CS706 Fall 2016  
Homework #2

1. Exercise 2.1 part (a) only
   1. Mutual Exclusion: Yes
      1. Lines 2-5 guarantee mutual exclusion. The deciding variable is in fact the “turn” variable which breaks ties and decides which thread gets to go.
   2. Progress: (assuming neither thread can die) Yes
      1. Line 2 controls the progress. If the other threads flag is set to false, then the thread that wants the critical section can simply enter.
      2. Failure Example: Here is a series of events where progress is not guaranteed if a thread dies.
         1. Turn = 0 (init) → T0 (1) → CS → T1 (1) → T0 (2) → T0 Dies → CS → T1(2) → T1 (3) → T1(4) → T1(5) Infinitely...
   3. Bounded Waiting: (assuming neither thread can die) Yes
      1. Line 2,3,6,8, and 9 control bounded waiting. Importantly, the variable “turn” enforces a ping pong effect when both threads want the critical section. When this occurs, it forces the threads to alternate.
      2. Failure Example: Assuming that a thread can die, this fails for the same reason as above
2. Exercise 2.2. You may want to consider condition at-most-once on page 28  
    to justify your answer.
   1. In Peterson's algorithm, every **assignment** statement **is** atomic. This can formally be proved since every assignment statement meets the second condition of the at most once principle, “*e contains no critical references, in which case x may be read or written by other threads*.” However, **not every expression** in Peterson’s algorithm is atomic. For example, the while loop condition expressions violate at most once since they contain more than one critical reference  
        
      *while (intendToEnter1 && turn == 1)*

*while (intendToEnter0 && turn == 0)* Both of those conditional expressions have more than one critical reference.

1. Exercise 2.7

The algorithm simply finds the largest number, and increments it by 1. Each thread keeps incrementing the maximum value with the expression “number[i] = max(number)+1”. And since increment is all that is done, It grows withourt bound and eventually overflow.

No, this solution will not work because as soon as one thread overflows the ticket value in number[], its ticket value, *number[i]*, will revert back to 0, allowing it to enter the critical section along with another thread that also has ticket 0. This violates the mutual exclusion.

1. Exercise 2.9
   1. No, it is not a solution to the n thread critical section problem because it does not satisfy bounded waiting or progress.
   2. volatile int v = 1  
      volatile int lock\_v = 0  
        
      while (DecrementAndCompoare(v)){  
       while (lock\_v){;} //busy loop  
       lock\_v = 1;  
       v++;  
       lock\_v = 0;  
      }  
        
      Critical section;  
        
      while(lock\_v) {;} busy loop  
      lock\_v=1;  
      v++;   
      lock\_v=0;  
      exit;
2. Exercise 2.11 part (a) (Declare an AtomicBoolean object b and use b.getAndSet() and b.set().)  
    (See <http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/AtomicBoolean.html>)

AtomicBoolean lock = new AtomicBoolean(false);

Each thread executes

while (true) { (1)while (lock.getAndSet(true) == true) {;}

(2) critical section

(3) lock.set(false);

(4) noncritical section

}

1. Read the article "Brief Announcement: A partitioned Ticket Lock" and answer the questions: <http://cs.gmu.edu/~rcarver/cs706/spaa11-PartitionedTicketLock.pdf>
   1. Why is the lock considered to be "partitioned"?
      1. The lock is considered to be partitioned because of the array of the Grant variables in this system as opposed to the naive Ticket lock system that has a single Grant variable. Wherein the naive system all the threads spin on the single Grant variable, In the Partitioned lock the pool of the spinning Threads gets partitioned across the indexed Grant variables of the array..
   2. The code uses expression(T & (GRANTSLOTS - 1)). Read the following article and answer these questions: <http://psy-lob-saw.blogspot.com/2014/11/the-mythical-modulo-mask.html>
   3. What is the purpose of this expression? Answer this by showing an equivalent expression that uses operator %.

It is well known that % of power of 2 numbers(notice that GRANTSLOTS is 4) can be replaced by a much cheaper AND operator to the same power of 2 - 1 (that is GRANTSLOTS - 1).

