CS111 - Project 2C: Lock Granularity and Performance

INTRODUCTION:

In part B, the mutex and spin-lock are bottlenecks, preventing parallel access to the list. In this project, you will extend part B to deal with lock granularity and performance profiling. Part C (this part!) can be broken up into three major steps:

- Implement a new option to divide a list into sublists and support synchronization on sublists, thus allowing parallel access to the (original) list.
- Do performance measurement and instrumentation.
- Do performance profiling.

RELATION TO READING AND LECTURES:

Partitioned lists and finer granularity locking are discussed in chapter 29.

PROJECT OBJECTIVES:

- demonstrate the ability to recognize bottlenecks on large data structure
- experience with lockings on multi-list to allow parallel access
- experience with basic performance measurement and instrumentation
- experience with performance profiling

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 (*lab2c.c*) that compiles cleanly (with no errors or warnings), and implements the specified command line options (--threads, --iterations, --yield, --sync, --lists), 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:
    - (corrected) average time per operation (for none, mutex, and spin-lock) vs the ratio of threads per list.
- a README file contains:

- 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 2C.1, 2C.2, and 2C.3 (below).
- gprof profiling report.

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
- gprof (http://www.thegeekstuff.com/2012/08/gprof-tutorial/)
- 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.

Add a new **--lists=#** option to your previous lab2b program (now lab2c):

- break the single (huge) sorted list into the specified number of sub-lists (each with its own list header and synchronization object).
- change your lab2b to select which sub-list a particular key should be in based on a simple hash of the key, modulo the number of lists.
- figure out how to (safely and correctly) obtain the length of the list, which now involves enumerating all of the sub-lists
- each thread
  - starts with a set of pre-allocated and initialized elements **(--iterations=#)**
  - inserts them all into the multi-list (which sublist the key should go into determined by a hash of the key)
  - gets the list length
  - looks up and deletes each of the keys it inserted
  - exits to re-join the parent thread

Suggested sample output:

```
% ./lab2c --threads=10 --iterations=1000 --lists=10 --yield=id
--sync=m
10 threads x 1000 iterations x (insert + lookup/delete) = 20000
operations
elapsed time: 23155406ns
per operation: 1157ns
```

Rerun all three versions (unprotected, spin, mutex) of your program (without yields) for a range of *lists* values. Note that you will only be able to run the unprotected version for a single thread.

Graph the per operation times (for each of the three synchronization options) vs the ratio of threads to lists.

QUESTION 2C.1A

> **Explain the change in performance of the synchronized methods as a function of the number of threads per list.**

QUESTION 2C.1B

> **Explain why threads per list is a more interesting number than threads (for this particular measurement).**

Performance Analysis

Next, use gprof to do performance profiling. Compile the program with "-pg" option, and do the following:

- Run the program with a single (huge) list (**--lists=1**) for a (large) range of iterations (without yields and lock). Report in README:
  - where the time is spent (how the time is distributed among different calls).
  - the number of calls for each reported function.
  - the time spent on each function per call.
- Repeat the above step with a range of *lists* values (use multiple threads, but the number of threads and iterations should be fixed) and use **mutex** for synchronization.
- Repeat the above step with a range of *lists* values (use multiple threads, but the number of threads and iterations should be fixed) and use **spinlock** for synchronization.

QUESTION 2C.2A

> **Compare the time per operation when increasing the *lists* value. Explain your observations.**

QUESTION 2C.2B

> **Compare the time per operation between mutex and spinlock. Explain your observations.**

Sleep/Wakeup Races

In section 30.1, after Figure 30.3, Arpaci-Dusseau explains that the pthread_cond_wait operation takes a mutex as a parameter, and automatically/atomically releases it after putting the process to sleep, and reacquires it before allowing the process to resume.

QUESTIONS 2C.3A

> **Why must the mutex be held when pthread_cond_wait is called?**

QUESTION 2C.3B

> **Why must the mutex be released when the waiting thread is blocked?**

QUESTION 2C.3C

Why must the mutex be reacquired when the calling thread resumes?

QUESTION 2C.3D

Why must mutex release be done inside of pthread_cond_wait? Why can't the caller simply release the mutex before calling pthread_cond_wait?

QUESTION 2C.3E

Can pthread_cond_wait be implemented in user mode? If so, how? If it can only be implemented by a system call, explain why?

SUBMISSION:

Project 2C is due before midnight on Monday, May 9, 2016.

Your tarball should have a name of the form lab2c-*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%)
4%	overall readability and reasonableness
4%	multi-list implementation
4%	thread correctly sums up the length across sublists
4%	mutex use on multi-lists
4%	spin-lock use on multi-lists
	Results (40%) ... (reasonable run for a range of --lists= values)
5%	lists
10%	correct mutex
10%	correct spin
5%	reasonable time reporting

5% graphs (showed what we asked for)
5% gprof report

Analysis (30%) … (reasonably explained all results in README)

3% general clarity of understanding and completeness of answers
3% (Q2C.1A) explain change in performance of synchronized methods
3% (Q2C.1B) explain why threads per list is a more interesting number than threads
3% (Q2C.2A) compare the time per operation when increasing --lists= values
3% (Q2C.2B) compare the time per operation between mutex and spinlock
3% (Q2C.3A) why is mutex held when cond_wait is called
3% (Q2C.3B) why must mutex be released when the waiting thread is blocked
3% (Q2C.3C) why must mutex be reacquired when calling thread resumes
3% (Q2C.3D) why must mutex release be inside of cond_wait?
3% (Q2C.3E) can pthread_con_wait be implemented in user mode? How/Why not?