un programa concurrente

- Asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos.
  (por ejemplo, los factores de la multiplicacion, o mirad las prácticas)
- Si los recursos de los diferentes procesos son diferentes no hay problema (mira por ejemplo la práctica de filtrado de matriz),
- Si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?
- Vimos ya ayudas Java como `AtomicInteger`, o el `synchronized`, o los otros...
- Levantamos el nivel a algo más abstracto, revisamos los ya mencionados y luego implementamos a nivel bajo.

un programa concurrente

Tenemos básicamente tres opciones para tratar posibles escrituras concurrentes:

- se implementa exclusión mutua, es decir, solamente un proceso tiene acceso, los demás esperan;
- se implementa comportamiento idéntico, es decir, desde el algoritmo se garantiza que todos los procesos actuan igual (sobre todo: escriben lo mismo en caso que escriban concurrentemente);
- se implementa comportamiento transaccional, es decir, solo un proceso gana, lo que hacen los demás no influe en su resultado (con la opción que los demás se notifiquen en caso de fracaso) (con la opción que cualquiera o uno específico gana).

¿Qué es exclusión mutua?

- Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control
  - que permite la entrada de un proceso si el recurso está disponible y
  - que prohibe la entrada de un proceso si el recurso está ocupado.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Para implementar exclusión mutua se necesita algo básico a nivel hardware.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles en el hardware.
  (Eso veremos más adelante.)

estructura general basada en protocolos

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue (si protegemos código):

```
P0                          ... Pi
...                             ...
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...                             ...
```

Enteros con `lock`

```
import java.util.concurrent.locks.*;

// Implementation of out integer with a reentrant lock.
class Int {
  private int i;
  private ReentrantLock lock;
  Int(int i) {
    this.i=i;
    lock=new ReentrantLock();
  }
  void Add(Int I) {
    lock.lock(); // Entrance protocol.
    try {
      i+=I.i;
    } finally {
      lock.unlock(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

conceptos

- El concepto de usar estructuras de datos a nivel alto libera al/a programador/a de los detalles de su implementación.
- Se puede asumir que las operaciones están implementadas correctamente y se puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.
- Para que se puedan utilizar con provecho hay que entender en detalle las propiedades de tales estructuras de datos.

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 alguna manera (depende del lenguaje de programación en concreto) 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

- El compilador asegura que la variable $\mathbb{V}$ tenga un protocolo de entrada y salida 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 de los protocolos para controlar el acceso con el posible error de olvidarse de alguna parte esencial.
- 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 ésta está controlada por la misma variable $\mathbb{V}$ el proceso obtiene automáticamente también acceso a dicha región.

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 pueden funcionar 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 puede ser interesante realizar un control más sofisticado del acceso a la región crítica dando paso directo de un proceso a otro.

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, 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();
  }
}
```

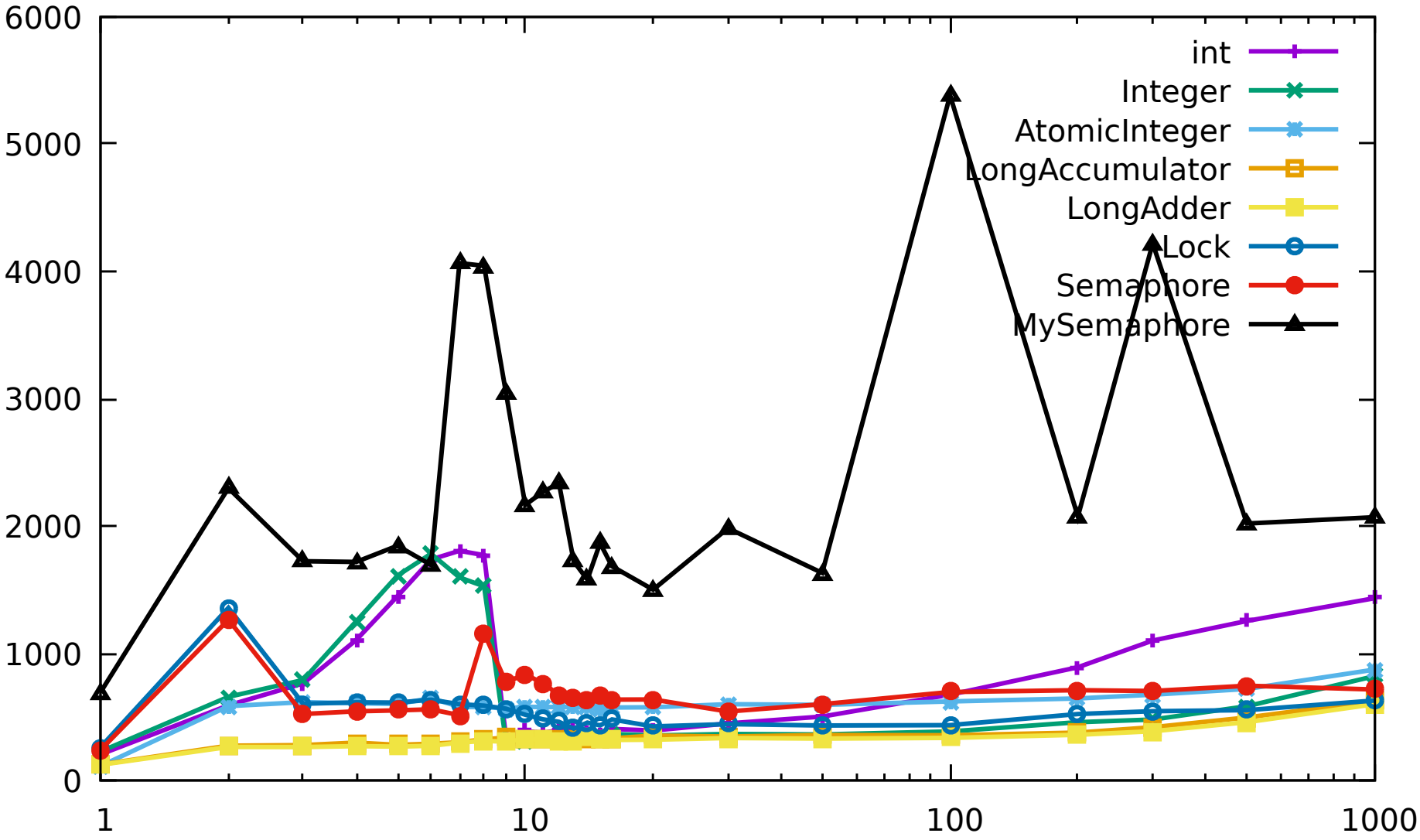
principales desventajas de semáforos

- 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.

Mediciones con más diferentes clases (variando #hilos)



tiempo versus #hilos de 10*1000000 en mi sistema *corona*