Lab 4. Synchronization

Project goals

This is not meant to be a difficult project, but is intended to force you to directly engage (at a low level) with a range synchronization problems. The educational and skill demonstration goals of this project are:

- 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.

Assignment overview

The assignment is divided into three general parts and an optional design problem.

1. conflicting read-modify-write operations on a single variable.
2. conflicting search and update operations in an ordered doubly linked list.
3. understanding the crux issue in sleep/wakeup races
4. (design problem) correct a significant measurement artifact

In this assignment, you will:

- implement a few basic operations on shared variables and complex data structures.
- implement test programs to exercise those operations.
- demonstrate race conditions resulting from unprotected parallel execution.
- implement and demonstrate the effectiveness of a few different serialization mechanisms.
- measure the relative performance of these implementations as a function of the number of competing threads and operations, and explain your results.
- explain the correct handling of a very important critical section that was only briefly covered in the reading.

Your deliverables for this assignment will include:

- object modules that build cleanly and implement specified APIs
- programs that build cleanly and implement specified CLIs
- executions of those programs to demonstrate specified problems and solutions
- timing results and graphs
- brief (1 paragraph) explanations of those results

To perform this assignment, you will need to learn to exploit a few new tools:

- clock_gettime(2) … high resolution timers
- GCC atomic builtins

- gnuplot(1) … you may also find it useful to learn to use this versatile package for producing your graphs.

Part 1 – parallel updates to a shared variable

Summary of Deliverables

- the source for a C program and Makefile that cleanly (no warnings) builds using gcc on an x86-64 GNU/Linux system, implements the (below) specified command line options, drives one or more parallel threads that do adds and subtracts to a shared variable, and reports on the final sum and performance.
- graphs 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.
- written brief (a few sentences per question) answers to questions 1.1, 1.2 and 1.3.

Detailed instructions

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 addtest 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
  - exits 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:

```
$ ./addtest --iter=10000 --threads=10
10 threads x 10000 iterations x (add + subtract) = 200000 operations
ERROR: final count = 374
elapsed time: 6574000 ns
per operation: 2 ns
```

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).

QUESTIONS 1.1:

1. Why does it take this many threads or iterations?
2. 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=1` 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.

QUESTIONS 1.2

1. Why does the average cost per operation drop with increasing iterations?
2. How do we know what the “correct” cost is?
3. Why are the --yield runs so much slower? Where is the extra time going?
4. 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 eliminates 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 of all four versions (unprotected, mutex, spin-lock, compare-and-swap) vs the number of threads.

QUESTIONS 1.3

1. Why do all of the options perform similarly for low numbers of threads?
2. Why do the three protected operations slow down as the number of threads rises?
3. Why are spin-locks so expensive for large numbers of threads?

Part 2 – parallel updates to complex data structures

Summary of Deliverables

- the source for a C module and Makefile that cleanly (no warnings) builds using gcc on an x86-64 GNU/Linux system, and implements insert, delete, lookup, and length methods for a sorted doubly linked list (described in a provided header file, including correct placement of pthread_yield calls).
- the source for a C program and Makefile that cleanly (no warnings) builds using gcc on an x86-64 GNU/Linux system, implements the (below) specified command line options, drives one or more parallel threads that do operations on a shared linked list, and reports on the final list and performance.
- graphs of:
    - average time per unprotected operation vs number of iteration (single thread)
    - (corrected) average time per operation (for none, mutex, and spin-lock) vs number of threads.
    - (corrected) average time per operation (for none, mutex, and spin-lock) vs the ratio of threads per list.

  o
- written brief (a few sentences per question) answers to questions 2.1, 2.2 and 2.3.

Detailed instructions

Review the interface specifications for a sorted doubly linked list package described in the header file `SortedList.h`, and implement all four those methods. 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`

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

Write a test driver program (you can start with your driver from part 1) called `sltest` 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. 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 of which will use the above list function to

    - insert each of its elements (iterations parameter) into the list
    - gets the list length
    - look up each of the keys it inserted
    - deletes each returned element from the list
    - exits to re-join the parent thread
- wait 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:

```
$ ./sltest --threads=10 --iterations=1000 --lists=10 --sync=m 10
threads x 100 iterations x (ins + lookup/del) x (100/2 avg len) = 100000 operations
elapsed time: 4225760 ns
per operation: 4 ns
```

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 with increasing numbers of iterations. Graph the time per operation vs the number of iterations (for a `--threads=1`).

QUESTIONS 2.1

- Explain the variation in time per operation vs the number of iterations? 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 seem 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 2.2

- Compare the variation in time per protected operation vs the number of threads in Part 2 and in Part 1. Explain the difference.

Add a new `--lists=#` option to your test driver 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 test driver 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) re-implement the length function, which now needs to enumerate all of the sub-lists.

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.

QUESTIONS 2.3

- Explain the the change in performance of the synchronized methods as a function of the number of threads per list.
- Explain why threads per list is a more interesting number than threads (for this particular measurement).

Part 3 – sleep/wakeup races

Summary of Deliverables

- brief (a few sentences per question) answers to questions 3-1

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 3-1

- Why must the mutex be held when pthread_cond_wait is called?
- Why must the mutex be released when the waiting thread is blocked?
- Why must the mutex be reacquired when the calling thread resumes?
- Why must this be done inside of pthread_cond_wait? Why can't the caller simply release the mutex before calling pthread_cond_wait?
- Can this be done in a user-mode implementation of pthread_cond_wait? If so, how? If it can only be implemented by a system call, explain why?

Part 4 – Design Problem – Artifact Correction

At the beginning of Questions 1-2 you explored a measurement artifact: an error in our time per operation computation that was introduced by the way in which we implemented the test driver. In part 1 you used a large number of iterations to dilute the artifact per operation. That isn't a bad approach, but a better approach would be to eliminate the artifact (avoid measuring it, or find a way to subtract it from our results).

Can you design and implement a change to the measurement program to eliminate this artifact? Having done

so, what data can you generate to testify to the elimination (or at least substantial reduction) of that artifact?

If you choose to do this:

- add a Part 4 in which you explain your approach.
- add a `--correct` switch to your `addtest` and `sltest` programs that enables your artifact correction.
- We will review your design and test the effectiveness of your correction.

$Id: lab4.html,v 1.3 2016/03/02 19:38:11 eggert Exp $