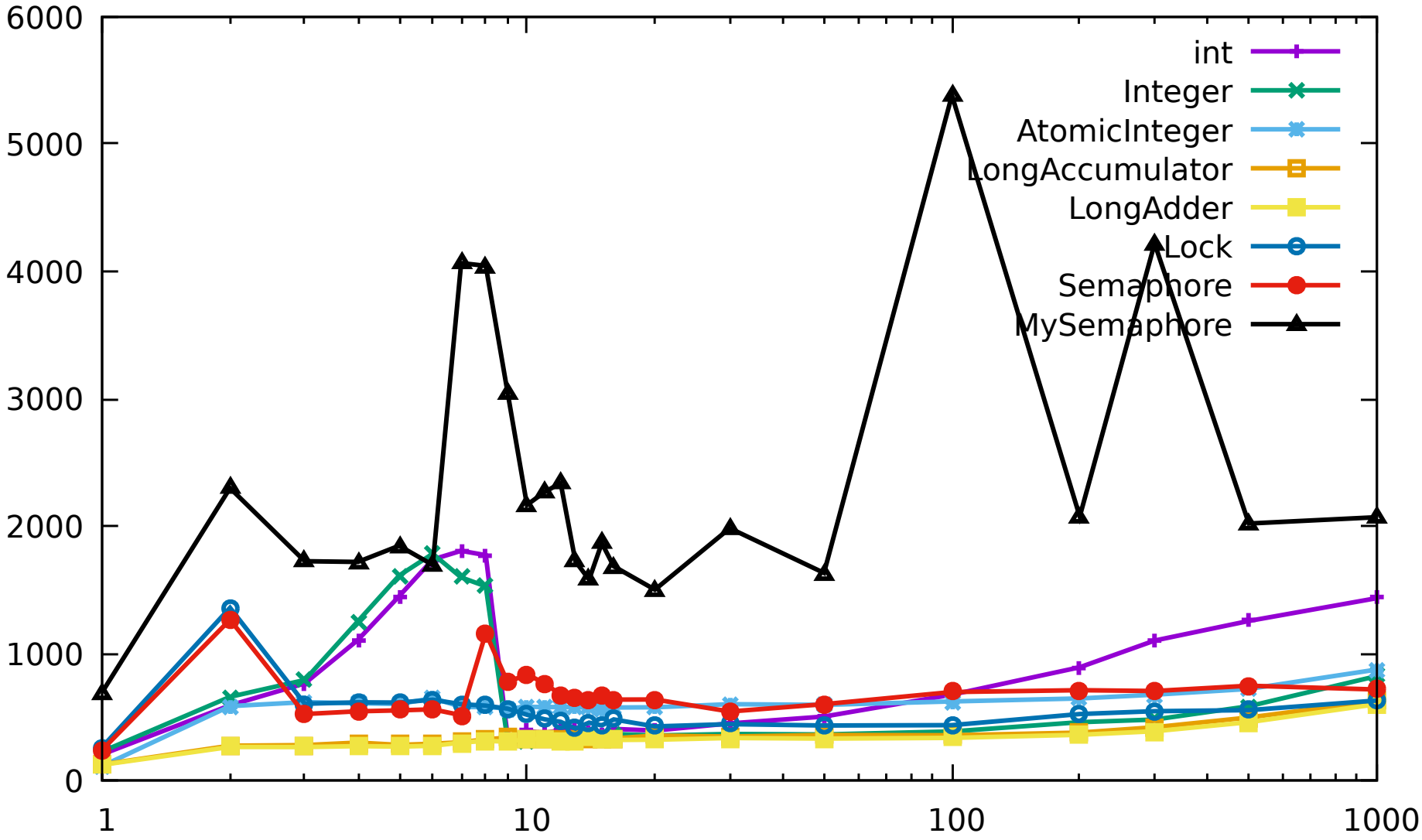
Mediciones con más diferentes clases (variando #hilos)



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

Multiplicación concurrente correcto

El algoritmo no era eficiente, ya que hay mucha congestión accediendo a las variables compartidas `q` y `r` con exclusión mutua:

- En cada iteración del bucle de cálculo todos los hilos compiten por el acceso exclusivo a `q` y `r`.
- Es más eficiente dividir el trabajo por hacer en trozos que puede realizar cada hilo independiente de los demás.
- Y solamente al final se une los resultados individuales.

