Project 4: Locks and Threads

Overview

In this project, you will be getting a feel for threads, locks, and performance. The first entity you will build is called a spin lock. A spin lock uses some kind of powerful hardware instruction in order to provide mutual exclusion among threads. Then, you may be not satisfied with performances of the simple spin lock, and step further to develop a mutex. With locks in hand, you can build thread-safe versions of three common data structures: counter, list and hash table. Finally, you will try to make a nice report by comparing different lock implementations and concurrency levels. Ready? Welcome to parallel universes!

Readings

OSTEP Chapter 27, Chapter 28, Chapter 29.

Part 1: Spin Locks

To build a spin lock, you will use the x86 exchange primitive. As this is an assembly instruction, you will need be able to call it from C. Fortunately, gcc conveniently lets you do this without too much trouble: xchg.c.

For those interested in learning more about calling assembly from C with gcc, see here (Note that you may need to add the gcc option `-std=gnu99` since your code is no longer in ansi C).

To learn more about this instruction, you should read about it in the Intel assembly instruction manual found here. However, I bet you can figure it out without looking.

The lock you build should define a `spinlock_t` data structure, which contains any values needed to build your lock, and two routines:

```
spinlock_acquire(spinlock_t *lock)
spinlock_release(spinlock_t *lock)
```

These routine(s) should use the xchg code above as needed to build your spin lock.

Part 2: Mutex

Mutex is another implementation of a lock which causes threads to sleep rather than spin when the lock is unavailable. Linux provides the system calls named *futex* for this purpose. Want to know details? Go and read its man page.

One difficulty on using futex is that it is a low level lock primitive. The man page says

> Bare futexes are not intended as an easy-to-use abstraction for end-users. (There is no wrapper function for this system call in glibc.)

However, we are not end-users, we are locksmiths! To call futex, sys_futex.c is a wrapper.

Again, the mutex you build should define a `mutex_t` data structure and two routines:

```
mutex_acquire(mutex_t *lock)
mutex_release(mutex_t *lock)
```

Hint: starting from the simplest one and refining your mutex step by step.

Part 3: Using Your Lock

Next, you will use your locks to build three concurrent data structures. The three data structures you will build are a thread-safe counter, list, and hash table.

To build the counter, you should implement the following code:

```
void counter_init(counter_t *c, int value);
int counter_get_value(counter_t *c);
void counter_increment(counter_t *c);
void counter_decrement(counter_t *c);
```

You will make these routines available as a shared library, so that multi-threaded programs can update a shared counter. The library will be called `libcounter.so`

To build the list, you should implement the following routines:

```
void list_init(list_t *list);
void list_insert(list_t *list, unsigned int key);
void list_delete(list_t *list, unsigned int key);
void *list_lookup(list_t *list, unsigned int key);
```

The routines do the obvious things. The structure `list_t` should contain whatever is needed to manage the list (including a lock). Don't do anything fancy; just a simple insert-at-head list would be fine. This library will be called `liblist.so`

To build the hash table, you should implement the following code:

```
void hash_init(hash_t *hash, int size);
void hash_insert(hash_t *hash, unsigned int key);
void hash_delete(hash_t *hash, unsigned int key);
void *hash_lookup(hash_t *hash, unsigned int key);
```

The only difference from the list interface is that the user can specify the number of buckets in the hash table. Each bucket should basically contain a list upon which to store elements. This library will be called `libhash.so` The hash table should simply use one list per bucket. How can you make sure to allow as much concurrency as possible during accesses to the hash table?

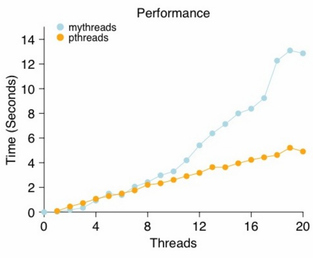
Part 4: Comparing Performance

Finally, you will write up a report on some performance comparison experiments. Specifically, you will

- Compare the performance of **your locks** versus the performance of pthread locks
- Compare different implementations of **your locks** (spin lock versus mutex, different mutex)
- Compare the fairness of **your locks**
- If your locks have parameters (e.g., two-phrase locks), how parameters influence the performances of your locks?
- Give some analysis for each experiment. Why you can observe the result? What cause the performance difference?
- ...

You will do this for each of your data structures (counter, list, hash table).

The result of comparison should be presented accurately and clearly. Figures and tables are required. For example, you can give such an figure that along the x-axis, you vary the number of threads contending for the data structure, while the y-axis will plot how long it took all of the threads to finish running. The figure might look like this:



This graph plots the average performance of many threads updating a shared counter: each thread would call `counter_increment(&counter)` in a loop max times.

For this experiment, max was set to 1000000, and each curve in the figure shows the performance of either using your own spin lock or a pthreads lock inside the counter library.

Similar plots should be made for:

- A list insertion test - where you have threads each insert say 10e6 items into a list (and scale the number of lists threads)
- A list insert delete test - where you have threads each first insert say 10e6 items into a list, then delete all items. How about random insertion and delete?
- A hash table test - same as above but with a hash table with a reasonable bucket size
- A hash table scaling test - this test should fix the number of threads (say at 20) and vary the number of buckets
- ...

Rather than only comparing pthread lock and your spin lock, you could also plot curves for your different mutex implementations.

Timing should be done with `gettimeofday()` . Read the man page for details. One thing that is good to do: write a wrapper which returns the time in seconds as a floating-point value. This makes the timing routine really easy to use.

To make your graph less noisy, you will have to run multiple iterations, as well as to make sure to let each experiment run long enough so as to be meaningful. You might then plot the average (as done above) and even a standard deviation.

Hand In

- Source files (.c, .h) of your locks and three libraries (counter, hash, and list)
- Makefile which builds each of the libraries
- The **PDF** version of your report.

Adapted from WISC CS537 by Remzi Arpaci-Dusseau