CS5510: Multiprocessor Programming : Homework II

Sergio Sainz

September 2019

1 Part I : Problems

1.1 [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:

1. Does this violate the safety property of the solution? **Answer** No, safety property pertains to not allowing more than one thread accessing the critical section. Safety property is avoiding something wrong happens, in this case the wrong thing is more than one threads accessing critical section. If thread crashes in critical section, it is still the only thread in the critical section. Therefore safety property is not broken. Bounded waiting property is broken though.

2. Does this violate the liveness property? **Answer** Talking about the **deadlock-freedom**, it says: any thread that is waiting to access the critical section is waiting because some other thread is accessing the critical section at the time. In this case other threads are waiting because the crashed thread is accessing the critical section. Therefore does not violates deadlock-freedom.
   About **starvation freedom**, it says: every thread that wants to enter critical section eventually does it. In this case, if the crashed thread would not have crashed, then it would have continued and exit critical section. After such thread exits critical section other threads can enter critical sec-

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:

tion. Thus achieving starvation freedom. Starvation freedom property is not violated.

3. Does this violate the bounded waiting part of liveness? **Answer** Yes, it does. Bounded waiting refers to waiting to enter critical section is at most x amount of time, because the crashed thread will stay crashed forever, we can say other threads will wait for infinite time. $\infty > x$, thus *bounded waiting* is broken.

1.2 [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

Algorithm 2

1. Algorithm provides mutual exclusion:
   No, it does not. By code inspection, we know both threads could end up

in critical section by following this sequence of steps:

$$read_{p_0}(flag[1] = false) \rightarrow read_{p_1}(flag[0] = false) \rightarrow$$
$$write_{p_0}(flag[0] = true) \rightarrow write_{p_1}(flag[1] = true) \rightarrow$$
$$CS_{p_0}, CS_{p_1}$$

2. Algorithm provides deadlock freedom
   Yes it does! Prove it by contradiction. Assume both threads wait in the while loop indefinitely.

$$read_{p_0}(flag[1] = true) \rightarrow read_{p_1}(flag[0] = true) \rightarrow \text{wait forever}$$
$$write_{p_0}(flag[0] = true) \rightarrow CS_{p_0} \rightarrow write_{p_0}(flag[0] = false)$$
$$write_{p_1}(flag[1] = true) \rightarrow CS_{p_1} \rightarrow write_{p_1}(flag[1] = false)$$

   Only way for both threads to wait forever is if both reach the while loop with both flags as true. But because we know as soon as the threads exit their critical sections they both set their flags as false, then, the values they will be reading is $read_{p_0}(flag[1] = false)$ or $read_{p_1}(flag[0] = false)$. Therefore a contradiction. Also the case when only one thread enters the algorithm works fine because the other thread's flag will be off. Then the thread that is entering algorithm is able access critical section.

3. Algorithm provides starvation freedom
   It is not starvation freedom! By code analysis we know the thread p_0, for example, could be starved forever if every time it checks for p_1 flag is set to true $\text{read}_{p_0}(flag[1] = true)$, and this is possible if thread p_1 is always in the critical section whenever p_0 reads p_1 flag. Thread p_1 could do any number of sequences such as:

$$write_{p_1}(flag[1] = true) \rightarrow CS_{p_1} \rightarrow write_{p_1}(flag[1] = false) \rightarrow$$
$$\ldots$$
$$write_{p_1}(flag[1] = true) \rightarrow CS_{p_1}$$

   As long as p_1 is in CS_{p_1} when p_0 reads the flag, then p_0 will starve.

Algorithm 3

1. Algorithm provides mutual exclusion
   It is mutual exclusion. Proven by contradiction. Suppose both threads are in their critical sections:

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:

$$CS_{p_0}, CS_{p_1}$$
$$read_{p_0}(turn = 1) \rightarrow CS_{p_0}$$
$$read_{p_1}(turn = 0) \rightarrow CS_{p_1}$$
$$read_{p_0}(turn = 1) \rightarrow read_{p_1}(turn = 0)$$

Above means that in order for both threads to be in the critical section at the same time, turn needs to be both 1 and 0 at the same time , contradiction.

