CSC 2/458: Parallel and Distributed Systems

Scalable Synchronization Algorithms

Due Date: Midnight, Thursday, February 19th, 2015.

The purpose of this assignment is to analyze the tradeoffs among different synchronization algorithms in terms of their latency, fairness, scalability, traffic, and storage requirements. The performance levels of the algorithms used varies widely due to interaction between multiple factors, including the length of the critical section and the latency of communication. Your mission is to reason about the performance of these locks in terms of: latency, scaling behavior, effects of preemption (i.e., saturated CPU resources), and fairness.

Create a program that launches *t* threads and have them all work on a shared counter. Try experiments where each thread increments the counter a total of *i* times, and where each thread increments the counter *i* times (for a total of *t***i*).

- with no synchronization
- with pthread locks
- with TAS locks
- with TATAS locks
- with Backoff included in TATAS
- with ticket locks
- with MCS locks
- with FAI

Test the scalability of several global barrier implementations. Create a program that launches *t* threads and have them arrive *i* times at a barrier implemented using

- pthreads-based condition synchronization (as provided in the example `sor` application)
- a centralized sense-reversing barrier
- a tree-based barrier of your choice

Try each option with varying numbers of threads, both greater and fewer than the number of processors in the machine. (To find out how many processors there are, run `psrinfo -v`) on the sparcs, or examine `/proc/cpuinfo` on x86/Linux machines. Report final counter values, execution times, and the number of times each thread incremented the counter in the case of the first experiment. Try any other tests that occur to you. Explain your results (in writing).

To simplify testing of your code, please write your program to take the number of threads *t* and the number of iterations *i* as command-line arguments, specified with "`-t` *t*" and "`-i` *i*" (in either order). If the arguments are not specified, use *t* = 4 and *i* = 10,000.

Be sure to include a README.pdf file that explains what you did and what you learned. It should include your timing results and analysis of the various locks. We will be grading the assignment on a roughly equal mixture of completeness and correctness; programming style; and quality of write-up.

Notes

- For more information on pthreads, you may want to consult the `pthreads` man page and the tutorial from Lawrence Livermore National Lab. On the man page, you will see that Sun provides somewhat more extensive facilities for its `threads` package, but these are not portable across platforms.
- Pseudocode for locking algorithms can be found at www.cs.rochester.edu/research/synchronization/pseudocode/ss.html.

- The Gnu C compiler provides a very flexible mechanism to insert assembly language instructions (e.g. the various atomic primitives) into your code. Unfortunately it's a rather confusing mechanism, so we've written the magic incantations for you: `atomic_ops.h`. This file also includes implementations of all the locks. Since the primitives are written in barebone assembly they are machine specific. We have used #defines were appropriate. The "__i386" defines are to be used on 32 bit x86s (cycle1,e series, a series, "__x86_64__" on the 64bit x86s and "__sparc__" on the SUN boxes.

- Pthreads are not part of the C standard library. To use them you must link in a separate library explicity. Add `-lpthread` to the end of your g++ command line. We strongly recommend that you create a Makefile for your assignment, even if you have only a few files. Take a look at the Makefiles from the previous assignement. Start with the machine specific files where appropriate. Unlike the previous assignment, the .h files here require the g++ compiler.

- As with the last assignment, use the fine-grain `gethrtime` timer provided at `/u/cs(2or4)58/hrTimer/` Run your tests multiple times. See which results seem repeatable, and which vary greatly from one run to the next. (And try to explain why.) To gain the absolute maximum performance you may have to resort to techniques such as binding a thread to a processor.

- In order to see race conditions, you will need to ensure that your threads to run at roughly the same time. (If you don't do anything special, it's possible for a newly created worker thread to finish all its counter increments before the master thread manages to create the next worker.) You can use a *barrier* to accomplish this (we've provided code for a centralized sense-reversing barrier). You should also use it to make sure all your threads are done before you check results and timing. (You don't want to use `thr_join()` for this; like create_thread it's so expensive it can hide what you're looking for.) If you want to perform multiple timing tests in a single program execution, you can safely call the barrier multiple times. Your code should look something like this:

```
barrier()                       // threads are all together
if (tid == 0) {
    counter = 0
    start = gethrtime()         // thread 0 checks the time
}
barrier()                       // other threads wait for 0 to catch up
for (i = 0; i < iters; i++) {   // the test itself
    counter++
}
barrier()                       // make sure all threads are done
if (tid == 0) {
    end = gethrtime()
    print counter, end-start
}
```

```
// more tests
```

- To get accurate timings you'll need to run when no one else is running. We suggest that you do code development on a uniprocessor. When you're happy with your code, run timing experiments on node2x6.csug.rochester.edu (on the undergraduate side) or cycle3.cs.rochester.edu (on the grad side) while using `top` to ensure you have the machine to yourself (and coordinate amongst yourselves if there is contention).

- Machines: See the Machines section of the build instructions for programming assignments.

- In order to avoid incompatibilities, you should compile your programs on the same machine you plan to collect statistics on since we are using machine-specific assembly. The provided files have been tested on the i386 and Sun machines, so we suggest you restrict your experimentation to these machines. You are more than welcome to add support for and try other machine types as well. To determine the version of gcc you are using type: `gcc -E -\ dM -x c /dev/null`. This will list a bunch of #defines in gcc. Search for x86_64 or i386 etc.
- Please be careful not to clog these machines with run-away processes. In particular, run `ps -Af` before you log out and make sure you kill any run-away processes.