COL380 Homework 1 Submission

Shashwat Shivam
2016CS10328

1. Solution

   Let us first calculate the total number of instructions required to complete the task for 40K words. Since alternate i is produced and consumed, 20K words are called with consumeItem() and 20K words are called with produceItem(). The total number of instructions required are:-

   $$20K * 20 + 20K * 25 = 400000 + 500000 = 900000$$

   a. SIMD Architecture

      In SIMD architecture the code is the same for all parallel processing units. If we consider that each processor gets a thread with variable i at gaps of 4, all threads alternatively call produceItem() and consumeItem() taking 20 and 25 instructions respectively. If a single instruction stream is sent for all data then completing one thread will wait for it's case in if/else. So the time for individual cases of produceItem() and consumeItem() will be added.

      Considering 99% cache hit ratio, we can assume 99% of words processed take 1ns whereas 1% of words have an extra 100ns wait time for the cache to be fetched.

      In total a SIMD architecture to process 40K words spread over 4 processing units we need (weighting times according to the cache hit ratio and adding latency),

      $$0.99 * 10K * (20 + 1)ns + 0.01 * 10K * (20 + 100\ ns) + 0.99 * 10K * (25 + 1)ns + 0.01 * 10K * (25 + 100\ ns)$$
      $$= 465300\ ns + 24500\ ns$$
      $$= 489300\ ns$$

      So the peak performance in GFLOPs is,

      900000 / 489300ns = **1.8394 GFLPOs**

   b. MIMD Architecture

      In MIMD architecture different processors can execute different code independently. This allows the threads with different if/else to execute simultaneously. The processors alternatively take 20 and 25 cycles to complete a thread. We can assume on average every processor takes 22.5 cycles per thread.

Using the same cache explanation as in SIMDs.

In total a MIMD architecture to process 40K words spread over 4 processing units we need (weighting times according to the cache hit ratio and adding latency),

$$0.99 * 10K * (22.5 + 1)\ ns + 0.01 * 10K * (22.5 + 100\ ns)$$
$$= 232650 + 12250\ ns$$
$$= 244900\ ns$$

So the peak performance in GFLOPs is,

900000 / 244900ns = **3.675 GFLPOs**

Using MIMD architecture we get a speedup of approx. 100%

2. The Peterson's Lock is as follows:-

```
bool flag[2];
Int turn;

void lock() {

        int i = tid(i);
        int j = 1-i;
        flag[i] = true;
        turn = i;
        while(flag[j] && turn == i);

}
```

Here the steps which affect variables common to the 2 threads are the line which assigns turn variable and the while loop which reads the turn variable and the flag variable of the opposite thread.

When both the threads are trying to access the critical section simultaneously, we can have the following cases (with properties of regular registers):-

1. W_1(turn) -> R_1(flag,turn) -> W_2(turn) -> R_2(flag,turn) (Without loss of generality similarly covers case when 1 and 2 are interchanged.) :- At R1 there are 2 cases. If the flag of thread 2 is false, thread 1 will enter critical section and at R2 thread 2 will see the flag of thread 1 true and turn written by the same thread so thread 2 will not enter the critical section. If the flag of thread 2 is true , thread 2 will not enter the critical section, but after W2, at R2 thread 2

will not be able to enter the critical section and when step R1 again occurs it will enter the critical section.

2. W_1(turn) -> W_2(turn) -> R_1(flag,turn) -> R_2(flag,turn) :- At R1 thread 1 sees turn set by other thread, therefore it enters critical section. At R2 flag is true and turn is set by same thread so it cannot enter critical section.
3. W_1(turn) -> W_2(turn) -> R_2(flag,turn) -> R_1(flag,turn) :- At R2 flag is true and turn is set by same thread so it cannot enter critical section. At R1 thread 1 enters critical section as turn is set by other thread.
4. W_2(turn) -> W_1(turn) -> R_2(flag,turn) -> R_1(flag,turn) :- At R2 turn is set by other thread so R2 will enter critical section. At R1 flag of thread 2 is true and turn is set by thread 1 so it will not enter critical section.

