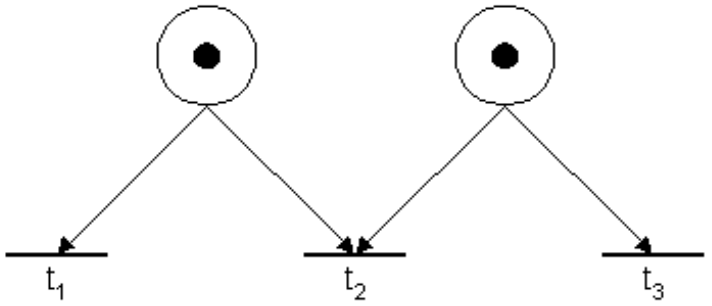




Redes de Petri : Patrones básicos

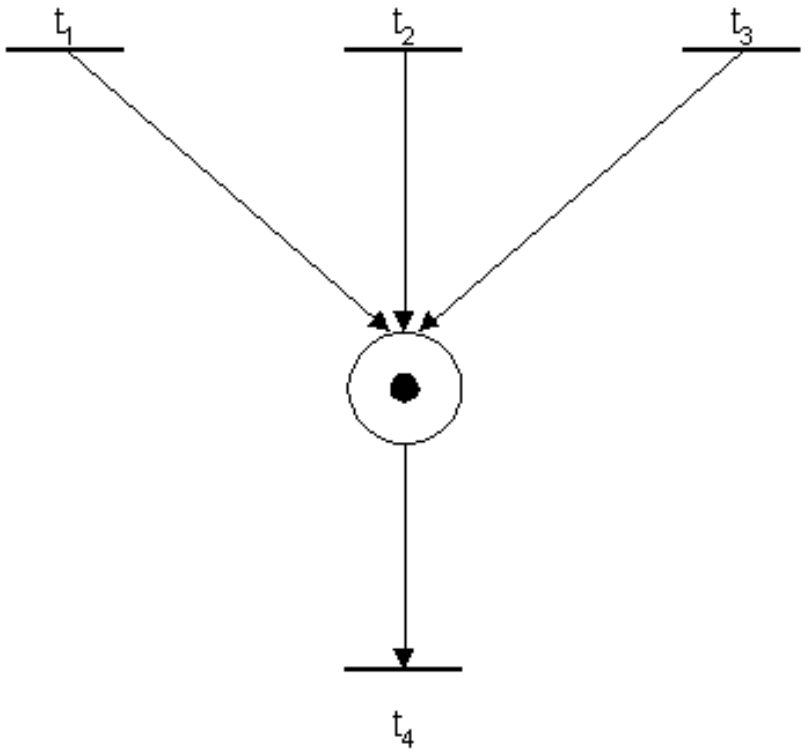
- Sincronización
- Unión





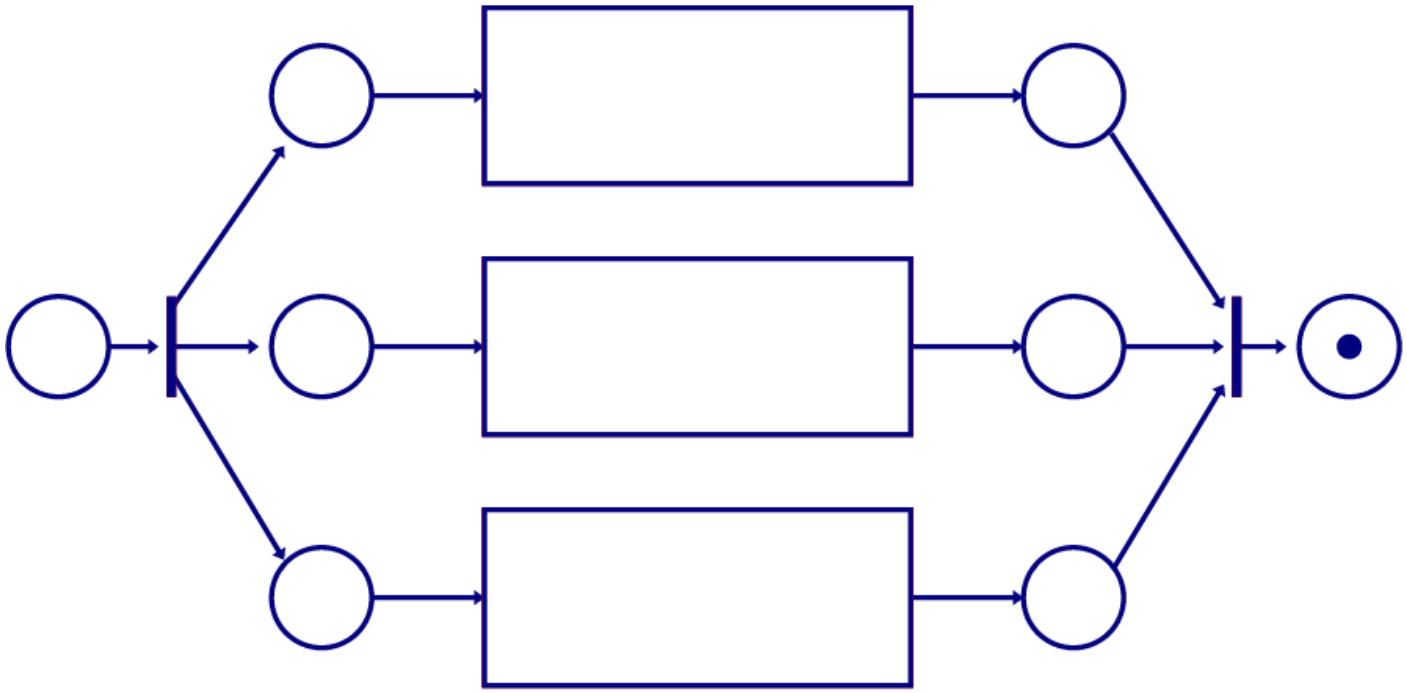
