Tema 4. Semáforos

Programación Concurrente

17 de noviembre de 2020

Introducción

- Problemas de la **espera ocupada**
  - La **espera activa** es ineficiente
    - Los procesos ocupan el procesador incluso aunque no puedan avanzar
- El uso de *variables compartidas* genera esquemas:
  - Demasiado complejos
  - Muy artificiales
  - Propensos a errores: No hay una separación entre las variables de los cómputos de los procesos y las utilizadas para sincronización.
  - Inadecuados para grandes esquemas de protección.
  - Fuertemente acoplados: los procesos deben de *saber* unos de otros.
- Los semáforos son señales simples que pueden obligar a un proceso a detenerse en una posición determinada.
- Con la estructura adecuada de señales podemos satisfacer cualquier requisito de coordinación.
- Los procesos se bloquean o ejecutan condicionados únicamente por el valor de una variable entera

Definición

```
struct Semaphore {
     unsigned value;
     Queue q;
}
```

- Es un TAD formado por un valor numérico y una cola de procesos bloqueados.
- Fueron inventados por Edsger Dijkstra (1965) y se usaron por primera vez en el sistema operativo THEOS (1974). En este otro enlace puedes ver un poco más de la intrahistoria de THEOS.
- Un semáforo bloquea un proceso cuando éste no puede realizar la operación deseada, en vez de desperdiciar tiempo de CPU.
- Pseudocódigo para inicializar a 1: `Semaphore s = 1`

Operaciones sobre semáforos

- P: del holandés "*proberen te verlagen*"(intentar decrementar). Si el contador es mayor que `0` lo decrementa, en caso contrario bloquea el proceso que lo llamó.

  Más conocida como **wait**, **acquire** o **lock**.
- V: del holandés "*verhoog*"(incrementar). Si hay algún proceso bloqueado en el semáforo, lo desbloquea, en caso contario, incrementa el valor de la variable.

  Más conocida como **signal**, **release** o **post**.

```
1  def wait(s):
2     if s.value > 0:
3       s.value -= 1                  (1)
4     else:
5          add(process, s.queue)
6          block(process)             (2)
```

- En **(1)** si es mayor que `0`, se decrementa el valor del semáforo
- En **(2)** si es `0`, se bloquea el proceso que llamó a *wait*

```
1  def signal(s):
2     if empty(s.queue):
3       s.value += 1           (1)
4     else:
5       process = get(s.queue)
6       sched(process)         (2)
```

- En **(1)** si no hay procesos bloqueados en la cola del semáforo, se incrementa su valor
- En **(2)**, en caso contrario, se desbloquea a un proceso
- Otras operaciones que dependen de la implementación concreta:
  - Asignar un valor al semáforo
  - Obtener el valor del semáforo
  - Intentar wait sólo si no produce bloqueo del proceso (`try_lock`)
  - Elección de la política de desbloqueo de procesos
  - ...
- La definición de un semáforo implica la exclusión mutua en la ejecución de las operaciones por él definidas
  - Esta exclusión mutua se consigue mediante funciones proporcionadas por el sistema operativo (típicamente espera con bloqueo)

Problemas inherentes a la *programación concurrente*

- Los semáforos son una herramienta para resolver problemas tanto de exclusión mutua como de sincronización.

  Exclusión mutua: sólo uno de los procesos debe estar en la sección crítica en un instante dado.

  Sincronización: un proceso debe esperar la ocurrencia de un evento para poder seguir ejecutándose.

Exclusión mutua

```
1  Semaphore s = 1
2  ...
3  s.wait()
4  critical_section()
5  s.signal()
6  ...
```

- Cuando el primer proceso entre a la sección crítica decrementará su valor y continuará.
- Si otro proceso desea entrar, `value` será cero, por lo que se bloqueará hasta que el primero ejecute `signal`.

- Veamos los programas `semaphore.c` y `semaphore.py`
- En `java` también hay semáforos, pero no se suelen utilizar si no se está portando código de otros lenguajes.

Candados, mutexes o locks

Semáforos binarios optimizados para exclusión mutua

- Inicializados a `1` por defecto
- Sólo el proceso que hizo `wait` puede hacer `signal`.
- Algunos sistemas permiten que el mismo hilo haga varios `wait`, si ya es propietario del `lock` continúa su ejecución. Se denominan *reentrantes*.

