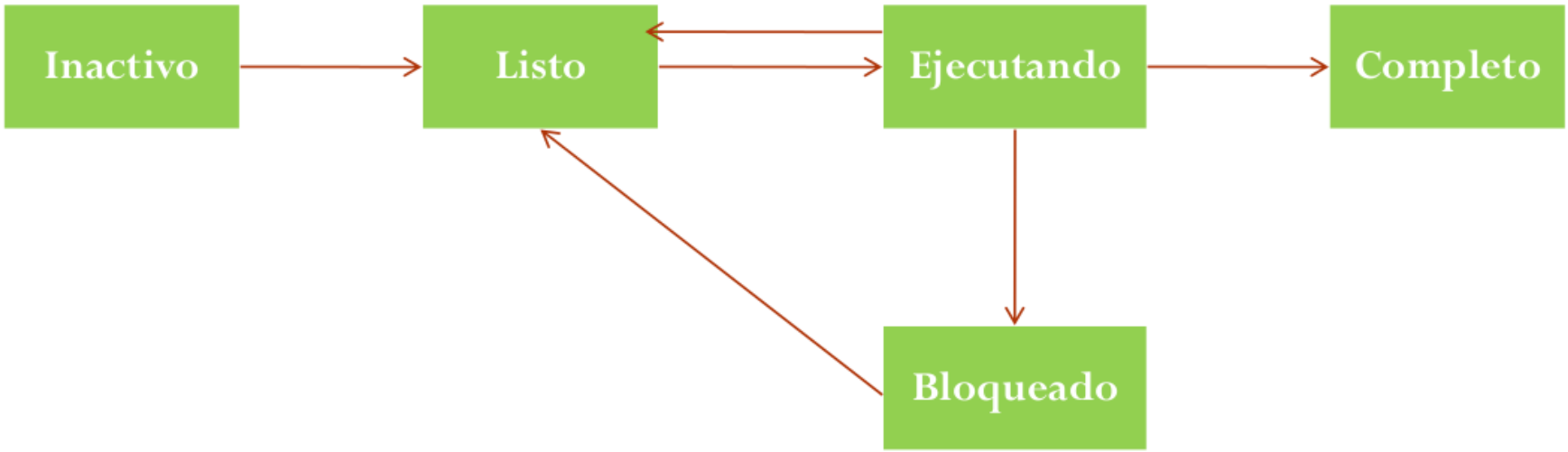
Semáforos

Programación concurrente

Hasta ahora vimos

- Características de la concurrencia
  - Ventajas
  - Problemas
  - Exclusión mutua
  - Operaciones atómicas

Estados de un proceso



Ejemplo: Puente

Semáforo

Un semáforo es un tipo abstracto de datos con las operaciones:

- `acquire`
- `release`

Semáforo

Un semáforo es un tipo abstracto de datos con las operaciones:

- `acquire`
- `release`

Una posible representación interna consta de:

- `entero: permisos`
- `conjunto: procesos`

Acquire

`Acquire` consume un permiso o espera si no hay uno disponible.

Acquire

Acquire consume un permiso o espera si no hay uno disponible.

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

Release

`Release` libera un permiso (despierta a un thread bloqueado)

Release

Release libera un permiso (despierta a un thread bloqueado)

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

Invariantes de un semáforo

Invariantes de un semáforo

- $\texttt{permisos} \geq 0$

Invariantes de un semáforo

- $\texttt{permisos} \geq 0$
- $\texttt{permisos} = \texttt{initial} + \#\texttt{releases} - \#\texttt{acquires}$

Invariantes de un semáforo

- $\texttt{permisos} \geq 0$
- $\texttt{permisos} = \texttt{initial} + \#\texttt{releases} - \#\texttt{acquires}$

Nota: Se considera que un proceso bloqueado no realizó la operación `acquire`.

Mutex

Llamaremos `mutex` a un semáforo que sólo admite 0 o 1 permisos.

```
atomic release() {
  assert(permisos <= 1); // invariante de mutex
  if (procesos.empty()) {
    permisos++;
  } else {
    wakingThread = procesos.removeAny();
    wakingThread.state = READY;
  }
}
```

Mutex

Llamaremos `mutex` a un semáforo que sólo admite 0 o 1 permisos.

```
atomic release() {
  assert(permisos <= 1); // invariante de mutex
  if (procesos.empty()) {
    permisos++;
  } else {
    wakingThread = procesos.removeAny();
    wakingThread.state = READY;
  }
}
```

(Usaremos un `Semaphore` y nos aseguraremos de nunca liberar permisos inválidos.)

