ECE/CS **5510 Multiprocessor Programming** **Fall 2019**

Homework 2

Part I: Problems [50 points]:

Solve the following problems.

1. [9 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. [9 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⟩:
1: flag[0]:=true	1: flag[1]:=true
2: turn:=1	2: turn:=0
3: **while** flag[1] and turn=1 **do** wait	3: **while** flag[0] and turn=0 **do** wait
⟨Critical Section⟩:	⟨Critical Section⟩:
⟨Exit⟩:	⟨Exit⟩:
4: flag[0]:=false	4: 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. [5 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
1 start: x := i
2        if y != 0 then
3                await y=0;
4                goto start;
5        end if;
6        y := 1;
7        if x != i then
8                y := 0
9                await x = 0;
10               goto start;
11       end if;
12       <critical section>
13       y := 0;
14       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] Consider the proposed algorithm to the mutual exclusion problem for two threads in Figure 6.

   a. Prove or dispute the safety and liveness properties of the algorithm.

   b. If the `turn` variable was initialized to `-1`, how will this affect the algorithm's correctness properties?

   c. How does this algorithm compare to the Peterson's algorithm. Discuss the differences in design choices that impact the safety and liveness properties of the algorithms. Is one better than the other?

```
01: var flag: array [0..1] of boolean;
02: turn: 0..1;
03:
04: repeat
05:   flag[i] := true;
06:   while flag[j] do
07:     if turn = j then
08:     begin
09:       flag[i] := false;
10:       while turn = j do no-op;
11:       flag[i] := true;
12:     end;
13:
14:   // critical section
15:
16:   turn := j;
17:   flag[i] := false;
18:
19:   // remainder section
20: until false;
```

Figure 6

6. [8 points] Reentrant Locks:
   A reentrant mutual exclusion lock provides the same basic behavior and semantics as the regular lock, but with extended capabilities. A `ReentrantLock` is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking `lock` will return, after successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.

   State whether the following algorithms can be trivially modified to be reentrant capable, without any additional variables? Give reason.

   a. Peterson's Lock

   b. Bakery Lock

   c. Filter Lock

   d. Lock in Figure 6 (irrespective of its correctness)

   By trivial modification, we mean using an *if* conditional statement to check whether the current thread owns the lock and returning immediately.

Part II: Programming assignment [50 points]

Attached with this homework is a Java project that contains the implementation of *Peterson* and *Filter* locks, and simple benchmarks for testing the working and performance of these algorithms using a shared counter. Familiarize yourself with the code. Please configure IntelliJ to use JDK 8 to run the provided code and your development.
(http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)

Do the following tasks:

1. [15 points] Timestamp and Bakery lock:
    a. Provide an implementation of the `Timestamp` and `TimestampSystem` interfaces in Figure 2.10 of the textbook.
    b. Implement the *Bakery* lock algorithm using the implementation of the Timestamp interface.
    c. Compare the performance of *Bakery* lock with *Filter* and *Peterson* locks for different thread counts – 2 to as many threads your machine supports (e.g. my machine supports 8 threads), and preferably a high thread count value. For this purpose, you may modify the Test2.java which measures the runtime of each thread and reports the average. Show how the average thread runtime scales with the number of threads for both *Filter* and *Bakery* lock.

       Fix the instantiation code for *Filter* lock so that it can be easily instantiated for any number of threads. Note that you should run Peterson's algorithm won't support more than two threads.
    d. Among the *Bakery* locks and *Filter* locks, which one performs better? Explain the reason.

2. [20 points] 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 by inserting trace statements in you code).
    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. Modify Test2 to run the new lock implementation. 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.

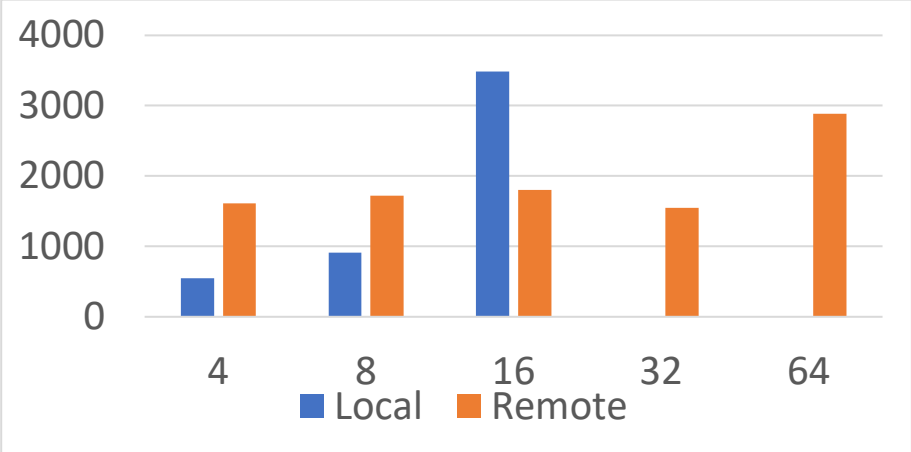
You may use any machine in the Rlogin cluster to satisfy the high-core count server requirement. The Rlogin cluster consists of 24 40-core server machines and 2 64-core server machines. Your report must indicate the core-count of the machine you used. The instructions to setup an account to access these 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 into 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 may use the provided binary tree implementation using Java generics. Balance the binary tree using an appropriate order of nodes insertions. Feel free to implement your own classes for this problem.]

3. [15 points] 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 your implementation of the Bakery algorithm (from Question 1) and convert it into an L-exclusion algorithm. Call this class LBakery. Reason about the correctness of your design in about 50-100 words. Write a test program (not a unit test) and produce a trace to show that more than one (but less than L) threads are in the critical section at the same item.

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

Rlogin Registration instructions

- Create an account on - https://admin.cs.vt.edu/
- Create the SLO password according to the password policy (7 characters, one uppercase letter, one lower case letter, one special character or number). This is the password that will be used to access Rlogin.
- Wait for an hour, and use the following command on Terminal (Windows users, see below) to login:
  `ssh your_pid@rlogin.cs.vt.edu`
- In case you can't login into Rlogin, you will have to use Virginia Tech's Remote Access VPN. https://www.nis.vt.edu/ServicePortfolio/Network/RemoteAccess-VPN.html
- Rlogin accounts of non-CS majors are deleted at the end of every semester. So, if you are a non-CS major and had an account previously, you will have to create a new account.

For Windows users:

- You can either use PuTTy or MobaXterm to login into Rlogin.

- Or you can install a Unix-like environment for Windows such as Cygwin or msys2. I suggest msys2 (Follow the guide here: https://www.msys2.org/). Follow the guide in the link. Install OpenSSH through the following command: pacman -S openssh

- You also have an option of using ‘Windows Subystem for Linux’ - https://docs.microsoft.com/en-us/windows/wsl/install-win10