Sistemas Operativos

Concurrencia y sincronización

Marcelo Arroyo

Dpto. de Computación - FCEFQyN - Universidad Nacional de Río Cuarto

2014

Concurrencia

Procesos secuenciales

- *Determinísticos*: Una única traza de ejecución. Siempre arrojan la misma salida ante una entrada dada
- *Corrección*: Razonamiento mas simple

Concurrencia

Procesos secuenciales

- *Determinísticos*: Una única traza de ejecución. Siempre arrojan la misma salida ante una entrada dada
- *Corrección*: Razonamiento mas simple

Procesos concurrentes

- *No determinísticos*: Varias trazas de ejecución posibles.
- *Corrección*: Razonamiento mucho mas complejo
- *No determinismo observable*: Es posible obtener diferentes salidas ante la misma entrada en diferentes instancias de ejecución posibles

Concurrencia

Procesos independientes

No realizan tareas cooperativas con otros procesos. Usa sus recursos en forma exclusiva.

Concurrencia

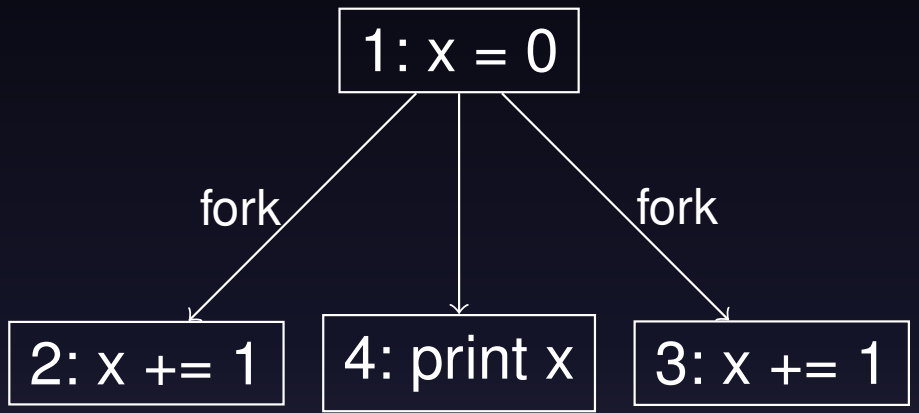
Procesos independientes

No realizan tareas cooperativas con otros procesos. Usa sus recursos en forma exclusiva.

Procesos cooperativos

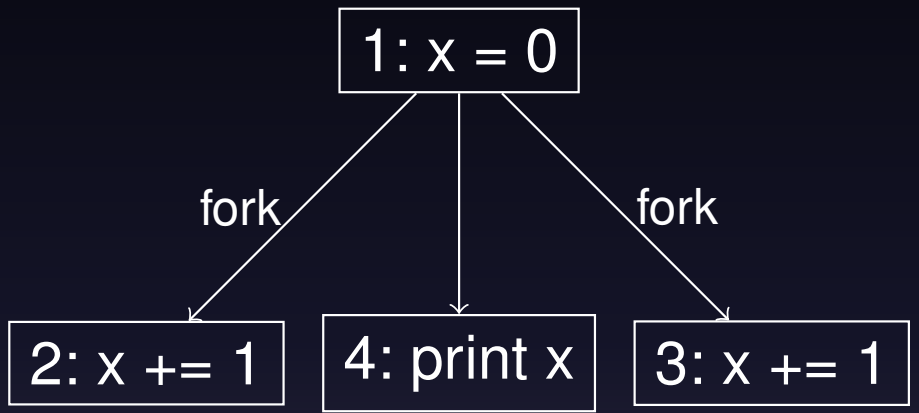
- Comparten recursos (ej: archivos)
- Hilos en el mismo espacio de memoria (threads)
- Utilizan algún mecanismo de comunicación entre procesos (IPC):
  - Archivos (compartidos)
  - Memoria compartida, pipes, fifos, sockets, ...
  - Mensajes (sistemas distribuidos)

Concurrencia (mem. compartida)



Salida	**Traza**
x=0	$< 1, 4, 2, 3 >$
x=0	$< 1, 4, 3, 2 >$
x=1	$< 1, 2, 4, 3 >$
x=1	$< 1, 3, 4, 2 >$
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$

