http://cs.LTH.se/EDA040

Real-Time and Concurrent Programming

Lecture 3 (F3):

More on concurrency and semaphores.

Klas Nilsson

2014-09-16

1. More about mutual exclusion
2. More about semaphores
3. More about Lab1

Mutual exclusion - without system calls?

```
class T extends Thread {
  public void run() {
    while (true) {
      nonCriticalSection();
      preProtocol();
      criticalSection();
      postProtocol();
    }
  }
}
```


```
class T extends Thread {
  public void run() {
    while (true) {
      nonCriticalSection();
      preProtocol();
      criticalSection();
      postProtocol();
    }
  }
}
```


Critical Section (CS)

- Like the three lines of code from the bank account example.
- We will concentrate on the construction of pre/postProtocol.
- Assumption: A thread will not block inside its critical region.
- Requirements:
  Mutual exclusion, No deadlock, No starvation, and Efficiency.

Required Mutex Properties

R1. Mutual exclusion: Execution of code in critical sections must not be interleaved.

R2. No deadlock: If one or more threads tries to enter a CS, one must do so eventually.

R3. No starvation: A thread must be allowed to enter its CS eventually.

R4. Efficiency: Small overhead when only one active thread.

Can that be accomplished by ordinary (Java) code?

Mutual exclusion – version 1

```
int turn=1;
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      while (turn!=1);
      CS1();
      turn = 2;
    }
  }
}
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      while (turn!=2);
      CS2();
      turn = 1;
    }
  }
}
```

R1. Mutual exclusion: OK

R2. No deadlock: OK since one of the threads can always proceed.

R3. No starvation: Alternating protocol; OK.

R4. Efficiency: Does not work for one thread only. Busy-wait; inefficient!
No good for many threads.

#: Not acceptable!

Mutual exclusion - version 2

```
int c1,c2; c1=c2=1;
```

```
class V2 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      while (c2!=1);
      c1 = 0;
      CS1();
      c1 = 1;
    }
  }
}
```

```
class V2 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      while (c1!=1);
      c2 = 0;
      CS2();
      c2 = 1;
    }
  }
}
```

R1. Mutual exclusion: NO!

Fails e.g. with that interleaving →

#: Not a solution, but could work for a long time (until interrupt in pre1 or pre2)!

```
c1 = 1;
          c2 = 1;
while (c2!=1);
          while (c1!=1);
c1 = 0;
          c2= 0;
CS1();
          CS2();
```

Mutual exclusion - version 3

```
int c1,c2; c1=c2=1;
```

```
class V3 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      c1 = 0;
      while (c2!=1);
      CS1();
      c1 = 1;
    }
  }
}
```

```
class V3 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      c2 = 0;
      while (c1!=1);
      CS2();
      c2 = 1;
    }
  }
}
```

R1. Mutual exclusion: OK

R2. No deadlock: Fails while also using the CPU! E.g.:

```
c1 = 0;
    c2 = 0;
while (c2!=1);      // Forever ..
    while (c1!=1); // ..and ever.
....
    ....
```

#: Not a solution, but could work for a long time!

Mutual exclusion - version 4

```
int c1,c2; c1=c2=1;
```

```
class V4 extends Thread {
  //..
      nonCS1();
      c1 = 0;
      while (c2!=1){
        c1 = 1;   //**
        c1 = 0;
      }
      CS1();
      c1 = 1; //..
}
```

```
class V4 extends Thread {
  //..
      nonCS2();
      c2 = 0;
      while (c1!=1){
        c2 = 1;   //**
        c2 = 0;
      }
      CS2();
      c2 = 1; //..
}
```

R1. Mutual exclusion: OK (as for V3).

R2. No deadlock: OK (yield at //**).

R3. No starvation: Failure, a thread may execute but never get the resource (called Livelock; threads neither block progress).