Multiplicación concurrente correcto y eficiente

Usamos variables privados dentro de los trabajadores:

```
class Mul implements Runnable {
  private final int id;
  private final int n;
  private int p;
  private int q;
  private Int r;

  Mul(int id, int n, Int p, Int q, Int r) {
    this.id=id;
    this.n=n;
    this.p=p.Get();
    this.q=q.Get()/n;
    this.r=r;
  }
```

Multiplicación concurrente correcto y eficiente

Ejecutamos el bucle central con las variables locales, solamente al final se suma a la variable compartida (con exclusión mutua).

```
public void run() {
  try {
      // Here, we run on local variables.
    int local_r=0;
    while(q>0) {
      local_r+=p;
      --q;
    }
    final Int My_r=new Int(local_r);
    r.Add(My_r);
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
}
```

Multiplicación concurrente correcto y eficiente

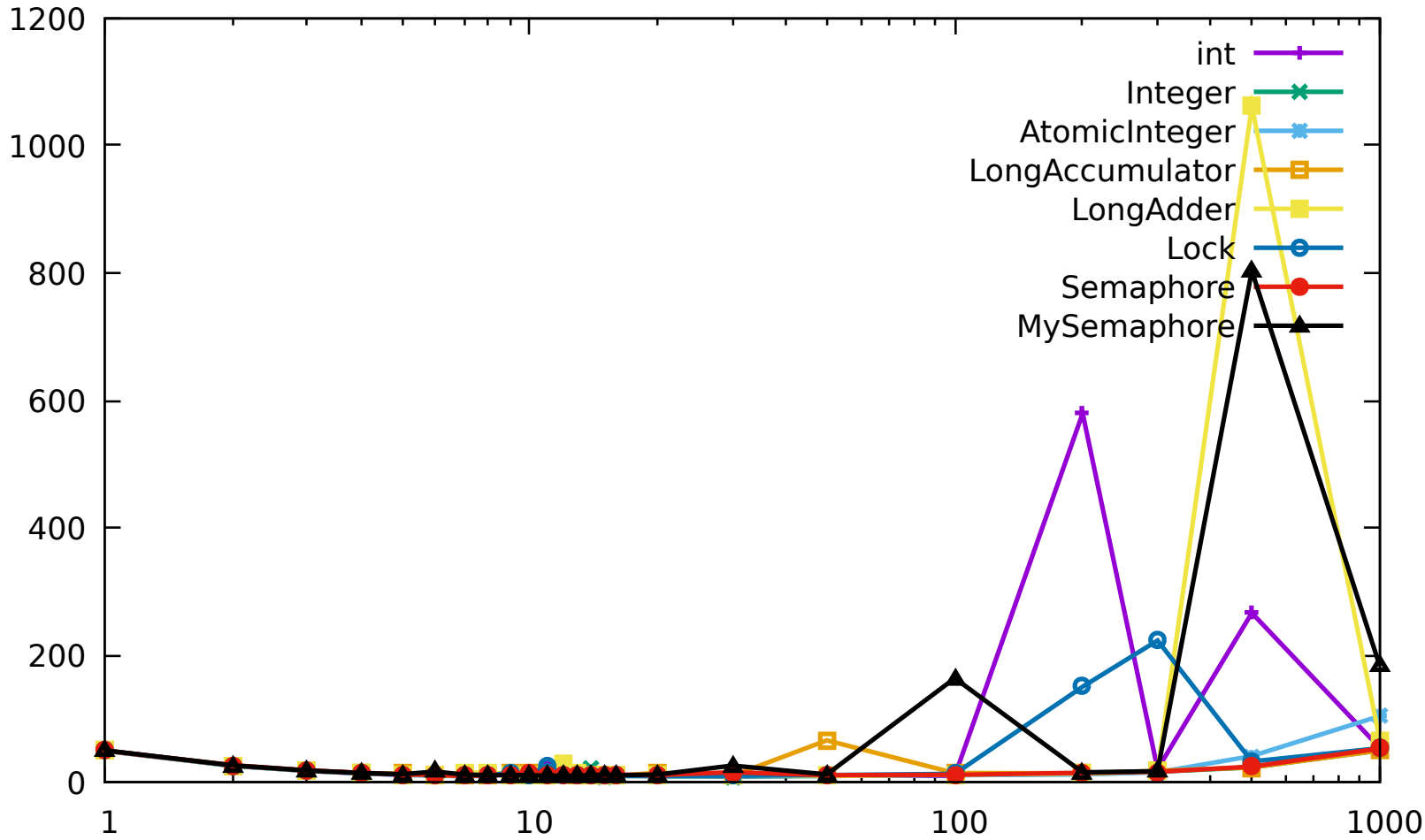
El hilo principal ayuda con el trabajo no distribuido:

```
for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}
  // Here we help with the left-overs.
int local_p=p.Get();
int local_q=q.Get()%n;
int local_r=0;
while(local_q>0) {
  local_r+=local_p;
  --local_q;
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
final Int My_r=new Int(local_r);
r.Add(My_r);
```