Uso de candados

```
1  Mutex mutex
2  ...
3  mutex.lock()
4  critical_section()
5  mutex.unlock()
6  ...
```

- En **C** los mutex están en la biblioteca *POSIX Threads* y no son reentrantes (`mutex.c`).
- En **Python** tenemos `threading.Lock` (no reentrante) y `threading.Rlock` (reentrante). Se pueden usar también con la cláusula `with` (programas `lock.py` y `lock_with.py`).
- En la biblioteca estándar de **C++** tenemos mutex, thread, condition variable, etc...
- En **D** tenemos la parte de *bajo nivel* de su biblioteca estándar `core.sync` (mutex, semaphore, condition, etc...) y otros componentes de *alto nivel* como: std.parallelism o std.process.
- En **Vala** tenemos un API *orientado a objetos* sobre GLib (Mutex, Thread).
- En la biblioteca estándar de **rust** disponemos de hilos y primitivas de sincronización (mutex, condvar, etc...), pero **no de semáforos**. En éste último caso hemos de recurrir a bibliotecas externas.

POSIX Mutex

- Si nuestra aplicación se compone de un sólo proceso con múltiples hilos, la biblioteca `pthread.h` proporciona mutex (semáforos binarios para hilos).

```
1  #include<pthread.h>
2
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4
5  // inicializar con los atributos por defecto
6  int pthread_mutex_init (...)    // para otras inicializaciones
7  int pthread_mutex_destroy(...)  // destruir el candado
8  int pthread_mutex_lock(...)     // bloquear candado o espera si bloqueado
9  int pthread_mutex_trylock(...); // bloquear candado y no espera si bloqueado
10 int pthread_mutex_unlock(...)   // desbloquear candado
11 ...
12
13 // Declaración de los datos compartidos y del candado que los protege
14
15 struct datos_compartidos {
16    declaración de los datos
17    pthread_mutex_t candado;
18 } datos;
19 ...
20 void codigo_hilo(void *arg){
21    ...
22    pthread_mutex_lock(&datos.candado);
23    // seccion critica
24    pthread_mutex_unlock(&datos.candado);
25    ...
26 }
27
28 void main(){
29    pthread_t hilo_1, hilo_2;
30    int error;
31    ...
32    // Inicializacion del candado
33    error = pthread_mutex_init (&datos.candado, NULL);
34    //pthread_mutex_t datos.candado = PTHREAD_MUTEX_INITIALIZER;
35    if (error)
36       // tratamiento del error
37
38    // Creación de los hilos
39    error = pthread_create(&hilo_1, NULL, codigo_hilo, NULL);
40    if (error)
41       // tratamiento del error
42    error = pthread_create(&hilo_1, NULL, codigo_hilo, NULL);
43    if (error)
44       // tratamiento del error
45    ...
46 }
```

Sincronización genérica

- El valor del semáforo puede interpretarse como un indicador del número de recursos disponibles (*permits*).
- Indica el número de llamadas que se pueden hacer a `wait` sin provocar bloqueo.
- En exclusión mutua debe ser `1`. En este caso reciben el nombre de semáforos binarios.
- Mecanismos más complejos requieren otros valores (por ejemplo, un número máximo de procesos, de impresoras, de procesadores, ...). Se denominan *multiplexes*.

Algoritmo de Barz

- Sirve para simular semáforos generales con semáforos binarios

```
 1    BinarySemaphore mutex = 1
 2    BinarySemaphore gate = 1
 3    value = k
 4
 5  def generalWait():
 6       gate.wait()       # si no es el primero debe esperar
 7       mutex.wait()
 8       value -= 1
 9       if value > 0:
10          gate.signal()  # permite que entre otro si el valor es positivo
11       mutex.signal()
12
13  def generalSignal():
14       mutex.wait()
15       value += 1
16       if value == 1:
17          gate.signal()  # antes estaba en cero, permite que entre otro
18       mutex.signal()
```

Condición de sincronización

- Se asocia un semáforo general a cada condición.
- Un proceso espera a que una condición se cumpla ejecutando -*en principio*- una operación `wait`.
- Cuando un proceso ha hecho que se cumpla una determinada condición, lo indica ejecutando un `signal` sobre dicho semáforo.

- Ejemplo de 2 procesos concurrentes donde **Q** no puede ejecutar $\mathbf{Q_2}$ hasta que **P** haya ejecutado $\mathbf{P_1}$. ¿Qué haríais?