2. Algorithm provides deadlock freedom
   No, algorithm 3 does not hold property deadlock freedom. If one of the threads, for example p_0, do not call algorithm (not interested in enter critical section ever), then the algorithm 3 deadlocks for the thread that did enter the critical section p_1. On the other hand if both threads run the algorithm concurrently, they would run without deadlock in this special case, still not satisfy deadlock freedom.

3. Algorithm provides starvation freedom
   By similar argument from previous point, algorithm 3 is not starvation free. Assume thread p_0 does not enter algorithm ever. Then thread p_1 enters to the lock method, waits for *turn* to turn to 1, but it will never turn to 1 unless p_0 enters its critical section, exits its critical section and turns *turn* to 1. Because thread p_0 will never enter algorithm, then p_1 starves.

```
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:

1.3 [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.

Below answers assume that the execution starts for one thread at line before line 1 (while loop). And for the second thread execution starts at line before line 7 (while loop). This is not clear in the question. If both threads start at line before line 1, then both threads will loop in lines 1, critical section line before 6 and line 6 indefinitely without ensuring mutual exclusion.

- Safety property: Mutual exclusion. Means no more than one thread can be in the critical section at the same time.
  Answer Mutual exclusion is supported. Consider proof by contradiction. Consider both threads are in the critical section:

$$
\begin{aligned}
read_{p_0}(b = 0) &\rightarrow CS_{p_0} \\
read_{p_1}(a = 0) &\rightarrow CS_{p_1} \\
write_{p_0}(a = 1) &\rightarrow read_{p_0}(b = 0) \\
write_{p_1}(b = 1) &\rightarrow read_{p_1}(a = 0)
\end{aligned}
$$

  Last two step means that both threads need to write their respective flags a, b to 1 before reading for the other thread's flags as zero. Because the flags are not set to 0 until after critical section. And by that time the thread need to start over again setting their flags as 1, then it is not possible for threads to write 1 in their flags and then expect to see 0 in both flags so that both threads enter critical sections at same time. contradiction.

- Liveness property: comprised of deadlock freedom, which means if one thread is waiting is because other threads are entering and existing the critical section. And starvation freedom, which means if one thread is waiting it will eventually enter critical section.

  Algorithm is deadlock free, consider that one thread p_0 does not call the routine, while the other thread p_1 calls the routine. In that case p_1 will not be deadlock because the while loop in line 8 will always read $a = 0$.
  Algorithm is not starvation freedom. Consider if thread p_0 is waiting for flag b to turn to 0. Thread p_0 will be starved if every time it checks for b thread p_1 is in its critical section. In theory it is possible, but highly unlikely.

```
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:

1.4 [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

1. Show that it satisfies mutual exclusion and deadlock-freedom for two threads.
   We show it satisfies **mutual exclusion** with contradiction, assume it does not and that both thread 1 and 2 are in their critical section. If so, the following events must have happened.

$$
\begin{aligned}
&write_1(x = 1) \rightarrow read_1(y = 0) \rightarrow \\
&write_1(y = 1) \rightarrow read_1(x = 1) \rightarrow CS_1 \\
&write_2(x = 2) \rightarrow read_2(y = 0) \rightarrow \\
&write_2(y = 1) \rightarrow read_2(x = 2) \rightarrow CS_2
\end{aligned}
$$

   This is telling us that for both threads to enter the critical section at the same time, both must have read x = i (1 or 2). But the only time the x is set is in line one. So we could argue that thread 1 went all the way to

its critical section while thread 2 was still not executed line 1:

$$read_1(x = 1) \rightarrow write_2(x = 2)$$

And we know that after thread 2 sets x to 2, then it must read y is 0 to continue to its critical section, but y is set to 1 already by thread 1 and it wont be set back to 0 until after critical section in line 13, thus a contradiction:

$$\begin{aligned} &read_1(x = 1) \rightarrow write_2(x = 2) \rightarrow \\ &read_2(y = 0)\ \textit{y already set to 1 by thread 1} \end{aligned}$$

Also, another branch of actions is that both threads perform same operations at same time up to reading x at line 7:

$$\begin{aligned} write_1(x = 1) \rightarrow write_2(x = 2) \rightarrow \\ read_1(y = 0) \rightarrow read_2(y = 0) \rightarrow \\ write_1(y = 1) \rightarrow write_2(y = 1) \rightarrow \\ read_1(x = 1) \rightarrow read_2(x = 2) \rightarrow \end{aligned}$$

But x cannot take both 2 and 1 at same time, therefore a contradiction in this last branch of actions.