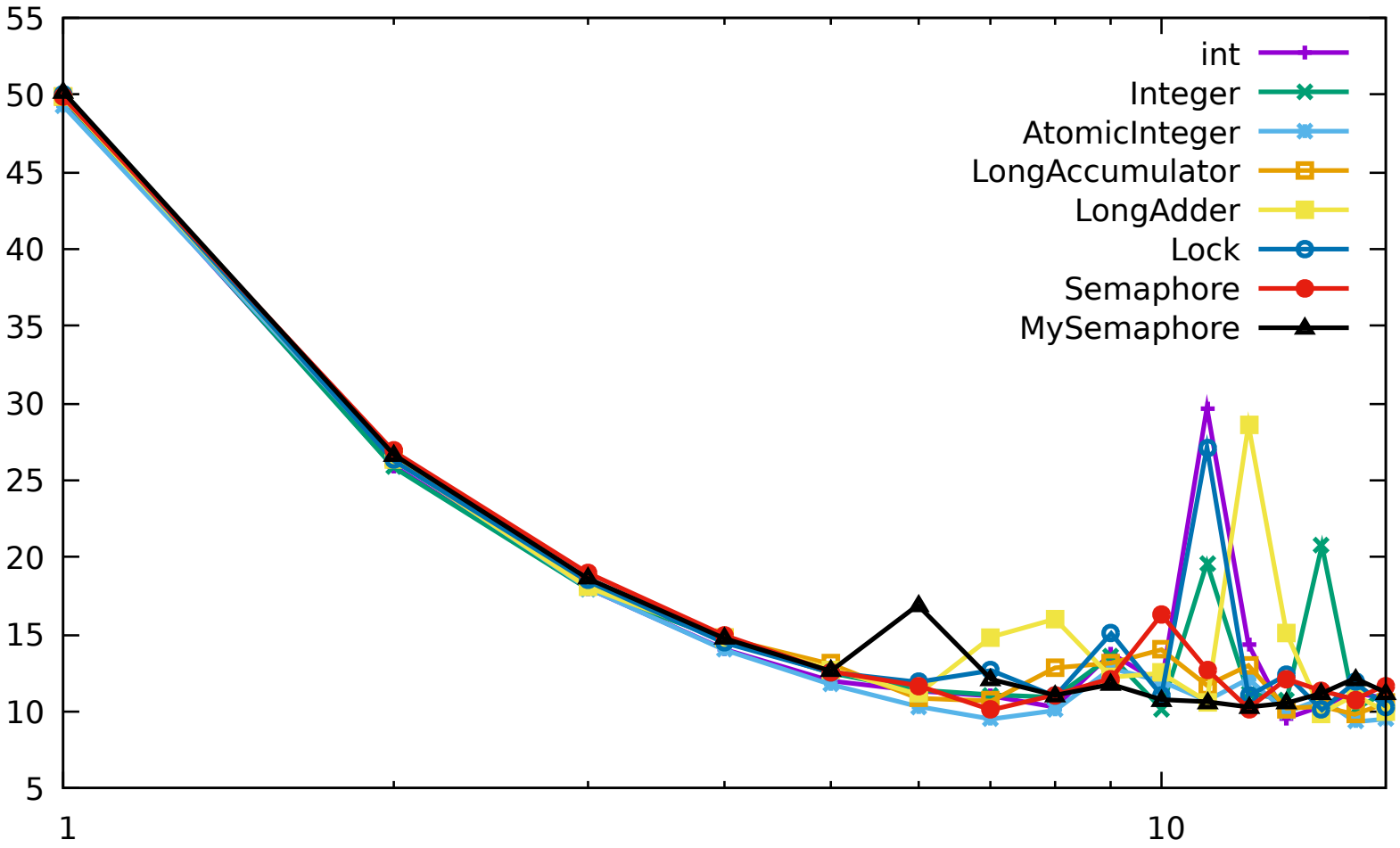
Multiplicación concurrente correcto y eficiente (variando #hilos)