Exclusión mutua usando `mutex`

Exclusión mutua usando mutex

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

thread {                          thread {
  // seccion no critica               // seccion no critica
  mutex.acquire();                    mutex.acquire();
  // seccion critica                  // seccion critica
  mutex.release();                    mutex.release();
  // seccion no critica               // seccion no critica
}                                 }
```

Exclusión mutua usando `mutex`

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

thread {                          thread {
  // seccion no critica             // seccion no critica
  mutex.acquire();                  mutex.acquire();
  // seccion critica                // seccion critica
  mutex.release();                  mutex.release();
  // seccion no critica             // seccion no critica
}                                 }
```

Esta solución no usa busy waiting ya que un proceso bloqueado en el acquire pasa al estado bloqueado y no vuelve al estado listo hasta tanto se le de permiso.

Invariantes de la exclusión mutua con semáforos

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$
- $\texttt{\#criticalSections} = \texttt{\#acquires} - \texttt{\#releases}$

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$
- $\texttt{\#criticalSections} = \texttt{\#acquires} - \texttt{\#releases}$

Esto garantiza:

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$
- $\texttt{\#criticalSections} = \texttt{\#acquires} - \texttt{\#releases}$

Esto garantiza:

- Mutex: $\texttt{\#criticalSections} \leq 1$

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$
- $\texttt{\#criticalSections} = \texttt{\#acquires} - \texttt{\#releases}$

Esto garantiza:

- Mutex: $\texttt{\#criticalSections} \leq 1$
- Garantía de entrada: no sucede que $\texttt{permisos} = 0$ y $\texttt{\#criticalSections} = 0$

Invariantes de la exclusión mutua con semáforos

- $\texttt{\#criticalSections} + \textit{permisos} = 1$
- $\texttt{\#criticalSections} = \texttt{\#acquires} - \texttt{\#releases}$

Esto garantiza:

- Mutex: $\texttt{\#criticalSections} \leq 1$
- Garantía de entrada: no sucede que $\texttt{permisos} = 0$ y $\texttt{\#criticalSections} = 0$
- No hay *starvation* entre dos procesos

El problema del comensal

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

comensal() {
  while (true) {
    mutex.acquire();
    comer();
    mutex.release();
  }
}

repeat (N)
  thread comensal();
```

El problema del comensal

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

comensal() {
  while (true) {
    mutex.acquire();
    comer();
    mutex.release();
  }
}

repeat (N)
  thread comensal();
```

- Si el semáforo usa un conjunto (débil) este programa tiene starvation.

El problema del comensal

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

comensal() {
  while (true) {
    mutex.acquire();
    comer();
    mutex.release();
  }
}

repeat (N)
  thread comensal();
```

- Si el semáforo usa un conjunto (débil) este programa tiene starvation.
- Si el semáforo usa una cola (fuerte) este programa no tiene starvation.

Semáforos fuertes

Cuando tenemos más de dos procesos existe posibilidad de *starvation*.

Esto puede solucionarse cambiando el conjunto por una cola.

Semáforos fuertes

Cuando tenemos más de dos procesos existe posibilidad de *starvation*.

Esto puede solucionarse cambiando el conjunto por una cola.

En java se indica en la construcción

```
/** Creates a Semaphore with the given number of permits
    and the given fairness setting. */
Semaphore(int permits, boolean fair)
```

Mutex para evitar pérdida de sumas

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

incrementador() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
}

repeat (N)
  thread incrementador();
```

Sincronización de procesos

¿Cómo imprimimos el total del contador de N incrementadores?

Sincronización de procesos

¿Cómo imprimimos el total del contador de N incrementadores?

```
repeat (N)
  thread incrementador();
print("Total = " + contador);
```

Sincronización de procesos

¿Cómo imprimimos el total del contador de N incrementadores?

```
repeat (N)
  thread incrementador();
print("Total = " + contador);
```

¿Qué sucede al ejecutar este código?

