CS111 - Project 2A: Atomic Operations

INTRODUCTION:

In this project, you will engage (at a low level) with a range of synchronization problems. Part A of the project (this part!) deals with conflicting read-modify-write operations on a single variable, and can be broken up into three major steps:

- Write a multi-threaded application using pthread that performs parallel updates to a shared variable.
- Demonstrate the race condition in the provided “add” routine, and address it with different synchronization mechanisms.
- Do performance measurement and instrumentation.

RELATION TO READING AND LECTURES:

The basic shared counter problem was introduced in section 28.1.
Mutexes, test-and-set, spin-locks, and compare-and-swap are described (chapter 28).

PROJECT OBJECTIVES:

- primary: demonstrate the ability to recognize critical sections and address them with a variety of different mechanisms.
- primary: demonstrate the existence of race conditions, and efficacy of the subsequent solutions
- secondary: demonstrate the ability to deliver code to meet CLI and API specifications.
- secondary: experience with basic performance measurement and instrumentation
- secondary: experience with application development, exploiting new library functions, creating command line options to control non-trivial behavior.

DELIVERABLES:

A single tarball (.tar.gz) containing:

- a single C source module (*lab2a.c*) that compiles cleanly (with no errors or warnings).
- a *Makefile* to build the program and the tarball.
- graphs (.png) of:
    - the average time per operation vs the number of iterations
    - the average time per operation vs the number of threads for all four versions of the add function.
- a README file containing:
    - descriptions of each of the included files and any other information about your submission that you would like to bring to our attention (e.g. limitation, features, testing methodology, use of slip days).

- brief (a few sentences per question) answers to each of the questions under 2A.1, 2A.2, and 2A.3 (below).

PROJECT DESCRIPTION:

To perform this assignment, you will need to learn a few things:

- pthread (https://computing.llnl.gov/tutorials/pthreads/)
- clock_gettime(2) … high resolution timers
- GCC atomic builtins (http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html)
- gnuplot(1) … you may find it useful to learn to use this versatile package for producing your graphs. However, you can also use other tools such as matlab, R, … to produce graphs.

Start with a basic add routine:

```
void add(long long *pointer, long long value) {
      long long sum = *pointer + value;
      *pointer = sum;
}
```

Write a test driver program called **lab2a** that:

- takes a parameter for the number of parallel threads (**--threads=#**, default 1)
- takes a parameter for the number of iterations (**--iterations=#**, default 1)
- initializes a (long long) counter to zero.
- notes the (high resolution) starting time for the run (using clock_gettime(2))
- starts the specified number of threads, each of which will use the above add function to
  - add 1 to the counter the specified number of times
  - add -1 to the counter the specified number of times
  - exit to re-join the parent thread
- wait for all threads to complete, and notes the (high resolution) ending time for the run.
- checks to see if the counter value is zero, and if not log an error message to stderr
- prints to stdout
  - the total number of operations performed
  - the total run time (in nanoseconds), and the average time per operation (in nanoseconds).
- If there were no errors, exit with a status of zero, else a non-zero status

  suggested sample output:

```
% ./lab2a --iterations=10000 --threads=10
10 threads x 10000 iterations x (add + subtract) = 200000
operations
ERROR: final count = 374
```

```
elapsed time: 6574000ns
per operation: 32ns
```

Run your program for a range of *threads* and *iterations* values, and note how many threads and iterations it takes to (fairly consistently) result in a failure (non-zero sum).

QUESTION 2A.1A:

Why does it take this many threads or iterations to result in failure?

QUESTION 2A.1B:

Why does a significantly smaller number of iterations so seldom fail?

Extend the basic add routine to be more easily cause the failures:

```
int opt_yield;

void add(long long *pointer, long long value) {
      long long sum = *pointer + value;
      if (opt_yield)
            pthread_yield();
      *pointer = sum;
}
```

And add a new **--yield** option to your driver program that sets opt_yield to 1. Re-run your tests and see how many iterations and threads it takes to (fairly consistently) result in a failure. It should now be much easier to cause the failures.

Compare the average execution time of the yield and non-yield versions with different numbers of threads and iterations. You should note that the --yield runs are much slower than the non-yield runs, even with a single thread.

Graph the average cost per operation (non-yield) as a function of the number of iterations, with a single thread. You should note that the average cost per operation goes down as the number of iterations goes up.

QUESTION 2A.2A:

Why does the average cost per operation drop with increasing iterations?

QUESTION 2A.2B:

How do we know what the "correct" cost is?

QUESTION 2A.2C:

Why are the --yield runs so much slower? Where is the extra time going?

QUESTION 2A.2D:

Can we get valid timings if we are using --yield? How, or why not?

Implement three new versions of the add function:

- one protected by a pthread_mutex, enabled by a new **--sync=m** option.
- one protected by a spin-lock, enabled by a new **--sync=s** option. You will have to implement your own spin-lock operation. We suggest that you do this using the GCC atomic __sync_ builtin functions **__sync_lock_test_and_set** and **__sync_lock_release**.
- one that performs the add using the GCC atomic __sync_ builtin function **__sync_val_compare_and_swap** to ensure atomic updates to the shared counter, enabled by a new **--sync=c** option

Use your yield option to confirm that (even for large numbers of threads and iterations) all three of these serialization mechanisms eliminate the race conditions in the add critical section.

Using a large enough number of iterations to overcome the issues raised in the previous section, note the average time per operation for a range of *threads* values, and graph the performance (time per operation) of all four versions (unprotected, mutex, spin-lock, compare-and-swap) vs the number of threads.

QUESTION 2A.3A:

Why do all of the options perform similarly for low numbers of threads?

QUESTION 2A.3B:

Why do the three protected operations slow down as the number of threads rises?

QUESTION 2A.3C:

Why are spin-locks so expensive for large numbers of threads?

SUBMISSION:

Project 2A is due before midnight on Monday, April 25, 2016.

Your tarball should have a name of the form lab2a-*studentID*.tar.gz and should be submitted via CCLE.

We will test it on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

RUBRIC:

Value	Feature
	Packaging and build (10%)
3%	untars expected contents
3%	clean build w/default action (no warnings)
2%	Makefile has working clean and dist targets
2%	reasonableness of README contents
	Code review (20%)
4%	overall readability and reasonableness
4%	yields correct and in appropriate places
4%	mutex correctly used
4%	spin lock correctly implemented and used
4%	compare-and-swap correctly implemented and used to implement atomic add
	Results (50%) (reasonable run)
3%	threads and iterations
4%	correct yield
5%	correct mutex
10%	correct spin
10%	correct cas
8%	reasonable time reporting
10%	graphs (showed what we asked for)
	Analysis (20%) … (reasonably explained all results in README)
2%	general clarity of understanding and completeness of answers
2%	(Q2A.1A) explain why it takes many threads or iterations
2%	(Q2A.1B) explain why small number of iterations seldom fails
2%	(Q2A.2A) explain cost/op vs iterations
2%	(Q2A.2B) how to know the correct cost
2%	(Q2A.2C) explain why yield runs much slower
2%	(Q2A.2D) can we get valid timings with yield
2%	(Q2A.3A) similarity of low thread-count results
2%	(Q2A.3B) explain why three protected operations slow down as number of threads rises
2%	(Q2A.3C) explain why spins are so expensive for large number of threads