Multicore Programming (R204)

Exercise sheet (Tim Harris' section)
Please hand to student admin, with a coversheet
Deadline: 16 January 2018, 12 noon

1. Lock implementation

A machine has 4 processors, each with its own cache. A program runs with 1 thread per processor, and 1 mutual exclusion lock used by all of the threads. Each thread must acquire the lock, execute a short critical section, and then release the lock.

For each of the following locks (i) describe the cache line transfers that occur in a simple MESI cache protocol in the best-case execution of the lock, and (ii) describe any additional cache line transfers that may occur if execution does not follow the best case.

- Test-and-set lock.
- Test-and-test-and-set lock.
- MCS lock.

You may assume that, by chance, the thread on processor 1 successfully acquires the lock first, then processors 2, 3, and 4. Also, you may assume that each QNode structure for the MCS lock is a single cache line in size.

[5 marks]

2. Linearizability

Consider the following history of operations on a set implemented over a linked list. The set is initially empty. A call to insert(X) returns true if it succeeds in adding X to the set. A delete_ge(X) operation deletes the next value above or equal to X. It returns the value deleted, or -1 if there is no such value.

```
Thread 1 : Calls delete_ge(10)
                                Thread 2 : Calls insert(30)
                                Thread 2 : insert(30) returns true
                                                                Thread 3 : Calls insert(20)
                                                                Thread 3 : insert(20) returns true
                                                                                                Thread 4 : Calls insert(30)
                                                                                                Thread 4 : insert(30) returns false
Thread 1 : delete_ge(10) returns 30
```

Show that this concurrent history *is not* linearizable. Then, for each of the following alternatives, show that the resulting history *would be* linearizable:

(i) If the delete_ge(10) operation had returned -1 to thread 1.
(ii) If the delete_ge(10) operation could delete any value greater than or equal to 10 in the set.
(iii) If the operations in thread 4 executed before those in thread 3.

[5 marks]

3. Lock-free lists and memory management

Consider a lock-free linked list of integers, held in sorted order and shared between a large number of threads. Threads perform search, insert, and delete operations on the list.

Initially, assume that a garbage collector is used to reclaim storage automatically. Describe workloads (i) where the lock-free list is likely to perform better than a list protected by a well-implemented mutual exclusion lock, and (ii) where the lock-free list is lilkely to perform less well than the lock-based list.

Suppose that a per-list-node reference counting scheme is used instead of garbage collection in the lock-free list. Are there now any cases where the lock-free list would be preferable? (Note that reference counting *would not* be needed in the lock-based list.)

[5 marks]

4. Transactional memory

Transactional memory implementations are often classified as making eager or lazy updates and performing eager or lazy conflict detection.

Describe two workloads, one of which would perform well under eager-eager, and one which would perform well under lazy-lazy. Justify your answer in terms of (i) the amount of work executing the transactions initially, (ii) the amount of additional work attempting to commit the transactions, and (iii) the amount of additional work caused by transactional re-execution.

[5 marks]

Multicore Programming (R204)

Exercise sheet (Tim Harris' section)
Please hand to student admin, with a coversheet
Deadline: 16 January 2018, 12 noon

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 47. 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 49). 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 56), 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?]