Solutions for CP Exercise Class 4

1. Solution for Andrews Ex. 4.24

The idea is that a writer should "take all n coconuts" to make sure that no readers are active. This has to be done through n single P-operations, but this could lead to deadlock if started by more than one writer. Therefore, writers first have to get the right to start this operation by entring a critical region protected by another semaphore $mutex_w$.

var rw : $semaphore$:= n;
    $mutex_w$: $semaphore$:= 1;

process $Reader[i : 1..n]$ =	**process** $Writer[j : 1..m]$ =
...	...
P(rw);	P($mutex_w$);
read the database;	**for** k **in** $1..n$ **do** P(rw);
V(rw);	read the database;
...	**for** k **in** $1..n$ **do** V(rw);
	V($mutex_w$);
	...

This solution is fair towards both readers and writers if the semaphores are strongly fair (e.g. FIFO).

2. In general, the semaphores of the Sema.3 program may be signalled when they are already 1 (see solution to point 3. below). The effect for the various kinds of binary semaphores are:

(i) $V(S)$: $\langle$ **if** $s = 0$ **then** $s := 1 \rangle$ or just $\langle s := 1 \rangle$
Here a signal may be lost with the result that two operations op_A and op_B start to alternate rather than run concurrently.

(ii) $V(S)$: $\langle s := (s + 1)$ **mod** $2 \rangle$
Here two signals may be lost with the result that the system deadlocks.

(iii) $V(S)$: $\langle s = 0 \rightarrow s := s + 1 \rangle$
In the given program, this variant just makes a fast process wait a little before it would otherwise do, so this does not influence the synchronization of the operations. In general, however, this kind of binary semaphore is more prone to deadlock. That is, you can construct more complex programs which will deadlock with this kind of semaphore but not with general semaphores.

3. Solution for Sema.3

(a) The solution to the meeting problem (see [Basic]) does not work for binary semaphores since the following execution is possible:

Process A signals S_B.

Process B signals S_A.

Process A passes $\mathtt{P}(S_A)$, executes OP_A and signals S_B.

Now, two signallings on S_B have been performed without an intermediate wait. The precise effect of this depends on the particular kind of binary semaphore and should generally be avoided.

(b) A solution with binary semaphores is obtained by (getting the idea of) interchanging `P` and `V` in one of the processes:

$$
\begin{array}{llll}
SYNC_A\colon & \mathtt{V}(S_B); \qquad\qquad & SYNC_B\colon & \mathtt{P}(S_B); \\
 & \mathtt{P}(S_A); & & \mathtt{V}(S_A);
\end{array}
$$

That this solution still ensures that the two operations do not deviate from each other is proven by using the semaphore invariants as before. Furthermore, we may show that the values of the semaphores can never exceed 1. From the program, we obtain the following inequalities:

$$
\begin{array}{ccccc}
\#\mathtt{P}(S_A) & \leq & \#\mathtt{V}(S_B) & \leq & \#\mathtt{P}(S_A)+1 \\
\#\mathtt{V}(S_A) & \leq & \#\mathtt{P}(S_B) & \leq & \#\mathtt{V}(S_A)+1
\end{array}
$$

Since S_B is initialized to 0 we have the semaphore invariant $\#\mathtt{P}(S_B) \leq \#\mathtt{V}(S_B)$. Together with the above we get:

$$\#\mathtt{V}(S_A) \leq \#\mathtt{P}(S_B) \leq \#\mathtt{V}(S_B) \leq \#\mathtt{P}(S_A)+1$$

Subtracting $\#\mathtt{P}(S_A)$ from both sides we get:

$$\#\mathtt{V}(S_A) - \#\mathtt{P}(S_A) \leq 1$$

or, as the left hand side is precisely the semaphore value s_a,

$$s_a \leq 1$$

That is, using general semaphores the value of S_A can never exceed 1. Thus, S_A may as well be implemented by a binary semaphore. Correspondingly we can show that $s_b \leq 1$.

4. Solution for Andrews Ex. 4.6

To implement the *sleep*/*wakeup* mechanism, we need a semaphore for mutual exclusion and one for suspension. A counter keeps track of the number of waiting processes.

$$
\begin{array}{llll}
\textbf{var} & S & : & \textit{semaphore} := 1; \\
 & Q & : & \textit{semaphore} := 0; \\
 & K & : & \textit{integer} := 0;
\end{array}
$$

sleep: $\mathtt{P}(S)$;
$K := K + 1$;
$\mathtt{V}(S)$
$\mathtt{P}(Q)$;
$K := K - 1$;
if $K > 0$ **then** $\mathtt{V}(Q)$
else $\mathtt{V}(S)$

wakeup: $\mathtt{P}(S)$;
if $K > 0$ **then** $\mathtt{V}(Q)$
else $\mathtt{V}(S)$

Note that a solution in which *sleep* increments K and waits on Q and *wakeup* signals K times does **not** work, since the signals may be taken by new processes arriving later. The above solution using the *baton* technique ensures that new processes are not allowed to interfere during the cascaded wakeup.

5. Solution for Mon.4

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.]

6. 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** $\{OK := \mathit{false};\ \ \mathit{signal}(c)\}$
else $\{OK := \mathit{true};\ \ \mathit{wait}(c)\}$

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.

7. Solution for Mon.2

We introduce a counter that keeps track of how many processes that have arrived at the meeting point. The last process starts all the others using *signal_all*.

```
monitor Barrier

  var count : integer := 0;
      c : condition;                    – Common waiting queue

  procedure SYNC()
    count := count + 1;
    if count < N then wait(c);
                 else {count := 0;  signal_all(c)}

end
```

[This solution is not robust towards *spurious wake-ups*.]