Redes de Petri : Patrones básicos

- Confusión



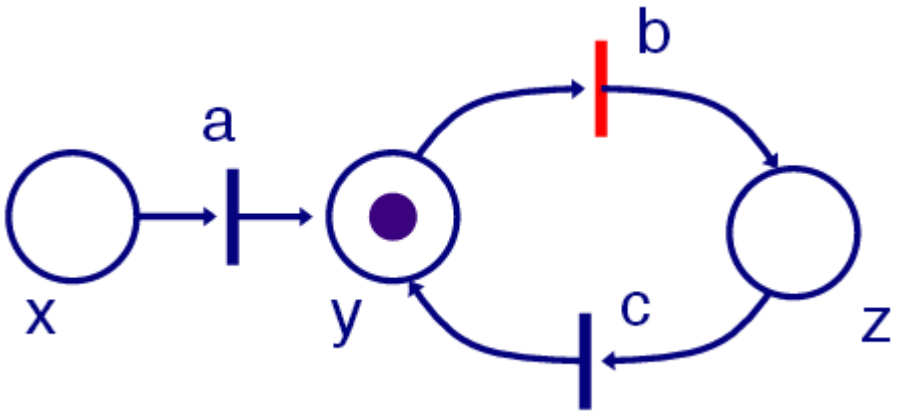
Redes de Petri : Patrones básicos

- Fork y Join



Redes de Petri: Propiedades formales

