ECE/CS 5510 Multiprocessor Programming Fall 2016
Homework 2

Part I: Problems [50 points]:

Solve the following problems.

1. [12 Points] Consider the Bakery Algorithm, in which both the safety (mutually exclusive) and liveness (guarantees progress and bounded-waiting) properties hold. The proof for these properties depends on the following assumption: any process will stay in the critical section only for a finite amount of time. Suppose a process crashes while inside its critical section (for example, the program does a division that causes an uncaught division by zero exception in that thread), while other processes are still running:

    a. Does this violate the safety property of the solution?

    b. Does this violate the liveness property?

    c. Does this violate the bounded waiting part of liveness?

    For each of the questions above, provide a very brief explanation for your answer.

2. [6 Points] The Peterson's mutual exclusion algorithm, which ensures no starvation, is presented in Figure 1. Consider the algorithms in Figures 2 and 3. Either prove or refute the following claims about the algorithms in Figures 2 and 3.

    a. Algorithm provides mutual exclusion

    b. Algorithm provides deadlock freedom

    c. Algorithm provides starvation freedom

Initially flag[0] and flag[1] are false and turn is 0

code for p_0	code for p_1
⟨Entry⟩:	⟨Entry⟩:
flag[0]:=true	flag[1]:=true
turn:=1	turn:=0
while flag[1] and turn=1 **do** wait	**while** flag[0] and turn=0 **do** wait
⟨Critical Section⟩:	⟨Critical Section⟩:
⟨Exit⟩:	⟨Exit⟩:
flag[0]:=false	flag[1]:=false
⟨Remainder⟩:	⟨Remainder⟩:

Figure 1

Initially flag[0] and flag[1] are false

code for p_0	code for p_1
⟨Entry⟩:	⟨Entry⟩:
1: **while** flag[1] **do** wait	1: **while** flag[0] **do** wait
2: flag[0]:=true	2: flag[1]:=true
⟨Critical Section⟩:	⟨Critical Section⟩:
⟨Exit⟩:	⟨Exit⟩:
3: flag[0]:=false	3: flag[1]:=false
⟨Remainder⟩:	⟨Remainder⟩:

Figure 2

Initially turn is 0

code for p_0	code for p_1
⟨Entry⟩:	⟨Entry⟩:
1: **while** turn=1 **do** wait	1: **while** turn=0 **do** wait
⟨Critical Section⟩:	⟨Critical Section⟩:
⟨Exit⟩:	⟨Exit⟩:
2: turn:=1	2: turn:=0
⟨Remainder⟩:	⟨Remainder⟩:

Figure 3

3. [6 Points] A solution to the mutual exclusion problem has been proposed in Figure 4. The line numbers are on the right. Does this solution satisfy the safety and liveness properties of the mutual exclusion problem? Give an informal proof or a violating adversary schedule for each of these properties.

```
int a = 0, b = 0;

while (1) {
  a = 1;              1
  while (b) {         2
    a = 0;            3
    while (b) ;       4
    a = 1;            5
    }
  critical section
  a = 0;              6
}


while (1) {
  b = 1;              7
  while (a) {         8
    b = 0;            9
    while (a) ;      10
    b = 1;           11
    }
  critical section
  b = 0;             12
}
```

Figure 4

4. [12 Points] The following algorithm (Fig. 5) is for two threads 1 and 2, and it makes use of two registers: x which can hold three values (0, 1, and 2); and y which can hold two values (0 and 1). Both threads can read and write registers x and y. The symbol i is used to designate the thread-id, and can be 1 or 2.

```
Initially: x=0 and y=0
start: x := i
       if y != 0 then
              await y=0;
              goto start;
       end if;
       y := 1;
       if x != i then
              y := 0
              await x = 0;
              goto start;
       end if;
       <critical section>
       y := 0;
       x := 0
```

Figure 5

a. Show that it satisfies mutual exclusion and deadlock-freedom for two threads.

b. Does it satisfy starvation-freedom for two threads?

c. Does it satisfy deadlock-freedom for three threads? That is, i can be 1, 2 or 3.

d. Does it satisfy mutual exclusion for three threads? That is, i can be 1, 2 or 3.

For each question, you should either sketch a proof, or display an execution where it fails.

5. [7 Points] Prove, by way of a counterexample, that the sequential time-stamp system T^3, started in a valid state (with no cycles among the labels), does not work for three threads in the concurrent case. Note that it is not a problem to have two identical labels since one can break such ties using threadIDs. The counterexample should display a state of the execution where three labels are not totally ordered.

6. [7 Points] The sequential time-stamp system T^3 had a range of $O(3^n)$ different possible label values. Design a sequential time-stamp system that requires only $O(n2^n)$ labels. Note that in a time-stamp system, one may look at all the labels to choose a new label, yet once a label is chosen, it should be comparable to any other label without knowing what the other labels in the system are. Hint: think of the labels in terms of their bit representation.

Write your answers for Part I in a PDF file named *hw2_<yourPID>.pdf*

Part II: Programming assignment [50 points]

Attached with this homework is a Java project that contains the implementation of *LockOne*, *LockTwo*, *Peterson,* and *Filter* locks, and a simple benchmark for testing the performance of these algorithms using a shared counter. Familiarize yourself with the code.