  - ¿Cuántos semáforos utilizaríais y a qué los inicializaríais?
  - ¿Dónde pondríais las operaciones `wait()` y `signal()` de cada semáforo?

```
P           Q
...         ...
P1          Q1
P2          Q2
...         ...
```

La solución usando un semáforo con valor inicial `0`

```
   Semaphore sync = 0
P               Q
...             ...
P1              Q1
sync.signal()   sync.wait()
P2              Q2
...             ...
```

- Ejemplo de 2 procesos concurrentes donde ambos procesos deben esperarse en el punto `1`

  - ¿Cuántos semáforos utilizaríais y a qué los inicializaríais?
  - ¿Dónde pondríais las operaciones `wait()` y `signal()` de cada semáforo?

```
P           Q
...         ...
P1          Q1
(1)         (1)
P2          Q2
...         ...
```

La solución usando dos semáforos `s` y `t` con valores iniciales `0`

```
   Semaphore s = 0
   Semaphore t = 0
P               Q
...             ...
P1              Q1
s.signal()      t.signal()
t.wait()        s.wait()
P2              Q2
...             ...
```

- Ejemplo de 3 procesos concurrentes donde **Q** solo puede pasar a ejecutar **Q2** si **R** ha ejecutado **R1** ***o*** **P** ha ejecutado **P1**

```
P           Q           R
...         ...         ...
P1          Q1          R1
P2          Q2          R2
...         ...         ...
```

La solución utilizando un semáforo con valor inicial `0`

```
   Semaphore sync = 0
P               Q            R
...             ...          ...
P1              Q1           R1
sync.signal()   sync.wait()  sync.signal()
P2              Q2           R2
...             ...          ...
```

- Ejemplo de `3` procesos concurrentes donde **Q** solo ejecuta **Q2** si **R** ha ejecutado **R1** ***y*** **P** ha ejecutado **P1**

```
P           Q           R
...         ...         ...
P1          Q1          R1
P2          Q2          R2
...         ...         ...
```

Solución 1: ¿Es correcta?

```
   Semaphore sync = 0
P               Q            R
...             ...          ...
P1              Q1           R1
sync.signal()   sync.wait()  sync.signal()
P2              sync.wait()  R2
...             Q2           ...
                ...
```

Solución 2: Usar dos semáforos `s` y `t` con valor inicial `0`

- Implica dos sincronizaciones: de **P** y **Q** por un lado y **Q** y **R** por otro.

```
   Semaphore s = 0
   Semaphore t = 0
P               Q            R
...             ...          ...
P1              Q1           R1
s.signal()      s.wait()     t.signal()
P2              t.wait()     R2
...             Q2           ...
                ...
```

Barreras de sincronización

- Los casos anteriores son ejemplos de barreras de sincronización, los programas concurrentes se sincronizan por *fases*: todos los procesos deben esperar que termine la fase actual para comenzar simultáneamente la siguiente.
- Cuando el número de procesos es pequeño basta con utilizar un semáforo por cada punto de espera
- Cuando aumenta el número de procesos esta solución es altamente ineficiente por el coste de los semáforos

Barreras para n procesos

- El problema general se puede representar mediante sincronizaciones cíclicas del tipo:

```
while True:
   do_phase()
   barrier(n)
```

- Y se puede solucionar con dos semáforos y un contador con actualización atómica
- Si no se dispone de este tipo de contador se necesita además un candado
- Algunos lenguajes implementan, o están en proceso de implementar, este tipo de barreras.

```
1       Semaphore arrival = 1
2       Semaphore departure = 0
3       counter = 0
4
5  def barrier(n):
6        arrival.wait()
7        getAndAdd(counter, 1)
8        if counter < n:
9            arrival.signal()   (1)
10       else:
11           departure.signal() (2)
12
13       departure.wait()       (3)
14
15       getAndAdd(counter, -1)
16       if counter > 0:
17           departure.signal() (4)
18       else:
19           arrival.signal()   (5)
```

- En **(1)** si no llegaron todos los procesos, permite la llegada de otro
- En **(2)** si llegaron todos, autoriza la salida de un proceso
- En **(3)** espera la autorización para continuar
- En **(4)** si no salieron todos, autoriza la salida del siguiente
- En **(5)** si llegaron todos, comienza nuevamente el ciclo de llegadas
- Veamos cómo queda el código completo (`barrier.py`)

