COL380 Homework-1

Pradyumna Meena

2016CS10375

Q 1

Overall memory access time = 0.99*1 + 0.01*100 = 1.99 ns

For SIMD Architecture:

Only one instruction will be issued in this case. If a processing element performs consume instruction then the next instruction it will perform will be a produce instruction. Hence the total time to execute 2 instructions (produce and consume) = 1.99*2 + 20 + 25 = 48.98 ns.
Therefore performance in this case is 2/48.98 ns = 40.83 MFLOPS

For MIMD Architecture:

Due to multiple instructions issued each processing element will be able to do its respective operation. And since every processing element is performing one operation only one data access will be required. Therefore the total time to execute 2 instructions (produce and consume) = 1.99 + 25 = 26.99 ns.

(Since each processing element is executing a single instruction and consume operation takes more time hence time for execution of the 2 instructions is taken to be 25 ns)

Therefore performance in this case is 2/26.99 ns = 74.1 MFLOPS

Q 2

The difference between *regular* registers and *atomic* registers only comes in picture when read operations overlaps with any write operation. In such a case since the write in *regular* registers is not atomic hence the value can be either the old value or the new value unlike *atomic* registers. If the requests for lock by two threads A and B are non-concurrent then there will not be any issue as there won't be any overlaps. In concurrent requests for lock, issues will arise. Consider two threads A and B where A's execution is stalled in the while loop and B changed its flag to true. Since the read by A and write by B overlap, using the *regular* registers can lead to unexpected outcomes. Total of 4 possibilities arise shown as below along with their consequences. In every possibility A enters its critical section.

	Old victim value (corresponding to A)	New victim value (corresponding to B)
Old flag value of B	A enters CS_A	A enters CS_A
New flag value of B	A enters CS_A	A enters CS_A

If reading of victim by A overlaps with writing of victim by B then either it will read old value (A itself) or new value (B). If it reads new value then A will enter its critical section but in the other case it will keep looping until B has finished writing into victim and then A will enter its critical section. Hence in all cases A will enter its critical section and B will wait until A has finished its critical section and unlocks the lock. Therefore the algorithm is correct even if we use regular registers.

Q 3

Mutual Exclusion:

From the code we can say that

$$w_A(turn=A) \text{ -> } r_A(busy==false) \text{ -> } w_A(busy=true) \text{ -> } r_A(turn==A) \text{ -> } CS_A$$
$$w_B(turn=B) \text{ -> } r_B(busy==false) \text{ -> } w_B(busy=true) \text{ -> } r_B(turn==B) \text{ -> } CS_B$$

Suppose the algorithm does not satisfy mutual exclusion and A and B both are in their critical sections. Let A be the last thread to write to turn. That means

$$\mathbf{r_B(turn==B) \text{ -> } w_A(turn=A)}$$

Since A is in its critical section it must have read busy as false. Hence we get

$$w_B(turn=B) \text{ -> } r_B(busy==false) \text{ -> } w_B(busy=true) \text{ -> } r_B(turn==B) \text{ -> } w_A(turn=A) \text{ -> } r_A(busy==false)$$

There was no other write to busy variable. Hence we arrive at contradiction and therefore mutual exclusion holds.

Deadlock freedom:

The algorithm is not deadlock free. Consider the execution of the following steps by two threads A and B

$$w_A(turn=A) \text{ -> } r_A(busy==false) \text{ -> } w_A(busy=true) \text{ -> } w_B(turn=B) \text{ -> } r_B(busy==false)$$

After execution of these steps by the two threads both of them will keep looping into the while loop as the busy variable has been set to true and since both of them are stuck in while loop the value cannot be changed and hence a deadlock.

Starvation freedom:

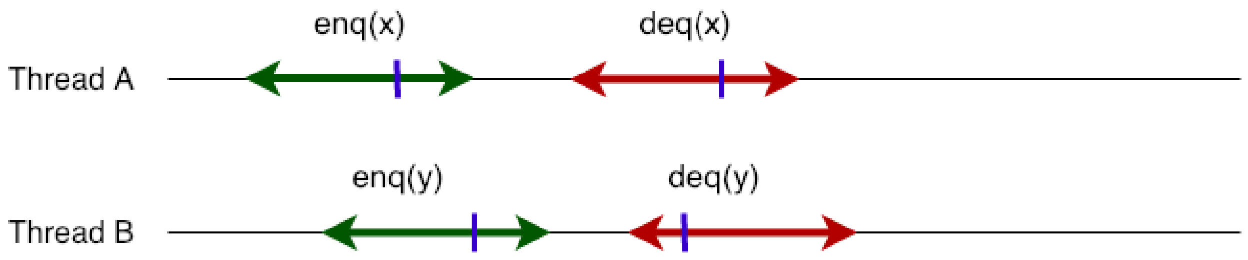
Since the algorithm is not deadlock free it can never be starvation free as both threads who are stuck in deadlock attempted will never be able to access their critical sections. So the given algorithm is not starvation free.

Q 4

Sequential consistency means that result of any execution of the program is same as if we had re-written the program sequentially. Also the instructions of each parallel component appear in the same order in our sequential execution as they were in the original program. Since the method calls by different threads are unrelated by the program order, if there is any blocking encountered we can always reorder the method calls by the threads and can make the execution non-blocking.

Q 5

Two threads are trying to enqueue two values and x and y and then dequeue them as represented in the diagram below. The blue lines show the set of linearization points. According to sequential behaviour the final state of the queue object should be an empty queue. But these set of linearization points show different behaviour as x is enqueued before y but at the time of dequeue y is seen before x. Hence the queue object is not linearizable.



Q 6

Since the lock variable is shared changing its value by a single thread will make other threads discard their own copies of the variable present in their own memory. In first implementation every time the lock variable is set to true while waiting all other threads are made to discard their own copy of lock variable and fetch new value from the memory. On the other hand in the second implementation the value is not changed until the very last moment and hence is computationally less expensive. Hence Lock1 implementation will be slower as compared to Lock2 implementation.