Universitetet i Oslo
Institutt for Informatikk
PMA
Einar Broch Johnsen, Martin Steffen

INF 4140: Models of Concurrency

Høst 2014 — **Series 4** — 26. 9. 2014

Topic Monitors

Issued: 26. 9. 2014

Exercise 1 (From the book) Solve: 5.2, 5.3, 5.7, 5.8

Remark: it may be hard to get all the signaling details of 5.7b) right. However, you should try to describe the waiting conditions for each kind of process.

For 5.8c): Only describe the changes you need to do, any actual programming is not necessary. Do you need any additional data-structure? If yes, how should this structure be manipulated?

Exercise 2 (Monitor solution to the Readers/writers problem) This monitor is used to control reader- and writer access to a shared resource. (fig. 5.5 in Andrews)

```
monitor RW_Controller {
  int nr := 0, nw = 0;  ## (nr = 0 OR nw = 0) AND nw <= 1
  cond oktoread;    #  signaled when nw = 0
  cond oktowrite;   #  signaled when nr = 0 and nw = 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr := nr + 1;
  }
  procedure release_read() {
    nr := nr - 1;
    if (nr = 0) signal(oktowrite);  # awaken one writer
  }
  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw := nw + 1;
  }
  procedure release_write() {
    nw := nw - 1;
    signal(oktowrite);      # awaken one writer and
    signal_all(oktoread);   # all readers
  }
}
```

You may assume that signaling is handled by the *signal and continue* discipline.

1. In the monitor the primitive `signal_all` is used. Modify the monitor so that it uses `signal`.

2. In the given monitor readers take presence over writers. Modify the monitor such that writes take presence over readers.

   Someone comes up with the following modified versions of `request_read` and `release_write` to solve this problem:

   ```
   procedure request_read() {
     while (nw > 0 || !empty(oktowrite)) wait(oktoread);
     nr := nr + 1;
   }

   procedure release_write() {
     nw := nw - 1;
     if (!empty(oktowrite) signal(oktowrite);
      else signal_all(oktoread);
   }
   ```

   Even though it may seem like a straight forward solution, it does not guarantee preference to writers. (Try to imagine why before continue reading.)

   Consider the situation where exactly one writer process is delayed on `oktowrite` when another writer starts execution of `release_write`. After `signal(oktowrite)`, the waiting writer is moved back into the entry queue of the monitor. Now, `empty(oktowrite)` is **true** and `nw = 0`. Thus, a newly arriving reader is given access to the shared resource.

   Thus, `!empty(oktowrite)` is *not* a sufficient condition to guarantee absence of waiting writers. In this and the remaining parts of the exercise, we will therefore follow the outline from exercise 5.7 and use counters to count the number of delayed processes.

3. Modify the monitor so that readers and writers is allowed to access the resource in turns, if both readers and writers want to access the resource.

4. Modify the monitor such that both readers and writers access the resource in a first-come-first-served (FCFS) manner. Allow more than one reader to access the resource as long as the FCFS-principle is satisfied.

Assume given a (FIFO) queue `q` with the following operations; `enqueue(q,X)` returns `q` with the element `X` added at the end of the queue. The operation `dequeue(q)` returns `q` with the first element removed and `inspect(q)` returns the first element of the queue without altering `q`. This queue will be used to order the processes. The operation `empty(q)` returns true only when `q` is empty, and an empty queue is declared by the statement

```
queue q := empty
```

Exercise 3 (Additional exercise: cigarette smokers) As an extra challenge, you may try to solve the Cigarette Smokers Problem, Exercise 4.27 in Andrews. This is a surprisingly hard problem.

You might take the following discussion as a starting point. First we model the agent.

```
# Initially, the agent is ready to put ingredients on the table
# This semaphore is used to make the agent wait for a smoker to finish
sem go := 1;

# These are one if the corresponding ingredience is on the table
sem tobacco := 0, paper := 0, match := 0;

process Agent {
  co
    while (true) {
      P(go); V(tobacco); V(paper);
    }
  ||
    while (true) {
      P(go); V(tobacco); V(match);
    }
  ||
    while (true) {
      P(go); V(paper); V(match);
    }
  co
}
```

A first attempt to model the smokers might be something like this (the process called `Match` is the one needing matches and so on).

```
process Match {
  while (true) {
    P(tobacco);
    P(paper);
    # make cigarette
    V(go);
  }
}
```

```
process Tobacco
  while (true) {
    P(paper);
    P(match);
    # make cigarette
    V(go);
  }
}
```

```
process Paper
  while (true) {
    P(tobacco);
    P(match);
    # make cigarette
    V(go);
  }
}
```

However, this solution has serve deadlock problems. For instance, if `tobacco` and `paper` is on the table, the process `Match` should make a cigarette. However the `Paper` process may pick up the tobacco before `Match`, leading to a deadlock.

Notice that the agent is only allowed to communicate with the smokers through the four given semaphores. It is therefore no solution to add three other semaphores used to announce which ingredient the agent did *not* put on the table.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.