Deadlock freedom . It supports deadlock freedom. The only places where the algorithm could be deadlocked waiting indefinitely is line 3 and line 9. Going to prove it for the three potential cases that cause deadlock: (1) Thread 1 in line 3 and thread 2 in line 3 as well: In this case both threads are waiting for y to become 0. Then it must be true that y is 1. But we can see from code that whenever y is set to 1 (in line 6), it is set back to 0 by the same thread that read it before reaching line 3 (set to 0 in line 8 and 13). Thus if both threads are in line 3 y value is 0, contradiction.

$$\begin{aligned} read_1(y = 1) \rightarrow read_2(y = 1) \rightarrow \text{waitingIn3}_1, \text{waitingIn3}_2 \\ write_1(y = 1) \rightarrow write_1(y = 0) \rightarrow \\ write_2(y = 1) \rightarrow write_2(y = 0) \rightarrow read_i(y = 1) \end{aligned}$$

(2) Thread 1 in line 3 and thread 2 in line 9 : In this case thread 1 is reading y is 1, and thread 2 is reading x is 1.

$$\begin{aligned} write_2(y = 1) \rightarrow waitingIn3_1(y = 1) \\ write_1(x = 1) \rightarrow waitingIn9_2(x = 1) \end{aligned}$$

We know thread 2 wrote y as 1 because thread 1 sets it to 0 in line 13 last iteration, therefore, only thread 2 could have set y to 1 in line 6. From code inspection we know:

$$write_2(y = 1) \rightarrow write_2(y = 0) \rightarrow waitingIn9_2(x = 1)$$
$$write_2(y = 1) \rightarrow write_2(y = 0) \rightarrow waitingIn3_1(y = 1)$$
$$write_1(y = 0) \rightarrow waitingIn3_1(y = 1)$$

Thus a contradiction that before thread 2 goes on waiting on line 9 it sets y as zero, and thread 1 cannot read y as 1.

(3) Thread 1 in line 9 and thread 2 in line 9: In this case both threads are reading x as:

$$waitingIn9_2(x = 1), waitingIn9_1(x = 2)$$

But is not possible for x to be both 1 and 2 at the same time. Contradiction.

2. Does it satisfy starvation-freedom for two threads?
   No, it does not, thread 1 starves if it is waiting in line 3 and then every time it reads for y thread 2 is in its critical section. Thread 2 goes around the algorithm entering and exiting critical sections as many times as needed. As long as thread 2 is in critical section when thread 1 checks y, then thread 1 will starve indefinitely.