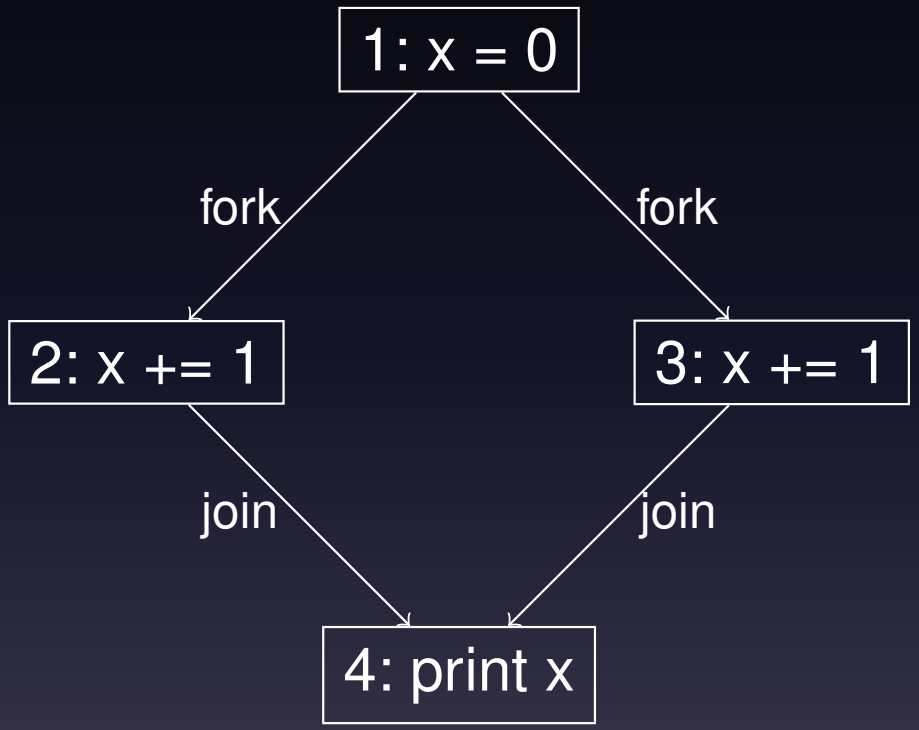
Concurrencia (mem. compartida)



Salida	**Traza**
x=0	$< 1, 4, 2, 3 >$
x=0	$< 1, 4, 3, 2 >$
x=1	$< 1, 2, 4, 3 >$
x=1	$< 1, 3, 4, 2 >$
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$

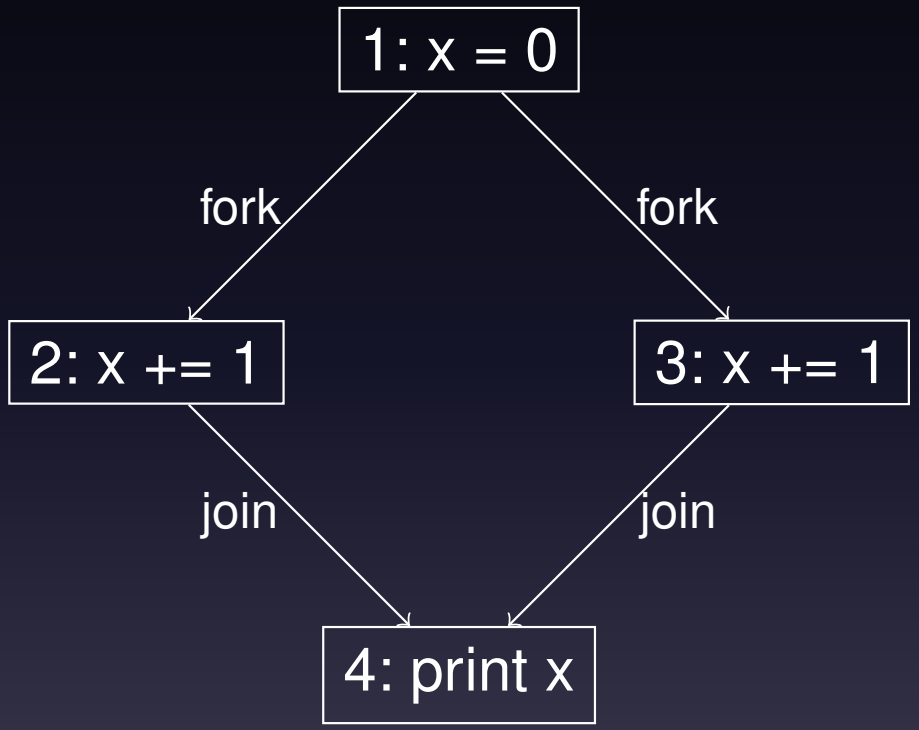
La traza elejida depende del planificador (scheduler) de procesos o threads