The equivalent expression is (T % GRANTSLOTS), it was replaced by a much cheaper expression(T & (GRANTSLOTS - 1))

* 1. The article shows 4 different methods: moduloLengthNoMask, moduloConstantLengthNoMask, moduloLengthMask, and moduloMask. Which one of these corresponds to the expression (T & (GRANTSLOTS - 1))?

moduloLengthMask(), However the method accesses the ints.length property which may have different benchmark performance than the enum GRANTSLOTS.

moduloMask is also similar but the GRANTSLOTS -1 is preclaculated.

* 1. Which of these 4 methods corresponds to the expression you gave in your answer to (c)?
     1. moduloConstantLenthNoMask
  2. How much faster is the expression (T & (GRANTSLOTS - 1)) than the expression you gave in your answer to (c)?
  3. It is almost 3 times faster. Again in the article, ints.length property is used which may have different benchmark performance than the enum GRANTSLOTS
  4. Explain how the partitioning works by showing how a ticket T=1 and a ticket T=2 will wait on different slots of array Grants.

“L­->Grants[T & (GRANTSLOTS­-1)].\_Grant” is the expression to be evaluated to show that the ticket T=1 and ticket T=2 will wait on different slots.

GRANTSLOTS = 4

The array Grants has slot0, slot1, slot2 and slot3

For T=1:

L­->Grants[1 & (4-1)].\_Grant

1 & 3 ===> (001) & (011) ===> 001 = 1

Therefore, T=1 waits on the slot1 of the array.

For T=2:

L­->Grants[2 & (4-1)].\_Grant

2 & 3 ===> (010) & (011) ===> 010 = 2

Therefore, T=2 waits on the slot2 of the array.

1. Read the article "Degrees Of (Lock/Wait) Freedom" and answer these questions:  
   <http://psy-lob-saw.blogspot.com/2015/05/degrees-of-lockwait-freedom.html>
   1. What degree of lock/wait freedom do the spin-loops in Section 2.3.1 have?

Only Lock Freedom And Obstruction Freedom. It lacks wait Freedom because it doesn't guarantee bounded waiting.

* 1. What degree of lock/wait freedom does your answer to (5) have? You may want to look at http://stackoverflow.com/questions/17418001/atomicboolean-where-is-the-lock or other internet references.
     1. If don’t consider JVM into our system, then the solution to 5 **has Lock Freedom** and obstruction Freedom but **not Wait Freedom**. Because again it doesn’t guarantee bounded waiting.

1. Read the paper "Internally Deterministic Parallel Algorithms Can Be Fast" and answer these questions:  
   <http://www.cs.gmu.edu/~rcarver/cs706/InternallyDeterministic.pdf>
   1. Define external determinism

External Determinism means program always produces the same output when run on the same input. Program executions for a given input may vary but it “converges” to the same output each time.

* 1. A trace is a path through a DAG of the program and the final state of the program. Define internal determinism.
     1. The Internal Determinism is defined as: A program is internally deterministic if for any fixed input I, all possible executions with input I result in equivalent traces.

Basically it means that given an input, the trace, and structure of the DAG is unique and fixed for all the possible interleavings with that particular input.

* 1. Consider operations \* and + on integers, do these operations commute?

if + and \* executed separately, they commute

x+y== y +x

x\*y == y\*x

if executed together.They don't commute.

ex. p,q,r= 5

operations 1) x= p + q; and 2) y= x \* r;

1. -> 2) gives 50 and 2) -> 1) gives 30
   1. Consider operation increment() on integers. Do increment operations commute? Note that commutativity does not imply atomicity.
      1. Yes, because the order of increments doesn’t matter, the final state remains the same.
   2. Consider operation increment() on integers. Is a sequence of increment() operations linearizable? Always? Possibly?
      1. A series of increments is linearizable as long as they are atomic. Otherwise, we can use locks.
   3. Consider the priority write operation defined in the paper.
      1. explain why priority writes commute

By definition two operations commute if the final states of their execution are equivalent as well as the return values for each operation are equivalent.

