

Arneish Prateek [2014CH10786]

## Solution 1

1. Performance in the SIMD architecture:  

$$10000 \text{ iterations times average time per iteration} = 10000 \times [(avg \text{ time per } \texttt{produceItem}() \text{ call}) + (avg \text{ time per } \texttt{consumeItem}() \text{ call})]$$

$$= 10000 \times [20 \times (0.99 \times 1 + 0.01 \times 100) + 25 \times (0.99 \times 1 + 0.01 \times 100)] ns$$

$$= 0.896 ms$$
2. Performance in the MIMD architecture:  

$$10000/2 \text{ times } [(avg \text{ time per } \texttt{produceItem}() \text{ call}) + (avg \text{ time per } \texttt{consumeItem}() \text{ call})]$$

$$= 10000/2 \times [20 \times (0.99 \times 1 + 0.01 \times 100) + 25 \times (0.99 \times 1 + 0.01 \times 100)] ns$$

$$= 0.448 ms$$

## Solution 2

1. For a single-threaded execution, Peterson's two-thread ME algorithm would work with regular registers since the execution would follow a total order on reads and writes to the memory registers corresponding to `flag[]` and `victim`.
2. Read-write overlaps can arise in a two-threaded race to acquire the lock, corresponding to shared memory accesses to `victim` and `flag` variables, as explained hereafter: without loss of generality, assume `thread A` is at the `while-loop` where it must read `flag[B]` and `victim` to proceed. Assume that while `A` is reading `flag[B]`, concurrently, `thread B` is writing `flag[B]` to `True`. With regular registers, two behaviours are possible while locking:
  - I. `Thread A` reads `flag[B]==False` (old value)
  - II. `Thread A` reads `flag[B]==True` (new value, written by `thread B`)
3. Following this, corresponding to the shared memory register for `victim`, again two behaviours are possible:
  - I. `Thread A` reads `victim==A` (old value)
  - II. `Thread A` reads `victim==B` (new value, written by `thread B`)
 In any combination of the scenarios above, `thread A` would succeed (at least *eventually*) in acquiring the lock (since `B` must necessarily write `victim=B`, and `flag[A]==True` is unaffected in the above scenario). `Thread B` will be stuck in the `while-loop`.
4. Similarly, during unlocking (assuming `WLOG thread A` is writing `flag[A]=False`, and concurrently, `thread B` is reading `flag[A]` in its `while-loop`), `thread B` could read either:
  - I. `flag[A]==True` (old value)

II. `flag[A]==False` (new value, written by `thread A`)

In both cases, B succeeds (*immediately* in the second case, *eventually* in the first) in acquiring the lock and A in unlocking.

Therefore, accuracy of the Peterson's two-thread lock is **preserved** while using regular registers.

## Solution 3

1. The proposed `n`-thread generalisation of the Peterson's algorithm is **mutually exclusive**. *Proof:* By contradiction, assume concurrent threads A and B are executing in the critical section simultaneously. For each thread, the program order immediately prior to entering the CS would be as follows. For `thread A` and `thread B` respectively:

1.1:  $W_A[\text{turn}=A] \rightarrow R_A[\text{busy}==\text{False}] \rightarrow W_A[\text{busy}=\text{True}] \rightarrow R_A[\text{turn}==A] \rightarrow \text{CS}_A$

1.2:  $W_B[\text{turn}=B] \rightarrow R_B[\text{busy}==\text{False}] \rightarrow W_B[\text{busy}=\text{True}] \rightarrow R_B[\text{turn}==B] \rightarrow \text{CS}_B$

Without loss of generality, assume `thread A` is the last thread to write to `turn`, i.e. `turn=A`. Since the value of `turn` remains A after the last write, the following order must hold:

1.3:  $R_B[\text{turn}==B] \rightarrow W_A[\text{turn}=A]$

Combining 1.1-1.3, we obtain:

1.4:  $W_B[\text{turn}=B] \rightarrow R_B[\text{busy}==\text{False}] \rightarrow W_B[\text{busy}=\text{True}] \rightarrow R_B[\text{turn}==B] \rightarrow W_A[\text{turn}=A] \rightarrow R_A[\text{busy}==\text{False}]$

Since `busy` is never set to `False` after the write by `thread B` in order 1.4, `thread A` cannot possibly read `busy==False`. Thus, we have a contradiction. Hence, proved.

2. The lock does **not** have **deadlock-freedom**. Consider the following order of events that constitutes a counterexample in a two-threaded concurrency involving `thread A` and `thread B`:

2.1:  $W_A[\text{turn}=A] \rightarrow R_A[\text{busy}==\text{False}] \rightarrow W_A[\text{busy}=\text{True}] \rightarrow W_B[\text{turn}==B] \rightarrow R_A[\text{turn}==B]$

Since `busy` once set to `True` remains `True` in the race to acquire the lock, both A and B in the above execution would loop infinitely in the first `do-while` block.

3. The lock does **not** have **starvation-freedom**. This is a direct consequence of fact 2 above: since starvation-freedom implies deadlock-freedom, absence of deadlock-freedom implies the absence of starvation-freedom. In the counterexample of point 2, both threads A and B issue a request for acquiring a lock but fail to ever acquire it.

## Solution 4

1. Sequential consistency requires that method calls act as if they occurred in a sequential order consistent with program order. In other words, a sequentially-consistent execution is equivalent to a sequential execution(s).
2. A sequential execution implies non-overlapping method call intervals.

From 1 and 2 above, the behaviour of overlapping method calls on a concurrent object in a multi-threaded environment following a sequentially-consistent execution is equivalent to some sequential execution of those calls. Thus, a method call being executed by one thread cannot block the execution of an overlapping call being executed by a different thread since otherwise, there cannot exist an execution equivalence to a sequential order of execution.

## Solution 5

1. Yes, the single-enqueuer, single-dequeuer `queue` implementation is **linearizable**.
2. Linearization points for the `enq()` and `deq()` methods are as follows, respectively:  
`enq()`: depending on the execution, either `line 6:throw FullException()` or `line 9:tail++`  
`deq()`: depending on the execution, either `line 13:throw EmptyException()` or `line 16:head++`
3. **Explanation:** For the `enq()` method, any interleaved `deq()` call(s) **cannot** dequeue the queued element (`line 8`) before the execution of `line 9:tail++`. Therefore, the *effects* of the `enq()` method only become visible after the execution of `line 9`, which justifies the choice of this as the linearization point. In case the `queue` is `Full`, the `enq()` method can be assumed to take effect at `line 6` where `FullException` is raised. This matches specification (c).  
Similarly, for the `deq()` method, any interleaved `enq()` calls **cannot** queue an element at the current `head` (`line 15`) before the execution of `line 16:head++`. Since the effects of `deq()` method only become visible after the execution of `line 16`, it can be chosen as the linearization point. In case the `queue` is `Empty`, the `deq()` method can be assumed to take effect at `line 13` where `EmptyException` is raised, matching specification (a).

## Solution 6

1. Lock2 is more optimal than Lock1. The loop in Lock1 `lock()` function involves writing `state` to `True` in every iteration when the returned value is `True`. To maintain **cache coherence**, following a write the caches corresponding to `state` are repopulated from

# Homework 1

COL 380  
January 27, 2019

---

the memory. This happens after every such write operation. In Lock2 however, the inner **while-loop** involves only a read operation on **state**, which does not invalidate the caches storing **state**. Thus, a thread spinning in Lock1 would trigger repeated fetch from the memory, more often than a thread spinning in Lock2 which only writes to **state** in the if-statement. Thus, because of this **false-sharing**, Lock2 would outperform Lock1.