Theory Assignment 2 Solutions

Q.1.

a) Yes, the solution satisfies mutual exclusion of the critical section (C.S). There are three possible cases to consider for processes contending for the C.Ss:
Case 1: flag[i] is true and flag[j] is false. In that case, process Pi will enter the C.S.
Case 2: flag[j] is true and flag[i] is false. In that case, process Pj will enter the C.S.
Case 3: both flag[i] and flag[j] are true. In that case, the value of turn will break the tie.
In all the three cases, only one process will be able to enter the C.S. and hence the mutual exclusion is guaranteed.

b) Yes, the solution guarantees the progress requirement. Based on the explanation in (a) above, the value of turn is consulted only when there is a tie (i.e., flag is true for each of the processes; refer to Case 3). This ensures that entries to the critical sections are not strictly governed by the value of turn and hence are not strictly alternate.

Moreover, deadlock is not possible. Deadlock (actually, livelock in this case) is possible if and only if each process is indefinitely waiting for the other process to allow it entry to the C.S. This is not possible, because whenever each process is waiting for the other process (when flag is true for both the processes), then the value of turn will determine which process will enter its C.S (refer to case 3 in (a)). Hence, such an indefinite wait is not possible.

c) There is an upper bound on waiting and hence starvation is not possible. Suppose process Pi is waiting to enter its C.S and process Pj is in its C.S. As soon as Pj finishes its C.S., it will set turn to i. Hence process Pi will be able to enter its C.S. next. Thus, there is an upper bound of 1 on how much a process needs to wait before it can enter its C.S., after it makes a request. Consequently starvation is not possible.

Q. 2.

a) Consider three concurrent processes A, B, and C, synchronized by three semaphores *mutex*, *goB*, and *goC*, which are initialized to 1, 0 and 0 respectively:

```
Process A          Process B          Process C
---------          ---------          ---------
wait (mutex)       wait (mutex)       wait (mutex)
…                  …                  …
signal (goB)       wait (goB)         wait (goC)
…                  signal (goC)       …
signal (mutex)     …                  signal (mutex)
                   signal (mutex)
```

(i) Suppose Process B executes first; then it will get blocked at wait(goB). Subsequently process A and process C will block at wait(mutex). In this situation, all three processes will block permanently because there is no process to send a signal to wake up a blocked process. Similar is the case when process C executes first**.** (ii) Suppose process A executes first. After Process A is finished, Process C gets executed and blocks at wait(goC). Then Process B executes and blocks at wait(mutex). In this scenario, process B and Process C will block permanently. (iii) If the processes execute to

completion in the following order: process A followed by process B followed by process C, then none of them will block.

b) Now consider a slightly modified example:

```
    Process A                          Process B
     ----------                         ----------
for (i = 0; i < m; i++) {          for (i = 0; i < n; i++) {
    wait (mutex);                      wait (mutex);
    …                                  …
    signal (goB);                      wait (goB);
    …                                  …
    signal (mutex);                    signal (mutex);
 }                                  }
```

(i) m > n: if process B executes first, then it will block at wait (goB). Subsequently process A gets blocked at wait (mutex). Under this scenario, both processes will block permanently.

In general, both processes will block permanently under this scenario: process A has completed M iterations of its loop (and hence sent M signals to goB) and then process B enters its N^{th} iteration of the loop where M < N. If the previous scenario never occurs, i.e., always M ≥ N, then the two processes will never block permanently.

(ii) m < n: if process B executes first, then it will block at wait (goB). Subsequently process A gets blocked at wait (mutex). Under this scenario, both processes will block permanently. In general, both processes will block permanently when: process A has completed M iterations of its loop, and then process B enters its Nth iteration of the loop where M < N.

The maximum value of goB is m, which means that Process B can only obtain total m signals when it calls wait(goB). However, in Process B, the wait(goB) will be called n times where n > m. Therefore, finally Process B will block at wait(goB) permanently.

Q. 3. a) The critical sections are code segments where shared variables are modified or read; in this case: sem.value--; sem.value<0; sem.value++; sem.value<=0; Enqueue(current, sem.queue); Dequeue(sem.queue); Enqueue(k, ReadyQueue); etc. These critical section code segments must be executed atomically so that shared data (e.g., the ready queue contents) remain consistent.

b) If the critical sections in the above code are not protected, then a whole lot of inconsistencies can result. For example: if sem.value is currently 1; then one process calls wait on the semaphore and another process concurrently calls signal on the semaphore. After these two operations, the correct value of sem.val should be 1. However, if atomicity is not preserved, then the final value of sem.val could be 0, 1, or 2; obviously two of these values are incorrect. Similarly, other inconsistencies can occur in manipulating the pointers in the shared queues.

c) The interrupt status is kept inside a hardware register called the interrupt status register (ISR), which is also a part of a process's context. When process B's context is restored, the ISR is loaded with the interrupt status of process B. If interrupt was enabled when B was executing the last time, then B will start executing with interrupts enabled.

Q.4. Here is an implementation of the monitor *FileControl*:

```
Monitor FileControl {
```

```
    // Initialization code below
    condition readers, writers;
    int activereaders = 0;
    boolean writing = False;
    int waitingwriters = 0;
    // Atomic methods below
    WriterEntry() {
        If (writing || activereaders > 0)) {
            waitingwriters++;
            writers.wait();
            waitingwriters--;
        }
        writing = True;
    }
    WriterExit() {
         if (waitingwriters > 0)
              writers.signal();
         else {
              writing = False;
              readers.signal();
         }
    }
    ReaderEntry() {
        if (writing || waitingwriters > 0)
            readers.wait();
        activereaders++;
    }
    ReaderExit() {
         activereaders--;
         if (activereaders == 0) {
              if (waitingwriters > 0) writers.signal();
              else readers.signal();
         }
    }
}
```

Q. 5. The given program has an atomicity bug due to which a reader and a writer can access the file simultaneously, which is not desired. Consider the following scenario: (i) Reader 1 arrives and since it is the first reader, it goes to the "if" part of the code, increments "numOfReaders" and starts reading the file. (ii) Reader 2 arrives and it enters the "else" part of the code. However, just before it calls "wait (mutex)", there is a context switch and control passes to Reader 1. (iii) Reader 1 finishes reading, executes the rest of the code, decrements "numOfReaders" which becomes zero, and then subsequently does "signal(writeBlock)". (iv) Subsequently, a writer arrives and starts writing to the file. (v) Reader 2 executes again from where it left, and gets access to the file. So, both Reader 2 and the writer have now concurrent access to the file which is not desired.

Note that the above scenario cannot arise in the code given in the textbook and the slides.