Semáforos

Programación Concurrente 2015
Universidad Nacional de Quilmes

Requerimientos de la exclusión mutua

1. **Mutex:** En cualquier momento hay como máximo un proceso en la región crítica.
2. **Ausencia de deadlocks y livelocks:** Si varios procesos intentan entrar a la sección crítica alguno lo logrará.
3. **Garantía de entrada:** Un proceso intentando entrar a su sección crítica tarde o temprano lo logrará.

Dekker y Peterson

- **Mutex:** Sí
- **Ausencia dead/live-locks:** Sí
- **Garantía de entrada:** Sí

Sólo sirven para dos procesos.

La demostración formal excede el alcance de este curso.

Algoritmo de Bakery

```
global boolean[] entrando = replicate(n,false);
global int[] numero] = replicate(n,0);

thread {
  id = 0;
  // seccion no critica
  entrando[id] = true;
  numero[id] = 1 + maximum(numero);
  entrando[id] = false;
  for (j : range(0,n)) {
    while (entrando[j]);
    while (numero[j] != 0 &&
          (numero[j] < numero[id] ||
          (numero[j] == numero[id] && j < id)));
  }

  // SECCION CRITICA

  numero[id] = 0;
  // seccion no critica
}
```

Este algoritmo resuelve el problema para *n threads*.

Test and Set

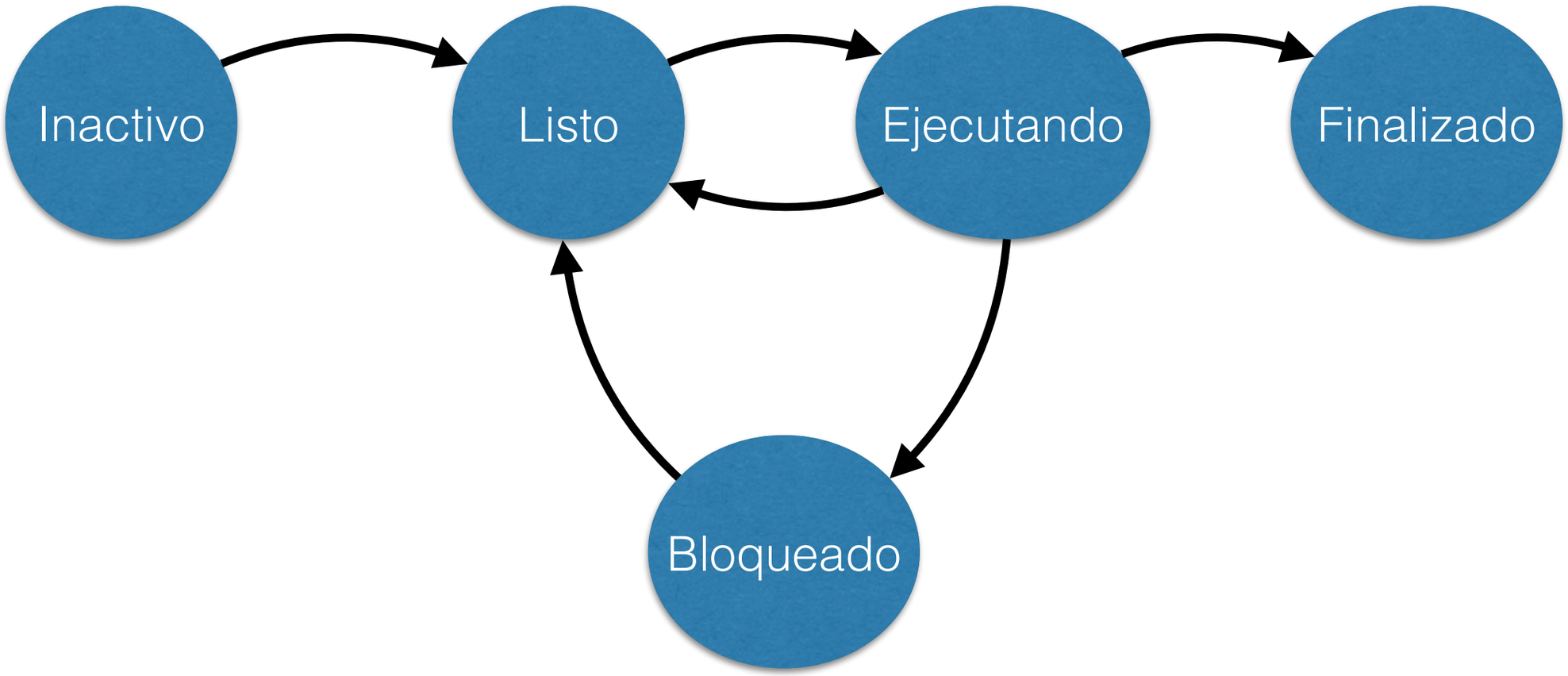
```
atomic boolean TestAndSet(ref) {
  result = ref.value;
  ref.value = true;
  return result;
}

global Object shared = record();
shared.value = false;
```