Concurrencia (mem. compartida)



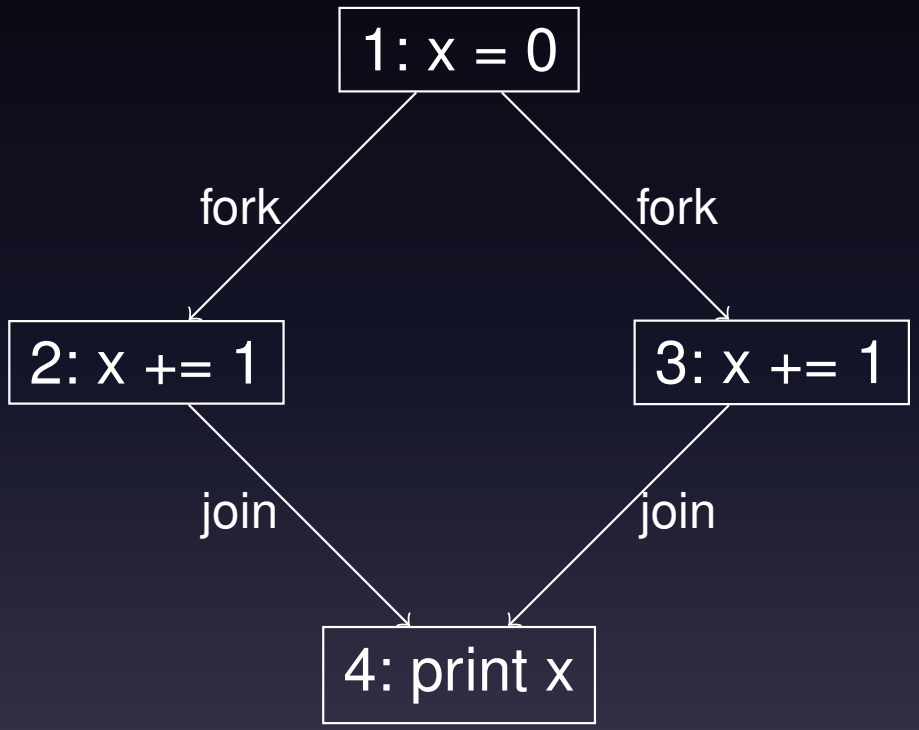
Salida	**Traza**
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$

Concurrencia (mem. compartida)



Salida	**Traza**
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$
x=1	¿Por qué?

Concurrencia (mem. compartida)



Salida	**Traza**
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$
x=1	¿Por qué?

Operación += no atómica

x += 1

load x
add 1
store x

Sincronización

Objetivo: No determinismo no observable

Lograr salidas determinísticas aún en un ambiente no determinístico o concurrente

Región crítica

Secuencia de código que utiliza recursos compartidos

Sincronización

Objetivo: No determinismo no observable

Lograr salidas determinísticas aún en un ambiente no determinístico o concurrente

Región crítica

Secuencia de código que utiliza recursos compartidos

Solución: restringir interleaving en regiones críticas (exclusión mutua)

