CS111 - Project 2B: Lock Granularity and Performance

INTRODUCTION:

In Project-2A, Part-2, the mutex and spin-lock proved to be bottlenecks, preventing parallel access to the list. In this project, you will do additional performance instrumentation to confirm this, and extend your previous solution to deal with this problem. Project 2B (this part!) can be broken up into a few major steps:

- Do performance instrumentation and measurement to confirm the cause of the problem.
- Implement a new option to divide a list into sublists and support synchronization on sublists, thus allowing parallel access to the (original) list.
- Do new performance measurements to confirm that the problem has been solved.

RELATION TO READING AND LECTURES:

Partitioned lists and finer granularity locking are discussed in sections 29.2-4

PROJECT OBJECTIVES:

- demonstrate the ability to recognize bottlenecks on large data structures
- experience with partitioning a serialized resource to improve parallelism
- experience with basic performance measurement and instrumentation
- experience with execution profiling
- experience with finding, installing, and exploiting new development tools

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 (*lab2_list.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 deliverable programs, output, graphs, and tarball. For your early testing you are free to run your program manually, but by the time you are done, all of the below-described test cases should be executed, the output captured, and the graphs produced automatically. The higher level targets should be:
  - **build** … compile all programs (default target)
  - **tests** … run all specified test cases to generate CSV results
  - **profile** … run tests with profiling tools to generate an execution profiling report
  - **graphs** … use gnuplot to generate the required graphs
  - **tarball** … create the deliverable tarball
  - **clean** … delete all generated programs and output
- **lab_2b_list.csv** - containing your results for all of the Part-2 performance tests.
- execution profiling report showing where time was spent :
  - In the un-partitioned mutex implementation, for 32 threads.
  - In the un-partitioned spin-lock implementation, for 32 threads.
- graphs (.**png** files), created by *gnuplot(1)* on the above csv data showing:
  - lab2b_1.png … throughput vs number of threads for mutex and spin-lock synchronized adds and list operations.
  - lab2b_2.png … mean time per mutex wait and mean time per operation for mutex-synchronized list operations.
  - lab2b_3.png … number of successful iterations for each synchronization method.
  - lab2b_4.png … throughput vs number of threads for mutexes with partitioned lists.
  - lab2b_5.png … throughput vs number of threads for spin-locks with partitioned lists.
  - a **README.txt** 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., limitations, features, testing methodology, etc.).
  - brief (a few sentences per question) answers to each of the questions (below).

PREPARATION:

To perform this assignment, you will need to research, choose, install and master a multi-threaded execution profiling tool. Execution profiling is a combination of compile-time and run-time tools that analyze a program's execution to determine how much time is being spent in which routines. There are three standard Linux profiling tools:

- The standard Linux *gprof(1)* tool is quite simple to use, but its call-counting mechanism is not-multi-thread safe, and its execution sampling is not multi-thread aware. As such, it is not usable for analyzing the performance of multi-threaded applications. There are other tools that do solve this problem. The two best-known are:
- valgrind … best known for its memory leak detector, which has an interpreted execution engine that can extract a great deal of information about where cycles are being spent, even estimating cache misses. It does work for multi-threaded programs, but its interpreter does not provide much parallelism. As such it is not useful for examining high contention situations.
- gperftools … a wonderful set of performance optimization tools from Google. It includes a profiler that is quite similar to gprof (in that it samples

real execution). This is probably the best tool to use for this problem.

This project is about scalable parallelism, which is only possible on a processor with many cores. You can do most of your development and testing on any Linux system, but if your personal computer does not have more than a few cores, you will not be able to do real multi-threaded performance testing on it. Lab servers are available if you need access to a larger machine for your final testing.

PROJECT DESCRIPTION:

Review your results from Lab 2 Parts 1 (lab2_add-5.png) and 2 (lab2_list-4.png). In Part-1, for Compare-and-Swap and Mutex throughput (total operations per second), we saw that adding threads took us from tens of ns per operation to small hundreds of ns per operation. Looking at the analogous results in Part-2, we see the (un-adjusted for length) time per operation go from a few microseconds, to tens of microseconds. For the adds, moderate contention added ~100ns to each synchronization operation. For the list operations, moderate contention added ~10us to each synchronization operation. This represents a 100x difference in the per operation price of lock contention.

Go back to your lab1 and lab2 data, and create a new plotting script that will graph the same data, but plotting the throughput (operations per second) rather than the time per operation:

- Mutex synchronized adds, 10,000 iterations, 1,2,4,8,12 threads
- Spin-lock synchronized adds, 10,000 iterations, 1,2,4,8,12 threads
- Mutex synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads
- Spin-lock synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads

In the previous lab, we gave you all of the necessary data reduction scripts. In this lab, you will have to create your own … but you can use the scripts provided in the previous lab as a starter:

- To get the throughput, divide one Billion (number of nanoseconds in a second) by the time per operation (in nanoseconds).
- Remember that for the list operations there is only one lock/unlock for an entire list, so we should not divide the time per operation by the list length.
- Call this graph lab2b_1.png

The most obvious differences, which we already knew:

- adds are much less expensive operations than inserts and searches in a long list.
- spin-locks waste increasingly more cycles as the probability of contention increases.

But these throughput graphs show us something that was not as obvious in the cost per operation graphs:

- The add throughput quickly levels off … we have saturated the CPU and the overhead of synchronization seems to increase only very slowly.
- The list operation throughput continues to drop, as the overhead of synchronization increases with the number of threads … and much worse for spin-locks.

The obvious conclusions (from both the cost-per-operation graphs you produced last week, and the throughput graphs you have just produced) are:

- The throughput of parallel synchronized linked list operations does not scale as well as the throughput of parallel synchronized add operations.
- The reduction in throughput with increasing parallelism is due to an increasing time per operation.

