http://cs.LTH.se/EDA040

# Real-Time and Concurrent Programming

Lecture 3 (F3):

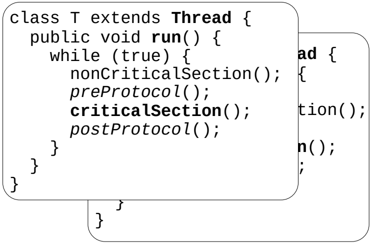
More on concurrency and semaphores.

Klas Nilsson

2016-09-13

## 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();
```

http://en.wikipedia.org/wiki/Dekker's_algorithm

## 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.

http://en.wikipedia.org/wiki/Semaphore_(programming)

## 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`

## Variants of Semaphores

Blocked-Set Semaphore

Give - wakes arbitrary waiting thread.

- Starvation when N$\geq$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$\geq$3 and two high priority threads, but that is desirable!

Binary Semaphore

- Uses a boolean instead of a counter internally. E.g., for asymmetric signaling (for quicker catch-up).

Multistep Semaphore

- To reserve several resources at once/atomically, i.e., getting all of nothing in one operation.

## Semaphore classes in LJRT

The `Semaphore` interface is implemented by the following classes:

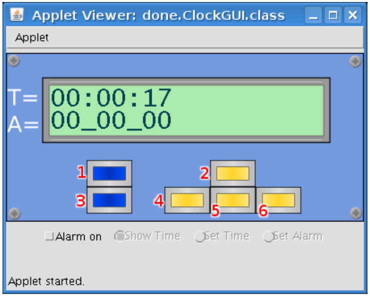
Counting Semaphore
: Classical counting semaphore, for signaling, see the `CountingSem` class.

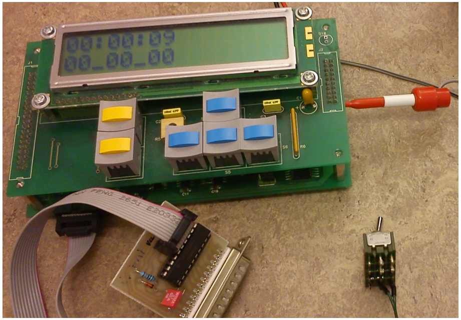
Binary Semaphore
: Efficient mutex-implementation in some RTOS, see the `BinarySem` class.

Mutex (Semaphore)
: Efficient mutex-implementation in some RTOS, see the `BinarySem` class.

Multistep Semaphore
: To reserve several resources at once/atomically see the `MultistepSem` class.

## Hardware and emulation of it





## Design considerations

- What threads, how many, blocking on what? Reflect concurrency (requirements) of the environment: One thread per independent sequence of input events.
- Any outputs that are asynchronous with all other threads, such that additional threads are needed?
- Data hidden locally in objects, but what about shared data?
- Operations on shared data, from an application point of view?
- Logic and operations that are not shared; where to place? Inside threads och as separate classes?
- Finite State machines (FSM): Defines behavior of the object or system. E.g. UML stateChart for FSM within (active or passive) object, while activity diagrams defines the interplay.

## Your application program (threads)

- What sequences of input data/events are present? The occurring order is unknown but it is known that appearance is sequential. Example: The clock buttons, will appear as some sequence of input events.
- For each such sequence, are input events occurring sparadically or periodically. How to block a receiving thread until data is available (to maintain a reactive system)?
- For the output (display and beep), is desired change synchronous with any of the input sequences?
  - If so, for some part of the output, should output be synchronous with input handling such that is can performed by the same thread of execution, or are there reasons for having separate output thread(s)?
  - If not, what threads are needed for handling the output, and what are they blocking on?
- Any other asynchronous activities needed?
- What threads are motivated by the above investigation?

## Your application program (shared data)

- What data needs to be shared between the asynronous activities (the threads according to previous page), i.e., at some point acessed by more than one thread?
- Shared data to be placed in a class with mutex protection;
  public methods the only proper way to manipulate data, and the content of all those methods should form critical sections using one and the same internally declared `MutexSem`
- To limit the number of execution combinations (avoiding concurrency faults), methods should have application meaning, and be as large and few as possible without adding complexity.
- Only one mutex per shared-data object!

## Mutual exclusion as part of interface

### In-line use of semaphores for mutual exclusion

Disadvantage: `take`/`give` tends to get spread out through the entire program (learned from exercise 1).

### Abstract data-types for mutual exclusion

Principle: `take`/`give` part of (mutually exclusive) methods that are kept together with the hidden data.

Monitor: Such a data-type with mutually exclusive access-functions is called a *Monitor*.

## Monitors (objects & concept)

- In OOP we use classes as a (more powerful) mean to accomplish abstract data-types.
- Objects with such mutually exclusive methods are then monitor objects.

For a class like `Account`:

```
class Account {
  // ...
  void deposit(int a){
    mutex.take();
    balance += a;
    mutex.give();
  }
}
```

the monitor concept is implemented by using semaphores.