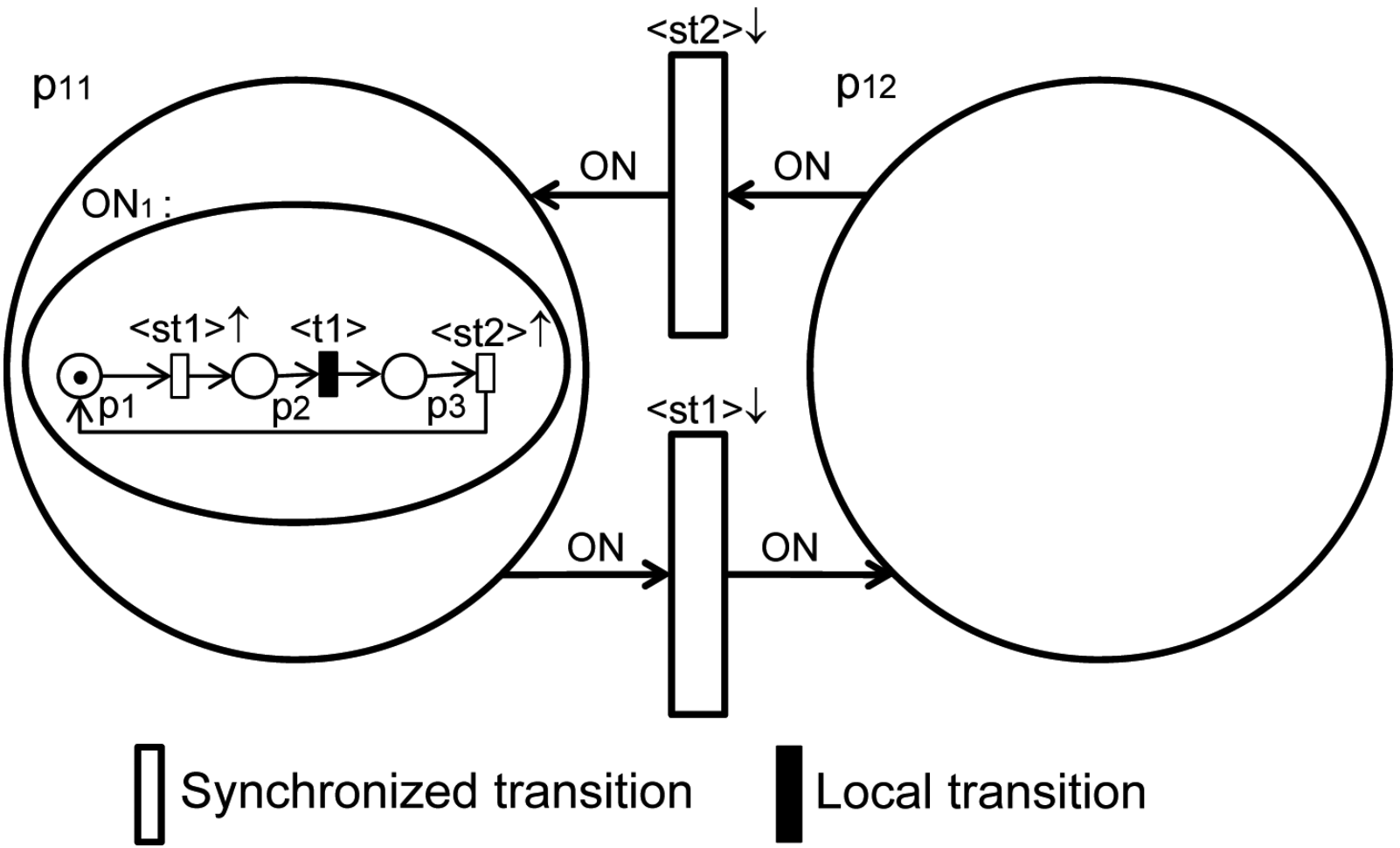
- Vivacidad (liveness)
  - Un transición tiene un deadlock si nunca se puede disparar
  - Una transición es viva si nunca puede llegar a un deadlock