Ahora hay mucho menos congestión... (unos 1000 veces más rápido, ¡aumentamos q por un factor de 500!)



Multiplicación concurrente correcto y eficiente (variando #hilos)

Inspección de zona con pocos hilos:



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.

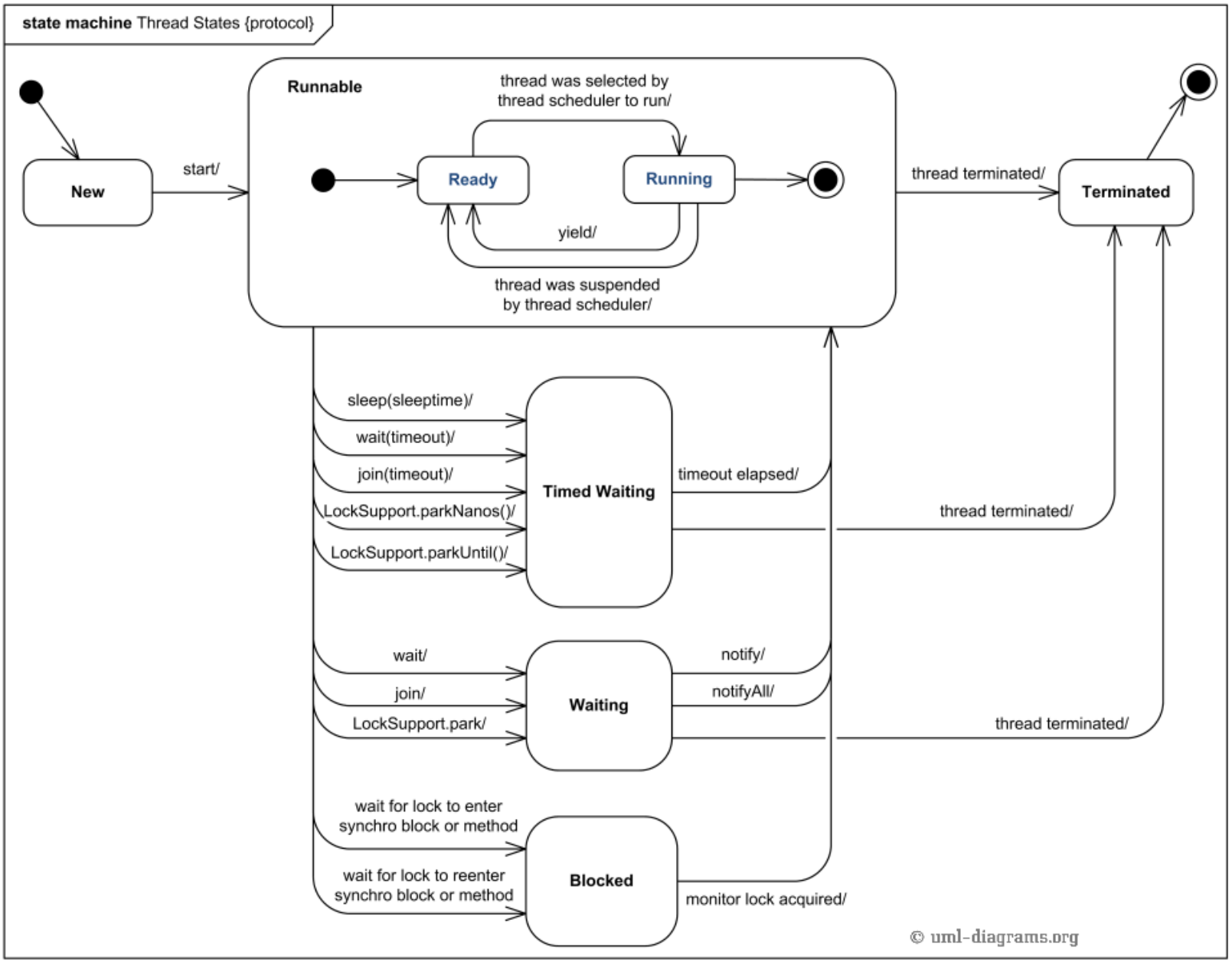
disponibilidad de monitores

- 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



instrucciones básicas

- Obviamente tenemos que asumir que ciertas acciones de un proceso se pueden realizar correctamente independientemente de las acciones de los demás procesos.
- Dichas acciones se llaman (también) atómicas (porque son indivisibles) y se garantizan por hardware.
- Normalmente, asumimos que podemos acceder a variables de cierto tipo (p.ej. enteros) de forma atómica con lectura y escritura (`load` y `store`)

Implementamos varios protocolos de exclusión mutua (aquí solamente para dos procesos) que solamente están basadas en instrucciones simples.

Un posible protocolo (asimétrico)

```
  P0                                  P1
a: loop                                loop
b:  non-critical section                non-critical section
c:  set v0 to true                      set v1 to true
d:  wait until v1 equals false          while v0 equals true
e:                                       set v1 to false
f:                                       wait until v0 equals false
g:                                       set v1 to true
h:  critical section                    critical section
i:  set v0 to false                     set v1 to false
j: endloop                             endloop
```

principio (teorema) de la bandera

El principio de la bandera es un teorema que podemos usar para comprobar si está garantizado la exclusión mutua para dos procesos en un código concreto.

- Si dos procesos primero levantan sus banderas
- y después miran al otro lado
- por lo menos uno de los procesos ve la bandera del otro levantado.

comprobación con contradicción

- asumimos P0 era el último en mirar
