20151030

dit
UPM

Programación concurrente – Fiabilidad

Juan Antonio de la Puente <jpuente@dit.upm.es>

ETSIT UPM

Referencias

- Scott Oaks & Henry Wong
  Java Threads
  O'Reilly Media; 3rd ed (2004)

- Kathy Sierra & Bert Bates
  Head First Java, ch. 15
  O'Reilly Media; 2nd ed (2005)

Propiedades de los programas concurrentes

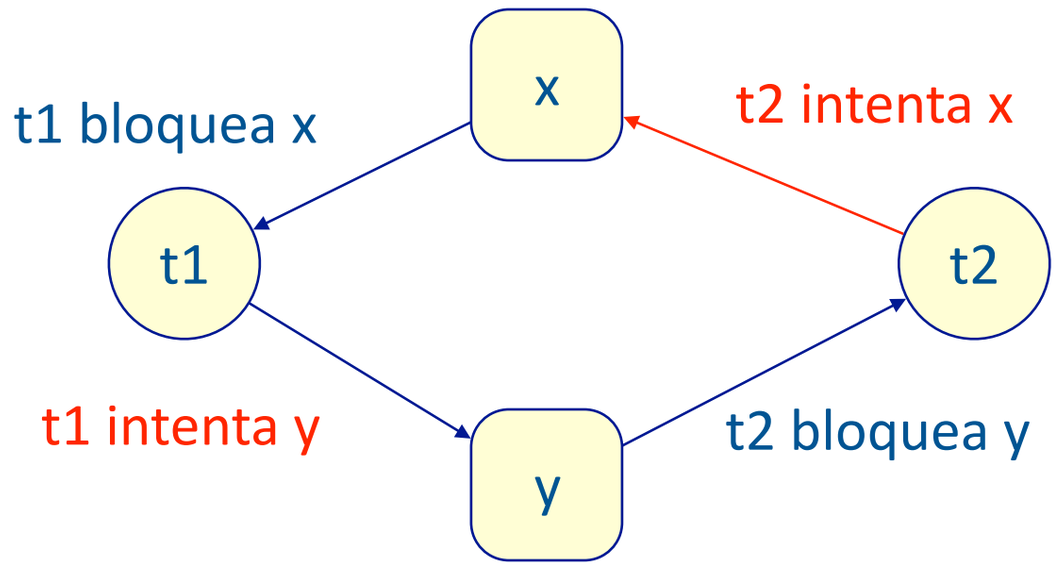
Propiedades de los programas concurrentes

- Corrección (*correctness*)
  - los resultados de los cálculos son correctos
- Seguridad (*safety*)
  - «nunca suceden cosas malas»
- Vivacidad (*liveness*)
  - «en algún momento ocurrirá algo bueno»
- Equidad (*fairness*)
  - ninguna hebra se ve perjudicada por el progreso de las otras

Interbloqueos

Interbloqueos (*deadlocks*)

- Situación en la que varias hebras están suspendidas esperando unas a otras y ninguna puede avanzar
  - es un problema de seguridad
- Se puede dar cuando se usan recursos de forma exclusiva y se asignan a distintas hebras
  - se dice que hay espera circular



Ejemplo

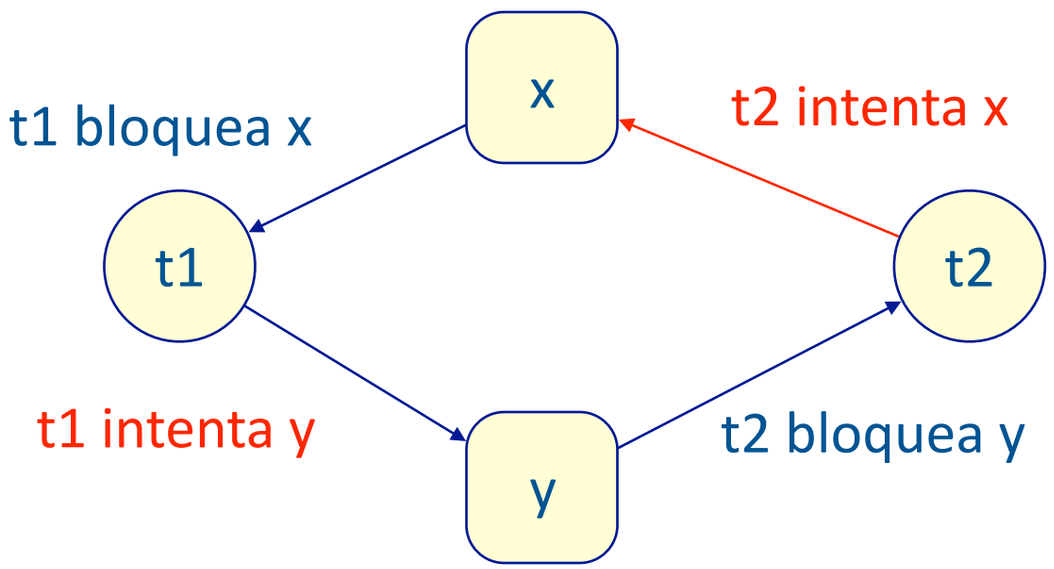
```
// hebra 1
…
synchronized(X){          // acceso exclusivo a x
  synchronized(y){…}      // acceso exclusivo a y
}
…
```

```
// hebra 2
…
synchronized(y){          // acceso exclusivo a y
  synchronized(x){…}      // acceso exclusivo a x
}
…

// también puede pasar con monitores
```