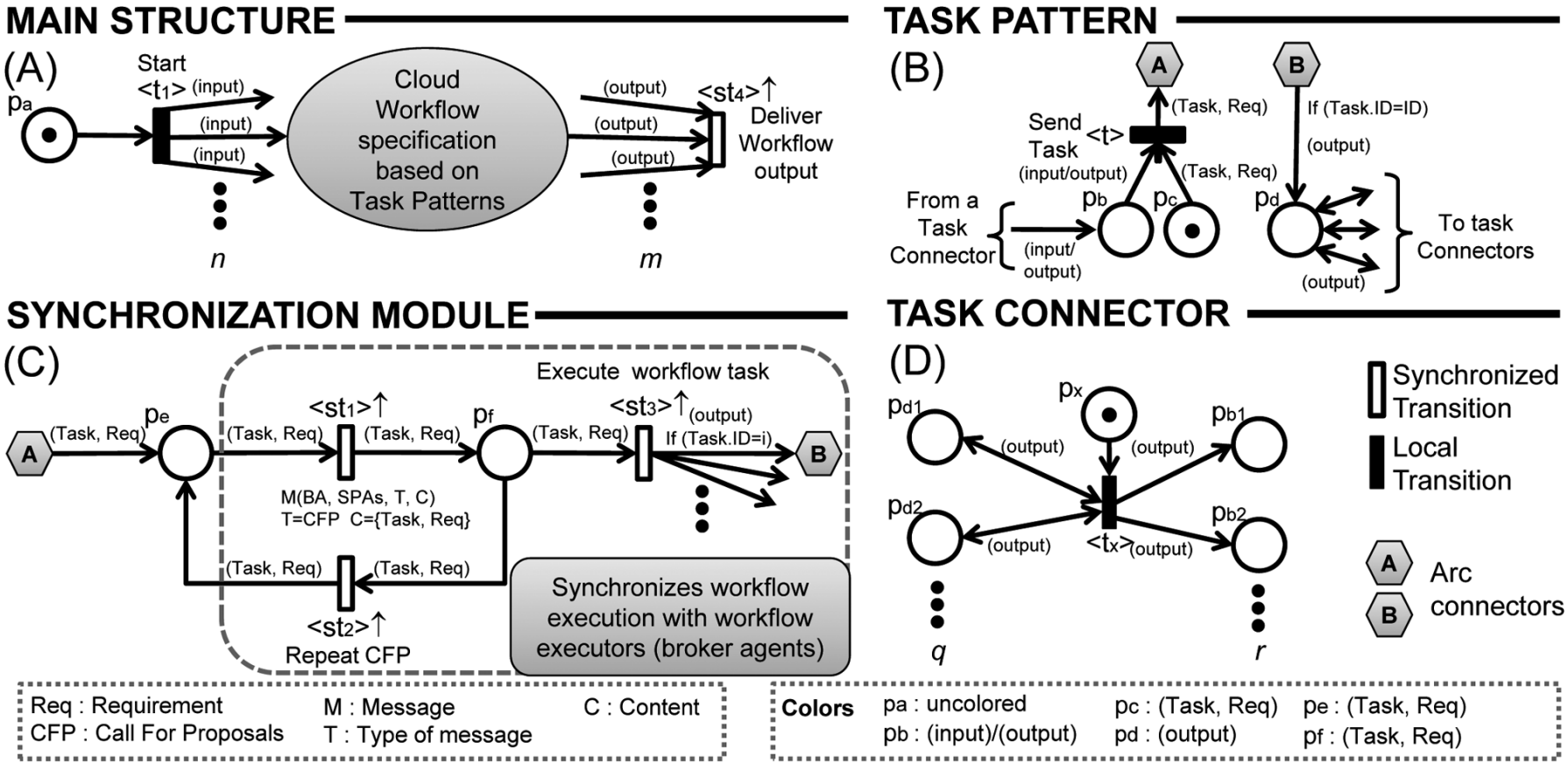
Múltiples tipos de redes de Petri

- Redes de Petri coloreadas
- Redes de Petri estocásticas
- Redes de Petri temporizadas
- Redes de Petri dentro de redes de Petri

