CSMC 23010-1 Due February 18, 2014

Homework 3

Professor: Hank Hoffmann *TA: Harper Zhang*

Theory

There is just a single theory problem for this assignment. Chapter 7, problem 85. Please use the time not spent on theory to begin testing and coding early. It is strongly suggested that you begin lock implementation and testing before the design review to give yourself enough time to complete the performance testing.

Programming

This assignment is all about locks and data contention. We will build a set of different lock types and use them to balance the load of our packet distribution system. In particular, we will be building the following set of lock types, L:

- **Test and Set Lock** (TASLock): This lock (Figure 7.2 in the text) makes use of the gcc atomic builtins, which encapsulate the read-modify-write atomic instructions on Intel hardware.
- **Exponential Backoff Lock** (BackoffLock): This lock (Figure 7.6 in the text) builds on a TTASLock by inserting a random pause after an unsuccessful attempt at grabbing the lock. You can make use of the nanosleep or usleep functions to implement this lock.
- **Mutex** (pthread_mutex_t): This is the standard lock in the POSIX threads library. We will compare our locks against the pthread mutex, but you should not use the mutex to implement any of the other locks.
- **Anderson's Array Lock** (ALock): This lock (Figure 7.7 in the text) has a fixed array of locations (equal to the total number of workers who might simultaneously try to grab the lock) furnishing each thread with a private location on which to spin. Take care to implement it with padding, as described in Figure 7.8.
- **CLH Queue Lock** (CLHLock): This lock (Figures 7.9 and 7.10 in the text) creates a linked list of nodes, so that each thread watches a distinct memory location. This is intended to decrease memory contention, at the expense of more work (*e.g.*, more overhead in managing the list etc.).
- **EXTRA CREDIT: MCS Queue Lock** (MCSLock): This lock (Figures 7.12 and 7.13 in the text) also creates a linked list (using a different algorithm) in order to reduce memory contention on NUMA architectures.

We will also be extending the locks to include a tryLock method. Essentially, tryLock returns true if the lock is acquired (the caller is then responsible to call unlock eventually) and false if the lock is held by another thread at the time of calling. Notice that there is a race here - you could test whether the lock is held, decide that it is not and then try to lock it, creating an opportunity for another thread to lock it in between the two steps. This behavior is acceptable - once the tryLock decides to grab the lock, it is permitted to wait in the case of this race condition. The purpose is to merely reduce the instances of threads waiting for locks that are already held.

In order to gauge the relative merits of this set of lock algorithms, we will conduct several experiments:

1. **Time-based Counter Test:** We will isolate the performance of each lock type in the simplified case where a single lock protects a counter that all threads increment. In addition to incrementing

the shared counter, the worker threads will increment a private counter which measures the distribution of how much work each worker does, a coarse measure of fairness. We will develop two different versions of this code:

- SerialCounter This application launches a single thread which increments the counter with wild abandon, free to do so because it is running solo. This code is configurable by:
  - numMilliseconds (M) the time in milliseconds that the experiment should run.
- ParallelCounter This application launches n threads, which all try to grab the lock and increment the counter leading to complete mayhem. Our goal is to measure the severity of the mayhem caused by the combination of extra overhead in the lock code *and* contention on the shared lock variables. This code is configurable by:
  - numMilliseconds (M) the time in milliseconds that the experiment should run.
  - numThreads (n) the total number of worker threads
  - lockType (L) the lock algorithm

2. **Work-based Counter Test:** Again, we will have all the threads increment a single counter protected by a lock. In this case, however, we will count to a fixed number (BIG) and each thread will be responsible for BIG/n increments. You will measure the time taken to perform all of the increments.

   - SerialCounter This is analogous to the above, a single thread that increments the counter. Configurable by:
     - B (B) the number to which we are counting.
   - ParallelCounter Again, this application launches n threads, which all try to grab the lock and increment the counter. This code is configurable by:
     - B (B) the number to which we are counting.
     - numThreads (n) the total number of worker threads
     - lockType (L) the lock algorithm

3. **Packet Processing:** We will extend the packet distribution system that we built in the last assignment to improve the load balancing characteristics. In particular, we will protect the dequeue interface to the Lamport queue that we designed in the previous assignment with a lock. The Dispatcher will be the only thread enqueueing data, so a lock is not necessary to protect the enqueue interface (*ie.* locking the dequeue side gives the illusion of single-reader / single-writer semantics).