- entonces la bandera de P0 está levantada
- asumimos que P0 no ha visto la bandera de P1
- entonces P1 ha levantado la bandera después de la mirada de P0
- pero P1 mira después de haber levantado la bandera
- entonces P0 no era el último en mirar

propiedades de interés del protocolo

Normalmente, un protocolo de entrada y salida debe cumplir con las siguientes condiciones:

- sólo un proceso debe obtener acceso a la sección crítica (garantía del acceso con exclusión mutua)
- por lo menos un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finito*.
- Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito.

propiedades de interés del protocolo

La propiedad de espera finita se puede analizar según los siguientes criterios:

justicia:
: hasta que medida influyen las peticiones de los demás procesos en el tiempo de espera de un proceso

espera:
: hasta que medida influyen los protocolos de los demás procesos en el tiempo de espera de un proceso

tolerancia a fallos:
: hasta que medida influyen posibles errores de los demás procesos en el tiempo de espera de un proceso.

análisis del protocolo asimétrico

Analizamos el protocolo de antes respecto a dichos criterios:

- ¿Está garantizado la exclusión mutua?
- ¿Influye el estado de uno (sin acceso) en el acceso del otro?
- ¿Quién gana en caso de peticiones simultáneas?
- ¿Qué pasa en caso de error?

soporte hardware

- Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más fácil o más difícil, además las soluciones pueden ser más o menos eficientes.
- Vimos que se pueden realizar protocolos seguros solamente con las instrucciones `load` y `store` de un procesador.
- Las soluciones no suelen ser muy eficientes, especialmente si muchos procesos compiten por la sección crítica. *Pero: su desarrollo y la presentación de la solución ayuda en entender el problema principal.*
- A veces no hay otra opción disponible.
- Todos los microprocesadores modernos proporcionan instrucciones básicas que permiten realizar los protocolos de forma más directo y en muchas ocaciones más eficiente.