For pwrite() no matter the order of execution, the final value of the memory cell is the maximum among all the possible values that could have been written by the sequence of operation.

The final state are equivalent.

The operation doesn't return any values, which implicitly makes the return value of operation equivalent.

Hence, It satisfies the definition of commutativity.

are priority writes linearizable? Explain.

From a look at the paper and its example, from an abstract level they are in linearizable, as the same state is generated no matter the execution order.

But at the instruction level, there are no locks available , and interrupts are enabled, we cannot ensure autonomic execution. Threads can be suspended and resumed at the operating systems .

* 1. Consider the algorithm for "Remove Duplicates", which is described in Section 5.3.This algorithm uses the dynamic map described in Section 3, which is implemented as a hash table that uses linear probing to resolve collisions. Suppose A and B have the same hashCode value of 6 in a hash table with places 0 .. 100.
     1. Where is A inserted in the table if A is hashed before B, i.e., B is not yet in   
         the table when A is inserted, and no other elements are in the table?

A will be stored in the first empty slot at the hash location 6.

* + 1. Where is A inserted in the table if A is hashed after B, i.e., B is in   
        the table when A is inserted, no other elements are in the table, and A has higher priority.
       1. A is put in the location 6 (slot of B) , and B is evicted. The linear probe will continue to find a slot for B and it will put B in the next empty or low-priority element's slot and so on.
    2. Where is A inserted in the table if A is hashed after B, i.e., B is in the table when A is inserted, no other elements are in the table, and B has higher priority than A.
       1. A will be stored in the next empty slot 7 after B.
    3. Where is A inserted in the table if A is hashed after B, i.e., B is in the table when A is inserted, no other elements are in the table, and A and B have equal priority.

The paper states: If the slot is occupied by an equal-priority element, either the new or old element is discarded (deterministically based on priority). It depends on what priority criteria it resorts to: FCFS or the latest value having higher priority.

Assuming FCFS, A will be discarded. And stored in the next empty slot 7.

* + 1. Suppose one thread inserts A and one thread inserts B, show how the result of inserting A and B is deterministic, i.e., you get the same result no matter which order A and B are inserted.
       1. If A and B have different priorities then, no matter the order of insertion or the interleaving may be , the element with the higher priority will finally end up in the hashed location.
       2. If they have priority, the order of execution and the underlying tie breaking priority mechanism does matter. For instance with (FCFS), the first element to be inserted will remain, and the other will be discarded.
       3. Assuming A and B have the same priority (6) and HashCode, It is deterministic, however ordering does impact the final result. The dynamic map implementation states that **"If the slot is occupied by an equal-priority element, either the new or old element is discarded (deterministically based on priority) and the operation completes"**. Assuming that the fixed scheme to break ties is first come first serve (FCFS), that means that the first element to be inserted will remain, and the other discarded.  
          However, if A and B do not have the same priorities, then it is deterministic since.
  1. Summarize the experimental results in a few sentences.  
       
     The single core performance for the internally deterministic algorithm was similar to the the serial/traditional algorithm. Its due to the complex coding overheads in the internal.

But on multi-core system, the performance is a lot better which goes to show that the internal deterministic algorithm can achieve great speed up.

In most of the cases, the internally deterministic approach on a single core was slightly slower than the serial versions. This was typically due to the overhead in coding style. However, when multiple cores were introduced, the internally deterministic algorithms pretty much blew almost all of the other algorithms out of the water. Which is actually really awesome to see!

* + 1. Comment on this paragraph from the paper: "It would also be interesting to conduct an empirical study supporting the programmability and debuggability claims for internal determinism. We have provided evidence that the programs in this paper have short code descriptions, but we have not studied how natural these programs are to develop in the first place."
       1. They are basically saying that they have not provided profound evidence that it is in fact simple to program and mentally comprehend a variety of problems in this internally deterministic style. Additionally, they never actually showed how to efficiently debugging their code.

Here they just discuss about how convenient or feasible would it actually be to develop these internally deterministic algorithms to provide such a speed up. How easy it will be to handle, debug such algorithms.