.UBAfiuba
FACULTAD DE INGENIERÍA

Técnicas de Programación Concurrente I

Sincronización

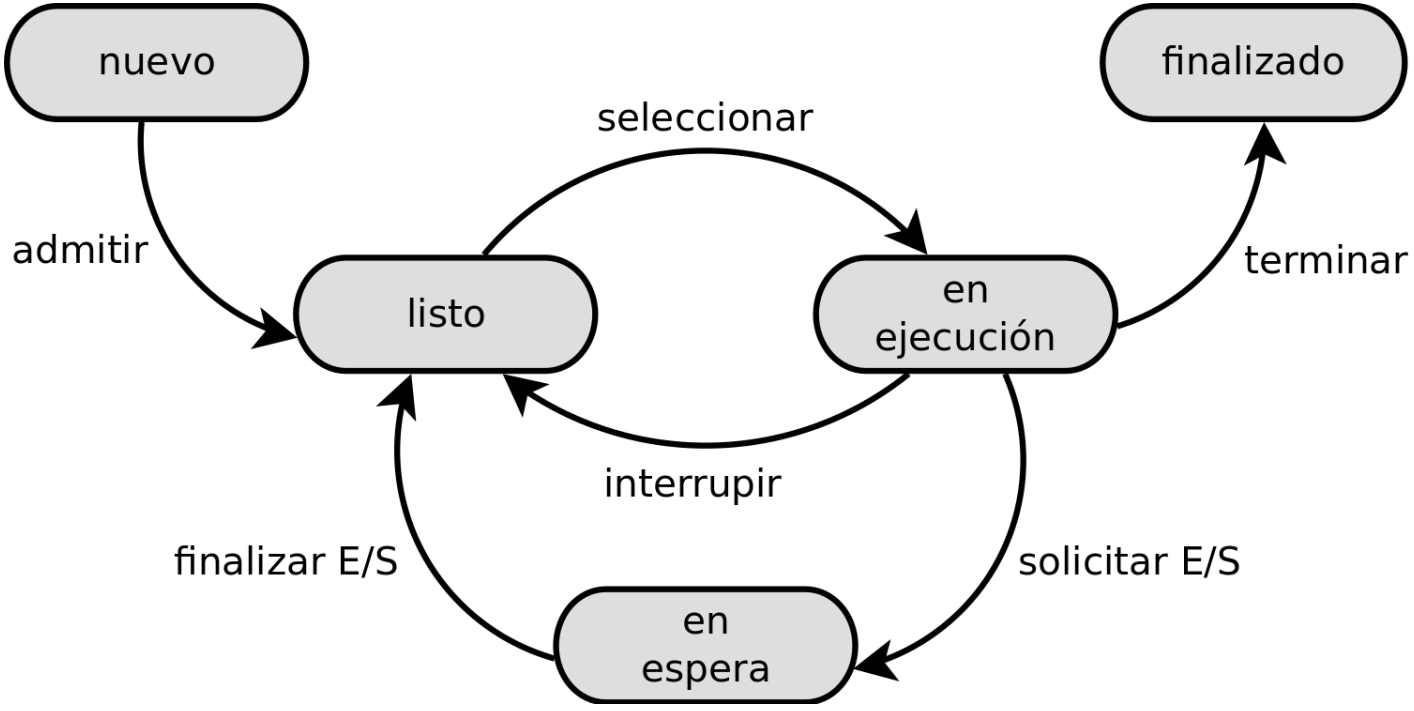
Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

Estados de ejecución de un proceso (teórico)



Semáforos

Mecanismos de sincronismo, implementado como una construcción de programación concurrente de más alto nivel.

- Tipo de dato compuesto por dos campos:
  - un entero no negativo llamado V
  - un set de procesos llamado L
- Se inicializa con un valor k ≥ 0 y con el conjunto vacío $\emptyset$
- Se definen dos operaciones atómicas sobre un semáforo **S**:
  - *wait(S)* también llamada p(S)
  - *signal(S)* también llamada v(S)

Semáforos

Mecanismos de sincronismo de acceso a un recurso

Un semáforo es un contador:

- Si contador $>0 \Rightarrow$ recurso disponible
- Si contador $\leq 0 \Rightarrow$ recurso no disponible
- El valor del semáforo representa la cantidad de recursos disponibles
- Si el valor es cero o uno, se llaman *semáforos binarios* y se comportan igual que los locks de escritura (también conocidos como **Mutex**)