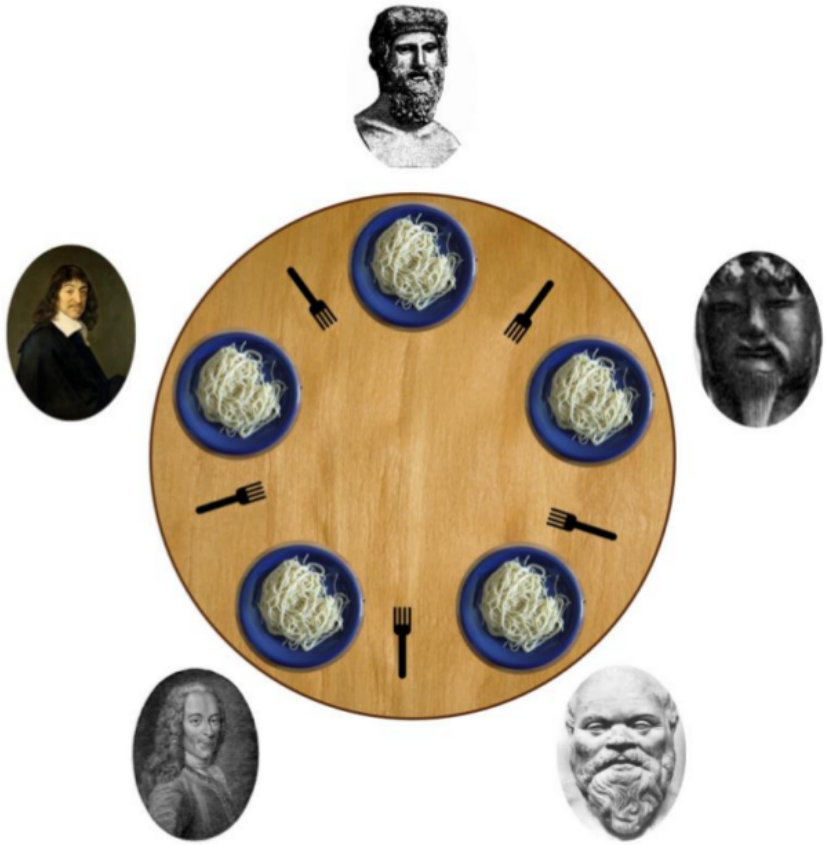
Sincronización de procesos (incrementadores)

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

incrementador() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
  finish.release();
}

