CS111 - Project 2B: Complex Critical Sections

INTRODUCTION:

In this project, you will engage (at a low level) with a range of synchronization problems. Part B of the project (this part!) deals with conflicting search and update operations in an ordered doubly linked list, and can be broken up into four major steps:

- Implement the four routines described in SortedList.h: SortedList_insert, SortedList_delete, SortedList_lookup, and SortedList_length.
- Write a multi-threaded application using pthread that performs parallel updates to a sorted doubly linked list data structure (using methods from the above step).
- Recognize the race condition when performing linked list operations, 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 the problems 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:

- SortedList.h - a header file containing interfaces for linked list operations.
- the source for a C source module (*SortedList.c*) that compiles cleanly (with no errors or warnings), and implements insert, delete, lookup, and length methods for a sorted doubly linked list (described in the provided header file, including correct placement of pthread_yield calls).
- the source for a C program (*lab2b.c*) that compiles cleanly (with no errors or warnings), and implements the specified command line options (--threads, --iterations, --yield,

--sync), drives one or more parallel threads that do operations on a shared linked list, and reports on the final list and performance.
- a *Makefile* to build the program and the tarball.
- graphs (.png) of:
    - average time per unprotected operation vs number of iteration (single thread)
    - (corrected) average time per operation (for unprotected, mutex, and spin-lock) vs number of threads.
- 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 2.1 and 2.2 (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.

Review the interface specifications for a sorted doubly linked list package described in the header file SortedList.h, and implement all four methods in a new module named **SortedList.c**. Note that the interface includes three software-controlled yield options. Identify the critical section in each of your four methods, and add calls to pthread_yield, controlled by the yield options:
- in SortedList_insert if opt_yield & INSERT_YIELD
- in SortedList_delete if opt_yield & DELETE_YIELD
- in SortedList_lookup if opt_yield & SEARCH_YIELD
- in SortedList_length if opt_yield & SEARCH_YIELD

to force a switch to another thread at the critical point in each method.

Write a test driver program called **lab2b** that:
- takes a parameter for the number of parallel threads (**--threads=#**, default 1)
- takes a parameter for the number of iterations (**--iterations=#**, default 1)
- takes a parameter to enable the optional critical section yields (**--yield**=[**ids**], **i** for insert, **d** for delete, and **s** for searches)
- initializes an empty list.

- creates and initializes (with random keys) the required number (threads * iterations) of list elements. Note that we do this before creating the threads so that this time is not included in our start-to-finish measurement.
- notes the (high resolution) starting time for the run (using clock_gettime(2))
- starts the specified number of threads
- each thread
    - starts with a set of pre-allocated and initialized elements **(--iterations=#)**
    - inserts them all into a (single shared-by-all-threads) list
    - gets the list length
    - looks up and deletes each of the keys it had previously inserted
    - exits to re-join the parent thread
- waits for all threads to complete, and notes the (high resolution) ending time for the run.
- checks the length of the list to confirm that it is zero, and logs an error to stderr if it is not.
- prints to stdout
    - the number of operations performed
    - the total run time (in nanoseconds), and the average time per operation (in nanoseconds).
- exits with a status of zero if there were no errors, otherwise non-zero

Suggested sample output:

```
% ./lab2b --threads=10 --iterations=1000 --yield=id --sync=m
10 threads x 1000 iterations x (insert + lookup/delete) = 20000
operations
elapsed time: 527103247ns
per operation: 26355ns
```

Run your program with a single thread, and increasing numbers of iterations, and note the average time per operation. These results should be quite different from what you observed when testing your add function (in Project 2A) with increasing numbers of iterations. Graph the time per operation vs the number of iterations (for **--threads=1**).

QUESTION 2B.1A:

Explain the variation in time per operation vs the number of iterations?

QUESTION 2B.1B:

How would you propose to correct for this effect?

Run your program and see how many parallel threads and iterations it takes to fairly consistently demonstrate a problem. Note that even if you check for most inconsistencies in the list, your program may still experience segmentation faults when running multi-threaded without synchronization. Then run it again using various combinations of yield options and see how many threads and iterations it takes to fairly consistently demonstrate the problem. Make sure that you can demonstrate:

- conflicts between inserts (**--yield=i**)
- conflicts between deletes (**--yield=d**)
- conflicts between inserts and lookups (**--yield=is**)
- conflicts between deletes and lookups (**--yield=ds**)

Add two new options to your program to call two new versions of these methods: one set of operations protected by pthread_mutexes (**--sync=m**), and another protected by test-and-set spin locks (**--sync=s**). Using your **--yield** options, demonstrate that either of these protections seems to eliminate all of the problems, even for large numbers of threads and iterations.

Rerun your program without the yields, and choose an appropriate number of iterations (or apply the correction you identified in response to question 2.1). Note that you will only be able to run the unprotected method for a single thread. Note the execution times for the original and both new protected methods. Graph the (corrected) per operation times (for each of the three synchronization options: unprotected, mutex, spin) vs the number of threads.

QUESTIONS 2B.2A:

Compare the variation in time per protected operation vs the number of threads in Project 2B and in Project 2A. Explain the difference.

SUBMISSION:

Project 2B is due before midnight on Monday, May 2, 2016.

Your tarball should have a name of the form lab2b-*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 clean and dist targets
2%	reasonableness of README contents
	Code review (20%)
5%	overall readability and reasonableness
5%	SortedList implementation and correct yield placement
5%	mutex use

5% spin-lock implementation and use

Results (55%) ... (reasonable run)

5% threads and iterations
10% correct yield
10% correct mutex
10% correct spin
10% reasonable time reporting
10% graphs (showed what we asked for)

Analysis (15%) … (reasonably explained all results in README)

3% general clarity of understanding and completeness of answers
3% (Q2B.1A) explain variation in time per operation vs the number of iterations
3% (Q2B.1B) how would you propose to correct for this effect
6% (Q2B.2A) compare and explain sync costs vs threads