   - SerialPacket This application features a single worker which calculates the checksums for each source serially. This version is configurable by:
     - numMilliseconds (M) the time in milliseconds that the experiment should run.
     - numSources (n) the total number of sources (i.e. worker threads)
     - mean (W) the expected amount of work per packet
     - uniformFlag = true for Uniform distributed packets, false for Exponentially distributed packets
     - experimentNumber non-negative integer, seeding the packet generator
   - ParallelPacket This applications differs from Parallel in the previous assignment in that the Worker threads grab the lock that is associated with each queue before calling dequeue. The Worker threads will also be designed to use any of the following strategies, S, for picking a queue:

- LockFree This is the same behavior as Parallel from the previous assignment (*i.e.* 1-to-1 mapping between queue and Worker).
- HomeQueue This is the same fixed mapping as in LockFree, but with the added requirement that the worker grabs the (albeit uncontended) lock for the associated queue.
- RandomQueue The Worker picks a random queue to work on for each dequeue attempt.
- LastQueue The worker uses a tryLock on a series of random queues until it finds one that is unlocked, then it proceeds to dequeue from that queue using the lock (*i.e.* not tryLock) method until it gets an empty return code from the dequeue method.
- Awesome Your design. You should first design, implement, test, and profile the four other strategies. When that portion of work is complete, use your judgment to design a new strategy that will be faster. You can design a strategy for a specific case (e.g., specific for uniform or exponential) or for a general case. You are responsible for describing your design, testing it, and implementing it. You are also responsible for forming a reasonable hypothesis about why your design is better. You are free to use any technique you wish in the workers. Do not change the dispatcher. Do not drop packets. More details follow.

This code is configurable by the same parameters as SerialPacket, plus:

- queueDepth (D) queue depth (always equal to 8 for this assignment).
- lockType (L) the lock algorithm
- strategy (S) the strategy for picking a queue to work on

Experiment

You should create a directory and expand the accompanying tar file into it. It will provide some basic components (*e.g.* utilities for timing code, a random packet generator, a payload checksum calculator etc.).

Next, you will perform the following set of experiments across various cross-products of these parameters. Each data point is a measurement taken on a dynamic system (a computer...) and is thus subject to noise. As a result, some care should be taken to extract representative data - we would propose running some reasonable number of experiments for each data point and selecting the median value as the representative. For Uniform packets and the Counter tests, something like 5 data points should suffice whereas for Exponentially Distributed packets 11 may be required to get relatively smooth plots - please use your own engineering judgment to decide how many trials you require. You should set the *seed* parameter for PacketSource to the trial number (or some deterministic function thereof) to ensure that you're seeing a reasonable variation in load from each source. Setting the experiment time M to 2000 (*i.e.* 2 seconds) should suffice to warm up the caches for all of the following experiments. In each of the following experiments, we describe a plot that you should produce, analyze and discuss in the writeup.

A note on describing performance: Use quantitative language when describing your observations. For example, strategy x provides a 10% improvement compared to strategy y. Avoid qualitative language of the type, "Strategy x is a little better than strategy y."

A note on measuring performance: All code should be compiled with optimizations on (`-O3`, for example). You should describe what optimizations you are using.

Counter Tests

1. **Idle Lock Overhead 1** Run SerialCounter for the Time-based counter experiment to find the throughput (increments / ms) of a single processor incrementing a counter (a volatile, which admittedly has some overhead). Then, run ParallelCounter with $n = 1$ and all L ($\in$ {TASLock, BackoffLock, Mutex, ALock, CLHLock, MCSLock}). Provide the speedup

(*i.e.* ratio of ParallelCounter throughput to SerialCounter) for each L in a table. Do these results make sense, given the relative complexity of the uncontended path through each lock method? You are responsible for justifying the answer to this question and persuading graders that you understand your implementations. (For example, what operations in your code lead to larger overhead?)

2. **Idle Lock Overhead 2** Run SerialCounter for the Work-based experiment to find the throughput (increments / ms) of a single processor incrementing a counter (a volatile, which admittedly has some overhead). Then, run ParallelCounter with $n = 1$ and all L ($\in \{\text{TASLock}, \text{BackoffLock}, \text{Mutex}, \text{ALock}, \text{CLHLock}, \text{MCSLock}\}$). Provide the speedup (*i.e.* ratio of ParallelCounter throughput to SerialCounter) for each L in a table. Do these results make sense, given the relative complexity of the uncontended path through each lock method? How do the results for the two different (time vs work based) approaches compare? (Depending on your implementations there may not be differences). You are responsible for justifying the answer to these questions. Your answers should persuade graders that you understand the internals of your code. If you cannot explain a result with data (taken from experiments or from code analysis) then you should ask for help.

3. **Lock Scaling 1** Optimize the DELAY times for ParallelCounter with L = BackoffLock for $n = 8$. Run ParallelCounter for all L and $n \in \{1, 2, 4, 8, 16, 32, 64\}$ with the tuned version of BackoffLock. Using the time-based implementation, plot speedup (ratio of ParallelCounter throughput to SerialCounter - it *should* be less than 1) for each L (*i.e.* one curve for each L) on the Y-axis vs. n on the X-axis. Describe on your results - what is your hypothesis about the performance, particularly of the more complex locking algorithms? Back up this answer with data that demonstrates your understanding.

