Solutions for Exercises, Week 7

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 ∨ 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 ¬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 in order to prevent starvation of writers. Likewise, writers should be favoured after a writing phase. Thus, *request_read*() and *release_write*() are modified to:

```
procedure request_read()
  while nw > 0 ∨ ¬empty(oktowrite) do wait(oktoread);
  nr := nr + 1;

procedure release_write()
  nw := nw − 1;
  if ¬empty(oktowrite) then signal(oktowrite)
                       else signal_all(oktoread)
```

(c) The solution below attempts to carefully alternate between readers and writers. Thus, the last reader should start a writer and an ending writer should start a group of readers. However, to prevent readers from starving writers, new readers should wait in a *prequeue* if there are writers waiting. Since all waiting readers should start together, the normal condition queue *oktoread* may be used for that purpose as well.

```
procedure request_read()
  if ¬empty(oktowrite) then wait(oktoread);
  while nw > 0 do wait(oktoread);
  nr := nr + 1;

procedure release_write()
  nw := nw − 1;
  if ¬empty(oktoread) then signal_all(oktoread)
                      else signal(oktowrite)
```

This solution, however, allows for new writers to overtake a woken writer and in theory a particular writer may be starved forever. For a truly fair solution, see (d).

It is also possible to use the general fairness technique of alternating a priority between the two groups. The priority is to be used only if both readers and writers are waiting. Here we use a boolean variable *reader_prio* indicating whether readers have priority (if not, writers have).

```
monitor Fairly_Fair_RW_Controller :

  var nr, nw : integer := 0;
      reader_prio : boolean := true;
      oktoread : condition;
      oktowrite : condition;

  procedure request_read()
    while nw > 0 ∨ (¬empty(oktowrite) ∧ ¬reader_prio) do
      wait(oktoread);
    nr := nr + 1;

  procedure release_read()
    reader_prio := false;
    nr := nr − 1;
    if nr = 0 then signal(oktowrite)

  procedure request_write()
    while nr > 0 ∨ nw > 0 ∨ (¬empty(oktoread) ∧ reader_prio) do
      wait(oktowrite);
    nw := nw + 1;

  procedure release_write()
    reader_prio := true;
    nw := nw − 1;
    if ¬empty(oktoread) then signal_all(oktoread)
                        else signal(oktowrite)
end
```

Here the priority is changed when (the first of) a group ends its operation. Again, in theory readers may still be starved, if they do not all get out of *request_read* before the first of the reader group changes the priority. However, in practice this would probably not be an issue if reading is a longer-lasting operation.

(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.]

2. Solution for Andrews Ex. 5.8

(a) The required invariant must state that the balance never becomes negative:

$$I \triangleq Bal \geq 0$$

The basic problem in this exercise is that the waiting condition for each *withdraw*(*amount*) operation depends on the parameter value *amount*. A general solution to this is to use a *covering condition*, i.e. to wake up all waiting processes, whenever the balance has improved. It is assumed that all amounts belongs to a type of positive integers *PosInt*.

```
monitor SimpleAccount

  var Bal : integer := 0;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    signal_all(positive);

  procedure withdraw(amount : PosInt)
    while Bal < amount do wait(positive);
    Bal := Bal − amount;

end
```

(b) Under the standard assumption the the condition queues are FIFO, the customers may be served FCFS by waking only one at a time, but only as long as the balance is large enough (using the magic *amount* function). Special care must be taken to prevent outside processes from making withdrawals before the woken processes. This could be accomplished by letting the deposit operation do the balance decrementation as in the FIFO Semaphore solution shown in Andrews Figure 5.3. Here, however, we take a more general approach. Whenever a withdrawal process is woken, the monitor is considered *busy*, and new processes will have to wait. Now the processes are started in FIFO order by a cascade wakeup:

```
monitor MagicFSCSAccount

  var Bal : integer := 0;
      Busy : boolean := false;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    if ¬Busy ∧ ¬empty(positive) ∧ amount(positive) ≤ Bal then
      Busy := true;
      signal(positive);

  procedure withdraw(amount : PosInt)
    if Busy ∨ ¬empty(positive) ∨ Bal < amount then
      wait(positive);
      Busy := false;
```

```
    Bal := Bal − amount;                                  — Bal assumed large enough
    if ¬Busy ∧ ¬empty(positive) ∧ amount(positive) ≤ Bal then
      Busy := true;
      signal(positive);
end
```

Note that deposit processes may increment *Bal*, even when the monitor is busy, but that will not violate the expectations of the woken withdrawal process.

(c) In order to implement the magic function *amount* giving the requested amount of the first withdrawal process, two ideas may be applied:

- A new, separate condition queue is used by the first waiting process and that processes may set a global amount variable. Further processes will have to wait on the old queue. Now great care must be taken to ensure that exactly one and only one of the waiting processes proceed to this queue.
- Within the monitor, a datastructure is maintained giving the amounts of the waiting processes. Thus the datastructure will be parallel to the condition queue. Since the queue is supposed to be FIFO, a list type will be appropiate.

Here we choose the latter approach, extending the above solution with a list type with operations *append*, *head* and *tail*:

```
monitor FSCSAccount

  var Bal : integer := 0;
      Busy : boolean := false;
      amounts : List of integer;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    if ¬Busy ∧ ¬empty(positive) ∧ head(amounts) ≤ Bal then
      Busy := true;
      amounts := tail(amounts);
      signal(positive);

  procedure withdraw(amount : PosInt)
    if Busy ∨ ¬empty(positive) ∨ Bal < amount then
      amounts := append(amounts, amount);
      wait(positive);
      Busy := false;
    Bal := Bal − amount;                                  — Bal assumed large enough
    if ¬Busy ∧ ¬empty(positive) ∧ head(amounts) ≤ Bal then
      Busy := true;
      amounts := tail(amounts);
      signal(positive);

end
```