

CS111 - Project 2A: Races and Synchronization

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 single variables and complex data structures (an ordered linked list). It is broken up into multiple steps:

- Part 1 - updates to a shared variable:
 - Write a multithreaded application (using pthreads) 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 instrumentation and measurement.
 - Analyze and explain the observed performance.
- Part 2 - updates to a shared complex data structure:
 - 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 and demonstrate the race conditions when performing linked list

operations, and address them with different synchronization mechanisms.

- Do performance instrumentation and measurement.
- Analyze and explain the observed performance.

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 were described in (many sections of) chapter 28.

Synchronization of partitioned lists was introduced in section 29.2.

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:

- Four C source modules that compile cleanly (with no errors or warnings):
 - **lab2_add.c** - a C program that implements and tests a shared variable add function, implements the (below) specified command line options, and produces the (below) specified output statistics.
 - **SortedList.h** - a header file (supplied by us) describing the interfaces for linked list operations.
 - **SortedList.c** - a C module that implements insert, delete, lookup, and length methods for a sorted doubly linked list (described in the provided header file, including correct placement of *yield* calls).
 - **lab2_list.c** - a C program that implements the (below) specified command line options and produces the (below) specified output statistics.
- 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 (over 200) specified test cases to generate results in CSV files. Note that the lab2_list program is expected to fail when running multiple threads without synchronization. Make sure that your Makefile continues executing despite such failures (e.g. put a '-' in front of commands that are expected to fail).
 - **graphs** ... use *gnuplot(1)* and the supplied data reduction scripts to generate the required graphs
 - **tarball** ... create the deliverable tarball
 - **clean** ... delete all generated programs and output
- **lab2_add.csv** - containing all of your results for all of the Part-1 tests.
- **lab2_list.csv** - containing all of your results for all of the Part-2 tests.
- graphs (.png files), created by *gnuplot(1)* on the above **csv** files with the supplied data reduction scripts:

- For lab2_add
 - **lab2_add-1.png** ...threads and iterations required to generate a failure (with and without yields)
 - **lab2_add-2.png** ... Average time per operation with and without yields.
 - **lab2_add-3.png** ... Average time per (single threaded) operation vs. the number of iterations.
 - **lab2_add-4.png** threads and iterations that can run successfully with yields under each of the three synchronization methods.
 - **lab2_add-5.png** Average time per (multi-threaded) operation vs. the number of threads, for all four versions of the add function.
- For lab2_list
 - **lab2_list-1.png** ... average time per (single threaded) unprotected operation vs. number of iterations (illustrating the correction of the per-operation cost for the list length).
 - **lab2_list-2.png** ... threads and iterations required to generate a failure (with and without yields).
 - **lab2_list-3.png** ... iterations that can run (protected) without failure.
 - **lab2_list-4.png** ... (corrected) average time per operation (for unprotected, mutex, and spin-lock) vs. number of threads.
- 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. limitation, features, testing methodology).
 - brief (1-4 sentences per question) answers to each of the questions (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) ... is a general and powerful tool for producing a wide variety of graphs, and is commonly used for organizing and reporting performance data. We are providing you with sample data reduction scripts for the first parts of this assignment:
 - [lab2_add.gp](#)
 - [lab2_list.gp](#)

To use these scripts you will need a recent version of *gnuplot* installed on your system.

These scripts take no arguments, read *comma-separated value* (CSV) input files with standard names (lab2_add.csv, lab2_list.csv), and produce graphical output .png files with standard names.

In the next and final part of this assignment, you can use these as a basis for creating your own graphing scripts.

PART 1: adds to a shared variable

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 **lab2_add** 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
- prints to stdout a comma-separated-value (CSV) record including
 - the name of the test (“add-none” for the most basic usage)
 - the number of threads (from **--threads=#**)
 - the number of iterations (from **--iterations=#**)
 - the total number of operations performed (threads x iterations x 2, the “x 2” factor because you add 1 first and then add -1)
 - the total run time (in nanoseconds)
 - the average time per operation (in nanoseconds).
 - the total at the end of the run (0 if there were no conflicting updates)
- If the run completes successfully, exit with a return code of zero. If any errors (other than a non-zero final count) are encountered, exit with a non-zero exit code.