#: Not acceptable!

```
c1 = 0;
    c2 = 0;
    while (c1!=1);
    c2 = 1;
while (c2!=1);
CS1();
c1 = 1;
nonCS1();
```

```
c1 = 0;
    c2 = 0;
    while ..
    c2 = 1;
while (c2!=1);
CS1();
c1 = 1;
nonCS1();
```

Dekkers Algorithm

```
int c1,c2,turn; c1=c2=turn=1;
```

```
class DA1 extends Thread {
  //..
    nonCS1();
    c1 = 0;
    while (c2!=1){
      if (turn==2){
        c1 = 1;
        while (turn==2);
        c1 = 0;
      }
    }
    CS1();
    c1 = 1;
    turn = 2;
  //..
```

```
class DA2 extends Thread {
  //..
    nonCS2();
    c2 = 0;
    while (c1!=1){
      if (turn==1){
        c2 = 1;
        while (turn==1);
        c2 = 0;
      }
    }
    CS2();
    c2 = 1;
    turn = 1;
  //..
```

R1. Mutual exclusion: OK

R2. No deadlock: OK.

R3. No starvation: OK.

R4. Efficiency: Not good!

#: Dekkers Algorithm (can be extended to many threads, but gets very complex) solves the mutex problem, but with busy-wait (CPU used also when nothing to do). Useful in some multi-processor systems.

Mutual exclusion – semaphore

```
MutexSem mutex = new MutexSem();
```

```
class M1 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      mutex.take();
      CS1();
      mutex.give();
    }
  }
}
```

```
class M2 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      mutex.take();
      CS2();
      mutex.give();
    }
  }
}
```

R1. Mutual exclusion: OK

R2. No deadlock: OK.

R3. No starvation: OK (give starts blocked thread directly).

R4. Efficiency: Works well also for a single thread, waiting threads are put to sleep (not using any CPU time).

#: Acceptable!

Test-and-Set

The problem in version 2 arose since the following is not atomic:

```
while (c2!=1) // Load
c1 = 0;       // Store
```

All computers have an instruction that corresponds to TestAndSet which performs both these instruction atomically. It stores a new value and returns the old value:

```
while (TestAndSet(c,0)==0) ;
CS();
c = 1;
```

- A simple solution is thus possible assuming hardware support.
- Still Busy-wait – inefficient, the waiting thread should be blocked.
- Useful for machines with several CPUs and shared memory.
- In recent JDKs there are `compareAndSet`-methods that implement Test-and-Set for built-in datatypes, as part of the package `java.util.concurrent.atomic`

1 More about mutual exclusion

2 More about semaphores

3 More about Lab1

Variants of Semaphores

Blocked-Set Semaphore

Give - wakes arbitrary waiting thread.

- Starvation when N≥3 if two threads happen to alternate.

Blocked-Queue Semaphore

Give wakes threads in FIFO order (the longest waiting thread first)

- Starvation impossible

Blocked-Priority Semaphore

Give wakes the thread that has the highest priority (FIFO order when equal)

- Starvation possible if N≥3 and two high priority threads, but that is desirable!

Binary Semaphore

- Efficient mutex-implementation in some RTOS, see the `BinarySem` class.

Multistep Semaphore

- To reserve several resources at once/atomically see the `MultistepSem` class.

Semaphore classes in LJRT

UNDER CONSTRUCTION

Binary Semaphore

- Efficient mutex-implementation in some RTOS, see the `BinarySem` class.

Multistep Semaphore

- To reserve several resources at once/atomically see the `MultistepSem` class.

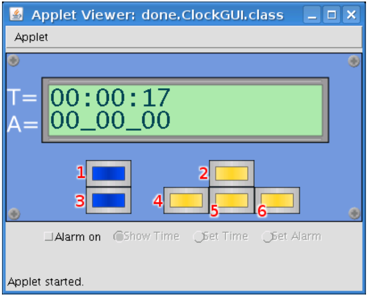
1 More about mutual exclusion

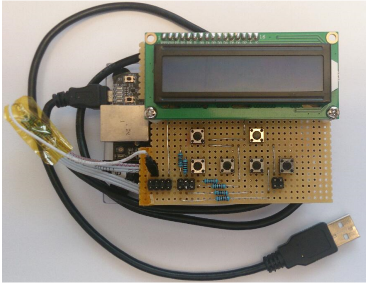
2 More about semaphores

3 More about Lab1

Hardware and emulation of it

UNDER CONSTRUCTION





Hardware-Software

UNDER CONSTRUCTION;
Shown on white/black-board during lecture

Your application program

UNDER CONSTRUCTION;
Shown on white/black-board during lecture