CS3211 Parallel and Concurrent Programming – Concurrency practice questions.

Question 1 [Message passing]

Consider a client-server encoding in Promela. The two servers are supposed to be independent of each other, as are the two clients. When a client sends a request, one of the servers should act on it and send a reply. Comment on the following Promela program which is supposed to do this task. Note that _pid correctly denotes the process identifier of the corresponding client or server process. Give detailed comments with sample traces.

```
chan request = [0] of { byte};     chan reply = [0] of {byte};

active[2] proctype server(){                      active [2] proctype client(){
   byte client;                                          byte server;
   do                                                    request!_pid
   ::  request ?client ->  reply!_pid                     reply?server
   od                                             }
}
```

Answer:

The channels are 0 capacity, so all send receives are handshake.

Client 1	Client 2	Server 1	Server2
request!c1			request?c1
	request!c2	request?c2	
reply?s1		reply!s1	
	reply?s2		reply!s2

So, client c1's request is received by server s2, and yet the reply comes from server s1, which is supposed to be independent!

Question 2 [Mutual exclusion]

Consider the following algorithm which tries to ensure mutual exclusion of accesses to critical section among two processes p and q. np and nq are shared variables, both of which are initially 0. Assume each statement is evaluated atomically.

```
Process    p                                   Process q
loop{                                         loop{
     x1:  /*non-critical section*/                  y1: /* non-critical section*/
     x2:  np = nq+1;                                y2: nq = np +1;
     x3: while (nq!=0 && np>nq){}                   y3: while (np!= 0 && nq >= np){ }
     x4: /* critical section*/                      y4: /* critical section */
     x5: np = 0;                                    y5: nq = 0;
}                                               }
```

Comment on whether mutual exclusion is preserved by the algorithm. Give detailed comments.

Answer:

```
Process    p                                          Process q
loop{                                                 loop{
     x1:  /*non-critical section*/                        y1: /* non-critical section*/
     x2:  np = nq+1;                                      y2: nq = np +1;
     x3: while (nq!=0 && np>nq){}                         y3: while (np!= 0 && nq >= np){ }
     x4: /* critical section*/                            y4: /* critical section */
     x5: np = 0;                                          y5: nq = 0;
}
```

Mutual exclusion is preserved. This can be understood as follows.
When p and q are in critical section – they are in x4 and y4 respectively.
We observe that
Process p is in x4 $\Leftrightarrow$ (nq == 0 $\vee$ np $\leq$ nq) [Property 1]
The right hand side is simply the negation of the guard in the while loop of process p.
Clearly when process p comes from x3 to x4 this property is true. We need to ensure that this property does not get falsified by the update of nq by process q in locations y2 and y5. However, we see that the property does not get falsified by such updates since y2 only ensures np $\leq$ nq and y5 only ensures nq == 0

In a similar way we can claim
Process q is in y4 $\Leftrightarrow$ (np == 0 $\vee$ nq < np) [Property 2]
Again this is the negation of the loop guard in process q.

From the 2 properties we have
Process p in x4 $\wedge$ process q in y4 $\Leftrightarrow$ (nq == 0 $\vee$ np $\leq$ nq) $\wedge$ (np == 0 $\vee$ nq < np)

But we observe that np is modified only by process p, and nq is modified only by process q. Hence we have
np == 0 $\Leftrightarrow$ process p is in locations x1 or x2
nq == 0 $\Leftrightarrow$ process 1 is in locations y1 or y2

Thus
Process p in x4 $\wedge$ process q in y4 $\Leftrightarrow$ (np $\leq$ nq) $\wedge$ (nq < np)

This shows that mutual exclusion is preserved.

Question 3 [Monitors]

Consider the following schematic code for the reader writer problem involving conditional synchronization.
Comment on whether the schematic code allows for starvation of readers.
Assume separate queues of waiting processes are maintained for the individual condition variables.

```
monitor RW{
    int readers = 0;  int readers = 0;
    condition OKtoRead, OKtoWrite;

      operation StartRead{
           if  (writers != 0  || ! empty(OKtoWrite)){
              wait_on_condition(OKtoRead);
           }
          readers++; signal_to_condition(OKtoRead);
      }
      operation EndRead{
              readers--;
              If (readers == 0){  signal_to_condition(OKtoWrite);}
      }
      operation StartWrite{
              if  (writers != 0 || readers != 0){  wait_on_condition(OKtoWrite);}
              writers++;
      }
      operation EndWrite{
              writers--;
               if empty(OKtoRead){   signal_to_condition(OKtoWrite); }
               else {signal_to_condition(OKtoRead);}
      }
}
```

Answer:

```
monitor RW{
      int readers = 0;  int readers = 0;
      condition OKtoRead, OKtoWrite;

        operation StartRead{
               if  (writers != 0  || ! empty(OKtoWrite)){
                  wait_on_condition(OKtoRead);
               }
              readers++; signal_to_condition(OKtoRead);
        }
        operation EndRead{
                  readers--;
                  If (readers == 0){  signal_to_condition(OKtoWrite);}
         }
         operation StartWrite{
                  if  (writers != 0 || readers != 0){  wait_on_condition(OKtoWrite);}
                  writers++;
         }
         operation EndWrite{
                  writers--;
                   if empty(OKtoRead){   signal_to_condition(OKtoWrite); }
                   else {signal_to_condition(OKtoRead);}
          }
}
```

If a reader is starved – he should blocked on OKtoRead. This means
writers!=0 $\vee$! empty(OKtoWrite).

If writers!=0, that means currently there are writers which have executed StartWrite but have not executed EndWrite. So, they will eventually execute EndWrite (due to the progress of the writers in writing the database) – and execute signal(OKtoRead) – unblocking readers.

If ! empty(OKtoWrite) - that means there are waiting writers. From the wait in StartWrite we know that this could be because writers!=0 $\vee$ readers != 0

We already saw that if writers != 0, readers will eventually get unblocked – due to the progress of the writers.

If readers!=0, this again means that there are readers who have executed StartRead and have not executed EndRead. Due to the progress of the readers in reading the database – they will all execute EndRead, and the last one will execute signal(OKtoWrite). This will unblock a writer, making writer!=0 true, reducing the argument to the previous case.

P.S. Note that we cannot have an infinite stream of readers (possibly constituted out of a finite number of readers) coming in and executing StartRead, thereby preventing a blocked writer from ever getting unblocked. This is because the wait in StartRead – blocks when there is a waiting writer --- so new readers cannot come in until the currently blocked writer is unblocked.