Operaciones

- p (*wait*): resta 1 al contador
- v (*signal*): suma 1 al contador

Semáforos

Operación *wait(S)*

```
if S.V > 0
  S.V := S.V - 1
else
  S.L add p
  p.state := blocked
```

Operación *signal(S)*

```
if S.L is empty
  S.V := S.V + 1
else
  sea q un elemento arbitrario del conjunto S.L
  S.L remove q
  q.state := ready
```

Semáforo binario o Mutex

- El valor V sólo puede tomar los valores 0 ó 1
- Se inicializa como $(0, \emptyset)$ o $(1, \emptyset)$
- La operación signal(S) se define como:

```
if S.V = 1
  // undefined
else if S.L is empty
  S.V := 1
else
  sea q un elemento arbitrario del conjunto S . L
  S.L remove q
  q.state := ready
```

Semáforos

- *wait()* y *signal()* son instrucciones atómicas
- Un semáforo debe ser inicializado con un valor entero no negativo
- La instrucción *signal()* debe despertar a uno de los procesos suspendidos, pero no está definido cuál de todos los procesos debe despertarse

Invariantes de Semáforos

- $S.V \geq 0$
- $S.V = k + \#signal(S) - \#wait(S)$, siendo k el valor inicial del semáforo

Introducción (II)

Tipos de semáforos

- System V
- POSIX

Un semáforo System V está compuesto por:

- El valor del semáforo
- El *process id* del último proceso que utilizó el semáforo
- La cantidad de procesos esperando por el semáforo
- La cantidad de procesos que está esperando que el semáforo sea cero

Índice

Semáforos en Rust

Usamos el crate *std-semaphore*.

- Inicializar el semáforo:

```
let sem = Semaphore::new(5);
```

- Obtener el acceso (wait):

```
fn acquire(&self)
```

- Liberar el semáforo (signal):

```
fn release(&self)
```

- (extra) Obtener el acceso con el patrón RAII (wait):

```
fn access(&self)
```

Barreras en Rust

Permiten sincronizar varios threads en puntos determinados de un cálculo o algoritmo.
Están en el módulo: `std::sync::Barrier`

- Creación de la barrera:

  ```
  fn new(n: usize) -> Barrier
  ```

- Bloquear al thread hasta que todos se encuentren en el punto:

  ```
  fn wait(&self) -> BarrierWaitResult
  ```

Operación interesante: **líder**
El método `BarrierWaitResult::is_leader()` devuelve *true* en el thread líder.
Las barreras son reutilizables automáticamente.

Índice

Productor - Consumidor

- Se definen dos familias de procesos: productores y consumidores
- Requisitos (premisas - propiedades - invariantes):
  1. No se puede consumir lo que no hay
  2. Todos los items producidos son eventualmente consumidos
  3. Al espacio de almacenamiento se accede de a uno
  4. Se debe respetar el orden de almacenamiento y retiro de los items

Productor - Consumidor

- Al utilizar un buffer de comunicación se presentan los siguientes problemas de sincronización:
  1. No se puede consumir si el buffer está vacío
  2. No se puede producir si el buffer está lleno
- Vamos a estudiar dos casos:
  1. Buffer infinito: sólo se presenta el primer problema
  2. Buffer acotado: se presentan ambos problemas

Buffer infinito

buffer := emptyQueue	*sem notEmpty (0, ∅)*
Productor	**Consumidor**

`dataType d`
`loop forever`
`p1: append(d, buffer)`
`p2: signal(notEmpty)`
	`dataType d`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: d <- take(buffer)`

Buffer acotado

buffer := emptyQueue sem notEmpty (0, ∅) sem notFull (N, ∅)

Productor	**Consumidor**

`dataType d`
`loop forever`
`p1: producir`
`p2:  wait(notFull)`
`p3:  append(d, buffer)`
`p4:  signal(notEmpty)`
	`dataType d`
	`loop forever`
	`q1:  wait(notEmpty)`
	`q2:  d <- take(buffer)`
	`q3:  signal(notFull)`
	`q4:  consume(d)`

Bibliografía

- **Principles of Concurrent and Distributed Programming**, M. Ben-Ari, Segunda edición (capítulos 6)
- Bibliografía de Rust.
- *The Design of the Unix Operating System*, Maurice Bach