Semáforos fuertes y débiles

Fuertes: la cola de procesos sigue una política *FIFO* estricta

Débiles: si los procesos se seleccionan aleatoriamente

Semáforos para procesos: System V

- Se compone de los siguientes elementos:
  - El valor del semáforo
  - El identificador del último proceso que lo manipuló
  - El número de procesos que hay esperando a que el valor del semáforo se incremente
  - El número de procesos que hay esperando a que el semáforo tome el valor `0`
- Ejemplos de código `sem-1.c` y `sinSem-1.c`

Semáforos POSIX

- Debemos incluir la cabecera `semaphore.h`
- Estándar relativamente reciente (1993)
- Utiliza funciones clásicas sobre semáforos
- El tipo del semáforo es `sem_t` y almacena toda la información relativa al semáforo

- Dos tipos:

  Semáforos nominales : para sincronizar procesos no relacionados. Tienen asociado un nombre en la jerarquía del sistema de ficheros

  Semáforos anónimos : para sincronizar procesos de una misma jerarquía o hilos de un mismo proceso

- Veamos el código de `semPosix.c` y de `buffer-circular-hilos.c`

Semáforos vs. Espera Ocupada

- Si los semáforos residen en la memoria compartida y por tanto hay que acceder a ellos en exclusión mutua, ¿no es la pescadilla que se muerde la cola?
- En algún momento habrá que recurrir a mecanismos de más bajo nivel...
- ¿Cuál pensáis entonces que es la ventaja fundamental de los semáforos?

Espera ocupada :

```
protocolo entrada espera ocupada
Sección Crítica (código usuario, duración no definida)
protocolo salida espera ocupada
```

Semáforo :

```
protocolo entrada espera ocupada
sección crítica wait (corta)
protocolo salida espera ocupada

sección crítica (código usuario)

protocolo entrada espera ocupada
sección crítica signal(corta)
protocolo salida espera ocupada
```

Variables de condición

Idea de funcionamiento

- Las variables de condición se introducen para simplificar una situación bastante habitual en la sincronización de hilos.

- La situación se da cuando un hilo necesita esperar a que se cumpla un predicado en el que intervienen varias variables compartidas con otro u otros hilos.
- Al ser variables compartidas se las protege con un `mutex`.
- Las variables de condición nos permiten que un hilo espere *dormido* el cumplimiento de ese predicado además de que el otro hilo le avise mediante la propia variable de condición de que ha habido un cambio en una o varias de las variables que conforman el predicado -*no que el predicado se cumple*-.
- Es entonces cuando el *otro* hilo despierta y comprueba si con los nuevos valores de las variables compartidas se cumple el predicado.
- Para ello:
  1. Obtiene el mutex
  2. Comprueba el predicado
  3. Si el predicado es cierto hace su trabajo y libera el mutex
  4. Si el predicado es falso llama a `cond_wait` y vuelve al punto `2` la siguiente vez.
  5. `cond_wait` bloquea el hilo y libera el mutex.

Ejemplo

- Supongamos que el predicado es que dos variables `x` e `y` tengan el mismo valor: `x == y`.

```
// 'v' es una variable de condicion
// 'm' es un mutex

// Hilo A                  // Hilo B
lock_mutex(&m);            lock_mutex(&m);
while ( x != y )           x += 3;
  cond_wait(&v, &m);       cond_signal(&v);
// Thread CS               // Thread CS
unlock_mutex(&m);          unlock_mutex(&m);
```

- **Importante:** trata de entender por qué el `Hilo A` llama a `cond_wait` en un bucle en lugar de con un `if`.

Posix API

```
#include <pthread.h>

// pthread_cond_t v = PTHREAD_COND_INITIALIZER
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Problemas clásicos

- Productor-consumidor
- Lectores-escritores
- Comida de los filósofos

Productor-consumidor

- Es un modelo muy común en informática.
- La mayoría de lenguajes proporcionan implementaciones nativas o en bibliotecas
- Características del problema:
  - Para que sea asíncrono es necesario un buffer
  - Productor deposita elementos en un buffer
  - Consumidor recoge los elementos del buffer
- Dos variantes:
  - Buffer limitado
  - Buffer ilimitado
- Restricciones:
  - El consumidor debe esperar a que el buffer tenga elementos a consumir.
  - Exclusión mutua en el acceso al buffer.
  - Si el buffer es *limitado*, el productor debe esperar que haya posiciones vacías si éste está lleno.

- Buffer infinito (`producer-consumer-infinite.py`)