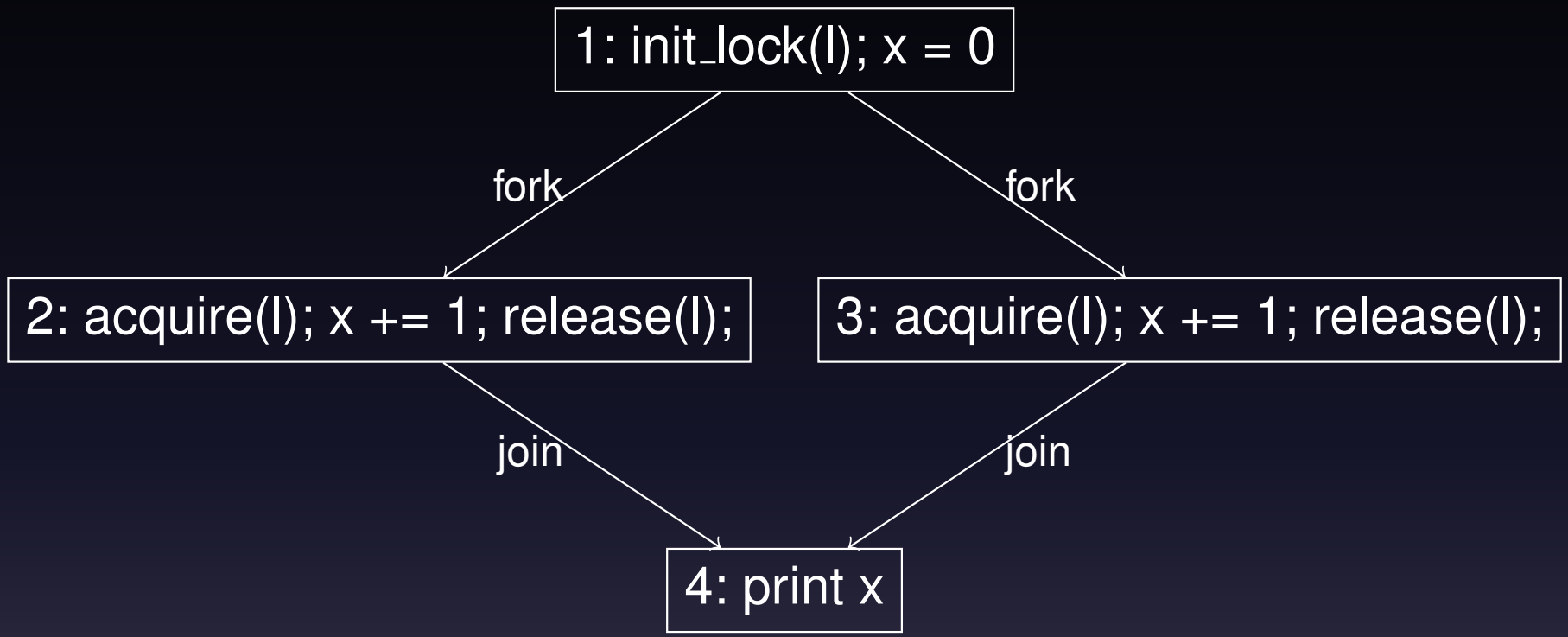
Mecanismos (en memoria compartida):

- **Locks**: *acquire(lock)* y *release(lock)*
- **Semáforos**: generalización de locks

Mensajes (primitivas *send(to,msg)* y *receive(from)*):

- **Comunicación sincrónica o asincrónica**

Uso de locks



Salida	**Traza**
x=2	$< 1, 2, 3, 4 >$
x=2	$< 1, 3, 2, 4 >$

Implementación de locks

Spin lock (espera ocupada)

```
1  void acquire(int lock)
2  {
3     while ( lock ) ;
4     lock = 1;
5  }
```

```
1  void release(int lock)
2  {
3     lock = 0;
4  }
```

Implementación de locks

Spin lock (espera ocupada)

```
1  void acquire(int lock)
2  {
3     while ( lock ) ;
4     lock = 1;
5  }
```

```
1  void release(int lock)
2  {
3     lock = 0;
4  }
```

Problema: dos procesos pueden conseguir un lock *(¿Cómo?)*

Solución

El lock en sí mismo es un recurso compartido.
acquire(lock) y *release(lock)* deben ser atómicas (indivisibles).
Usar instrucciones atómicas (ej: Intel *xchg*) o deshabilitar interrupciones.

Otras primitivas (mem. compartida)

Variables de condición

- **Lock** con una cola de procesos asociada
- *wait(c)*: espera por un evento (sleep)
- *signal(c)*: dispara un evento (wakeup)

Otras primitivas (mem. compartida)

Variables de condición

- **Lock** con una cola de procesos asociada
- *wait(c)*: espera por un evento (sleep)
- *signal(c)*: dispara un evento (wakeup)

Semáforos

- *typedef struct { int counter; task_queue q; } semaphore;*
- *init(semaphore s, int n)*: (counter=n, q=empty)
- *p(semaphore s)*: if counter==0 *sleep(q)*; - -counter
- *v(semaphore s)*: if ++counter *wakeup_one(q)*

No realiza espera ocupada.
Debe garantizarse la atomicidad de las operaciones.

