AST: scalable synchronization
Supervisors' guide 2004

keir.fraser@cl.cam.ac.uk

These are some notes about the topics that I intended the questions to draw on. Do let me know if you find the questions unclear or have suggestions for additional topics. They are not intended as 'model answers'; in particular they often discuss questions in vastly more detail than I would expect, either in supervision work or in the exam.

7.1 The problem here is that a thread could be interrupted during the call to `wait`. If this happens then the ticket it received will be lost and the system will permanently stall when that ticket is selected. The 'solution' of catching the exception and decrementing `nextTicket` is incorrect because subsequent tickets may already have been issued.

One option is to maintain an explicit queue of rejected tickets and to skip over these in `finished`. This is OK if interruption is rare.

Another solution is to relax the fairness requirement and to have a handshake between `finished` and `awaitTurn`. `Finished` would wait t for a signal from `await-Turn` and, if none is received, would increase the turn again. If `awaitTurn` was not scheduled within that window then it would take another ticket. Setting t is hard, but the solution may be simpler to implement (and analagous to the real-world solution). Again, it is likely to be poor if tickets are often lost.

A third solution with very different costs is to assign tickets but to keep the 'active' tickets in a separate data structure. A thread places its ticket into the structure when starting to wait and removes it when it is notified.

For further reading of a modern solution see Scott & Scherer's PPoPP 2001 paper.

7.2 For the single-threaded case to be faster:

$$\begin{aligned} 2(l+s) &< l+s+a+r \\ l+s &< a+r \end{aligned}$$

If we consider contention for the mutex and a mean waiting time of $\frac{1}{2}s$:

$$2(l+s) < l+s+a+r+\left(\frac{s}{s+l}\right)\left(\frac{s}{2}\right)$$

The assumption made here is unrealistic for two main reasons. Firstly, the mean waiting time of $\frac{1}{2}s$ does not take into account how long would be spent by the previous holder releasing the lock; some portion of r should probably be included. Secondly, the two threads' execution will not be independent, e.g. if s, a and r are very short then, once the lock has been contended once, the two threads will start their next phase of local-computation at roughly the same time and so will be likely to contend again. Going further than this analysis would need more information about the distributions of l and s (e.g. see the CSM material on $M/G/1$ queueing theory).

7.3 You should find that `Fairlock` performs much worse than a built-in mutex. After all, the `awaitTurn` and `finished` methods both involve acquiring and releasing a built-in mutex.

If you have access to a larger multi-processor machine, then you will find that the performance does not scale well. The use of `notifyAll` will cause every thread waiting for the lock to be woken whenever it becomes available. Compare this with the fair queue-based locks in Lecture 9 in which only the previous and next lock holders are involved.

7.4 Using separate connections is likely to be faster. The workload is I/O bound. Recall that TCP 'fairness' operates on a per-connection basis rather than a per-client one. If there is congestion within the network then one client will be better off by using many connections rather than by using few. The transfer would be best performed sequentially if a single connection is able to saturate one of the links – e.g. an ordinary dial-up phone link.

8.1 It is reasonable to have more threads than processors when the threads are not executing CPU-bound tasks. The goal is broadly to have as many *runnable* threads as there are CPUs and so a thread pool may be configured with more threads if many of them are expected to be blocked. Deciding how many is 'enough' will depend on the workload (e.g. what other resources the threads are using, or whether there are inter-task dependencies and the risk of deadlock if the thread pool is too small).

8.2 An unbounded input queue avoids the need to consider explicitly what happens with items that arrive when it is full. The downside is obviously that the queue can grow without bound. A related problem in OS design is *receiver livelock* in which the early stages of protocol processing (in this case depositing items in the queue) dominate execution time to the exclusion of servicing existing queues items. No useful progress is made. One approach is to perform some kind of admission control – perhaps not accepting new connections on a `ServerSocket` while the system load is beyond a certain level.

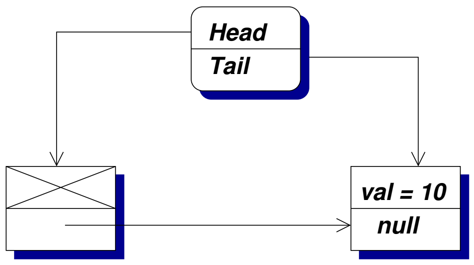
8.3 A reasonable scheme is to place generated items at the head of the queue and to steal items from the tail. This assumes that if one object refers to another then its likely that processing them in that order will give better locality than doing a simple first-come-first-served approach. (Some garbage collectors have the side effect of actively re-arranging the heap in this way while copying data).

Stealing from the tail makes it likely that the items are 'cold', i.e. not held in another processor's cache and not actively being scanned. By allowing only one thread to access the head of the queue it might be possible to use an implementation with better performance.

8.4 Adding `push_head` is reasonably straightforward. A thread proceeds by acquiring the head lock, allocating a new dummy node, linking it to the existing one and storing the value into the previous dummy node.

Note that acquiring the head lock will exclude other `push_head` and `pop_head` operations. `push_tail` operations access only the `next` field in the current tail – notice how this access is safe at any point during the above procedure.

8.5 Supporting two kinds of `pop` operation on a queue like this is a problem because you'll need to make sure that it works when the queue is almost empty. Consider the following case:



Both of the `pop` operations would have to return the 10 node if invoked at this time. There will need to be some arbitration between them – either by holding both of the locks at once (note the possibility of deadlock if they can be acquired in any order) or the use of a multi-word atomic operation (see Detlefs *et al*'s work on double-ended queues using `DCAS`).