4. **Lock Scaling 2** Optimize the DELAY times for ParallelCounter with L = BackoffLock for $n = 8$. Run the work-based approach to ParallelCounter for all L and $n \in \{1, 2, 4, 8, 16, 32, 64\}$ with the tuned version of BackoffLock. (Is the tuning the same? Why or why not?) Plot speedup (ratio of ParallelCounter throughput to SerialCounter - it *should* be less than 1) for each L (*i.e.* one curve for each L) on the Y-axis vs. n on the X-axis. Describe your results - what is your hypothesis about the performance, particularly of the more complex locking algorithms? Does the work-based test lead you to a different conclusion than the time-based test? Demonstrate your understanding and think about what we described in class and in the book as justifications for these various lock algorithms.

5. **Fairness** Find the standard deviation of the count for each worker in the time-based **Lock Scaling** experiment for $n = C$, where C is the number of cores on your test machine. Produce a scatter plot of throughput (Y-axis) vs. standard deviation of counts among workers (X-axis) with each lock, L, represented. Does this corroborate your hypothesis from the time-based **Lock Scaling** experiment? Again, your answer should demonstrate an understanding of the code you have submitted and how that code is translated into performance.

Packet Tests

1. **Idle Lock Overhead** Run ParallelPacket with $n = 1$, $S \in \{\text{LockFree}, \text{HomeQueue}\}$, all L and $W \in \{25, 50, 100, 200, 400, 800\}$ on the uniformly distributed packets. Plot the speedup of ParallelPacket (with S = HomeQueue) throughput relative to ParallelPacket (with S = LockFree) for each L on the Y-axis vs. W on the X-axis. Is this consistent with the insights you drew in the Counter Tests? How does the **Worker Rate** compare with your measurements in the last assignment at each W; i.e., what is the overhead of the locks in terms of Worker Rate?

2. **Speedup with Uniform Load** Run ParallelPacket and SerialPacket on the uniformly distributed packets with the number of workers $n \in \{1, 2, 3, 7, 15\}$, $W \in \{1000, 2000, 4000, 8000\}$, $S \in \{\text{LockFree}, \text{RandomQueue}, \text{LastQueue}\}$ and $L \in \{\text{TASLock}, \text{BackoffLock}, \text{Mutex}, \text{ALock}, \text{CLHLock}, \text{MCSLock}\}$. For each W (*i.e.* four graphs), make a plot of speedup (relative to SerialPacket with the same parameters) for each $\langle S, L \rangle$ combination on the Y-axis vs. $n + 1$ (which is the total number of threads) on the X-axis. How does the scalability of *ParallelPacket* change given the use of locks and load balancing? *Note:* $S = \text{LockFree}$ *does not need to be evaluated against all values of L - it doesn't use locks!* This answer should, again, demonstrate understanding. Particularly, you should form a hypothesis about expected behavior before running the experiments. If your hypothesis does not hold up, you may need to reevaluate your hypothesis, rework your code, or change your experiment. In particular, at small values of M results may be distorted by randomness. You may consider increasing both M and the number of trials over which you average. Also note that what may appear to be small differences may be meaningful – stick to quantitative language.

3. **Speedup with Exponential Load** Repeat the previous experiment, except with the exponentially distributed packets. How does the scaling change? Why? Again, you should be able to argue that your results make sense.

4. **Speedup with Awesome** Demonstrate the speedup of your design under either uniform or exponential load or for whatever scenario you have designed it to outperform the given strategies. You should describe why your design is better and demonstrate its superior qualities with an experiment. Note that you have a lot of freedom to design here.

Submission

As with previous assignments, a design and test document should be submitted one week prior to the due date for the full writeup. Note that the major area for design in this project is the Awesome scheme that you are coming up with. It will not be possible to write a comprehensive design for this scheme as it should be based on what you learn from other experiments. The best thing to discuss for this scheme is how you will evaluate opportunities for improving on the other approaches. You should discuss correctness and performance testing for all modules in this assignment.

You are responsible for submitting a writeup and all code associated with this project.

You write up should be concise, but it should primarily demonstrate your understanding of the code and experiments and your engagement with the problems. The writeup should include quantitative language that clearly supports your statements. The writeup should stand alone as a document that explains the code you developed, the experiments you ran and the conclusions you drew. The writeup should have a section for each of the experiments described above and an additional section describing your test plan. You should have a test plan for each lock and for each of the Random Queue, Last Queue, and Awesome strategies.

The grades will be primarily based on the understanding demonstrated in the writeup, so please spend time on them. This means you need to start early and get help if there is a result you don't understand. For partial credit, higher grades will be assigned to reports that clearly explain results for some smaller number of experiments than those that have a poor explanation for a large number of experiments. (I sincerely hope that everyone completes and explains all experiments, of course).

You should submit your working directory and all the associated code including test code. It should be easy for old people to run your tests and attempt to recreate your results (differences in machines might make this difficult, but in principle it should be doable).

You are responsible for understanding what is required of you on this assignment. If you are not clear about what to do, then ask clarifying questions.