Considering each case and following properties of regular registers, we can see that only one thread enters the critical section. So by case analysis, we can say that Peterson's Lock Algorithm mutual exclusion even if atomic registers are replaced by regular registers.

3. Solution

For a thread to get into the critical section, the following steps are followed by the algorithm:-

W(turn) -> R(busy) -> W(busy) -> R(turn) -> CS

Now consider the following:-

a. <u>Mutual Exclusion</u>

Consider that the algorithm doesn't guarantee Mutual Exclusion. Consider thread A which has gotten into critical section and there are k other threads trying to get into the critical section. For thread A to get into the critical section the above sequence of steps will have to be followed. This means that the last known value of turn was = A. After this any other thread which was not in the inner loop will have seen thread = A and gone back to the inner loop. Now in this loop for another thread to get into critical section the following sequence of steps will have to be followed:-

W_A(turn) -> R_A(busy) -> W_A(busy) -> R_A(turn) -> CS_A -> W_i(turn) -> R_i(busy) -> W_i(busy) -> R_i(turn) -> CS_i

For this to be possible, R_I(busy) will have to be false but considering registers to be atomic, the only way R_i(busy) will be false if it happens before W_A(busy) which will cause a cycle in the program execution order which is not possible. (Even if multiple threads are considered for this order, no thread will be able to exit the inner loop using the above argument). This contradicts our assumption that mutual exclusion is not guaranteed.

Therefore, the above implementation of Peterson's lock guarantees mutual exclusion.

b. Deadlock Freedom

Consider the following execution of 2 threads:-

$$W_1(turn) \rightarrow R_1(busy) \rightarrow W_1(busy) \rightarrow W_2(turn) \rightarrow R_1(turn)$$

Since at last step turn != 1, thread 1 will not get into the critical section and since busy = true, thread 2 will also not get out of the internal loop. This leads to both threads stuck in internal loops with no progress and thus a deadlock. (NOTE:- The above serialization is valid as it follows the individual order of events in each thread correctly.)

c. Starvation Freedom

Since the given algorithm is not deadlock-free, it is also not starvation free.

4. According to the definition of sequential consistency, we can assume the following:-
   a. Method calls should appear to happen in one-at-a-time, sequential order.
   b. Method calls should appear to take effect in program order.

   Considering the above conditions if we serialize a sequentially consistent program, then a blocking call will not allow any other call to occur in the serial history (since method calls can be reordered as long as they stay in program order). This will violate sequential consistency of the program. Thus sequential consistency has to be non - blocking.

5. Consider the case of enq() method for a single thread and non-empty queue.

   $$R(tail,head) \rightarrow W(items[tail\%items.lenght]) \rightarrow W(tail)$$

   Consider the following case when the 2 threads call enq() method simultaneously. (Assume queue is not full)

   $$R_1(tail,head) \rightarrow R_2(tail,head) \rightarrow W_1(items[head\%items.length) \rightarrow W_2(items[head\%items.length])$$

   We see that two threads write in the same location and cause one of the values being enqeued to be lost. This is against the sequential specification of the queue object. And since this happens before the linearization point (point where threads observer change in tail value), therefore this is possible during execution of the methods.

   Seeing the above example we can see that the given object is not linearizable.

6. The Lock1 implementation is able to acquire the lock if the previous state was false. This happens atomically. So the lock acquisition is atomic in lock1.

   The lock2 implementation goes through 2 loops to acquire a lock. For a lock to be acquired in lock2 first the state is checked to not be false, which allows the lock to exit the internal loop and then another if statement sets the lock to true and returns acquisition of the lock if the previous state was false again. So the lock2 needs to see state == false 2 times to acquire the lock.

   However, if we compare the number of writes, then lock1 does drastically more write to acquire the lock whereas lock2 only reads the value for state and tries to write only when it sees the state == false.

   Although the number of instructions required to acquire the lock is lesser in lock1 due to the overhead of writing to state many times, lock2 is more efficient than lock1 preventing too many writes.