NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 2 AY2012/2013

CS3211 – PARALLEL AND CONCURRENT PROGRAMMING

March 2013 **Time Allowed: 1 Hour**

INSTRUCTIONS TO CANDIDATES

1. This examination contains **four (4)** questions and comprises **SEVEN (7)** printed pages, including this page.
2. Answer **ALL** questions within the space in this booklet
3. This is an Open Book examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
1	**/ 6**	
2	**/ 4**	
3	**/ 5**	
4	**/ 5**	
Total	**/ 20**	

Question 1 [6 marks]

Consider the following multi-threaded program where x is an integer variable initialized to zero. You may assume that any assignment statement is executed atomically. Any condition evaluation is also executed atomically. The printf statement is also executed atomically.

```
Thread 1                    |          Thread 2
-------------------------------------------------------------
                            |
while (x < 2){              |        x = x + 1;
   printf("%d", x);}        |        x = x + 1;
                            |
```

Which of the following output sequences may be printed? For each, if it may be printed by the program, construct an interleaving that can print it. Also, if any sequence cannot be printed – give a reason why it cannot be printed.
(i) 012 (ii) 021 (iii) 12

Answer:

(i) This is possible, as shown by the following interleaving

```
while (x < 2)
     print x   // print 0
                                    x = x + 1     // x== 1
while (x <2)
     print x // print 1
while (x < 2)
                                    x = x + 1 //  x == 2
     print x // print 2
```

(ii) Not possible since the value of x is monotonically increasing with any execution of this program. Thus, prints in the left hand thread of the program should be printing higher values in later iterations of the loop (as compared to the earlier iterations).

(iii) This is possible, as shown by the following interleaving.

```
                                    x = x + 1 //   x == 1
while (x < 2)
     print  x // print 1
while (x < 2)
                                    x = x + 1 // x == 2
     print x // print 2
```

Question 2 [4 marks]

Consider the following encoding in Promela for the critical section problem. Processes are trying to access critical section, and we should ensure mutual exclusion of access, no deadlock, and eventual entry to critical section for each process. Comment on the following solution. You may assume that a false statement always blocks.

```
byte turn = 1;
active proctype P(){
    do
    ::  if
        ::  true
        ::  true -> false
        fi
        turn == 1;
       // critical section
        turn = 2;
    od
}
```

```
active proctype Q(){
      do
       ::  turn == 2;
           // critical section
           turn = 1;
       od
 }
```

Answer:

The only challenge comes from the following structure

```
if
 :: true
 :: true -> false
fi
```

Otherwise – it is simply a round-robin scheme which satisfies all the three properties.

Due to this if structure – the process P may block.
This will prevent process Q's attempt to enter critical section, since it waits forever for turn == 2 to be true.

Question 3 [5 marks]

Consider an atomic operation flip, such that

```
int flip (int lock){ lock =(lock +1)%3; return lock}
```

This is a variation of an example we discussed in class, where we had `lock =(lock +1)%2`. Suppose 2 processes are executing the following code, with lock initialized to 0. Will the solution work – i.e. it ensures mutual exclusion and no starvation? Give detailed comments.

```
/* Lock acquisition */
while (flip(lock) != 1)
      while (lock!= 0) {};
CRITICAL SECTION  /* Does not alter the value of lock */
/* Lock release */
lock = 0;
```

Answer:

For the class example, mutual exclusion was violated with `lock =(lock +1)%2`

Mutual exclusion is now preserved with `lock =(lock +1)%3`. Initially lock is 0, and any arbitrary process, say process 1, executes flip(lock) which returns 1, gaining entry to critical section. Since subsequent flip(lock) executions return 2, process 2 enters the inner loop where it is stuck until process 1 exits from the critical section and sets lock to 0.

No starvation is not guaranteed as shown by the following execution

Process 1	Process 2

flip(lock) returns 1 // exit outer loop
CRITICAL SECTION	

	flip(lock) returns 2 // enter outer loop
	lock == 2 // enter inner loop and stuck

lock = 0	
	lock == 0 // exit inner loop

flip(lock) returns 1 // exit outer loop
CRITICAL SECTION	

	flip(lock) returns 2// enter outer loop

< The pattern above may repeat forever>

EMPTY PAGE

Question 4 [5 marks]

The readers-writers problem for concurrently accessing a shared database was discussed in class. In this problem, several reader and writer threads try to access a shared database. At any time only one writer or several readers (but not both) should be allowed to access the database.

A. Following is a Java solution of the problem using monitors. Comment on the solution in terms of progress of readers/writers in eventually accessing the database. Give detailed comments.
B. Comment in general about the ability of monitors in Java in ensuring that a thread trying to enter a monitor will eventually (and quickly) enter the monitor.

```
class RWmonitor{
   private int readers = 0; private boolean writing = false;

  public synchronized void StartRead(){
           while (writing){
              try{ wait();
                 } catch(InterruptedException e){}
           }
           readers++; notifyAll();
  }

  public synchronized void StartWrite(){
           while (writing || (readers != 0)){
                try{ wait();
                   } catch(InterruptedException e){}
           }
           writing = true;
   }
}

public synchronized void EndRead(){
        notifyAll();
        readers--;
}

public synchronized void EndWrite(){
         notifyAll();
         writing = false;
}
```

Answer

A. A writer will wait whenever *readers > 0*. Hence readers can starve out a writer if more and more readers continue to acquire access to the database by executing StartRead. Even if there are fixed number of readers, and they are forced to exit reading after bounded time --- we can have reader i+1 acquire access to the database immediately after reader i relinquishes access. This can go on forever, starving the writer.

   The readers also wait whenever *writing == true*. Thus, if there are several hungry writers, they will also continue to access the database, starving out the readers.

B. In Java, the process executing notify (the signaling process) has to release the lock. Even after executing notify/notifyAll --- it continues to hold the lock until it returns from a synchronized method or encounters a wait itself. The notified process (which was waiting) therefore has to re-check the condition on which it was waiting, and the condition may no longer be true. This allows for starvation in monitor entry.

To avoid such starvation – one could allow for the signaling process to immediately pass control to the chosen waiting process. However, this is not done in Java implementations.

In addition, notifyAll notifies all waiting processes – processes waiting on the object, and only one of them is chosen. So, a process may keep on getting ignored.

END OF PAPER