MPhil course in Multicore Programming (R204)

Exercise sheet (Tim Harris' section)
Please hand to student admin, with a coversheet.

1. In the slides, the pseudo-code for the MCS "acquireMCS" operation shows that the new QNode is only linked into the queue after performing a CAS operation. This makes the "releaseMCS" operation more complicated because a call to "releaseMCS" might need to wait until it sees that a lock holder has linked its QNode into the queue. Show why it would be *incorrect* to optimize the "acquireMCS" operation by initializing prev->next to point to the new QNode before performing the CAS (that is, moving the line at Label 2 to occur earlier at Label 1).

   [2 marks]

2. A "ticket lock" is implemented using two shared counters, T and C, both initially 0. A thread wanting to acquire the lock uses an atomic fetch-and-add on T to obtain a unique sequence number. The thread then waits until C is equal to this sequence number. After releasing the lock, a thread increments C.

   What are the advantages / disadvantages of this ticket lock compared with (i) a test-and-test-and-set lock, and (ii) compared with the MCS queue lock?

   [3 marks each]

3. Programmers often use the phrase "lock-free" informally to mean that an algorithm is fast and scalable, even if it does not provide the lock-free progress property. Describe a workload where the singly-linked list in the slides will *not* be fast and scalable, but a normal lock-based list could be better.

   [3 marks]

4. Consider a simple shared counter that supports an "Increment" operation. Each increment advances the counter's value by 1 and returns the counter's new value – 1, 2, 3, etc.

   a) Explain whether or not the following history is linearizable:

   - Time 0 : Thread 1 invokes Increment
   - Time 10 : Thread 1 receives response 1
   - Time 11 : Thread 1 invokes Increment
   - Time 20 : Thread 2 invokes Increment
   - Time 21 : Thread 1 receives response 3
   - Time 22 : Thread 1 invokes Increment
   - Time 30 : Thread 2 receives response 2
   - Time 31 : Thread 1 receives response 4

   [2 marks]

b) In pseudo-code, give a lock-free implementation of “Increment” using an atomic compare and swap operation.

[1 mark]

c) Explain whether or not your implementation is also wait-free.

[2 mark]

[Optional: if your counter is not wait-free, then can you see a way to build a wait-free one from compare and swap, or can you see how to write a proof-sketch that it is impossible to build one?]

5. The array-based deque in the slides supports one thread on the “top” and, and multiple threads stealing from the “bottom” end (slide 59).

   Consider instead the case of a simpler array-based queue supporting a fixed maximum number of elements (N) and only a single producer (calling “pushTop”) and a single consumer (calling “popBottom”). A push should return “true” if it succeeds (adding the item to the queue), and “false” otherwise (if the queue is full). A pop should return a data item if there is one in the queue, or NULL if the queue is empty.

   In pseudo-code, give a lock-free linearizible implementation of this queue building on atomic compare and swap, read, and write.

   [4 marks]

MPhil course in Multicore Programming (R204)

Practical exercise (Tim Harris' section)
Please hand to student admin, with a coversheet.

The aim is to investigate the practical performance of different reader-writer lock implementations on a real machine.

The written report that is submitted should include:

(*i*) Graph(s) showing the performance of the different implementations developed. Graphs should include results from an appropriate number of runs, and include error bars.

[12 marks in total, 2 each for Q2-7 below]

(*ii*) A summary of the machine being used – how many processors, cores, and hardware threads it has, which language and operating system were used. If possible, indicate how the software threads are allocated to the hardware threads in the machine (e.g., in a machine with 2-way hyperthreading, different results would be expected if 2 software threads are running on the hyperthreads in a single core, as opposed to running on different cores).

[2 marks]

(*iii*) A short description explaining the reasons for the performance that you see – 500 words is sufficient.

[6 marks]

The problems can be tackled in any suitable programming language on a multi-core machine or other parallel computer. However, please make sure that the machine has at least 4 cores, 4 processors, or 4 hardware threads (the CL's teaching lab includes suitable machines). C, C++, and Java are all possible languages to use. The course web page includes a link to example code to help you get started.

When timing experiments please use "wall-clock" time (measured from starting the program until when it finishes). Each experiment should take a few seconds to run, and so cycle-accurate timing is not needed: from a UNIX shell prompt you could use the "time" utility.

1. Check that the example code builds and runs correctly. In particular, try passing in a large value to the "delay" function and make sure that the compiler is not optimizing the loop away. (For this exercise it is best to use a timing loop like this, rather than a proper "sleep" function, to reduce interactions between the test program and the OS).

2. Extend the "main" function to take a command line parameter saying the number of threads to use (N). The harness should start N threads. The program should only exit once all the threads are done.

   To check that the harness works correctly, start off by having each thread call "delay" with a parameter for a delay of about 1s. Plot a graph showing the execution time as you vary N. Start with N=1 and raise N until it is twice the number of hardware threads on your machine.

Check that:

a) If N is <= the number of cores on your machine then the execution time should stay at about 1s (as with a single thread).

b) The execution time should rise above 1s as you raise N above the number of cores on the machine, and then rise substantially once N is above the number of hardware threads.

3. Implement a read-only test harness: Have the threads share a single array of X integers, and write a sum() function to calculate the sum of these integers. Each thread will loop, calling sum() repeatedly. Arrange that the program exits when thread 0 has performed a fixed number of these calls (other threads should keep executing these sum() operations until signalled to exit by thread 0). Try the experiments with X=5 and with X=5000.

   How fast is the original program on a single core if you do not use any locking?

   How fast is this program if you run it on multiple cores, but acquire a built-in mutex for each call to sum()? (e.g., in Java, you could make sum a synchronized method, and in C you could use a pthread mutex). Plot a graph showing the execution time as you vary N. As before, start with N=1 and raise N until it is twice the number of hardware threads on your machine. This version is overly pessimistic – all of the operations are being serialized by the lock, even though they are read-only.

4. Implement a test-and-test-and-set mutual exclusion lock, and repeat using that instead of the built in lock. Since this is just a mutual exclusion lock, all of the readers will still be serialized unnecessarily.

5. Implement a test-and-test-and-set reader-writer lock, based on the example on slide 65. This will allow multiple readers to acquire the lock at the same time, but it involves more synchronization than the basic mutual exclusion lock. Repeat the experiment with the different values of X and N and plot the results – is the reader-writer lock faster than the mutual exclusion lock?

6. Implement the flag-based reader-writer lock (slide 67). Repeat the experiment with the different values of X and N and plot the results – does the flag-based lock actually scale better than the test-and-test-and-set reader-writer lock?

7. Finally, try the version number scheme (slide 74), and repeat the experiments and plot the results as before.

   [Optional: This workload only includes read operations. Suppose that every 100 operations each thread performs a write to an entry in the shared array, and so it needs to acquire the lock in write mode. Does this small number of writes change the relative performance and scaling of the different locks? Do you see starvation of reads or writes under any of the different implementations?]