Condiciones de Coffman

- Condiciones necesarias para que se produzca un interbloqueo
  - Exclusión mutua en el uso de recursos
  - Tener y esperar: una hebra tiene recursos y los mantiene mientras espera conseguir otros
  - Sin expulsión: no se puede quitar un recurso a una hebra
  - Espera circular: hay un conjunto de hebras con recursos, esperando por otros recursos, formando una cadena circular



Cómo prevenir interbloqueos

- Los interbloqueos no se pueden detectar mediante pruebas (tests)
  - sólo ocurren de vez en cuando
  - un programa puede pasar las pruebas y dar problemas más tarde

- Para prevenir interbloqueos hay que evitar que se dé alguna de las condiciones necesarias
  - La más sencilla es evitar la espera circular es acceder a los recursos siempre en el mismo orden
    - requiere disciplina por parte de los programadores
    - ponerse de acuerdo en seguir algunas reglas

Ejemplo

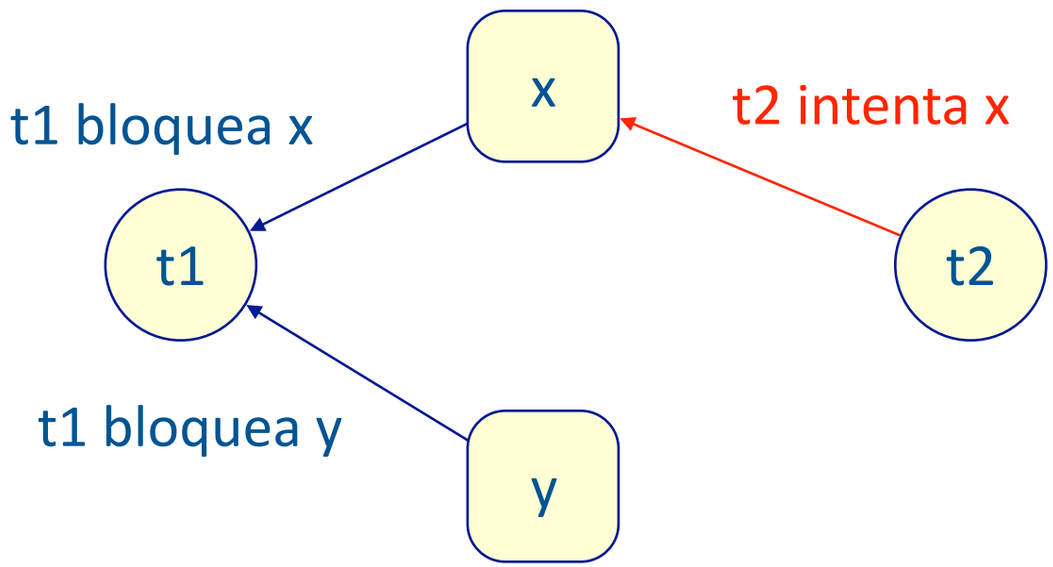
```
// hebra 1

synchronized(x){          // acceso exclusivo a x
  synchronized(y){…}      // acceso exclusivo a y
}

// hebra 2

synchronized(x){          // acceso exclusivo a x
  synchronized(y){…}      // acceso exclusivo a y
}
```

¡Ahora no hay interbloqueo!



Bloqueos vivos e inanición

Bloqueos “vivos”

- Podemos intentar evitar interbloqueos invalidando la condición de “tener y esperar”

- Ejemplo

```
enum Estado {LIBRE, OCUPADO}
estado recurso[] = {Estado.LIBRE, Estado.LIBRE};
boolean success = false;
```

Ejemplo (continuación)

```
while (!success) {
   while (true) {                              // adquirir recurso 0
      synchronized(recurso[0]) {
         if (recurso[0] == Estado.LIBRE) {
             recurso[0] = Estado.OCUPADO;
             break;
         }
      }
   }

   synchronized (recurso[1]) {                 // adquirir recurso 1
      if (recurso[1] == Estado.LIBRE) {
          recurso[1] = Estado.OCUPADO;
         success = true;
      } else {                                 // devolver recurso 0
         synchronized (recurso[0]) {
            recurso[0] = Estado.LIBRE;
         }
      }
   }
}
```

Análisis

- Si hay mala suerte, uno de los procesos no conseguirá nunca los recursos: inanición (*starvation*)
- Si hay peor suerte, ninguno conseguirá los recursos: bloqueo “vivo” (*livelock*)
- Siempre que se busca evitar el interbloqueo, hay que tener mucho cuidado de no meterse en estos problemas

Resumen

Resumen

- El uso de recursos compartidos puede causar problemas difíciles de detectar
  - Interbloqueo (*deadlock*) — problema de *seguridad*
    - evitar espera circular y asignar recursos de forma ordenada
  - Bloqueo vivo (*livelock*) — problema de *vivacidad*
  - Inanición (*starvation*) — problema de *equidad*