Ej: productor-consumidor

```
int queue[N], first,
last;
lock ql;
semaphore empty,
full;
...
```

```
void main()
{
    init(empty,0);
    init(full,N);
    new_thread(prod);
    new_thread(cons);
}
```

```
void enqueue(int i)
{
    acquire(ql);
    queue[...] = i;
    ...
    release(ql);
}
```

```
void prod() {
  while (true) {
      int item=calc();
      p(full);
      enqueue(item);
      v(empty);
  }
}
```

```
int dequeue()
{
    acquire(ql);
    item = queue[...]
    release(ql);
    return item;
}
```

```
void cons() {
  while (true) {
      p(empty);
      item = dequeue();
      v(full);
      process(item);
  }
}
```

Mensajes

Características

- Memoria no compartida
- Permite el desarrollo de *sistemas distribuidos*
- Los procesos pueden correr en diferentes procesadores

Primitivas

- *send(dest,msg)*: envía *msg* al proceso *dest*
- *receive(from)*: retorna el *msg* emviado por el proceso *from*

Comunicación

- *Sincrónica*: se espera a que el par ejecute la operación complementaria
- *Asincrónica*: no se espera
- *Híbrida*: por ej: *send()* asincrónico y *receive()* sincrónico

Problemas de la concurrencia

Locks y modularidad

- Locks protegen invariantes
- Locks interfieren con la *modularidad*
- Locks recursivos: el *invocado* puede adquirir un lock ya adquirido por su invocante (no protegen invariantes)

Deadlock

Condiciones que posibilitan deadlocks

- Acceso exclusivo en una región crítica *(exclusión mutua)*
- No quita de recursos *(no preemption)*
- Mientras se espera por un recurso se retienen otros
- Espera circular

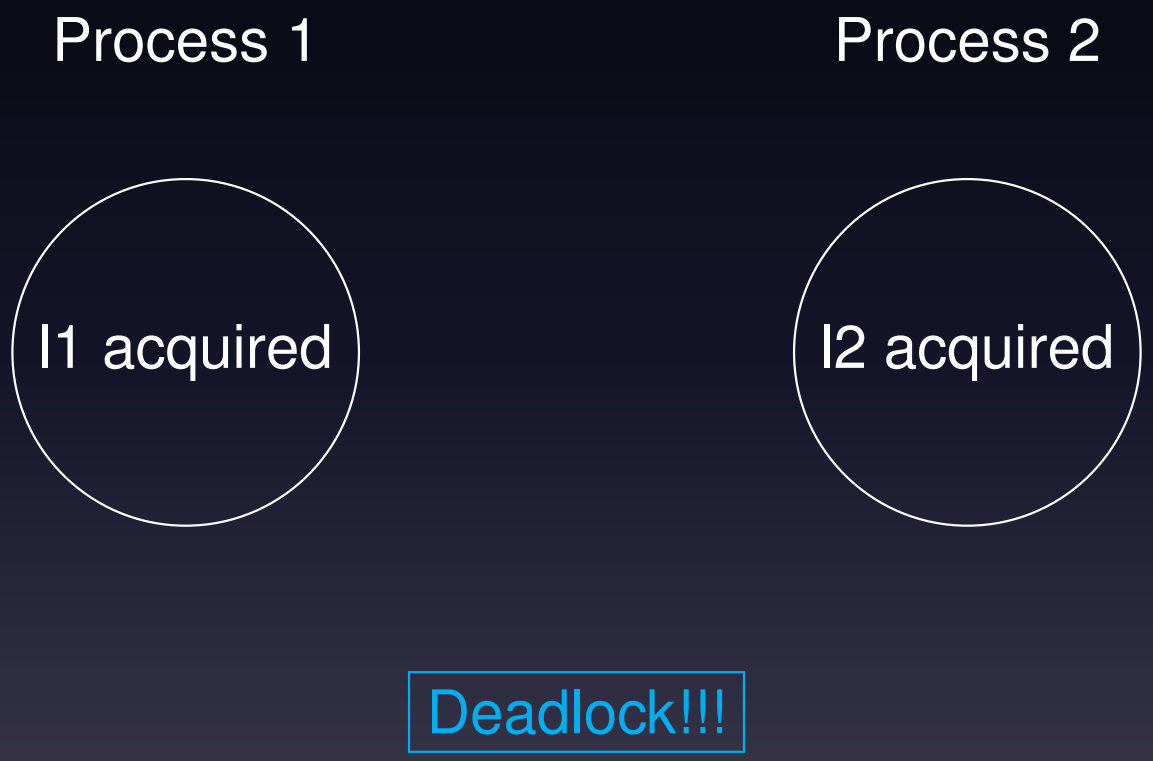
Ejemplo de deadlock (con locks)

