Assignment 1

ENGI 9869/7894 Adv. Concurrent Programming
Electrical and Computer Engineering
Memorial University
Due on May 30, 2019 11:59 PM

Note that the work that you turn in for this assignment must represent your individual effort. You are welcome to help your fellow students to understand the material of the course and the meaning of the assignment questions, however, the answer that you submit must be created by you alone.

1 Introduction [50]

1.1

Safety property asserts nothing bad happens during the program execution whereas the liveness asserts something good happens eventually. Essentially the mentioned properties depends on the some notable scheduling properties including mutual exclusion, deadlock free, starvation free and eventual consistency. How do you classify the following specifications into safety and/or liveness properties based on the mentioned subsequent properties above? [20]

1. "This switch can tern on and off the light bulb" [5]
2. "Your content will be published three days after your submission"[5]
3. "I will reply to your email if I receive it' '[5]
4. "You will not be able to receive my call" [5]

1.2

The semantics *co* implies it's body should be executed concurrently- at least conceptually depending on the number of processors. Program statements need to executed atomically always enclosed with $\langle$*atomic* action $\rangle$. Consider the following pseudo code with critical section semantics, assume four arbitrary threads are fighting to perform the assignments. [30]

int x = 1; y = 1;
co $\langle$ x = x + 1; $\rangle$
y = 10 ;

|
x = x * y ;
oc

1. Does this program holds the **at-most once property** (i.e mutual exclusion) ? [6]
2. Does the *co* semantics allow to run both assignments at same time? If so make a reasonable change to given pseudo code for parallel assignment. [6]
3. Change the number of threads to 8, does that make any difference? if so what would be the final value of x and y. [6]
4. Reason the same problem for n number of threads. What changes you distinguish from the previous two examples. [6]
5. Repeat 4 answer for 1,2, and, 3. Explain your reasoning. [6]

2 Mutual Exclusion [30]

Consider a simplest form of locks i.e *spin-lock*, where one variable is enough to switch the control between two threads over the critical section(part of code must be executed sequentially by one thread at a time). Consider the following pseudo code and determine some of properties of this lock you have studied in course.

```
    bool lock := false;
process CS1
    {
    while(true)
    {
    ⟨ await (!lock) lock := true; ⟩
    critical section;
    lock := false;
    noncritical section;
    }
    }
process CS2
    {
    while(true)
    {
    ⟨ await (!lock) lock := true; ⟩
    critical section;
    lock := false;
    noncritical section;
```

1. Mutual exclusion is the primary requirement for *critical section*, conceptually the the mutual exclusion is there due, but how you can assert that the both processes do no see the same value of lock, reason the possible ways ? [5]

2. How you can prove that the design is deadlock free? [5]

3. You need to make sure that the one process must be successful getting into *critical section*. It this code has freedom of starvation? if so, How you prove it? If not, make a reasonable change to let the processes run without starvation. [5]

4. The most subjective property for the lock is fairness, what you can do with this code to make it a fair solution(**hint:** *see some tie breaking solutions*) Explain the bakery algorithm and why reason it's fairness. [5]

5. Use semaphore to change and given code into one asserting the same critical section semantics. [5]

6. Discuss the scalability of a the solution with semaphores if you have n number of processes. [5]

3 Concurrent Computing [20]

1. State the difference among the terms concurrent computing, parallel processing and distributed computing. [5]

2. State the Amdahl's law, is it an accurate measure speed up? . [5]

3. Suppose an implementation takes 50 seconds to run on one processor, while 28% of the problem is parallelizable and rest is inherently serial portion. How many processors you need to run this application in less than 25 seconds? *(ignore the communication cost for time being)* [10]