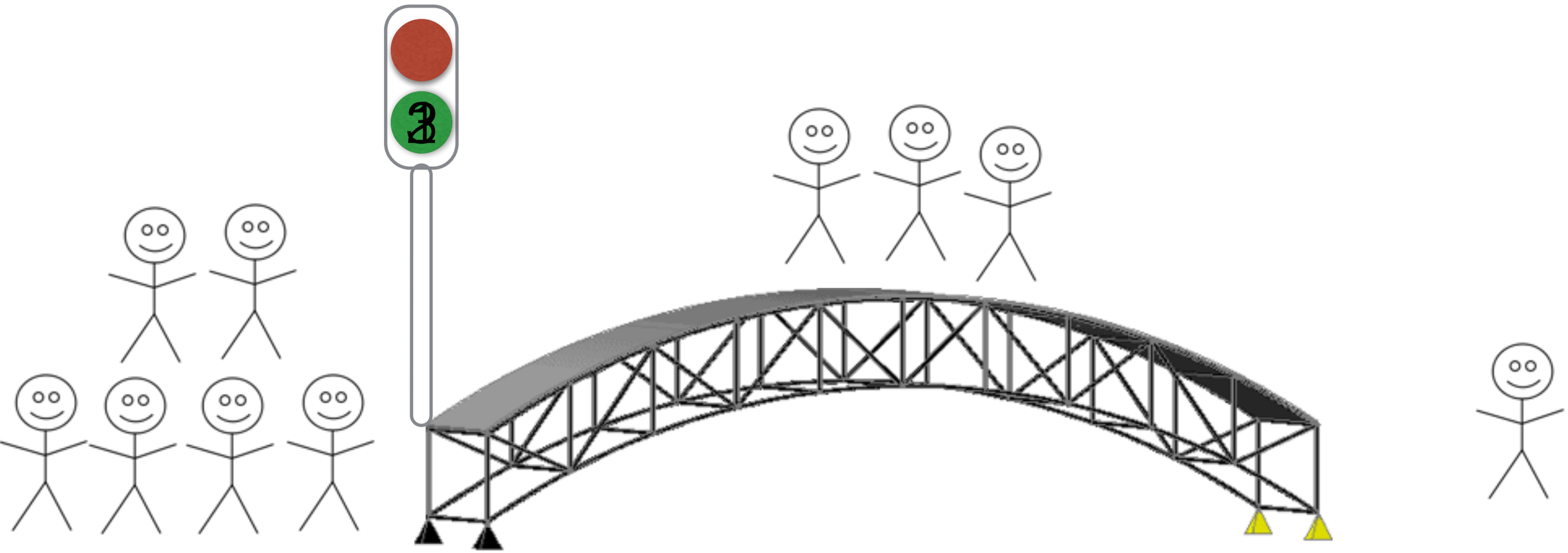
```
thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

```
thread {
  // seccion no critica
  while( TestAndSet(shared) );
  // SECCION CRITICA
  shared.value = false;
  // seccion no critica
}
```

Estados de un proceso/thread



Semáforos



Qué es un Semáforo?

- Es un tipo abstracto de datos (TAD) con 2 operaciones posibles:
  - `acquire()`
  - **`release()`**
- Internamente puede estar representado usando:
  - Un entero: La cantidad de permisos
  - Un conjunto: Los procesos esperando el semáforo

Acquire()

- `Acquire` consume un permiso o espera si no hay un permiso disponible

```
atomic acquire() {
 currentProcess = Process.getCurrentProcess();
 if (permisos>0) {
  permisos--;
 } else {
  procesos.add(currentProcess);
  currentProcess.state=BLOCKED;
 }
}
```

Release()

- `Release` libera un permiso (y despierta un proceso si hay alguno esperando)

```
atomic release() {
 if (procesos.isEmpty()) {
  permisos++;
 } else {
  wakingProcess = procesos.removeAny();
  wakingProcess.state=READY;
 }
}
```

Mutex

- Llamamos `Mutex` a un semáforo que sólo admite 1 permiso.

```
atomic release() {
  if (permisos==1) {
   ERROR!
  } else if (procesos.isEmpty()) {
   permisos++;
  } else {
   wakingProcess = procesos.removeAny();
   wakingProcess.state=READY;
  }
}
```

Exclusión Mutua

```
// Crea un mutex

global Semaphore mutex = new Semaphore(1);

// Adquiere el mutex (o se bloquea si no hay)

mutex.acquire();

// Libera el mutex

mutex.release();
```

Exclusión Mutua (usando semáforos)

```
global Semaphore mutex = new Semaphore(1);

thread {                              thread {
  //…Sección no-crítica                 //…Sección no-crítica
  mutex.acquire();                      mutex.acquire();
  //…Sección crítica                    //…Sección crítica
  mutex.release();                      mutex.release();
  //…Sección no-crítica                 //…Sección no-crítica
}                                     }
```

Exclusión Mutua (usando semáforos)

```
global Semaphore mutex = new Semaphore(1);

thread {
  //…Sección no-crítica
  mutex.acquire();
  //…Sección crítica
  mutex.release();
  //…Sección no-crítica
}
```

