Multicore Semantics & Programming

Practical report (Tim Harris' section)

The report forms 25% of the course total, with marks for individual questions identified below

Core exercises (Part II and ACS/Part III)

The course included a sketched graph comparing the performance of a basic spin lock and improvements such as a test-and-test-and-set lock. The aim of this exercise is to investigate the practical performance of these alternatives and assess whether the sketch in the slides fits with the behaviour of current hardware. The exact results that you see will be dependent on the machine being used.

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.
>
> [8 marks in total, 2 each for Q2-5 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.
>
> [10 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 48. 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?

Additional exercises (ACS/Part III only)

Reader-writer locks often perform poorly in practice for short critical sections – even when the lock is only acquired in read mode. This extension involves extending the test harness to include more advanced reader-write locks, and also exploring a workload with a mix of reads and writes.

6. Implement the flag-based reader-writer lock (slide 50). Repeat the experiment with a read-only workload using different values of X (the work done in the critical section) and N (the number of threads). Using results from the experiment to justify your answer, what guidance would you provide for when the flag-based lock is worthwhile from a performance viewpoint?

   [5 marks]

7. Now consider the case where the lock is acquired in write-mode every K operations. If K=1000000 then does this small number of writes change the relative performance and scaling of the locks from Q2—Q6? Do you see starvation of reads or writes under any of the different implementations? What if K=100?

   [5 marks]