repeat (N)
  thread incrementador();
```

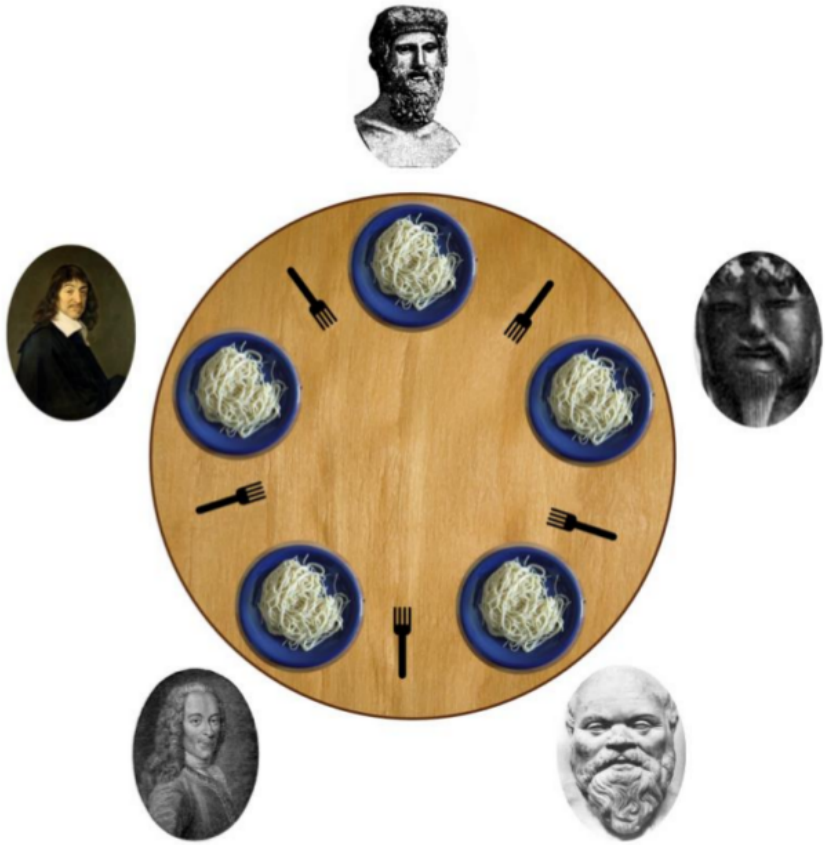
Sincronización de procesos (incrementadores)

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

incrementador() {
  repeat (100) {
    mutex.acquire();
    contador++;
    mutex.release();
  }
  finish.release();
}

repeat (N)
  thread incrementador();

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

Filósofos comensales

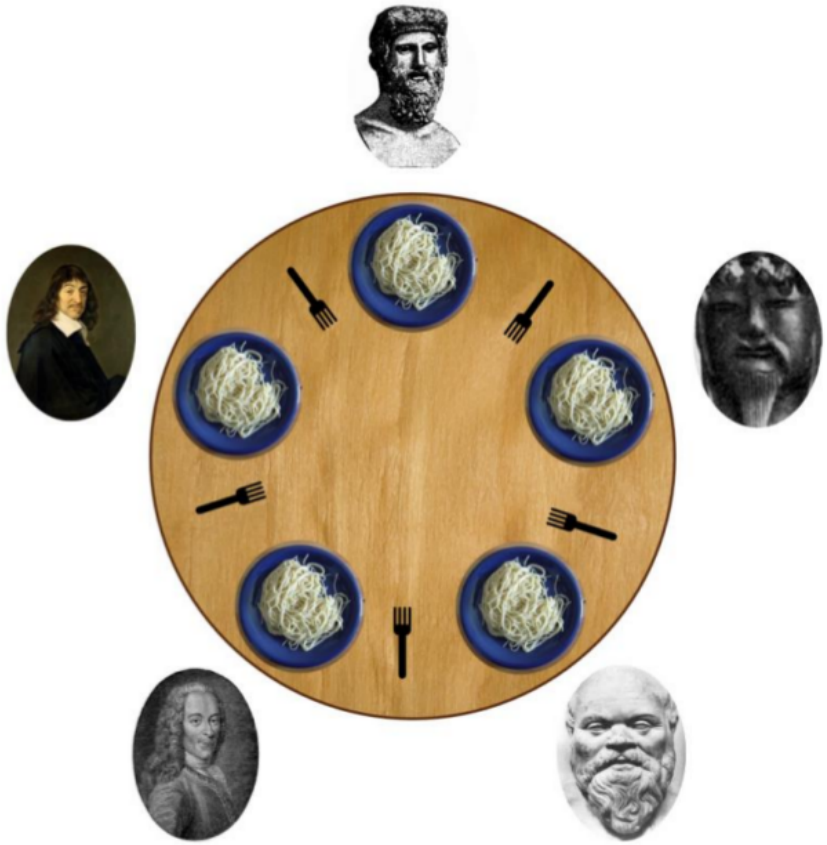


Filósofos comensales



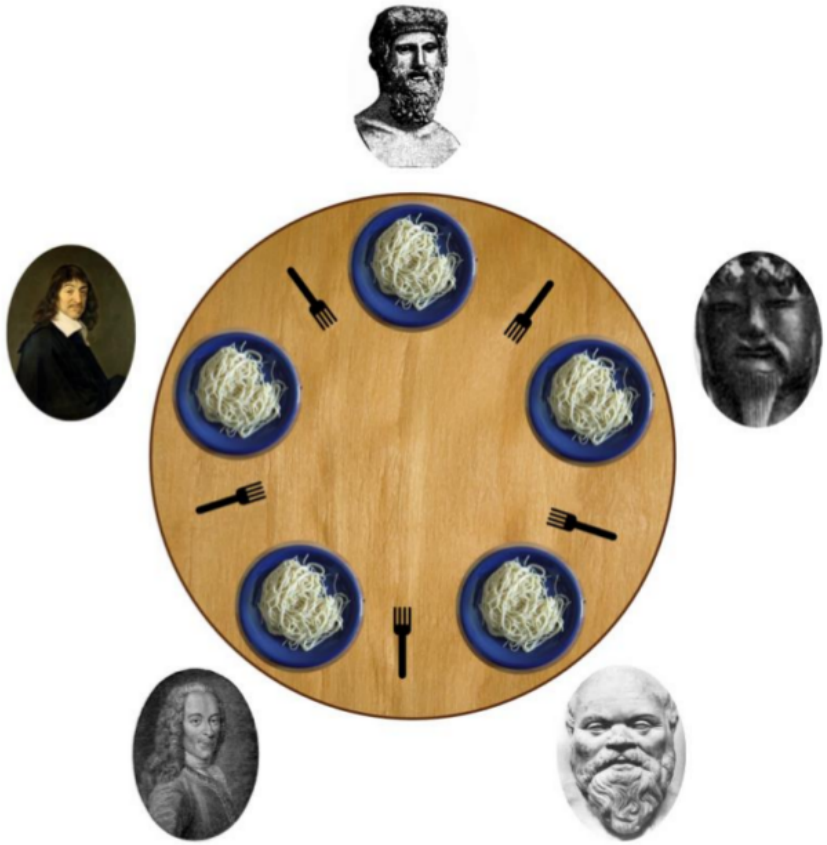
- Los filósofos piensan y comen alternadamente

Filósofos comensales



- Los filósofos piensan y comen alternadamente
- Sólo puede comer cuando tiene dos tenedores

Filósofos comensales



- Los filósofos piensan y comen alternadamente
- Sólo puede comer cuando tiene dos tenedores
- Sólo se pueden tomar los tenedores a izquierda y derecha

Filósofos comensales (características)

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

Filósofos comensales (características)

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

- Mutex: en un momento dado sólo un filósofo puede tener un tenedor dado
- Sincronización: un filósofo sólo puede comer cuando tiene dos tenedores
- Libre de Deadlock/Livelock
- Libre de Starvation

Filósofos comensales (intento naive)

```
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();
  }
}
```

Filósofos comensales (intento naive)

```
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: Si todos toman el tenedor izquierdo se produce una espera circular.

Filósofos comensales (semáforo general)

```
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();
  }
}
```

Filósofos comensales (ruptura de simetría)

```
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();
  }
}
```