```
 1  Queue buffer
 2  Semaphore mutex = 1
 3  Semaphore notEmpty = 0
 4
 5  # Productor
 6  while True:
 7      data = produce()
 8
 9      mutex.wait()
10      buffer.add(data)
11      mutex.signal()
12
13      notEmpty.signal()
14
15  # Consumidor
16  while True:
17      notEmpty.wait()
18
19      mutex.wait()
20      data = buffer.get()
21      mutex.signal()
22
23      consume(data)
```

- Buffer limitado (`producer-consumer.py`)

```
 1  Queue buffer
 2  Semaphore mutex = 1
 3  Semaphore notEmpty = 0
 4  Semaphore notFull = BUFFER_SIZE  (*)
 5
 6  # Productor
 7  while True:
 8      data = produce()
 9
10      notFull.wait()               (*)
11
12      mutex.wait()
13      buffer.add(data)
14      mutex.signal()
15
16      notEmpty.signal()
17
18  # Consumidor
19  while True:
20      notEmpty.wait()
21
22      mutex.wait()
23      data = buffer.get()
24      mutex.signal()
25
26      notFull.signal()             (*)
27
28      consume(data)
```

Lectores-escritores

Características del problema :

- Lectores desean leer un recurso, dos o más pueden acceder simultáneamente al recurso
- Escritores actualizan la información en un recurso, sólo uno puede acceder al recurso: acceso exclusivo al recurso

Dos versiones distintas del problema:

- Prioridad en la lectura: ningún lector debe esperar salvo que un escritor haya ya obtenido permiso para usar el recurso. Ningún lector debe esperar a que otros lectores acaben por el simple hecho de que un escritor esté esperando
- Prioridad en la escritura: una vez que el escritor está esperando, pasará el primero, es decir, ningún lector podrá iniciar su lectura
- ¿Qué problema presenta la primera versión? ¿Y la segunda?

Lectores-escritores: prioridad lectura

- Código completo (`rw_lock.py`)

```
 1  readers = 0          # num. lectores en la Secc. crítica
 2  Semaphore writer = 1 # mutex para escritor
 3  Semaphore mx = 1     # mutex acceso readers y
 4                       # barrera para los escritores
 5
 6  def writer_lock():     def writer_unlock():
 7      writer.wait()          writer.signal()
 8
 9  def reader_lock():     def reader_unlock():
10      mx.wait()              mx.wait()
11      readers += 1           readers -= 1
12      if readers == 1:       if readers == 0:
13         writer.wait()          writer.signal()
14      mx.signal()            mx.signal()
```

Lectores-escritores: prioridad escritura (versión 1)

- Archivo: `rw_lock_fair.py`

```
1  # Las mismas variables que antes, más
2  Semaphore entry = 1     # bloquea lectores cuando
3                          # un escritor quiere entrar
4
5  def writer_lock():      def writer_unlock():
6      entry.wait()            writer.signal()
7      writer.wait()           entry.signal()
8
9  def reader_lock():      def reader_unlock():
```