9.1 When there are i processors contending for the lock, they will each perform a `CAS` operation (one succeeding, the others failing) and then the lock holder will subsequently perform a write when releasing the lock. Note that this write should also be treated as a worst-case access: although the other `CAS` invocations have failed, they will have invalidated the lock-holders cached copy of the field.

$$\begin{aligned}\sum_{i=1}^{n}(i+1) &= \frac{1}{2}n(n+1)+n \\ &= \frac{1}{2}n^2+\frac{3}{2}n \text{ accesses} \\ &= 85(n^2+3n) \text{ cycles}\end{aligned}$$

What looked like an $O(n)$ operation is now $O(n^2)$.

9.2 The `ReadThenCASLock` will at least not prevent the spinning processors from sharing the cache block in which the `locked` field is located. However, there will still be a 'stampede' whenever the field is set back to `false`, so quadratic behavior is still possible if we assume that all of the processors will observe this.

The alternative version allows a new invocation of `lock` to proceed without executing a failing `CAS` (but still with the stampede problem). A disadvantage of it is that there are more operations to execute in the 'fast' case when the lock is not held – but if the lock is being contended then the initial cache miss will still dominate execution and so the cost may be insignificant.

9.3 The array of integers will most likely a packed in memory. A cache blocks of, say, 64 bytes would therefore hold 16 of the entries. Although there is contention between readers for a single entry in the array, there will be contention for the cache block. The array should perhaps be spread out so that each entry lies on a separate cache block.

9.4 See the Mellor-Crummey & Scott paper, Figure 5.

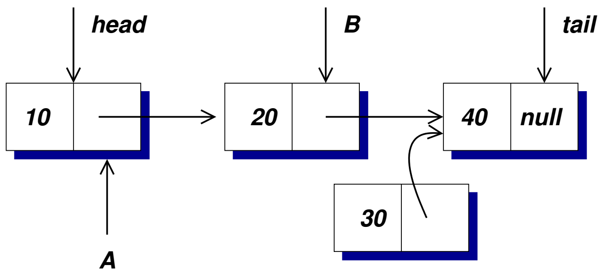
9.5 The queue could be used as a basic work stealing queue, but note that it does not, as it stands, support the usual set of operations of `push_head`, `pop_head`, `pop_tail`. It would be much nicer if the thread that 'owns' the queue could perform more lightweight operations, and to execute them generally independently of stealing. See Shavit & Hendler's work for a proper discussion of how to build a variety of work management queues ("shavit hendler work stealing' on google).

However, the new `push_head` operation could readily be defined: simply create the new head in shared storage pointing to `prev`.

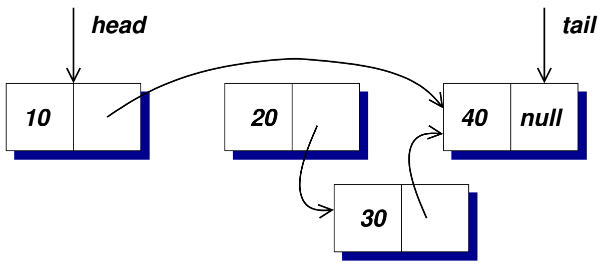
10.1 A wait-free system provides a strong progress guarantee to each invocation of an operation on it – i.e. 'this operation will complete within k cycles'. In principle this makes it suitable for hard real-time systems. A lock-free system provides a system-wide guarantee, so assuming a finite number of threads 'some operation will complete within k cycles'. It says nothing about fairness or about which operation will finish next.

An obstruction free system provides a progress guarantee only when threads run in isolation, i.e. 'if a thread executes in isolation then it can complete an operation within k cycles'. While this sounds like a weak guarantee, notice that 'isolation' does not mean that it is the only thread running, merely that it does not experience contention, e.g. failing `CAS` operations. We've seen in this course how contention impacts performance (e.g. exercise 9.1) and so the argument is that obstruction free designs are OK because contention should be low. There is anecdotal evidence that obstruction free designs are much easier to implement than lock-free ones (see Moir, Luchangco & Herlihy's recent papers).

10.2 Consider two threads in this situation:



One thread is about to delete the 20 node by performing a CAS at A, changing it from the 20 node to the 40 node. Another thread is about to insert the 30 node by performing a CAS at B, changing it from the 40 node to the 30 node. Both can succeed, leaving this non-linearizable mess:



10.3 DCAS would make deletions straightforward. In this example, the deletion would try to atomically set A to refer to the 40 node *and* to set B to be null. Note that *either* that DCAS could succeed *or* the inserting CAS could succeed. Forcing them to contend for the next field in the 20 node prevents both from going ahead. (Several alternative list designs have been developed, see Valois's PODC paper, or Tim Harris's paper at DISC 2001).

10.4 Building wait-free operations over an STM is hard because, order to get a per-thread progress guarantee, we would need to bound the number of times that a transaction might have to retry. It would be unacceptable to allow threads to execute transactions themselves in a FCFS order (that would not be non-blocking – a thread could be pre-empted or fail before its 'turn' arrives).

An alternative would be for threads to 'help' transactions to complete in an FCFS order, but the design becomes intricate, even if the threads can only perform local computation and memory accesses. Also note that 'helping' will usually generate contention, either between helpers or with the original thread.