protocolo simple: primer intento

Usamos una variable `v` que nos indicará cual de los dos procesos tiene su turno.

```
  P0                                P1
a: loop                              loop
b:  wait until v equals P0            wait until v equals P1
c:  critical section                  critical section
d:  set v to P1                       set v to P0
e:  non-critical section              non-critical section
f: endloop                           endloop
```

primer intento: propiedades

- Está garantizada la exlusión mutua porque un proceso llega a su línea `c:` solamente si el valor de `v` corresponde a su identificación (que asumimos siendo única).
- Obviamente, los procesos pueden acceder al recurso solamente alternativamente, que puede ser inconveniente porque acopla los procesos fuertemente.
- Un proceso no puede entrar más de una vez seguido en la sección crítica.
- Si un proceso termina el programa o no llega más por alguna razón a su línea `d:`, el otro proceso puede resultar bloqueado.
- La solución se puede ampliar fácilmente a más de dos procesos.

primer intento en Java (comenzar es fácil)

El protocolo del primer intento para implementar un ping pong.

```
public class PingPong {
  volatile static int turn;  // It's v.

  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);

    ping1.start();
    ping2.start();

    turn=1;
  }
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

un método para abreviar:

```
  static void Wait(int us) {
    try {
      Thread.sleep(us);
    } catch(InterruptedException e) {
      System.out.println("sleeping interrupted");
    }
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
public class PingPong {
  volatile static int turn;  // It's v.
  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);
    ping1.start();
    ping2.start();
    System.out.println("playing some seconds");
    turn=1;
    Wait(2000);
    System.out.println("waiting for players");
    turn=3;  // Try to stop :-)
    try {
      ping1.join();
      ping2.join();
    }
    catch(InterruptedException e) {
      System.out.println("got interrupted");
    }
    System.out.println("finished");
  }
```

primer intento en Java (comenzar es fácil)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.Wait(100);        // And blocking!!!
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

segundo intento

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indican si el correspondiente proceso está usando el recurso.

```
  P0                                  P1
a: loop                                loop
b:  wait until v1 equals false          wait until v0 equals false
c:  set v0 to true                      set v1 to true
d:  critical section                    critical section
e:  set v0 to false                     set v1 to false
f:  non-critical section                non-critical section
g: endloop                             endloop
```

segundo intento: propiedades

- Ya no existe la situación de la alternancia.
- Sin embargo: el algoritmo no es correcto, porque los dos procesos pueden alcanzar sus secciones críticas simultáneamente.

- El problema está escondido en el uso de las variables de control. `v0` se debe cambiar a verdadero solamente si `v1` sigue siendo falso.
- ¿Cuál es la intercalación maligna?

tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
  P0                                P1
a: loop                              loop
b:  set v0 to true                    set v1 to true
c:  wait until v1 equals false        wait until v0 equals false
d:  critical section                  critical section
e:  set v0 to false                   set v1 to false
f:  non-critical section              non-critical section
g: endloop                           endloop
```

tercer intento: propiedades

- Está garantizado que ambos procesos no entren al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intenten simultáneamente que resultaría en una espera infinita.

- ¿Cuál es la intercalación maligna?

cuarto intento

Modificamos la instrucción `c:` para dar la oportunidad que el otro proceso encuentre su variable a favor.

```
  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  repeat                         repeat
d:   set v0 to false                set v1 to false
e:   set v0 to true                 set v1 to true
f:  until v1 equals false          until v0 equals false
g:  critical section               critical section
h:  set v0 to false                set v1 to false
i:  non-critical section           non-critical section
j: endloop                        endloop
```

cuarto intento: propiedades

- Está garantizado la exclusión mutua.
- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)

- ¿Cuál es la intercalación maligna?

algoritmo de Dekker: quinto intento

```
Initially:  v0,v1 are equal to false, v is equal to P0 o P1

  P0                              P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  loop                           loop
d:   if v1 equals false exit        if v0 equals false exit
e:   if v equals P1                 if v equals P0
f:    set v0 to false                set v1 to false
g:    wait until v equals P0         wait until v equals P1
h:    set v0 to true                 set v1 to true
i:   fi                             fi
j:  endloop                        endloop
k:  critical section               critical section
l:  set v0 to false                set v1 to false
m:  set v to P1                    set v to P0
n:  non-critical section           non-critical section
o: endloop                        endloop
```

quinto intento: propiedades

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos, donde se asume que la lectura y la escritura de un valor íntegro de un registro se puede realizar de forma atómica.