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

```
% ./lab2_add --iterations=10000 --threads=10
add-none,10,10000,200000,6574000,32,374
%
```

Run your program for ranges of *threads* (2, 4, 8, 12) and *iterations* (100, 1000, 10000, 100000) values, capture the output, and note how many threads and iterations it takes to (fairly consistently) result in a failure (non-zero sum).

QUESTION 2.1.1 - causing conflicts:

Why does it take many iterations before errors are seen?

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

There are ways to cause a thread to immediately yield (rather than waiting for a time slice end to preempt it). Posix includes a `sched_yield` operation, and Linux includes a `pthread_yield` operation. Extend the basic add routine to more easily cause the failures:

```
int opt_yield;

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

Add a new **--yield** option to your driver program that sets `opt_yield` to 1. If this option has been specified, each line of statistics output should begin with “**add-yield**”. Re-run your tests, with yields, for ranges of threads (2,4,8,12) and iterations (10, 20, 40, 80, 100, 1000, 10000, 100000), capture the output, 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 a range threads (2, 8) and of iterations (100, 1000, 10000, 100000). You should note that the **--yield** runs are much slower than the non-yield runs.

QUESTION 2.1.2 - cost of yielding:

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

Is it possible to get valid per-operation timings if we are using the --yield option? If so, explain how. If not, explain why not.

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

If you install *gnuplot*(1) and append all of your test output to a file called **lab2_add.csv**, you can use our sample data reduction scripts to produce this and all other required data plots.

QUESTION 2.1.3 - measurement errors:

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

If the cost per iteration is a function of the number of iterations, how do we know how many iterations to run (or what the “correct” cost is)?

Implement three new versions of the add function:

- one protected by a `pthread_mutex`, enabled by a new **--sync=m** option. When running this test, the output statistics line should begin with “add-m” or “add-yield-m”.
- 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**. When running this test, the output statistics line should begin with “add-s” or “add-yield-s”.
- 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. When running this test, the output statistics line should begin with “add-c” or “add-yield-c”.

Use your **--yield** option to confirm that, even for large numbers of threads (2, 4, 8, 12) and iterations (10,000 for mutexes and CAS, only 1,000 for spin locks) that reliably failed in the unprotected scenarios, all three of these serialization mechanisms eliminate the race conditions in the add critical section. Capture the output from these confirmation runs. [Note that we suggest a smaller number of threads/iterations when you test spin-lock synchronization]

Using a large enough number of iterations (e.g. 10,000) to overcome the issues raised in the question 2.1.3, test all four (no yield) versions (unprotected, mutex, spin-lock, compare-and-swap) for a range of number of threads (1,2,4,8,12) and capture the output. Graph the average time per operation (non-yield), vs the number of threads.

QUESTION 2.1.4 - costs of serialization:

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

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

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

PART 2: sorted, doubly-linked, list

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` or `sched_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 & LOOKUP_YIELD`
- in `SortedList_length` if `opt_yield & LOOKUP_YIELD`

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

Write a test driver program called **lab2_list** 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 (any combination of) optional critical section yields (**--**

yield=[idl], **i** for insert, **d** for delete, and **l** for lookups).

- initializes an empty list.
- creates and initializes (with random keys) the required number (threads x 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.
- prints to stdout a comma-separated-value (CSV) record including:
 - the name of the test, which is of the form: **list-yieldopts-syncopts**
 - where *yieldopts* = {**none,i,d,l,id,il,dl,idl**}
 - Where *syncopts* = {**none,s,m**}
 - the number of threads (from **--threads=#**)
 - the number of iterations (from **--iterations=#**)
 - the number of lists (always 1 in this project)
 - the total number of operations performed (threads x iterations x (insert + lookup + delete))
 - the total run time (in nanoseconds) for all threads
 - the average time per operation (in nanoseconds).
- exits with a status of zero if there were no errors, otherwise non-zero

In part one, a synchronization error merely resulted in the subtracts and adds not balancing out. In this part, a synchronization error is likely to result in a corrupted list. If, at any time, you

find evidence of a corrupted list (e.g. you cannot find a key that you know you inserted, or the list length is not zero at the end of the test), you should log an error message (to stderr) and exit immediately without producing the above results record. Note that in some cases your program may not detect an error, but may simply experience a segmentation fault. When you look at your test results, you should consider any test that did not produce output to have failed.

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

