Lab 2

Concurrent Programming

Due: 11:59pm, October 16, 2019

This lab should familiarize you with building concurrency primitives and the C/C++ memory model.

Summary: In this lab you'll write three locking algorithms (four if you're in a grad section) and one barrier algorithm. You'll both use these primitives in your BucketSort algorithm from Lab 1 and also in a stand-alone "counter" micro-benchmark. Your lab write-up will use `perf` to investigate the performance of your code.
The algorithms you'll write are:

- Test-and-set lock
- Test-and-test-and-set lock
- Ticket lock
- MCS Lock (grads only)
- Sense-reversal barrier

Code Requirements / Restrictions:
Your primitives should be written using C/C++ atomics and should not rely on external concurrency primitives (e.g. `pthread_mutex_t`). You should apply the code performance lessons you've learned in class — avoid false sharing and contention when possible.
Microbenchmark: You will write a small stand-alone program, called `counter`. The program increments a counter, using your new primitives. Either the counter is protected by a lock, or the threads use a barrier to rotate turns incrementing the counter (see Figure 1), and at program end it will print the time the test took. Your counter program will have options to use all locks/barriers you wrote, plus the pthread ones. Take note of the performance differences.

```
int cntr=0;
void thread_main(int my_tid){
  for(int i = 0; i<NUM_ITERATIONS; i++){
    if(i%NUM_THREADS==my_tid){
      cntr++;
    }
    bar.wait();
  }
}
```

Figure 1: Using a barrier to synchronize counter access

BucketSort: You should also use your primitives within your BucketSort code, replacing all pthreads primitives. Take note of how the performance changes.

Lab write-up: Your lab write-up will be longer this time. In addition to the normal requirements, you should describe how changing the lock and barrier types in the two programs changes performance. In particular, your write-up should include:

- A table of all different locks (for both programs), which includes run time, L1 cache hit rate, branch-prediction hit rate, and page-fault count

- The same table for all barriers
- A discussion explaining why the best and worst primitives have these results.

Your write-up should also include the normal requirements:

- A description of your algorithms
- A brief description of your code organization
- A description of every file submitted
- Compilation instructions
- Execution instructions
- Any extant bugs

I expect your lab write-up for this project will be longer, around 4 pages.

Code style: Your code should be readable and commented so that the grader can understand what's going on.

Submission: You will submit a zip file of your lab to canvas. When unpacked, the directory should contain all files required to build and run your program, along with a brief write-up. Pay particular attention to the requirements for compilation and execution, as some testing will be done using automatic scripts.

Compilation and Execution:
Your submitted zip file should contain a Makefile and the project should build using a single `make` command. Your makefile will generate two executables.
Counter: The counter program has the following syntax:

```
counter [--name] [-t NUM_THREADS] [-i=NUM_ITERATIONS]
[--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,mcs,pthread>] [-o out.txt]
```

The program launches `NUM_THREADS` and each increments the counter `NUM_ITERATIONS` times. The counter is synchronized using either the `bar` or `lock` argument (setting both is an invalid input). The time taken to increment the counter to its total (excluding the time for file I/O and to initially launch / finally join threads) should be printed to standard out in nanoseconds, and the final counter value should be written to the output file designated by the `-o` flag. See Figure 2 for `counter` syntax examples.
BucketSort: As before, the generated sorting executable should be called `mysort`. The `mysort` command should have the following syntax (it is slightly different from Lab 1 as it adds two arguments):

```
mysort [--name] [source.txt] [-o out.txt] [-t NUM_THREADS] [--alg=<fj,bucket>]
[--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,mcs,pthread>]
```

The `mysort` requirements have not changed, save for the new `bar` and `lock` arguments that chose which barrier and lock to use respectively. Using the `--name` option should print your name. Otherwise, the program should sort the source file. The `mysort` command should then sort all integers in the source file and print them sorted one integer per line to an output file (specified by the `-o` option). The time taken to sort the file (excluding the time for file I/O and to initially launch / finally join threads) should be printed to standard out in nanoseconds. The `-t` option specifies the number of threads that should be used, including the master thread, to accomplish the sorting. The `getopt` and `getopt_long` method calls are helpful for parsing the command line.

Testing: Testing can be done using the same methodology as Lab 0 (e.g. `shuf`, `sort`, and `cmp`).
It is also recommended that you insert `assert` statements into your code to check for data races and invariant violations (particularly to check that the counter is correctly incremented).
Test machines will be available for you to execute code on. These machines have multiple hyperthreaded cores in a single socket. In order to connect to these machines, you'll need to be logged into the University network (UCB Wireless or a vpn). Your user name will be your last name, all lower case. Your password will be the four digit time you submitted Lab 0.

Grading: Your assignment will be graded as follows:

Sort unit tests (40%) We will check your code using ten randomly generated input files and using combinations of locks and barriers. Correctly sorting a file is worth four points.

Counter tests (10%) We will check your code for different locks and barriers, verifying that counter is correctly incremented.

Code readability (10%) Readable code that meet the requirements will get full marks. Unreadable code will be docked points.

Lab write-up (40%) Lab write-ups that meet the requirements and demonstrate understanding of the code's performance will be given full points. Incomplete experimentation and analysis will be docked points.

Recall that late submissions will be penalized 10% per day late, and will only be accepted for three days after the due date. Canvas submissions include the submission time.

```
### print your name
./counter --name
# prints:
Your Full Name

### Increment the counter
./counter -o out.txt -t5 -i=100 --lock=tas
# prints time taken in seconds for 5 threads on tas lock,
# to increment 100 times each and outputs final counter value to out.txt
294759

cat out.txt
# prints:
500
```

Figure 2: Examples of your `counter` program's syntax