```
thread {
  //…Sección no-crítica
  mutex.acquire();
  //…Sección crítica
  mutex.release();
  //…Sección no-crítica
}
```

Mutex (Máximo 1 proceso en la región crítica): ?

Ausencia de deadlocks/livelocks: ?

Garantía de entrada (todo proceso que quiere acceder a la región crítica tarde o temprano lo logra): ?

Exclusión Mutua (usando semáforos)

```
global Semaphore mutex = new Semaphore(1);

thread {
  //…Sección no-crítica
  mutex.acquire();
  //…Sección crítica
  mutex.release();
  //…Sección no-crítica
}
```

```
thread {
  //…Sección no-crítica
  mutex.acquire();
  //…Sección crítica
  mutex.release();
  //…Sección no-crítica
}
```

Mutex (Máximo 1 proceso en la región crítica): SI

Ausencia de deadlocks/livelocks: SI

Garantía de entrada (todo proceso que quiere acceder a la región crítica tarde o temprano lo logra): SI

Invariantes de un Semáforo

- `permisos>=0`
- `permisos` = k- #acquires (terminados) + #releases

donde k es la cantidad de permisos iniciales en el semáforo.

Nota: se considera un acquire() como “terminado” si se le concedió un permiso.

Invariantes Exclusión Mutua (usando semáforos)

- #ProcesosEnSeccionCritica + `permisos` = 1

- #ProcesosEnSeccionCritica = #acquires - #releases

Esto garantiza:

- Exclusión Mutua (#ProcesosEnSeccionCritica<=1)

- Ausencia de deadlock (no sucede que permisos=0 y #ProcesosEnSeccionCritica=0)

- No hay *starvation* entre dos procesos

El Problema del Molinete

```
global int contador=0;
global Semaphore mutex = new Semaphore(1);

molinete() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
}
repeat (N)
  thread molinete();
```

Semáforos Fuertes

- En la clase `Semaphore` de Java se puede crear un semáforo fuerte seteando `fair==true`.

Semaphore

```
public Semaphore(int permits,
         boolean fair)
```

Creates a `Semaphore` with the given number of permits and the given fairness setting.

Parameters:

`permits` - the initial number of permits available. This value may be negative, in which case releases must occur before any acquires will be granted.

`fair` - `true` if this semaphore will guarantee first-in first-out granting of permits under contention, else `false`

Qué sucede al ejecutar este programa concurrente?

```
global int contador=0;
global Semaphore mutex = new Semaphore(1);

molinete() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
}
repeat (N)
  thread molinete();
print("Total = " + contador)
```

Qué sucede al ejecutar este programa concurrente?

```
global int contador=0;
global Semaphore mutex = new Semaphore(1);
global Semaphore finish = new Semaphore(0);

molinete() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
  finish.release();
}
repeat (N)
  thread molinete();

repeat (N)
  finish.acquire();
print("Total = " + contador)
```

Filósofos Comensales

- Cada filósofo come y piensa alternadamente (todo el tiempo)
- Sólo comen cuando tienen 2 tenedores
- Sólo pueden tomar el tenedor a izquierda y derecha de su posición

Filósofos Comensales

```
Filosofo(id) {
  while (true) {
    //… pensar
    //…tomar tenedores
    //…comer
    //…dejar tenedores
  }
}
```

- Mutex: Cada tenedor es sostenido a lo sumo por un filósofo en todo momento
- Sincronización: un filósofo puede comer cuando tiene a lo sumo dos tenedores
- Deadlock
- Livelock
- Starvation

Intento #1

```
global Semaphore[] tenedores = [1,…,1]; // N
Filosofo(id) {
 izq = id;
 der = (id+1)%N;
  while (true) {
    //… pensar
    tenedores[izq].acquire();
    tenedores[der].acquire();
    //…comer
    tenedores[izq].release();
    tenedores[der].release();
  }
}
```

Intento #1

```
global Semaphore[] tenedores = [1,…,1]; // N
Filosofo(id) {
 izq = id;
 der = (id+1)%N;
  while (true) {
    //… pensar
    tenedores[izq].acquire();
    tenedores[der].acquire();
    //…comer
    tenedores[izq].release();
    tenedores[der].release();
  }
}
```

Deadlock: Se produce si todos toman el tenedor de la izquierda al mismo tiempo!

Uso semáforo general extra

```
global Semaphore[] tenedores = [1,…,1]; // N
global Semaphore sillas = new Semaphore(N-1);

Filosofo(id) {
 izq = id;
 der = (id+1)%N;
  while (true) {
    //… pensar
    sillas.acquire();
    tenedores[izq].acquire();
    tenedores[der].acquire();
    //…comer
    tenedores[izq].release();
    tenedores[der].release();
    sillas.release();
  }
}
```

Ruptura de Simetrías

```
global Semaphore[] tenedores = [1,…,1]; // N
Filosofo(id) {
 if (id==0) {
    izq = 1; der = 0;
  } else {
    izq = id;
    der = (id+1)%N;
  }
  while (true) {
    //… pensar
    tenedores[izq].acquire();
    tenedores[der].acquire();
    //…comer
    tenedores[izq].release();
    tenedores[der].release();
  }
}
```