```
% ./lab2-list --threads=10 --iterations=1000 --yield=id
list-id-none,10,1000,1,30000,527103247,25355
%
```

Run your program with a single thread, and increasing numbers of iterations (10, 100, 1000, 10000, 20000), capture the output, and note the average time per operation. These results should be quite different from what you observed when testing your add function with increasing numbers of iterations. Graph the time per operation vs the number of iterations (for **--threads=1**).

If you append all of your test output to a file called **lab2_list.csv**, you can use the supplied data reduction script to produce this and all other required data plots.

You will observe that the time per iteration eventually becomes linear with the number of iterations! This is because the time to insert into or search a sorted list is proportional to the list length. This is to be expected ... but we are primarily interested in the cost of serialization, and so we would like to separate the per operation costs from the per-element costs. The easiest way to do this is to divide the cost per iteration (total / (threads * iterations)) by the average search distance (iterations/4). Why iterations/4?

- Inserts take list length from 0 to iterations, and then from iterations to 0. Thus, the average list length is iterations/2.
- Each insert or search operation, on average, has to run through half the list, which gives us an average search distance of iterations/4.

With this correction, your program should (modulo startup time) report more stable per-

operation costs. Note that the provided data reduction script graphs both the raw time per operation and the time corrected for the list length.

Run your program and see how many parallel threads (2,4,8,12) and iterations (10,100,1000) it takes to fairly consistently demonstrate a problem. Then run it again using various combinations of yield options and see how many threads (2,4,8,12) and iterations (2,4,8,16,32) 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=il**)
- conflicts between deletes and lookups (**--yield=dl**)

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 (12) and iterations (32).

Choose an appropriate number of iterations (e.g. 1000) to overcome start-up costs and rerun your program without the yields. Note that you will only be able to run the unprotected method for a single thread, but you should be able to run the protected methods for a wide range of numbers of threads (1, 2, 4, 8, 12, 16, 24). Graph the (corrected) per operation times (for each of the three synchronization options: unprotected, mutex, spin) vs the number of threads.

QUESTION 2.2.1 - scalability of Mutex

Compare the variation in time per protected operation vs the number of threads (for mutex-protected operations) in Part-1 and Part-2, commenting on similarities/differences and offering explanations for them.

QUESTION 2.2.2 - scalability of spin locks

Compare the variation in time per protected operation vs the number of threads for Mutex vs Spin locks, commenting on similarities/differences and offering explanations for them.

SUBMISSION:

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 Inxsrv09). 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, tarball
2%	reasonableness of README contents
Code review (20%)	
4%	overall readability and reasonableness
2%	add: yields correct and in appropriate places
4%	list: yields correct and in appropriate places
2%	mutex correctly used for add

- 2% mutex correctly used for list
- 2% spin lock correctly implemented and used for add
- 2% spin lock correctly implemented and used for list
- 2% compare-and-swap correctly implemented and used to implement atomic add

Results (50%) (reasonable run)

- 2% add: threads and iterations
- 2% add: correct output format
- 2% add: reasonable time reporting
- 3% add: correct yield
- 3% add: correct mutex
- 3% add: correct spin
- 3% add: correct cas
- 2% add: graphs (showed what we asked for)

- 2% list: threads and iterations
- 2% list: correct output format
- 2% list: reasonable time reporting
- 6% list: correct yield
- 6% list: correct mutex
- 6% list: correct spin
- 6% list: graphs (showed what we asked for)

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 (20%) ... (reasonably explained all results in README)

- 4% General clarity of thought and understanding
- 2% each 2.1.1-2.1.4
- 4% each 2.2.1-2

