Technische Universität Berlin

School IV – Electrical Engineering and Computer Science
Chair Communication and Operating Systems (KBS)
http://www.kbs.tu-berlin.de

Integrated Course

Parallel Programming

J. Schönherr

in Summer Semester 2013

Assignment 5

Issued: Wednesday, 19th June 2013

Due: Wednesday, 3rd July 2013

Information

Please upload your solution via the ISIS page of this course until 23:55 of the due date. Your upload should consist of a PDF document covering the theoretical parts and a zip/tar.gz archive with your source files. Please list the names and matriculation numbers of all group members inside your uploaded documents.

Exercise 1 – Lock-free data structures

In https://svn.kbs.tu-berlin.de/svn/ps/public/ss13/pp/a5

you find a FIFO queue that uses coarse-grained synchronization: the queue itself is protected by one lock, that is taken by both, enqueue() and dequeue(). This implementation will most likely work well if the expected contention is very low.

Side quiz: Why does it use explicit locks instead of the OpenMP *critical* directive?

a) Fine-grained synchronization (optional)

There is no good reason, why enqueuing an item should prevent a concurrent consumer to dequeue an item from the other end of the queue.

Implement a FIFO queue, where both ends of the queue can be accessed simultaneously without disturbing each other. That is, how can we fully decouple enqueue() and dequeue() and avoid that these functions need to access both, the head and the tail of the queue?

As an alternative, you may use explained pseudo code, paying attention especially to ordering constraints.

(This part is optional, but might give some ideas towards the next part.)

Base64-encoded hint: TGVhdmUgYSBkdW1teSBub2RlIGluIHRoZSBxdWV1ZSwgc28gdGhhdCBpdCBpcyBuZXZlciBlbXRweS4=

b) Lock-free synchronization

If we now want to allow other producers/consumers to make progress, even if one of the producers/consumer got context-switched away within the enqueue/dequeue operation, we need to remove the locks entirely. Thus, we have to rely on atomic memory operations which have to be carefully arranged.

Quite a few things must be considered for lock-free synchronization:

1. memory reclamation,
2. the ABA problem, and
3. observing other half-finished method calls.

Not all of them are subject of this exercise, we concentrate on the last one and try to mask out the first two (unless you want to cover them).

Memory reclamation:

Consider thread A has just read a pointer which points to a node N and is about to examine the node's contents. If thread A is now delayed for arbitrary reasons and another thread B removes the node N in the meantime and frees the memory associated with N, we can run into serious problems. The “good” case would be that A segfaults; however more likely, it will read invalid data and do funny things (but only rarely, because long delays that increase the likelihood of the memory being used for something else are uncommon – a nice race condition).

Therefore memory must not be freed as long as another thread might access it. This is where garbage collection is very handy. If garbage collection is not available, there are sometimes specific solutions depending on the context. Another general solution are *hazard pointers*.

For this exercise, I strongly suggest to simply leak memory and not to free anything. Or – at your choice – use Java.

ABA problem:

The ABA problem is the inability to observe the occurrence of other method calls in certain situations. Consider for example a LIFO queue (aka stack) with three linked elements head → A → B → C. If a thread tries to pop the first element from that stack, it would usually issue a compare and swap (CAS) operation: “if head points to A, then update head letting it point to B” – CAS(&head, A, B). This nicely catches situations when another thread pops A in between as the CAS will fail. However, if another thread not only pops A but also pops B and pushes A again onto the stack, the CAS will succeed and the stack is in an inconsistent state.

One possible solution is to extend the CAS to pairs, so that in addition to the pointer a counter is checked and incremented with every successful CAS (AtomicStampedReference in Java). Then you would only get erroneous behavior, if a thread is delayed for a full warp around of the counter, which has to be prevented by other means. Another solution is to simply disallow reuse.

I suggest to take the latter approach: disallowing reuse. The example code does not reuse nodes, and – if you followed the earlier advice of not freeing memory – you cannot run into the situation that the same piece of memory is reused for a different node.

Half-finished method calls:

Despite atomic operations, a method might encounter effects of other, half-finished method calls. These have to be handled. It is your task to devise an order of operations that allows no race conditions. Use compare and swap and memory barriers if appropriate. Note, that compare and swap already implies a memory barrier and that loads and stores of variables not exceeding the native width are usually atomic.

Implement a lock-free FIFO queue, where a delay in one thread has no effects on other threads.

As an alternative, you may use explained pseudo code, paying attention especially to ordering constraints.