Process 1

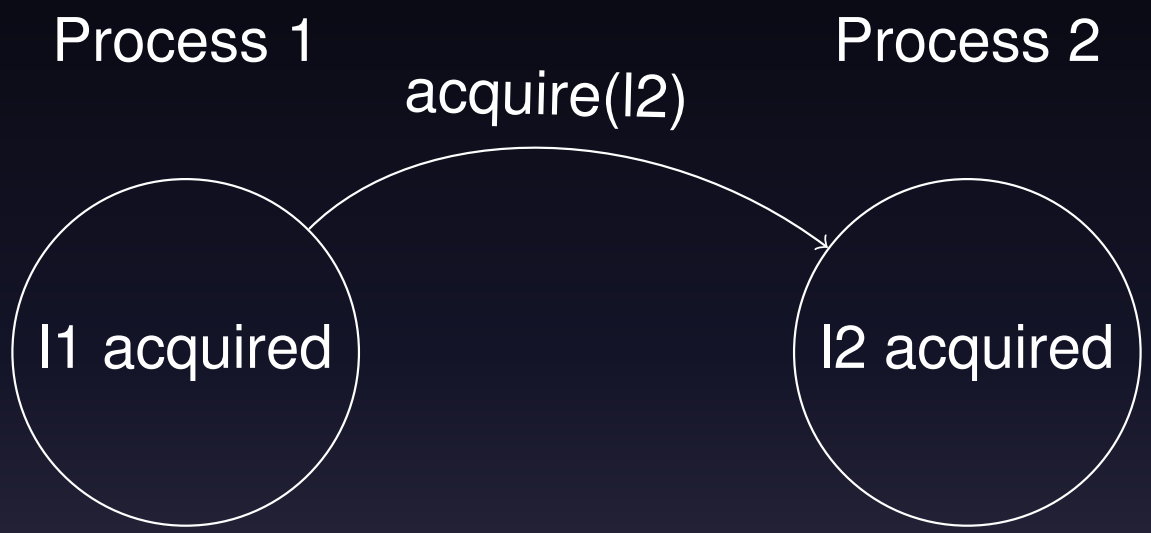
Process 2

Deadlock!!!

Ejemplo de deadlock (con locks)

