Solutions for CP Exercises, September 29

1. Solution for Andrews Ex. 5.4

Given Figure 5.5 in [Andrews] (here in our notation):

monitor $RW\_Controller$:

  var nr, nw : $integer$:= 0;
    $oktoread$: $condition$;
    $oktowrite$: $condition$;

  procedure $request\_read()$
    while $nw > 0$ **do** $wait(oktoread)$;
    $nr := nr + 1$;

  procedure $release\_read()$
    $nr := nr - 1$;
    if $nr = 0$ **then** $signal(oktowrite)$

  procedure $request\_write()$
    while $nr > 0 \vee nw > 0$ **do** $wait(oktowrite)$;
    $nw := nw + 1$;

  procedure $release\_write()$
    $nw := nw - 1$;
    $signal(oktowrite)$
    $signal\_all(oktoread)$

end

(a) The *signal_all* operation can be replaced with repeated signalling:

  while $\neg empty(oktoread)$ **do** $signal(oktoread)$;

Alternatively, *cascaded wakeup* can be used. Then $signal\_all(oktoread)$ is replaced by a single $signal(oktoread)$, and *request_read* becomes:

procedure $request\_read()$
  while $nw > 0$ **do** $wait(oktoread)$;
  $nr := nr + 1$;
  $signal(oktoread)$;

Cascaded wakeup is especially useful in situations where the number processes to be awakened is not known in advance, eg. may depend on parameters of the woken processes.

(b) To give preference to writers, readers should be held back if there are any pending writers. Likewise, writers should be favoured after a writing phase. Thus, $request\_read()$ and $release\_write()$ are modfied to:

procedure *request_read*()
  while $nw > 0 \vee \neg empty(oktowrite)$ **do** $wait(oktoread)$;
  $nr := nr + 1$;

procedure *release_write*()
  $nw := nw - 1$;
  if $\neg empty(oktowrite)$ **then** $signal(oktowrite)$
  else $signal\_all(oktoread)$

end

(c) A solution which is fair towards both readers and writer can be obtainted by running a batch of readers after each writer (if possible):

procedure *request_read*()
  if $\neg empty(oktowrite)$ **then** $wait(oktoread)$;
  while $nw > 0$ **do** $wait(oktoread)$;
  $nr := nr + 1$;

procedure *release_write*()
  $nw := nw - 1$;
  if $\neg empty(oktoread)$ **then** $signal\_all(oktoread)$
  else $signal(oktowrite)$

end

Here the **if**-statemente in *request* ensures fairness by holding back readers if any writers are waiting. However, once woken, the readers just enter if possible.

[Spurious wakeups just make a reader progress to the active reader groups which may be acceptable if these wakeups are sparse.]

(d) [Advanced] In order to get a strict First-Come-First-Served discipline both readers and writers must be processed in some common entrance queue. Further, if the first process of this queue discovers that it cannot start (eg. being a writer when readers are active), it will have to wait being the first to be considered next time. For this to work two condition queues can be used: *pre* where processes queue up in FIFO order and *front* where the (single) front process of the queue waits. To determine which queue to wait at, a count *ne* of the currently entering readers/writers is maintained. Only when being the only one entering, a process it can go directly to the front position. Whenever a process leaves the front position, the next process from the *pre*-queue (if any) is moved to the front.

```
monitor FCFS_RW_Controller :

  var nr, nw, ne : integer := 0;
      pre : condition;
      next : condition;

  procedure request_read()
    ne := ne + 1;
    if ne > 1 then wait(pre);
    if nw > 0 then wait(front);
                   signal(pre);
    nr := nr + 1;
    ne := ne − 1;

  procedure release_read()
    nr := nr − 1;
    if nr = 0 then signal(next);

  procedure request_write()
    ne := ne + 1;
    if ne > 1 then wait(pre);
    if nw > 0 ∨ nr > 0 then wait(next);
                            signal(pre);
    nw := nw + 1;
    ne := ne − 1;

  procedure release_write()
    nw := nw − 1;
    signal(next);

end
```

[Due to the wait conditions not being rechecked, this solutions is not robust towards spurious wakeups.]