Solutions for Exercise Class 3

1. Solution for Mon.1

monitor *Meet*

var OK : *boolean* := *false*; — Has the the other arrived?
c : *condition*; — First one waits here

procedure $SYNC_A()$
if OK **then** $\{OK := false;\ signal(c)\}$
else $\{OK := true;\ wait(c)\}$

procedure $SYNC_B()$ — As $SYNC_A$, but coded alternatively
$OK := \neg OK$;
if $\neg OK$ **then** $signal(c)$ **else** $wait(c)$;

end

If we utilize the possibility of asking whether a condition queue is empty or not, we may eliminate the variable OK and both operations become:

procedure $SYNC_{A/B}()$
if $empty(c)$ **then** $wait(c)$ **else** $signal(c)$

There are a number of other (more complex) solution where different queues are used, where each process has a flag etc.

2. See solution to point 6. below.

3. Both are used for letting a process wait until woken up, but they are indeed different:

- Semaphores may be used anywhere. Condition queues are associated with monitors and may only be used within these.
- The *wait* operation on a condition queue *atomically* realeases the monitor while putting the calling process on the queue.
  If a `P`-operation on a semaphore is made within a monitor operation, the process will just wait on the semaphore while the monitor's critical region remains locked.
- If there are no waiting processes, a `V` operation on a semaphore is remembered by incrementing the semaphore value, while signalling an empty condition queue has no effect.

4. Assuming that the variable `b` is protected by the monitor's critical region, this will lead to a deadlock since the critical region is not released during calls of `sleep()`.

5. Based on the second solution idea in point 1. above, adding two monitor variables, *first* and *sum*, the problem may be solved by the following:

```
monitor MeetAndSum

  var first, sum : integer := 0;
      c : condition;                          — First one waits here

  function SYNC_A/B(x : integer)
    if empty(c) then {first := x;  wait(c)}
                else {sum := first + x;  signal(c)};
    return sum

end
```

Notice, that since the last process at the meeting may immediately call *SYNC* again before the first of the processes has returned, the sum must be held in a separate variable, *sum*. This variabale will not be overwritten, because the monitor is assumed to be used by two dedicated processes only.

[Since conditions are not rechecked after the wait, this solutions would not work if *spurious wakeups* could occur. But according to the standard definition (in [Andrews]) they do not.]

6. Solution for Mon.4

(a)

```
monitor Event

  var c : condition;

  procedure sleep()
    wait(c)

  procedure wakeup()
    signal_all(c)

end
```

[Note that since no conditions are checked after the wait, this solution would **not** work if *spurious wake-ups* could occur.]

(b) To prevent spurious wakeups to let threads pass *sleep()* when no *wakeup()* is called, the current state of the synchronization mechanism must be registered in the monitor. First we try to use a flag *go* in combination with a *count* of the number of waiting threads:

```
monitor Event

  var c : condition;
      count : integer := 0;
      go : boolean := false;

  procedure sleep()
    count := count + 1;
    while ¬go do wait(c);
    count := count − 1;
    if count = 0 then go := false;

  procedure wakeup()
    if count > 0 then
      go := true;
      signal_all(c)

end
```

This solution is robust towards spurious wakeups, but is not quite faithful to the specification, since calls of `sleep()` made before all waiting threads have left the monitor are allowed to pass in the same round. Depending on the actual use, this may be acceptable.

It turns out to be very difficult to control exactly which threads are woken up explicity and which accidentally by spurious wakeups.

However, a simple and precise solution is obtained by using the idea of the *ticket algorithm* where processes keep track of their position themselves.

```
monitor Event

  var c : condition;
      round : integer := 0;

  procedure sleep()
    myround : integer := round;
    while myround = round do wait(c);

  procedure wakeup()
    round := round + 1;
    signal_all(c)

end
```

For this we need an indefinite integer to count the current round. In practice though, a 64-bit `long` will suffice.

Also here, we could count the waiting threads and only increment `round` when there are threads actually waiting.