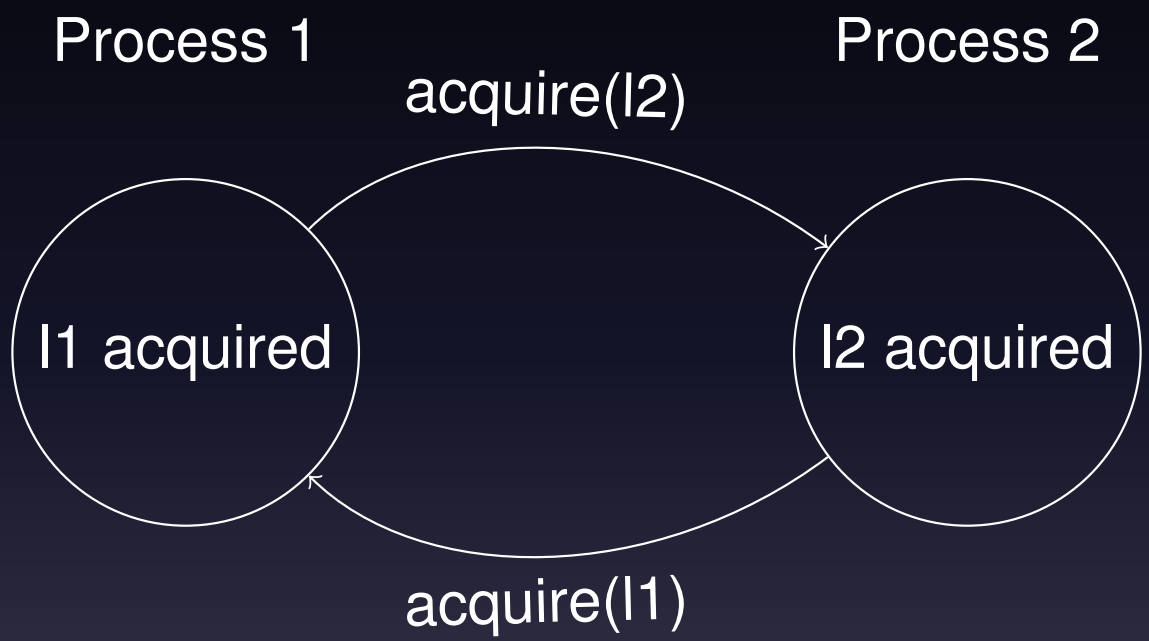


Ejemplo de deadlock (con locks)

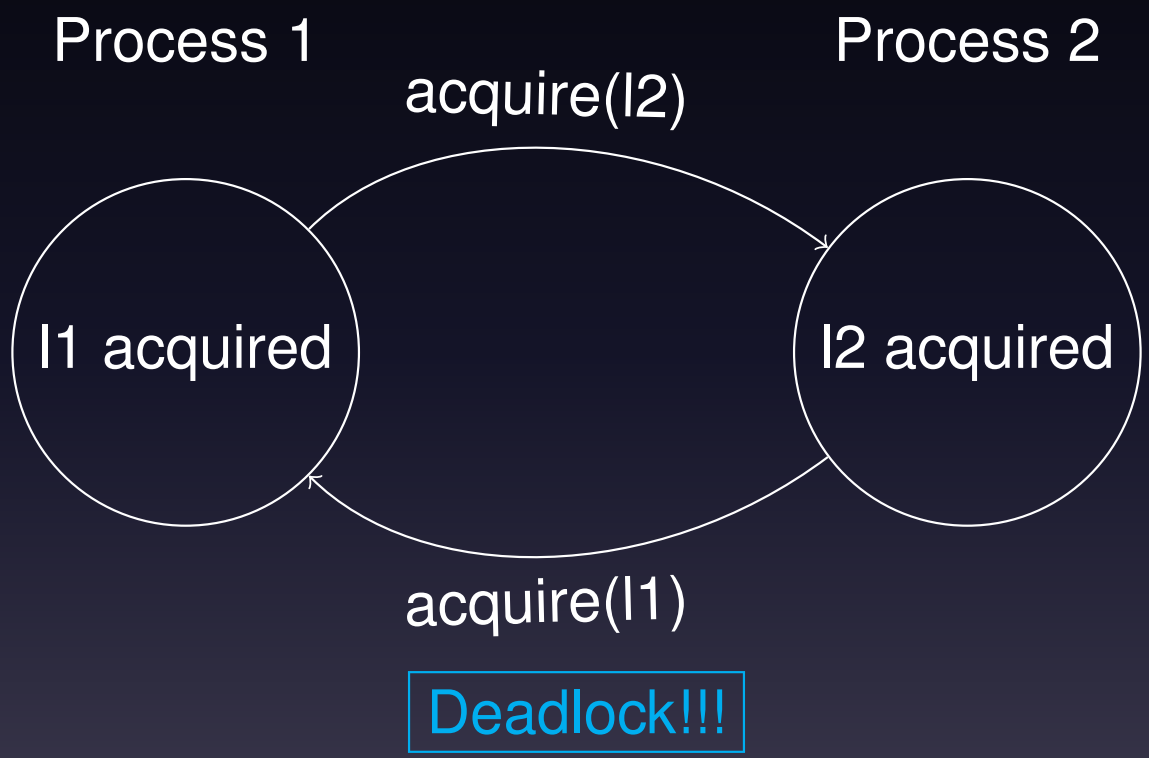


Deadlock!!!

Ejemplo de deadlock (con locks)



Ejemplo de deadlock (con locks)



Otros mecanismos

Mecanismos de alto nivel

- *Monitores*: un monitor encapsula operaciones que se ejecutan con exclusión mutua
- *Variables de flujo de datos* (data flow). Ej: Erlang variables
- *Memoria compartida distribuida*: ej: Linda tuples space
- *Colas de mensajes* (mailboxes)
- *Llamadas remotas a procedimientos (RPC)*: Sun RPC, Java RMI, Web services (XML-RPC, SOAP, . . .)
- *Código móvil*: migración de objetos, inyección de código, . . .