Concordia University
Computer Science & Software Engineering
Faculty of Engineering and Computer Science

# COMP 346 – Fall 2017
# Theory Assignment 2

Answer all questions

| | |
|---|---|
| **Due Date** | **By 11:59pm Friday October 27, 2017** |
| **Format** | Assignments must be typed and submitted online to Moodle system. **Scanned Hand-written assignments will be discarded**. |
| **Late Submission:** | none accepted |
| **Purpose:** | The purpose of this assignment is to help you learn the overview of computer operating system and Input and output mechanisms. |
| **CEAB/CIPS Attributes:** | Design/Problem analysis/Communication Skills |

## Question # 1

Consider a single processor, single core environment. Somebody suggested the following solution to the critical section problem involving two processes $P_0$ and $P_1$. It uses two shared variables *turn* and *flag*. Note that this is not the Peterson's solution discussed in class, but looks similar:

```
boolean flag [2]; // Initially False
int turn; // Initially 0
do {
      flag[i] = True;  // i == 0 for P0 and 1 for P1
      while (flag[j] == True) { // j = 1- i
          if (turn == j) {
                flag[i] = False;
                while (turn == j) ;  // Do nothing: just busy wait
                flag[i] = True;
          }
       }
      // Critical section code here
      // …
     turn = j;
     flag[i] = False;
    // Remainder section code here
    // …
 } while (True)
```

The above is the code for process $P_i$, i = 0 or 1. The other process is $P_j$, where j = 1 – i. Now answer the following questions:

a) Will the solution satisfy mutual exclusion of the critical section? You must prove or argue (in a way similar to we did in class for Peterson's solution) your answer.
b) Will the solution satisfy the "progress" requirement? You must prove your answer.
c) Will the solution satisfy the bounded waiting requirement? If so, what is the bound? You must prove your answer.

## Question # 2

Answer the following questions:

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)
```

Does there exist an execution scenario in which: (i) All three processes block permanently? (ii) Precisely two processes block permanently? (iii) No process blocks permanently? Justify your answers.

b) Now consider a slightly modified example involving two processes:

```
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) Let m > n. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

(ii) (ii) Now, let m < n. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

**Question # 3**

Consider the following solution for the bounded-buffer producer-consumer problem. One requirement is that the producer must print when the buffer is full and the consumer must print when the buffer is empty:

```
semaphore mutex = 1; semaphore empty = N; semaphore full = 0;
int inp = outp = 0;
```

Code for Producer process:

```
do {
   produce (item);
   wait (empty);
   wait (mutex);
   Buffer[inp] = item;
   inp = (inp + 1) % N;
   signal (mutex);
   signal (full);
```

Code for Consumer process:

```
do {
    wait (full);
    wait (mutex);
     item = Buffer [outp];
     outp = (outp + 1) % N;
     signal (mutex);
     signal (empty);
     if (empty == N)
```

```
    if (full == N)                                  printf ("Buffer is empty\n");
        printf ("Buffer is full\n");              consume (item);
} while (True)                                  } while (True)
```

N is the buffer size. There is at least one serious bug (non-syntactic) in the above program.

a) Find out all the bugs and describe what problems they can cause.
b) Fix all the bugs and rewrite the program. In your solution, the producer must print when the buffer is full and the consumer must print when the buffer is empty.

**Question # 4**

A file is shared between several reader and writer threads. Design a monitor to control the access of the file by the different threads so that the following constraints are satisfied: (i) at most one writer can be active on the file at a particular time. (ii) When a writer is writing to the file, no reader can read from the file. (iii) More than one reader can be reading from the file simultaneously. (iv) When a writer is waiting to write, no more **new** reader should be allowed to read. (v) When a writer is writing and some other writer is waiting to write, then the writer is given more preference over a reader waiting to read. The general structure of each reader and writer thread is shown in the following:

```
Monitor FileControl {

// Definition of the monitor class to be filled in by you
    …
    …
}

FileControl fc;   // An instance of the monitor

Writer Thread:                          Reader Thread:
while (True) {                          while (True) {
    …                                       …
    fc.WriterEntry();                       fc.ReaderEntry();
    Write (file);                           Read (file);
    fc.WriterExit();                        fc.ReaderExit();
    …                                       …
}                                       }
```

Fill in the pseudo-code for the monitor FileControl as shown above.

**Question # 5**

The textbook (section 5.8.3 of the 9th edition) and the slides discuss about a monitor implementation using semaphores. Now, suppose we impose a restriction that the *signal()* operation of a condition variable (e.g., x.signal()) can only appear as the last statement in a monitor procedure. Suggest how the implementation described in section 5.8.3 can be simplified under this restriction.