conceptos

- El concepto de usar estructuras de datos a nivel alto libera al programador de los detalles de su implementación.
- El programador puede asumir que las operaciones están implementadas correctamente y puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.
- Las implementaciones concretas de los tipos de datos abstractos tienen que recurrir a las posibilidades descritas anteriormente.

semáforo

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están compitiendo 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 contruido un semáforo binario.

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.

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
```

observamos los siguientes detalles

- Si algún proceso no libera el semáforo, se puede provocar un bloqueo.
- No hace falta que un proceso libere su propio recurso, es decir, la operación `signal()` puede ser ejecutada por otro proceso.
- Con simples semáforos no se puede imponer un orden a los procesos accediendo a diferentes recursos.

semáforos binarios/generales

Si existen en un entorno solamente semáforos binarios, 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()
```

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.

región crítica

- Un lenguaje de programación puede realizar directamente una implementación de una región crítica.
- Así parte de la responsabilidad se traslada desde el programador al compilador.
- De algúna manera se identifica que algún bloque de código se debe tratar como región crítica (así funciona Java con sus bloques sincronizados):

```
V is shared variable
region V do
  code of critical region
```

observaciones I

- El compilador asegura que la variable `V` tenga un semáforo adjunto que se usa para controlar el acceso exclusivo de un solo proceso a la región crítica.
- De este modo no hace falta que el programador use directamente las operaciones `wait()` y `signal()` para controlar el acceso con el posible error de olvidarse de algún `signal()`.
- Adicionalmente es posible que dentro de la región crítica se llame a otra parte del programa que a su vez contenga una región crítica. Si esta región está controlada por la misma variable `V` el proceso obtiene automáticamente también acceso a dicha región.

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.

regiones críticas condicionales

- En muchas situaciones es conveniente controlar el acceso de varios procesos a una región crítica por una condición.
- Con las regiones críticas simples, vistas hasta ahora, no se puede realizar tal control. Hace falta otra construcción, por ejemplo:

```
V is shared variable
C is boolean expression
region V when C do
  code of critical region
```

detalles de implementación I

Las regiones críticas condicionales funcionan internamente de la siguiente manera:

- Un proceso que quiere entrar en la región crítica espera hasta que tenga permiso.
- Una vez obtenido permiso comprueba el estado de la condición, si la condición lo permite entra en la región, en caso contrario libera el cerrojo y se pone de nuevo esperando en la cola de acceso.

detalles de implementación II

- Se implementa una región crítica normalmente con dos colas diferentes.
- Una cola principal controla los procesos que quieren acceder a la región crítica, una cola de eventos controla los procesos que ya han obtenido una vez el cerrojo pero que han encontrado la condición en estado falso.
- Si un proceso sale de la región crítica todos los procesos que quedan en la cola de eventos pasan de nuevo a la cola principal porque tienen que recomprobar la condición.

detalles de implementación III

- Nota que esta técnica puede derivar en muchas comprobaciones de la condición, todos en modo exclusivo, y puede causar pérdidas de eficiencia.
- En ciertas circunstancias hace falta un control más sofisticado del acceso a la región crítica dando paso directo de un proceso a otro.

desventajas de semáforos y regiones críticas

Todas las estructuras que hemos visto hasta ahora siguen provocando problemas para el programador:

- El control sobre los recursos está distribuido por varios puntos de un programa.
- No hay protección de las variables de control que siempre fueron variables globales.

monitor

Por eso se usa el concepto de monitores que implementan un nivel aún más alto de abstracción facilitando el acceso a recursos compartidos.
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.

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 colas de entrada que contengan todos los procesos bloqueados.
- Pueden existir varias colas y el controlador 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()`/`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 al otro.

disponibilidad de monitores

- Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (por ejemplo en Java).
- El uso de monitores es bastante costoso, porque se pierde eficiencia por bloquear mucho los procesos.
- Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema.

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()` (*race condition*).