3. Does it satisfy deadlock-freedom for three threads? That is, i can be 1, 2 or 3.
   This one is hard! I believe it satisfies deadlock-freedom. The proof is similar to the 2 thread case for the subcases where all three threads are in either line 3 or in line 9. That is, for case (1) all threads waiting in line 3: all threads cannot be in waiting in line 3 (waiting for y to become 0) because before reaching this line 3 all threads must have written $y = 0$ in line 8 or 13. For case (2) where all threads are waiting in line 9, is also not possible because for all threads to wait here means x is either 1,2 or 3. And it means at least one waiting thread i is equals to x. And if so, such thread i would not be waiting in line 9 because if thread i enters line 9 means x was different than i. So it means that some thread wrote x with i between ith thread's execution of line 7 and line 9, but only thread i can write x with i in line 1, therefore a contradiction.
   It becomes more tricky with the case where some threads waiting in line 3 and some threads waiting in line 9. The raw sketch is by contradiction like this: Assume some threads are waiting in line 3 and other threads waiting in line 9. This means y is 1, and x is different from the ids of the

```
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:

thread(s) waiting in line 9 (check previous paragraph to see why). If y is 1, it means some thread wrote y as 1. But the threads waiting in line 3 set the y as 0 in previous iteration (in line 13 and 8) so the threads waiting in line 3 cannot have set y to 1. At the same time, threads waiting in 9 specifically set y to 0 right before waiting. Therefore, threads waiting in 3 cannot have read y as 1.

4. Does it satisfy mutual exclusion for three threads? That is, i can be 1, 2 or 3.
   No, it does not. Here the sequence where both threads end up in critical section at same time.

$$\begin{aligned} write_2(x=2) \rightarrow write_1(x=1) \rightarrow \\ write_1(y=1) \rightarrow write_2(y=1) \rightarrow \\ write_2(y=0) \rightarrow waitingIn9_2(x=1) \rightarrow CS_1 \\ waitingIn9_2(x=1) \rightarrow write_3(x=3) \rightarrow read_3(y=0) \rightarrow \\ read_3(x=3) \rightarrow CS_3 \end{aligned}$$

1.5 [7 Points] Consider the proposed algorithm to the mutual exclusion problem for two threads in Figure 6.

1. Prove or dispute the safety and liveness properties of the algorithm.
   Mutual exclusion

Yes, it satisfies mutual exclusion. If both threads are in the critical section at the same time we will see that both threads should have read other's threads flag as false.

$$read_2(flag[1] = false) \rightarrow CS_2$$
$$read_1(flag[2] = false) \rightarrow CS_1$$

But then, that means the other thread should have turn down their flags before. And this only happens after critical section (line 17) or while yielding to the other thread (line 9), but ultimately before entering critical sections both threads turn their flags on (line 11 and 7), and therefore a contradiction.

$$write_0(flag[0] = true) \rightarrow read_1(flag[0] = false) \rightarrow CS_2$$
$$write_1(flag[1] = true) \rightarrow read_0(flag[1] = false) \rightarrow CS_1$$

Starvation freedom
It is starvation free. Assume that thread 0 is waiting in line 10, meanwhile other thread 1 keeps entering and existing critical section. Actually as soon as thread 1 exits critical section once, it sets the turn to be 0 (line 16). Then thread 0 will be able to move out of inner while loop into outer while loop. Meanwhile thread 1 may also try to enter critical section again, but this time because thread 0 flag is on ($flag[0] = true$) and is thread 0 turn ($turn = 0$) then thread 1 yields to thread 0 (line 7) and turns down its flag (line 9). And thread 0 is able to go next into critical section. While thread 1 waits in line 10. Therefore a contradiction , a thread cannot wait an indefinitely number of times while the other thread enters and exists critical section many times. Also, when only one thread is entering and existing the algorithm, that thread will be able to access its critical section as many times as it wants.

Deadlock freedom
Because algorithm is starvation free, it also is deadlock free.

2. If the turn variable was initialized to -1, how will this affect the algorithm's correctness properties?
   If turn is initialized to -1, then the mutual exclusion is still in place (proof sketch above does not consider turn). But deadlock freedom and starvation freedom are no longer satisfied:

$$\begin{aligned} &write_1(flag[1] = true) \rightarrow write_0(flag[0] = true) \rightarrow \\ &\quad read_1(turn = -1) \rightarrow read_0(turn = -1) \rightarrow \\ &read_0(flag[1] = true) \rightarrow read_1(flag[0] = true) \rightarrow \text{Loops forever.} \end{aligned}$$

3. * 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?
   No difference in regards to properties. Both satisfy mutual exclusion, deadlock freedom and starvation freedom. Both do not solve waiting problem. And both are fair. But, the style of this algorithm is convoluted and hard to follow. Meanwhile Peterson is succinct and with one less loop.