Redes de Petri dentro de redes de Petri

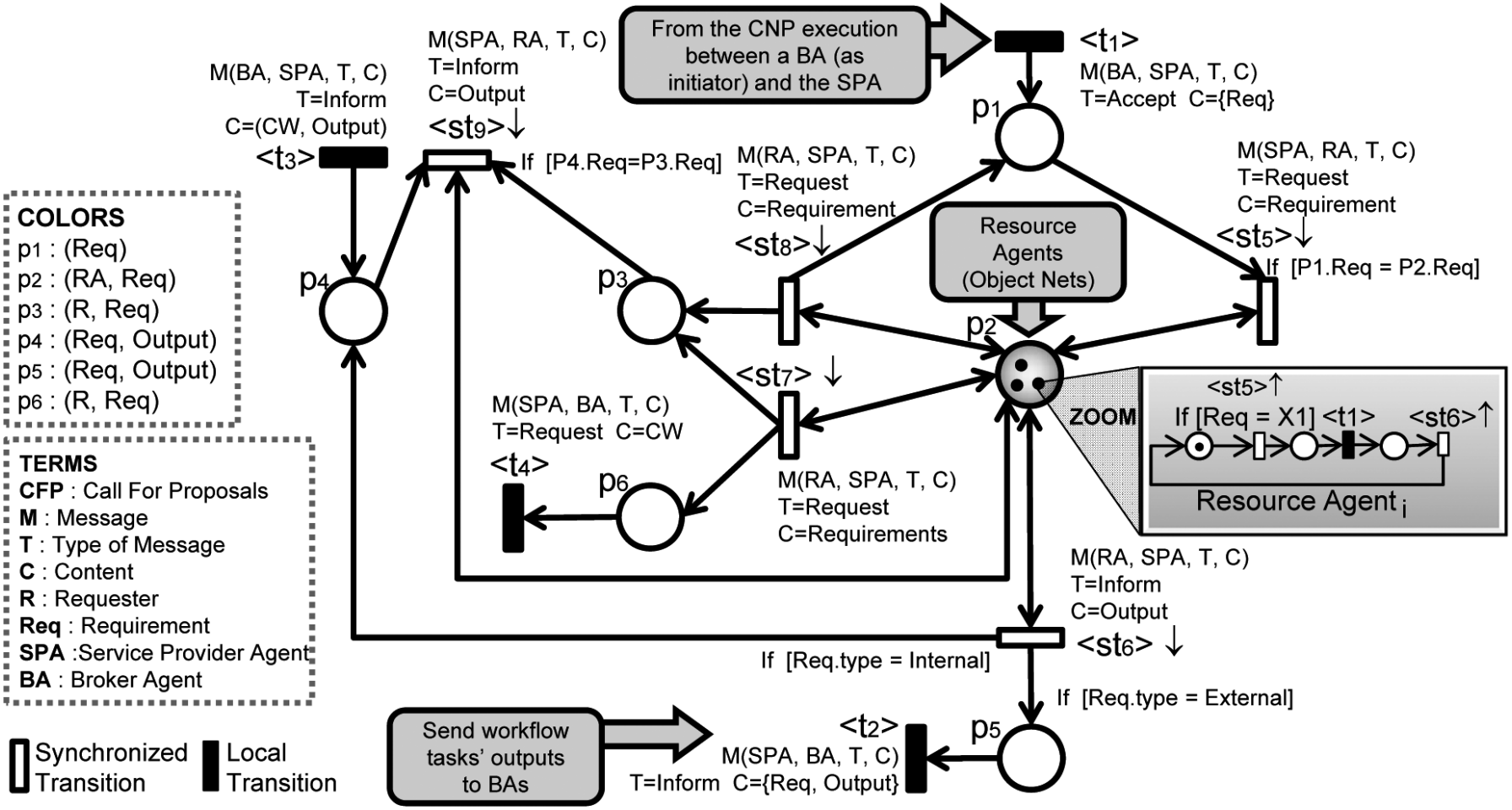


Redes de Petri dentro de redes de Petri



[Gutiérrez, 2012]

Redes de Petri dentro de redes de Petri



[Gutiérrez, 2012]

Material adicional

- **Advanced Topics in Programming Languages: Concurrency/message passing in Newsqueak by Rob Pike**
  - http://www.youtube.com/watch?v=hB05UFqOtFA
- **A tour of GO (a Google's concurrent programming language)**
  - http://tour.golang.org/#1
- **Google I/O 2012 - Go Concurrency Patterns by Rob Pike**
  - http://www.youtube.com/watch?v=f6kdp27TYZs
- **Google I/O 2013 - Advanced Go Concurrency Patterns by Sameer Ajmani**
  - https://www.youtube.com/watch?v=QDDwwePbDtw