Solutions for Exercises, Week 6

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] =
  ...
  P(rw);
  read the database;
  V(rw);
  ...

process Writer[j : 1..m] =
  ...
  P(mutex_w);
  for k in 1..n do P(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. 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.

```
var S : semaphore := 1;
    Q : semaphore := 0;
    K : integer := 0;

sleep: P(S);
       K := K + 1;
       V(S)
       P(Q);
       K := K - 1;
       if K > 0 then V(Q)
                else V(S)

wakeup: P(S);
        if K > 0 then V(Q)
                 else 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.

3. Solution for Andrews Ex. 4.14

In order to achieve concurrent deposits and concurrent deposits, we may try to do only the slot allocation under mutual exclusion. Then, however, the filling (and emptying) of the slots can occur out of order, rendering the use of the *front* and *rear* pointers unfeasible. We may choose to accept the loose ordering and protect each slot by private full/empty semaphores.

A more conservative option, shown here, is to keep the ordering by passing a signal down the slots indicating that the previous slot has been filled/emptied. This is done through two arrays of semaphores *prev_full* and *prev_empty*:

```
var buf[0..n − 1] : T;
    front, rear : integer := 0;
    full : semaphore := 0;
    empty : semaphore := n;
    prev_full[0..n − 1] : semaphore := 0;
    prev_empty[0..n − 1] : semaphore := 0;
    mutex_P, mutex_C : semaphore := 1;

    V(prev_full[0]); V(pref_empty[0]);          — first slot OK
```

```
process Producer[i : 1..M] =
  var data : T;
      inpos : integer;
  repeat
    data := produce;
    P(empty);
    P(mutex_P);
    inpos := rear;
    rear := (rear + 1) mod n;
    V(mutex_P);
    buf[inpos] := data;
    P(prev_full[inpos]);
    V(full);
    V(prev_full[(inpos + 1) mod n)]);
  forever
```

```
process Consumer[j : 1..N] =
  var result : T;
      outpos : integer;
  repeat
    P(full);
    P(mutex_C);
    outpos := front;
    front := (front + 1) mod n;
    V(mutex_C);
    result := buf[outpos];
    P(prev_empty[outpos]);
    V(empty);
    V(prev_empty[(outpos + 1) mod n)]);
    consume result;
  forever
```

This higher degree of concurrency may be beneficial if the data type T is a large datatype such that buffer insertions/removals take significant time.

4. 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 $\texttt{P}(S_A)$, 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:

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

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:

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

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

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

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

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

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 $s_b \leq 1$.

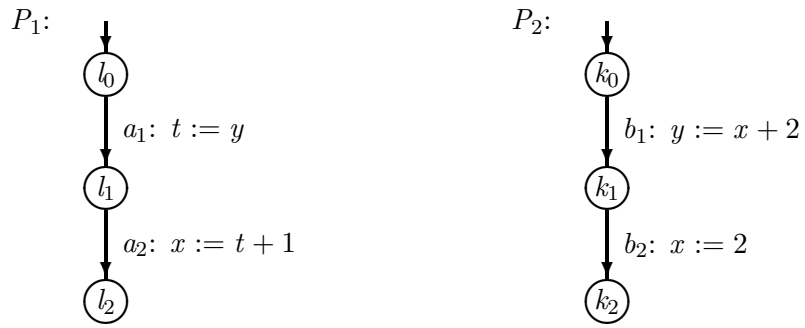
3. Solution for Concurrent Systems Exam December 2003, Problem 2

Question 2.1

The statements b, d, e, and f can be considered to be atomic since they only have only one critical reference each. Both a and c have two critical references.

Question 2.2

(a)



[Location and action labels not required.]

(b) Going through the 6 possible interleavings, the possible results for (x, y) are found to be:

$$(2,3), (2,2), (3,2), (1,2)$$

Question 2.3

P is preserved by a_2 and a_3. [By a_2 since $y < 0$ and P imply $x > 0$ and hence y also becomes positive.]

Q is preserved by all three actions. [Also by a_2 since it cannot be executed when Q holds.]

R is preserved only by a_2.

Question 2.4

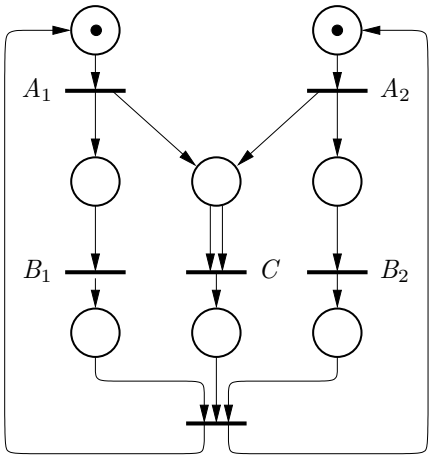
(a) The sequence $(0,1)(1,2)$ repeated forever will satisfy all parts of F.

(b) Assuming F, only the guard of a_3 is constantly true and hence only a_3 is guaranteed to be eventually executed under weak fairness. [If $x \neq 0$ it must be positive due to $\Box x \geq 0$ and hence $y > x > 0$ imply $y > 1$.]

(c) Assuming F, the guards of a_1, a_2, and a_3 will be infinitely often true, and hence they will be eventually executed under strong fairness. [The guard of a_4 is not necessarily true, for instance, it is never true in the state sequence proposed in (a).]

4. Solution for Concurrent Systems Exam December 2006, Problem 2

Question 2.1



Question 2.2

Corresponding to the Petri-net, we introduce a semaphore *DoneA* that counts the number of A-operations executed. C may then be executed after n P-operations on *DoneA*. Q controls the final synchronization by awaiting a signal from each finished B-operation on a semaphore *DoneB* and then signalling each process P_i on a private semaphore $GoA[i]$:

```
var DoneA : semaphore;          // Counts no. of A's done
    DoneB : semaphore;          // Counts no. of B's done
    GoA[1..n] : semaphore;      // OK to start A_i again.
```

All semaphores are initialized to 0

```
process P[i : 1..n];
  repeat
    A_i ;
    V(DoneA);
    B_i ;
    V(DoneB);
    P(GoA[i])
  forever
```

```
process Q;
  repeat
    for j in 1..n do P(DoneA);
    C;
    for j in 1..n do P(DoneB);
    for j in 1..n do V(GoA[j])
  forever
```

[It is **not** possible to replace $GoA[1..n]$ with a common semaphore since a P process may wait again immediately after a wait and thereby could consume a token destined for another process.]

Question 2.3

```
monitor Sync

  var adone : integer := 0;          // No. of A's done
      done : integer := 0;           // No. of B's and C done
      OkC : condition;               // Wait for all A's done
      Alldone : condition;           // Wait for all B's and C done

  procedure EndA()
    adone := adone + 1;
    if adone = n then signal(OkC)

  procedure StartC()
    while adone < n do wait(OkC);
    adone := 0

  procedure Done()
    done := done + 1;
    if done < n + 1 then wait(Alldone)
                    else done := 0;
                         signal_all(Alldone)

end
```

```
process P[i : 1..n];
  repeat
    A_i ;
    Sync.EndA();
    B_i ;
    Sync.Done()
  forever
```

```
process Q;
  repeat
    Sync.StartC();
    C ;
    Sync.Done()
  forever
```

[Solution assumes no spurious wake-ups.]