1.6 [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.

1. Peterson's Lock: we can implement reentrant lock in peterson's lock by adding a check at the beginning of lock method whether the $if(flag[i] == true)return;$. We can add that line after defining i and before setting $flag[i] = true$ (between lines 7 and 8 in the text book figure 2.6). In this way, if lock's owner reenters lock, it won't get blocked and will return and also won't affect the victim's variable. Every non-owner thread that calls the lock method must have its flag[i] as false. Why? Because if they do not own the lock and have flag as true, then won't be calling lock because they are waiting for the lock. And only the thread that calls lock again will have the flag[i] as true before calling lock.

2. Bakery Lock: we can implement reentrant lock to Bakery algorithm. Every non-owner thread that calls the lock method must have its flag[i] as false. Why? Because if they do not own the lock and have flag as true, then won't be calling lock because they are waiting for the lock. And only the thread that calls lock again will have the flag[i] as true before calling lock. Therefore, a solution is to check at the beginning of lock method whether $flag[i] == true$ and if so return. Referring to text book's figure 2.9, we can add that check ($if(flag[i] == true)return;$) between lines 12 and 13.

3. Filter Lock: we can add similar trick to the one made to the Peterson's Lock where we check in the lock method whether the caller is already in the topmost level (n-1) and if so return. Using as reference the algorithm in the text book (figure 2.7) add between lines 12 and 13: $if(level[me] ==$

$n-1)return;$. In line 12, variable me is defined, and in line 13 the for loop going up the level ladder starts. This will work because we do not modify any existing variable in the algorithm, we just read and if condition matches, return.

4. Lock in figure 6. Considering lock method comprises lines 5 - 12. We can add reentrant lock capabilities same trick. Because in this algorithm the only thread able to call lock again with flag as true is the one that is non-blocked, that is the owner as previously mentioned (other thread is blocked at line10 if already asked for lock or has not asked for lock in which case its flag is false. Then we can add line $if(flag[i] == true)return$; between lines 4 and 5 at the beginning of the lock part.

2 Programming assignment

2.1 [15 points] Timestamp and Bakery lock:

1. Provide an implementation of the Timestamp and TimestampSystem interfaces in Figure 2.10 of the textbook. **DONE**

2. Implement the Bakery lock algorithm using the implementation of the Timestamp interface. **DONE**

3. 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.

   Answer

   Comparison is below (640,000 max increment in each experiment and values are from the second execution (discards warm-up phase). Numbers are in milliseconds):

Lock \ Threads	2	4	6	8
Peterson	126			
Filter	123	589	1086	1901
Bakery	132	238	314	994

   My machine has 8 logical cores (4 physical cores).

4. Among the Bakery locks and Filter locks, which one performs better? Explain the reason.

Answer The Bakery lock is performing better.
(1) Contention: Bakery lock does not need to write to same location always. Filter lock all threads share the victim array, one index of the array for each level.
(2) Longer waiting: In Bakery lock every thread needs to read n locations to decide whether is its turn. In Filter lock, each thread needs to read all n locations just to move to next level, this additional check per each n-1 levels is causing further delay.

2.2 [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".)

1. 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).
Answer
Mutual exclusion
My proof sketch is based on four steps:
(1) for each and every peterson lock in the tree-lock, as long as its lock method is invoked by at most two threads, it satisfy mutual exclusion. We can see this is true from the text book.
(2) within the tree, every node has only two child nodes (binary tree).
(3) Only two threads reach a non-leaf node at most and non-leaf node also is made of peterson lock thus mutual exclusion is also satisfied in non-leaf nodes. As long as no more than two thread reach non-leaf node mutual exclusion is warranted. We know the leaf nodes are peterson's lock and from code analysis only the node that acquires leaf peterson's lock will continue and try to lock its parent's node lock. Therefore, only

two threads can reach a non-leaf node peterson lock at most (the winner from each child). This covers most other non-leafs even root. Hence mutual exclusion is warranted.