Implement the following requirements:

1. There is an implementation error in one of the locks. Find it, explain the problem, and fix it.
2. For each locking algorithm, run the benchmark (Test.java) 5~10 times and "measure" their following properties:
    a. mutual-exclusion
    b. fairness
    c. deadlock-freedom
    d. starvation-freedom

    An example (and simple) way to measure these properties is to record a sample execution pattern of threads through print statements. Feel free to modify or add probes to the code to detect thread execution patterns.
3. In the given benchmark, we are using only 2 threads. Consider a generalization of the two-thread Peterson lock by arranging a number of 2-thread Peterson locks in a binary tree, as follows. Each thread is assigned a leaf lock, which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1. In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (In other words, threads can take "naps", or even "vacations", but they do not "drop dead".)
    a. Run your new implementation against 16 competing threads modifying the Test class. Does each of the four properties mentioned above hold? Sketch a proof that it holds, or describe a, possibly infinite, execution where it is violated (you may also measure it as described previously).
    b. Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?
    c. The Test2 class measures the runtime of each thread and reports the average. Run this benchmark class on your local machine as well as a high-core count server for different THREAD_COUNT (4, 8, 16, 32, and 64). Make sure you do not run any other application when running the benchmark on your local machine. Include the core count of your local machine in the report.

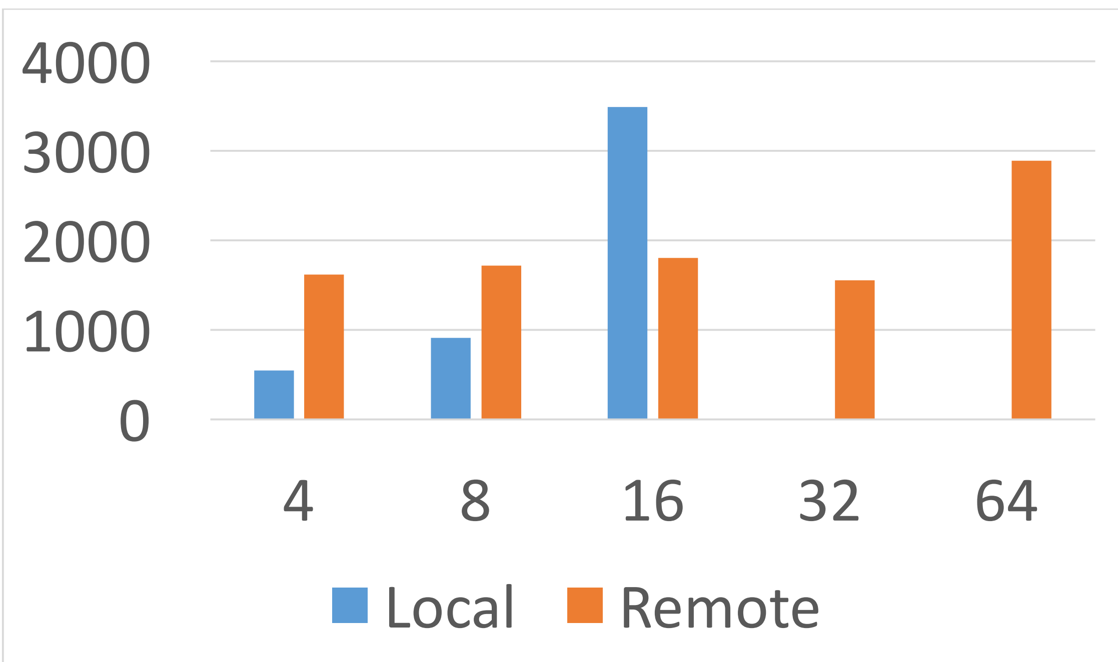
        You may use any machine in the Rlogin cluster to satisfy the high-core count server requirement. The Rlogin cluster consists of 24 40-core server machines and 2 64-core server machines. Your report must indicate the core-count of the machine you used. The instructions to setup an account to access these machines have been communicated in a previous announcement. The Rlogin is a shared cluster, thus there may be multiple users using the same machine at any given time, interfering with your benchmark. Thus, to overcome this, you have to run your benchmark at 4-5 different times of the day (For example, 12AM, 6AM, 9AM, 12PM, 6PM, 9PM). You have report these results in a table similar to below for each THREAD_COUNT.

Iterations	12AM	6AM	9AM	12PM	6PM
1					
2					
3					

Note that every time you login to Rlogin, you will be provided a random machine in the cluster. You must choose a particular machine for all your tests. Indicate the machine name in your report. You can use "hostname" to find the machine name initially. Then, even if you are given a different machine you can connect to the intended machine with the following command:

```
ssh <yourid>@<hostname>
```

You must also plot the average runtime vs THREAD_COUNT as a bar graph. There must be two bars (one for local machine and another for server machine). Essentially, you should compute the average value of each table and construct the graph. A sample has been provided below.



Note that as you move to higher THREAD_COUNT values on your local machine, you might experience machine freezes. In such cases, you may ignore reporting that and other higher THREAD_COUNT values. (For example, if you experience intermittent freezes when running with THREAD_COUNT=16, it is advisable to ignore THREAD_COUNT=16,32,64). However, you should not have any issues with the

Rlogin machines since they are capable of handling such high thread counts.

Hint: The benchmark uses a binary tree implementation using Java generics. Balance the binary tree using an appropriate order of nodes insertions.]

4. The L-exclusion problem is a variant of the starvation-free mutual exclusion problem, described as follows: as many as L threads may be in the critical section at the same time, and fewer than L threads might fail (by halting) in the critical section.

   An implementation of an algorithm for this problem must satisfy the following conditions:

   a. L-Exclusion: At any time, at most L threads are in the critical section.

   b. L-Starvation-Freedom: As long as fewer than L threads are in the critical section, then some thread that wants to enter the critical section will eventually succeed (even if some threads in the critical section have halted).

   Modify the n-process Filter mutual exclusion algorithm to convert it into an L-exclusion algorithm.

Zip up the PDF together with the source code for Part II, and save it in a file called h**w2_<myPID>.zip**. Submit the archive as Homework 2 on the class's Canvas page before the deadline.