```
10      entry.wait()            mx.wait()
11      mx.wait()               readers -= 1
12      readers += 1            if readers == 0:
13      if readers == 1:           writer.signal()
14         writer.wait()        mx.signal()
15      mx.signal()
16      entry.signal()
```

Lectores-escritores: prioridad escritura (versión 2)

- Los dos `wait` sobre los semáforos `entry` y `mx` hacen ineficiente al algoritmo anterior
- El código `rw_fair_faster.py` es una implementación posterior (2013) que evita este problema
- POSIX y Java ofrecen implementaciones para el problema de los lectores-escritores

Comida de filósofos

- Cinco filósofos se dedican a pensar y a comer en una mesa circular.
- En el centro de la mesa hay un cuenco con arroz, y la mesa está puesta con cinco platos y cinco palillos, uno por cada filósofo.
- Cuando un filósofo tiene hambre se sienta en la mesa a comer en su sitio.
- El filósofo sólo puede coger un palillo cada vez y no le puede quitar un palillo a un compañero que lo tenga en la mano.
- Cuando un filósofo tiene los dos palillos come sin soltarlos hasta que termine y vuelva a pensar.

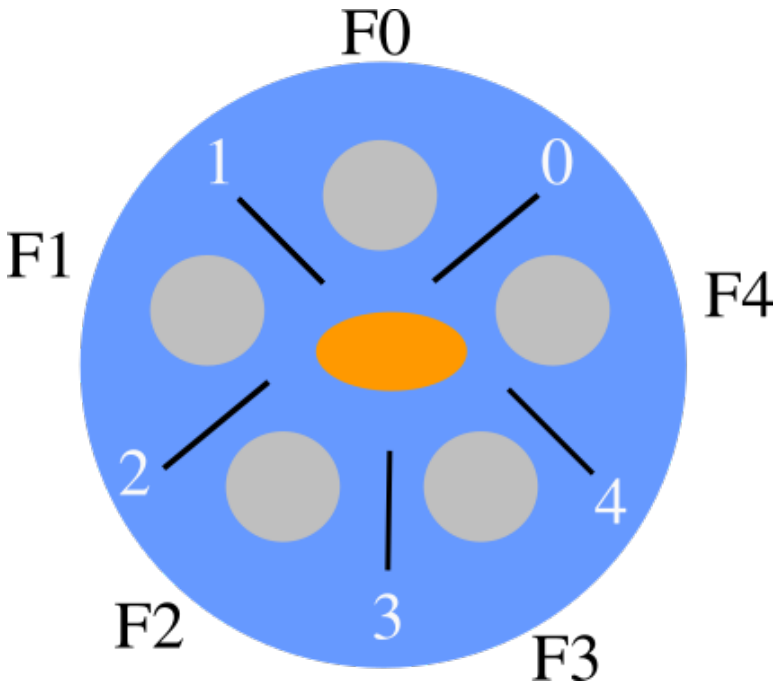

Figura 1: Comida filósofos.

- Ilustra los problemas básicos de interbloqueo.
- Representa los problemas relacionados con la coordinación de los recursos no compartibles de un sistema.
- Es objeto habitual de estudio y comparación entre los diferentes mecanismos de sincronización

```
1  def philosopher():
2  while True:
3       think()
4       pick()     (1)
5       eat()
6       release()  (2)
```

- En **(1)** se asegura que puede coger los dos palillos
- En **(2)** libera ambos palillos

Comida de filósofos (solución 1)

- Exclusión mutua a toda la mesa

- Sólo un filósofo puede comer a la vez

```
1  Semaphore table = 1
2
3  def philosopher():
4    while True:
5         think()
6         table.wait()
7         eat()
8         table.signal()
```

- Solución ineficiente, porque podrían comer dos

Comida de filósofos (solución 2)

- Exclusión mutua a cada palillo
- El filósofo **i** tendrá que coger los palillos **i** e **(i+1) %5**

```
 1  Semaphore forks[5] = [1, 1, 1, 1, 1]
 2
 3  def philosopher(i):    def pick(i):
 4    while True:               forks[i].wait()
 5         think()              forks[right(i)].wait()
 6         pick(i)
 7         eat()           def release(i):
 8         release(i)           forks[i].signal()
 9                              forks[right(i)].signal()
10  def right(i):
11    return (i+1)%5
```

Comida de filósofos (¿solución 2?)

- El filósofo coge primero el palillo izquierdo y después el derecho. Si alguno ya está cogido se queda esperando a que lo suelten
- Pero si todos quieren comer simultáneamente puede aparecer un problema:

```
 1  # Tenedor izquierdo
 2  forks[0].wait()         # Filosofo 0
 3    forks[1].wait()       # Filosofo 1
 4      forks[2].wait()     # Filosofo 2
 5        forks[3].wait()   # Filosofo 3
 6          forks[4].wait() # Filosofo 4
 7
 8  # Tenedor derecho
 9  forks[1].wait()         # Filosofo 0
10    forks[2].wait()       # Filosofo 1
11      forks[3].wait()     # Filosofo 2
12        forks[4].wait()   # Filosofo 3
13          forks[0].wait() # Filosofo 4
```

Interbloqueos

- Se pueden producir cuando hay competencia por recursos compartidos, por ejemplo:

```
P               Q

get(a)        get(b)
...           ...
get(b)        get(a)
```

- Ambos procesos se bloquean mutuamente esperendo a que el otro libere el recurso
- No hay progreso en la ejecución

Condiciones para interbloqueo

- Exclusión mutua
- Retención y espera
  - Un proceso mantiene los recursos asignados hasta la asignación de otro
- No apropiación
  - No se puede quitar un recurso ya asignado
- Espera circular
  - Se produce un ciclo cerrado de procesos esperando por recursos ya asignados

Resolviendo el interbloqueo

- Hay que impedir alguna de las circunstancias anteriores
- Lo más sencillo es romper la espera circular
- El filósofo que provoca el interbloqueo debe coger los palillos en otro orden (fíjate que el filosofo$_4$ ha cambiado el orden), por eso hacemos:

```
1  def pick(i):
2      if i < right(i):    # i == 0
3          forks[i].wait()
4          forks[right(i)].wait()
5      else:
6          forks[right(i)].wait()
7          forks[i].wait()
```

- Se resuelve el interbloqueo, pero...
- La solución no es óptima: en alguna traza podrían comer dos filósofos y sólo come uno
- Esperan los filósofos 0, 1, 2 y 4. Sólo come el 3

```
 1  forks[0].wait()         # Filosofo 0
 2    forks[1].wait()       # Filosofo 1
 3      forks[2].wait()     # Filosofo 2
 4        forks[3].wait()   # Filosofo 3
 5          forks[0].wait() # Filosofo 4, Wait en orden decreciente, se bloquea
 6
 7    forks[1].wait()       # Filosofo 0
 8      forks[2].wait()     # Filosofo 1
 9        forks[3].wait()   # Filosofo 2
10          forks[4].wait() # Filosofo 3, tiene tenedor 4 libre, come y
11                          # los otros esperan
```

¿Solución óptima?

- Cambio de enfoque: lo vemos como un problema de sincronización
- Un filósofo podrá estar en cualquiera de estos estados: `THINKING`, `HUNGRY`, `EATING`
- Un filósofo podrá comer si ninguno de sus vecinos está comiendo, en otro caso espera a que le notifiquen que han acabado
- No hay *deadlocks*, pero podría darse el caso de que un filósofo no comiera nunca...
- Disponemos de un array para representar el estado de los filósofos y un semáforo (`mutex`) controla la *exclusión mutua* a este array. El array `sync` de semáforos porporciona la sincronización entre filósofos.
- Destacar las líneas `28` y `29`, en ellas se evita un *deadlock*.

```
 1  # philosofers_2.py
 2  status[5] = [THINKING, ..., THINKING]
 3  Semaphore sync[5] = [0, 0, 0, 0, 0]
 4  Semaphore mutex = 1
 5  def pick(i):
 6     mutex.acquire()
 7     status[i] = HUNGRY
 8     canEat(i)
 9     mutex.release()
10     sync[i].acquire()
11
12  def right(i):
```

```
13     return (i+1)%5
14
15  def left(i):
16     return (i-1)%5
17
18  def canEat(i):
19     if status[i] == HUNGRY and
20        status[left(i)]  != EATING and
21        status[right(i)] != EATING:
22        status[i] = EATING
23        sync[i].release()
24
25  def release(i):
26     mutex.acquire()
27     status[i] = THINKING
28     canEat(left(i))
29     canEat(right(i))
30     mutex.release()
```

Problema de los Caníbales

- Una tribu cena en comunidad una gran olla que contiene **M** misioneros cocinados.
- Cuando un miembro de la tribu quiere comer, él mismo se sirve de la olla un misionero, a menos que esté vacía. Los miembros de la tribu se sirven de la olla de uno en uno.
- Si la olla está vacía, el que va a cenar despierta al cocinero y se espera a que esté llena la olla.
- Desarrollar el código de las acciones de los miembros de la tribu y el cocinero usando semáforos.

Usa las siguientes variables:

olla : Entero que indica el número de misioneros en la olla. Estará inicializada a **M**.

mutex : Semáforo para proteger la exclusión mutua sobre la variable olla. Inicializado a `1`.

espera : Semáforo utilizado para hacer que el que va a cenar se detenga hasta que el cocinero llene la olla cuando está vacía. Inicializado a `0`.

coci : Semáforo inicializado a `0` y usado para que el cocinero no haga nada cuando la olla no está vacía.

Aclaraciones

- **En ningún caso estas transparencias son la bibliografía de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.