semáforo

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están competiendo y que cumple la siguiente semántica:

- El estado interno del semáforo cuenta cuantos procesos todavía pueden utilizar el recurso. Se puede realizar, por ejemplo, con un número entero que nunca debe llegar a ser negativo.
- Existen tres operaciones con un semáforo: `init()`, `wait()`, y `signal()` que realizan lo siguiente:

operaciones del semáforo I

`init()`:

- Inicializa el semáforo antes de que cualquier proceso haya ejecutado ni una operación `wait()` ni una operación `signal()` al límite de número de procesos que tengan derecho a acceder el recurso.
- Si se inicializa con 1, se ha construido un semáforo binario.
- En lenguajes orientados a objectos, la operación `init()` se suele realizar en la construcción del objeto correspondiente.

operaciones del semáforo II

`wait()`:

- Si el estado indica cero, el proceso se queda atrapado en el semáforo hasta que sea despertado por otro proceso.
- Si el estado indica que un proceso más puede acceder el recurso se decrementa el contador y la operación termina con exito.
- La operación `wait()` tiene que estár implementada como una instrucción atómica. Sin embargo, en muchas implementaciones la operación `wait()` puede ser interrumpida.
- Normalmente existe una forma de comprobar si la salida del `wait()` es debido a una interrupción o porque se ha dado acceso al semáforo.
- Importante: esta comprobación se debe hacer!

operaciones del semáforo III

`signal()`:

- Una vez se ha terminado el uso del recurso, el proceso lo señaliza al semáforo. Si queda algún proceso bloqueado en el semáforo, uno de ellos sea despertado, sino se incrementa el contador.
- La operación `signal()` también tiene que estár implementada como instrucción atómica. En algunás implementaciones es posible comprobar si se ha despertado un proceso con exito en caso que había alguno bloqueado.
- Para despertar los procesos se pueden implementar varias formas que se destinguen en su política de justicia (p.ej. FIFO).

uso del semáforo

El acceso mutuo a secciones críticas se arregla con un semáforo que permita el acceso a un sólo proceso

```
S.init(1)
```

```
P1                              P2
a: loop                             loop
b:    S.wait()                        S.wait()
c:    critical section                critical section
d:    S.signal()                      S.signal()
e:    non-critical section            non-critical section
f: endloop                          endloop
```

Enteros con Semaphore

```
import java.util.concurrent.Semaphore;

// Implementation of our integer with a semaphore.
class Int {
  private int i;
  private Semaphore semaphore;
  Int(int i) {
    this.i=i;
    semaphore=new Semaphore(1);
  }
  void Add(Int I) {
    try {
      semaphore.acquire(); // Entrance protocol.
      i+=I.i;
    }
    catch(InterruptedException E) {
      System.out.println("got interrupted...??");
    }
    finally {
      semaphore.release(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

semáforos binarios/generales

Si existen en un entorno solamente semáforos binarios (es decir, cerrojos simples), se puede simular un semáforo general usando dos semáforos binarios y un contador, por ejemplo, con las variables `delay`, `mutex` y `count`.

- La operación `init()` inicializa el contador al número máximo permitido.
- El semáforo `mutex` asegura acceso mutuamente exclusivo al contador.
- El semáforo `delay` atrapa a los procesos que superan el número máximo permitido.

detalles de la implementación I

La operación `wait()` se implementa de la siguiente manera:

```
delay.wait()
mutex.wait()
decrement count
if count greater 0 then delay.signal()
mutex.signal()
```

detalles de la implementación II

La operación `signal()` se implementa de la siguiente manera:

```
mutex.wait()
increment count
if count equal 1 then delay.signal()
mutex.signal()
```

Implementación de este semáforo en Java

```
class MySemaphore {
  private int cnt;
  private ReentrantLock mutex;
  private ReentrantLock delay;
  MySemaphore(int n) {
    cnt=n;
    mutex=new ReentrantLock();
    delay=new ReentrantLock();
  }
  public void acquire() {
    delay.lock();
    mutex.lock();
    --cnt;
    if(cnt>0) delay.unlock();
    mutex.unlock();
  }
  public void release() {
    mutex.lock();
    ++cnt;
    if(cnt==1) delay.unlock();
    mutex.unlock();
  }
}
```

principales desventajas de semáforos

- No se puede imponer el uso correcto de las llamadas a los `wait()`s y `signal()`s.
- No existe una asociación entre el semáforo y el recurso.
- Entre `wait()` y `signal()` el usuario puede realizar cualquier operación con el recurso.

observaciones II

- Las regiones críticas no son lo mismo que los semáforos, porque no se tiene acceso directo a las operaciones `init()`, `wait()` y `signal()`.
- Con semáforos se puede emular regiones críticas pero no al revés.

monitor

Un monitor es un tipo de datos abstracto que contiene

- un conjunto de procedimientos de control de los cuales solamente un subconjunto es visible desde fuera,
- un conjunto de datos privados, es decir, no visibles desde fuera.

y permite realizar operaciones/transacciones con exclusión mutua.

detalles de implementación de un monitor

- El acceso al monitor está permitido solamente a través de los métodos públicos y el compilador garantiza exclusión mutua para todos los accesos.
- La implementación del monitor controla la exclusión mutua con mecanismos de entrada que contengan todos los procesos bloqueados mientras haya uno accediendo.
- Pueden existir varias colas (o estructuras de datos) y el controlador/planificador del monitor elige de cual cola se va a escoger el siguiente proceso para actuar sobre los datos.
- Un monitor no tiene acceso a variables exteriores con el resultado de que su comportamiento no puede depender de ellos.
- Una desventaja de los monitores es la exclusividad de su uso, es decir, la concurrencia está limitada si muchos procesos hacen uso del mismo monitor.

sincronización condicional

- Un aspecto que se encuentra en muchas implementaciones de monitores es la sincronización condicional, es decir, mientras un proceso está ejecutando un procedimiento del monitor (con exclusión mutua) puede dar paso a otro proceso liberando el cerrojo. (Estas operaciones se suele llamar `wait` o `delay`).
- El proceso que ha liberado el cerrojo se queda bloqueado hasta que otro proceso le despierta de nuevo. Este bloqueo temporal está realizado dentro del monitor.
- Dicha técnica se refleja en Java con `wait()` y `notify()` o `notifyAll()`.
- La técnica permite la sincronización entre procesos porque, actuando sobre el mismo recurso, los procesos pueden cambiar el estado del recurso y pasar así información de un proceso a otro.

disponibilidad de monitores

- Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (por ejemplo en Java: todos los objetos derivan de `Object` que contiene los métodos `wait()`, `notify`, y `notifyAll()`.
- El uso de monitores es bastante costoso, porque se puede perder eficiencia por bloquear los procesos innecesariamente y el trabajo adicional por el uso del monitor.
- Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema (alternativas libres de cerrojos (*lock free*) y/o libres de espera (*wait free*)).

desventajas de uso de sincronización a alto nivel

- No se distingue entre accesos de solo lectura y de escritura que limita la posibilidad de accesos en paralelo.
- Cualquier interrupción (p.ej. por falta de página de memoria) relantiza el avance de la aplicación,
- por eso las MVJ usan los procesos del sistema operativo para implementar los hilos, así el S.O. puede conmutar a otro hilo.
- Sigue presente el problema de llamar antes a `notify()`, o `notifyAll()` que a `wait()`
  (condición de carrera o *race condition*).

Java máquina de estados de hilos