(4) When locks are released mutual exclusion remains , the locks are being release from root to leaf. This allows nodes waiting in a node higher (closer to the root) to have priority over the tree-lock (root's lock). Starting from root, one it is unlocked, waiting thread can now enter root node critical section. For non-root nodes, once node's lock is released the waiting thread can acquire the lock just released. For example if the thread that just released the non-root node is the left hand side thread, then right hand side can acquire the node. Because lock is peterson lock, even though left thread wants to acquire lock right away, as long as right hand side thread is already waiting, it will see the victim flag changed to the left hand side thread and therefore be able to access.
Fairness Peterson lock is fair in that if a thread 0 is waiting for the other to finish, then once the other thread 1 finishes and wants to enter again the critical section, it cannot because thread 1 will set the victim to 1, yielding the right to go to thread 0.
When talking about the tree structure, the locks are made at every node of the three. Because in each node only two threads exists at one point in time, then fairness also exists in non-leaf nodes.
When unlocking the tree-lock the unlocking is done from root to leaf. This is important because it gives priority to the waiting nodes closer to the root. This means once a node is at a higher level (higher is closer to root) in the tree it has priority over all other nodes under the node and competes against all nodes under the sibling node. More over because peterson lock is fair, as soon as a thread is waiting for a node's lock and that lock is unlocked the node will acquire it (peterson's lock is fair). This places an upper bound on number of times the node needs to wait for the tree-lock. If n is number of threads, then if n-1 threads are already waiting in tree-lock, and thread n starts waiting to acquire the lock, thread n needs to wait for all n-1 threads to lock/unlock the tree-lock at most.

Starvation Freedom
Starvation freedom is that if one thread tries to acquire the lock it eventually acquires it. Because of sketch proof above (about fairness), we know that once a threads starts waiting a peterson lock, it will acquire it right after the owner thread unlocks it (even through the previous owner tries to acquire it again before the waiting thread checks flag and victim). And we know that once a thread acquires a node's lock, it goes to try to acquire lock from node's parent. And we can start same reasoning again for the parent node because it also uses peterson lock. We continue this logic to the root. Therefore, tree-lock is starvation free.
As mentioned in the description of this problem, threads will be busy taking naps and vacations but eventually return and unlock.
Deadlock freedom

Tree-lock is starvation freedom therefore deadlock freedom.

2. 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?
   Answer
   Yes, it is the number of nodes already waiting in the tree-lock, each one will need to lock and unlock the tree-lock. Plus the unlocking of the tree-lock made by the thread that is owner of the tree-lock at the time "a thread" starts waiting. Sketch of proof is same as the one in the fairness and starvation freedom above: Once thread starts waiting for the leaf peterson lock is certain it will be next one to take lock after leaf peterson lock is unlocked. After thread acquires lock it tries to acquire node's parent lock and so it waits to acquire it (if it is already owned by different thread). Because parent node is also peterson lock, thus same logic applies to parent. And same logic applies to parent's parent until root.

3. 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.

   Answer please find below the table with the performance raw data. Each number is the experiment *after* warm-up. Also please find in figure 7 the average performance of tree peterson comparing local vs. remote.

pawpaw.rlogin	640,000 max count	Time in millisec.	Thread count					
Time of day	Host	Iteration	2	4	8	16	32	64
08:30:00 AM	Local	1	195	431	975	6520		
08:30:00 AM	Local	2	208	467	977	5192	36723	
08:30:00 AM	Local	3	200	447	1014	5660	36047	271399
08:30:00 AM	Remote	1	747	1557	3893	8106	18609	54593
08:30:00 AM	Remote	2	783	1621	4819	8495	17867	76295
08:30:00 AM	Remote	3	821	1416	4034	5308	17632	70018
10:00:00 AM	Remote	1	770	1609	4499	5172	18330	58966
10:00:00 AM	Remote	2	788	1600	4059	7435	17046	78749
10:00:00 AM	Remote	3	796	2020	4303	6983	21346	74378
12:00:00 PM	Remote	1	824	1630	4321	6398	19654	58875
12:00:00 PM	Remote	2	794	2155	4072	7516	18383	78437
12:00:00 PM	Remote	3	792	1467	5021	6115	18519	90523
02:00:00 PM	Remote	1	781	1784	3249	8079	19717	52259
02:00:00 PM	Remote	2	775	1978	4454	5652	17864	56133
02:00:00 PM	Remote	3	777	1509	3687	5531	18498	66431

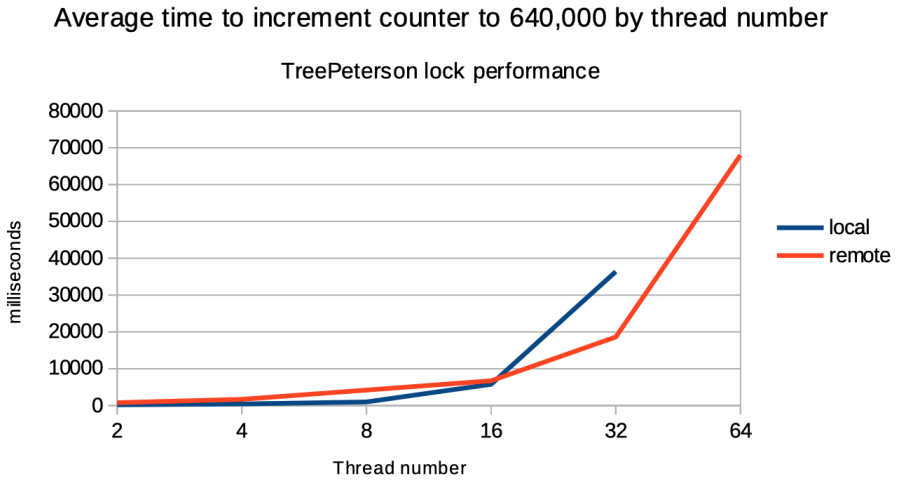

Figure 7:

2.3 [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:

1. L-Exclusion: At any time, at most L threads are in the critical section.
2. 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.

Answer

