THE AUSTRALIAN NATIONAL UNIVERSITY

Mid Semester Examination, September 2014

COMP2310 / COMP6310
(Concurrent and Distributed Systems)

Writing Period: 1 hours duration

Study Period: 0 minutes duration

Permitted Materials: One A4 page with handwritten notes on both sides. NO calculator permitted.

Questions are NOT equally weighted.
This exam will contribute 10% to your final assessment.
Write your answers in blue or black pen only. Answers written in pencil will not be marked.

The questions are followed by labelled, framed blank panels into which your answers are to be written. Additional answer panels are provided (at the end of the paper) should you wish to use more space for an answer than is provided in the associated labelled panels. If you use an additional panel, be sure to indicate clearly the question and part to which it refers to.

The marking scheme will put a high value on clarity so, as a general guide, it is better to give fewer answers in a clear and concise manner than to give a greater number in a vague and verbose fashion.

Please write clearly – if we cannot read your writing you will lose marks!

Student Number (please write clearly):							

Enrollment (circle one):
COMP2310
COMP6310

Official use only:

Q1 (14)	Q2 (15)	Q3 (16)	Q4 (5)	Total (50)

QUESTION 1 [14 marks]

(a) In the implementation of Java threads, what two steps must be performed before a new thread actually begins executing? Write the state of the thread after each step.

QUESTION 1(a) **[4 marks]**

(b) In a concurrent system, define the term *deadlock*. Illustrate this with a simple example where the components of the system on their own do not deadlock, but the overall system does.

QUESTION 1(b) **[4 marks]**

(c) Consider the problem of giving two concurrent threads mutual exclusive access to a critical section.

(i) State a suitable safety property and a suitable liveness property for this problem.

QUESTION 1(c)[i] **[2 marks]**

(ii) Explain why it is generally undesirable for a thread to sleep within the critical section.

QUESTION 1(c)[ii] **[1 mark]**

(d) Define the term *monitor* and briefly explain how a monitor may be used to implement mutual exclusion and condition synchronization.

QUESTION 1(d) **[3 marks]**

QUESTION 2 [15 marks]

(a) A system of two threads which repeatedly performs a unit of **work** (independently), followed by a **sync** (together).

```
THREADA = (a.work -> sync -> THREADA).
THREADB = (b.work -> sync -> THREADB).
||THREADS = (THREADA || THREADB).
```

(i) How many states are in the process `THREADA`?

QUESTION 2(a)[i] **[1 mark]**

(ii) If `THREADA` was redefined to perform the action `a.sync` instead of `sync`, and similarly for `THREADB`, how many states would the process `THREADS` have?

QUESTION 2(a)[ii] **[1 mark]**

(iii) Write either the Labelled Transition System for `THREADS`, or give an equivalent primitive FSP process definition (i.e. one that does not use the || operator). **COMP6310 students: you may use only the second option for your answer**.

QUESTION 2(a)[iii] **[4 marks]**

(b) To model such a synchronization for threads running on a computer, the **sync** action is broken into **arrive** and **exit** actions. Each thread will independently **arrive** (at a `barrier()` method), and only when both have arrived can they all **exit** the method (together). Rewrite the system of part (a) to reflect this.

QUESTION 2(b) **[3 marks]**

(c) Specify a *safety property* specifying that both threads must perform their **arrive** before the **exit** action can be performed. Write a composition of this property with your answer from (b). Would you expect the property to be violated? Briefly explain.

QUESTION 2(c) **[4 marks]**

(d) **COMP2310 students, answer the following:** Would you expect such a system to have problems in *progress*, that is can one thread repeatedly perform **work** actions while the other does not? Briefly explain.
COMP6310 students, answer the following: In a general context, explain how *progress* properties can be modelled in a language such as FSP.

QUESTION 2(d) **[2 marks]**

QUESTION 3 [16 marks]

(a) Consider the following (pseudo) Java code for a bounded buffer implemented using semaphores.

```
public class Semaphore {
    public Semaphore(int init);
    synchronized public void up();
    synchronized public void down();      }
class SemaBuffer {
    Semaphore full, empty;
    SemaBuffer(int size) {
        full  = new Semaphore(0);
        empty = new Semaphore(size);
        ... /*code to initialize buffer*/ }
   synchronized public void put(char c) {
        empty.down();
        ... // add c to buffer
        full.up();                         }
   synchronized public char get() {
        full.down();
        char c = ... // remove from buffer
        empty.up(); return(c);     }
}
```

Briefly explain what the problem that this code has and outline how you would write a correct bounded buffer using semaphores.

QUESTION 3(a) **[5 marks]**

(b) State the thread state transitions that are caused by a call to `sleep()` and `yield()`. For the former, what further actions must occur before the thread will run again?

QUESTION 3(b) **[3 marks]**

(c) In the context of a concurrent program, what is a *monitor invariant*? Describe an illustrative example using pseudo-code.

QUESTION 3(c) **[4 marks]**

Question 3 (continued)

(d) The Bakery algorithm aims to get mutual exclusion with N threads, where it is assumed that memory accesses are only via load or store operations. A enterprising student in CDS called Zed, after studying the algorithm and the ticketing system in a real bakery, decided that it can be simplified as follows.

Initialization: `t` is an integer array of length N, `int nextTktTaken = 0, lastTktCall = 0;`
Code for thread i, $0 \leq i < N$:

```
while (1) {
    // non-critical section i
    ...
    t[i] = nextTktTaken; nextTktTaken = t[i] + 1; // line A
    while (t[i] != lastTktCall) { /*spin*/ }      // line B
    // critical section i
    ...
    lastTktCall = lastTktCall + 1;                // line C
}
```

(i) Briefly explain why Zed's simplification does not give a correct mutual exclusion algorithm.

QUESTION 3(d)[i]	**[2 marks]**

(ii) Outline how you would fix this if the Java `synchronized` construct was available.

QUESTION 3(d)[ii]	**[1 mark]**

(iii) Outline how you would fix this using a test-and-set atomic operation, which you can assume can be access with a call to the following function.
`int testAndSet(volatile int *Lock)`:
{`int lv = *Lock; *Lock = 1; return lv;`} (atomically)

QUESTION 3(d)[iii]	**[1 mark]**

Student Number: .....

QUESTION 4 [5 marks]

(a) Describe how a *select* statement can be used to support condition synchronization in message passing programs.

QUESTION 4(a) **[3 marks]**

(b) Can a message passing program that uses synchronous message passing deadlock? Briefly explain with a simple example.

QUESTION 4(b) **[2 marks]**

Additional answers to QUESTION __(__)[__]

Additional answers to QUESTION __(__)[__]

Additional answers to QUESTION __(__)[__]