Since the code inside the critical section does not change with the number of threads, it seems reasonable to assume that the added execution time is being spent getting the locks.

QUESTION 2.3.1 - Cycles in the basic implementation:

> **Where do you believe most of the cycles are spent in the 1 and 2-thread tests (for both add and list)? Why do you believe these to be the most expensive parts of the code?**
>
> **Where do you believe most of the time/cycles are being spent in the high-thread spin-lock tests?**
>
> **Where do you believe most of the time/cycles are being spent in the high-thread mutex tests?**

It should be clear why the spin-lock implementation performs so badly with a large number of threads. But the mutex implementation should not have this problem. Now you have some theories about why these algorithms scale so poorly. But theories are only theories. We need some data to confirm our theories.

Execution Profiling

Build your program with debug symbols, choose your execution profiling package, install it, run the spin-lock list test (1,000 iterations 12 threads) under the profiler, and analyze the results to determine where the cycles are being spent.

The default output from google-pprof will show you which routine is consuming most of the cycles. If you then re-run google-pprof with the --list option (specifying that routine), it will give you a source-level breakdown of how much time is being spent on each instruction. You should get a very clear answer to the question of where the program is spending its time. Update your Makefile to run this test and generate the results automatically (make profile), include your profiling report in your submitted tarball, and identify it in your README file.

QUESTION 2.3.2 - Execution Profiling:

> **Where (what lines of code) are consuming most of the cycles when the spin-lock version of the list exerciser is run with a large number of threads?**
>
> **Why does this operation become so expensive with large numbers of threads?**

Timing Mutex Waits

In the mutex case, we are not spinning. A thread that cannot get the lock is blocked, and not consuming any cycles. How could we confirm that, in the mutex case, most threads are spending most of their time waiting for a lock?

Update your mutex implementation to:

- Note the high-resolution time before and after getting the mutex, compute the elapsed difference, and add that to a (per-thread) total.
- When the program completes, add up the total lock acquisition time (for all threads) and divide it by the number of lock operations to compute an average wait-for-lock, and add this number, as a new last column, to the output statistics for the run.

Run the list mutex test again for 1,000 iterations and 1, 2, 4, 8, 16, 24 threads, and plot the wait-for-lock time, and the average time per operation against the number of competing threads. Call this graph lab2b_2.png.

QUESTION 2.3.3 - Mutex Wait Time:

Look at the average time per operation (vs # threads) and the average wait-for-mutex time (vs #threads).

Why does the average lock-wait time rise so dramatically with the number of contending threads?

Why does the completion time per operation rise (less dramatically) with the number of contending threads?

How is it possible for the wait time per operation to go up faster (or higher) than the completion time per operation?

Addressing the Underlying Problem

While the details of how contention degrades throughput are different for these two mechanisms, all of the degradation is the result of increased contention. This is the fundamental problem with coarse-grained synchronization. The classic solution to this problem is to partition the single resource (in this case a linked list) into multiple independent resources, and divide the requests among those sub-resources.

Add a new **--lists=#** option to your lab2_list program:

- break the single (huge) sorted list into the specified number of sub-lists (each with its own list header and synchronization object).
- change your lab2_list program 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
- Include the number of lists as the fourth number (always previously 1) in the output statistics report.

The supported command line options and expected output are illustrated below:

```
% ./lab2_list --threads=10 --iterations=1000 --lists=5
--yield=id --sync=m
List-id-m,10,1000,5,20000,23155406,1157
%
```

Confirm the correctness of your new implementation:

- Run your program with --yield=id, 4 lists, 1,4,8,12,16 threads, and 1, 2, 4, 8, 16 iterations (and no synchronization) to see how many iterations it takes to reliably fail (and make sure your Makefile expects some of these tests to fail).

- Run your program with --yield=id, 4 lists, 1,4,8,12,16 threads, and 10, 20, 40, 80 iterations, --sync=s and --sync=m to confirm that updates are now properly protected.
- Graph these results (as you did last week) and include the results as lab2b_3.png.

Now that we believe the partitioned lists implementation works, we can measure its performance. Rerun both synchronized versions without yields for 1000 iterations, 1,2,4,8,12 threads, and 1,4,8,16 lists. For each synchronization mechanism, graph the aggregated throughput (total operations per second, as you did for lab2a_1.png) vs the number of threads, with a separate line for each number of lists. Call these graphs lab2b_4.png and lab2b_5.png

QUESTION 2.3.4 - Performance of Partitioned Lists

Explain the change in performance of the synchronized methods as a function of the number of lists.

Should the throughput continue increasing as the number of lists is further increased? If not, explain why not.

It seems reasonable to suggest the throughput of an N-way partitioned list should be equivalent to the throughput of a single list with fewer (1/N) threads. Does this appear to be true in the above curves? If not, explain why not.

SUBMISSION:

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%)
2%	untars expected contents
3%	clean build w/default action (no warnings)
3%	Makefile produces csv output, graphs, profiling reports, tarball
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%) ... produces correct output
5%	lists
5%	correct mutex
5%	correct spin
5%	reasonable time per operation reporting

5% reasonable wait for mutex reporting
5% graphs (showed what we asked for)
10% profiling report (clearly shows where the cycles wwent)

Note: if your program does not accept the correct options or produce the correct output, you are likely to receive a zero for the results portion of your grade. Look carefully at the sample commands and output. If you have questions, ask your TA.

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

2% General clarity of thought and understanding
7% 2.3.1 where the cycles go
7% 2.3.2 profiling
7% 2.3.3 wait time
7% 2.3.4 list partitioning