```
1 public void lock(){
2     int i = ThreadID.get();
3     flag[i] = true;
4     label[i] = max(label[0], ..., label[n-1]) + 1
5     while( true){
6         int counterThreads = 0;
```

```
7           for(int k = 0 ; k < stamps.length ; k++){
8               if(k != i){
9                   if(flag[k] && label[k] << label[i]){
10                      counterThreads++;
11                  }
12              }
13          }
14          if(counterThreads <= L-1){
15              break; //while loop
16          }
17      }
18  }
```

the unlock method remains as-is. Based on line numbers from above algorithm. Proof this works is made by sketch:
(1) The only threads ,k, for which condition in line 9 is true ($flag[k]\&\&label[k] << label[i]$) are the threads that are (A) either in critical section or, (B) are threads that are waiting to enter critical section whose $label[k]$ has not been set yet (for example these threads have got its label in line 4 but assignment has not been made to $label[k]$) or, (C) are threads that are waiting to enter critical section and its $label[k]$ is already set to an earlier time than thread's i $label[i]$.

(2) Because previous statements are true, then, we know we will never go above the limit of L number of threads in critical section.

Below is execution trace (the parameters are like this , 16 threads in total, each acquire lock twice. L is 4). We can see that the number of threads in critical section never goes above 4 (L is 4) although it does reach 4. And all the threads finish that is, starvation freedom is satisfied. The test class is *Test3.java*.

Table 1: A simple longtable example

Execution sequence	Total number of threads in critical section
Thread [0] inside CS !	1
Thread [0] about to exit CS !	0
Thread [4] inside CS !	1
Thread [4] about to exit CS !	0
Thread [5] inside CS !	1
Thread [5] about to exit CS !	0
Thread [6] inside CS !	1
Thread [6] about to exit CS !	0
Thread [7] inside CS !	1
Thread [7] about to exit CS !	0
Thread [8] inside CS !	1
Thread [8] about to exit CS !	0
Thread [9] inside CS !	1
Thread [9] about to exit CS !	0
Thread [10] inside CS !	1

Thread [10] about to exit CS !	0
Thread [0] inside CS !	1
Thread [0] about to exit CS !	0
Thread [4] inside CS !	1
Thread [3] inside CS !	2
Thread [4] about to exit CS !	1
Thread [3] about to exit CS !	0
Thread [6] inside CS !	1
Thread [6] about to exit CS !	0
Thread [7] inside CS !	1
Thread [7] about to exit CS !	0
Thread [8] inside CS !	1
Thread [8] about to exit CS !	0
Thread [5] inside CS !	1
Thread [2] inside CS !	2
Thread [1] inside CS !	3
Thread [2] about to exit CS !	2
Thread [1] about to exit CS !	1
Thread [10] inside CS !	2
Thread [3] inside CS !	3
Thread [10] about to exit CS !	2
Thread [3] about to exit CS !	1
Thread [11] inside CS !	2
Thread [9] inside CS !	3
Thread [11] about to exit CS !	2
Thread [1] inside CS !	3
Thread [5] about to exit CS !	2
Thread [2] inside CS !	3
Thread [11] inside CS !	4
Thread [2] about to exit CS !	3
Thread [11] about to exit CS !	2
Thread [9] about to exit CS !	1
Thread [12] inside CS !	2
Thread [1] about to exit CS !	1
Thread [14] inside CS !	2
Thread [13] inside CS !	3
Thread [12] about to exit CS !	2
Thread [14] about to exit CS !	1
Thread [12] inside CS !	2
Thread [13] about to exit CS !	1
Thread [14] inside CS !	2
Thread [12] about to exit CS !	1
Thread [15] inside CS !	2

Thread [14] about to exit CS !	1
Thread [13] inside CS !	2
Thread [15] about to exit CS !	1
Thread [13] about to exit CS !	0
Thread [15] inside CS !	1
Thread [15] about to exit CS !	0