CS416 Project 2: User-level Thread Library and Scheduler

Due: 03/02/2019

In Project 1, you learned how to use pThread library on multi-thread programming. In Project 2, you will get a chance to exploit the logic inside a thread library. In this assignment, you are required to implement a pure user-level thread library that has the same interface to pThread Library. Since you will be providing a multi-thread environment, you will also need to implement pthread mutexes which are used to guarantee exclusive access in critical section. This assignment is intended to illustrate the mechanics and difficulties of scheduling tasks within an operating system.

1. Description

You are given a code skeleton structured as follows:

- my_pthread_t.h
- my_pthread.c
- Makefile
- Benchmark

my_pthread_t.h contains thread library function prototypes and definition of important data structures. my_pthread.c includes the actual implementation of each API function. Benchmark folder includes the provided benchmarks that you should use to verify your implementation and do performance study.

You need to implement all of the API functions listed below in section 1.1 and the corresponding scheduler function and some auxiliary functions whatever you need.

1.1 Thread Library API

```
int my_pthread_create(my_pthread_t * thread, pthread_attr_t * attr,
                      void *(*function)(void*), void * arg);
```

Creates a thread that executes function. You could ignore attr.

```
void my_pthread_yield();
```

Explicit call to the my_pthread_yield in a thread will voluntarily give up CPU resource to other threads. That is to say, the thread context will be swapped out, and the scheduler context will be swapped in so that the scheduler could put current thread in run queue and choose the next thread to run.

```
void my_pthread_exit(void *value_ptr);
```

Explicit call to the my_pthread_exit library to end the pthread that called it. If the value_ptr isn't NULL, any return value from the thread will be saved.

```
int my_pthread_join(my_pthread_t thread, void **value_ptr);
```

Call to the my_pthread_join ensures that calling thread will not continue execution until the one it references exits. If value_ptr is not null, the return value of the exiting thread will be passed back.

```
int my_pthread_mutex_init(my_pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

This function initializes a my_pthread_mutex_t created by the calling thread. The 'mutexattr' is ignored.

```
int my_pthread_mutex_lock(my_pthread_mutex_t *mutex);
```

Locks a given mutex, other threads attempting to access this mutex will not run until it is unlocked.

```
int my_pthread_mutex_unlock(my_pthread_mutex_t *mutex);
```

Unlocks a given mutex.

```
int my_pthread_mutex_destroy(my_pthread_mutex_t *mutex);
```

Destroys a given mutex. Mutex should be unlocked before doing so.

1.2 User-level Context

It is strongly suggested that you investigate the system library ucontext.h. It has a series of commands to make, swap and get the currently running context in the user space. When a context is running, it will continue running until it completes.

1.3 Scheduler

Since your thread library is managed totally in user-space, you also need to have a scheduler in your thread library to determine which thread to run next. In the second part of the assignment, you are required to implement the following two scheduling policies:

1. **Pre-emptive SJF:** The first scheduling algorithm you are required to implement is a pre-emptive SJF which is also known as STCF. Unfortunately, you may have noticed that our scheduler DO NOT have the knowledge how long a thread will run. Hence, in our scheduler, we could book-keep the time quantum each thread has to run; this is based on an assumption that the more time quantum a thread has run, the longer this job will run to finish. Here are the hints to implement:
    - In the TCB of each thread, maintain a counter that indicates how many time quantum has run. And after the first time quantum finishes, increment that counter by one.
    - To schedule the shortest job, you will have to find the thread which has the minimum counter value, remove the thread from the queue, and then context switch to it.
2. **Multi-level Feedback Queue:** The second scheduling algorithm you need to implement is MLFQ. In this algorithm, you have to maintain a queue structure with multiple levels. Remember, the higher the priority, the shorter time slice its corresponding level of run queue will be. More description and logic of MLFQ is clearly stated in Chapter 8 of textbook. Here are the hints to implement:
    - Instead of a single run queue, you need multiple levels of run queue. It could be a 4-level or 8-level queue as you like.
    - When a thread has finished one "time quantum", move it to the next level of run queue.
    - Your scheduler should always pick a thread at the top-level of run queue.

For both of the two scheduling algorithms, you will have to set a time interrupt with some time quantum t ms so that every t ms your scheduler will preempt the current running thread. Fortunately, here are two useful library functions you might need: setitimer and sigaction

As you may find in the code and Makefile, your thread library is compiled with STCF as default scheduler. To change scheduling policy when compiling your thread library, passing variables when make:

```
> make SCHED=MLFQ
```

1.4 Benchmark

The code skeleton also includes a benchmark helps you verify your implementation and study the performance of your thread library. There are three programs in the benchmark (parallelCal and vectorMultiply are CPU-bound and externalCal is IO-bound). To run those benchmark programs, **please see README in the Benchmark folder**.

Here is an example of running the Benchmark program with different number of threads:

```
> make
> ./parallelCal 4
```

That create 4 user-level threads to run parallelCal. You could change this parameter to test how thread numbers affect performance.

You could use either pThread or your thread library to run benchmark. To use your thread library to run benchmark, please uncomment the following MACRO in my_pthread_t.h and recompile your thread library and benchmark.

```
#define USE_MY_PTHREAD 1
```

To use pThread again, just comment the MACRO above and then recompile your thread library and benchmark.

1.5 Report

Besides the thread library, you also need to write a report for your work. The report must include the following parts:

1. Detailed logic of how you implemented each API functions and scheduler
2. Benchmark results of your thread library with different configurations of thread number.
3. A short analyses of benchmark result and compare your thread library with pthread library.

2. Suggested Steps

Step 1. Designing important data structures for your thread library. For example, Thread Control Block, and Queue.

Step 2. Finishing my_pthread_create, my_pthread_yield, my_pthread_exit, pthread_join and scheduler function (with the simplest FCFS policy).

Step 3. After you finishing Step.2, then you could add your implementation of a mutex.

Step 4. Extend your scheduler function with preemptive SJF and MLFQ scheduling policy.

3. Tips and Resources

A POSIX thread library tutorial: https://computing.llnl.gov/tutorials/pthreads/ Another POSIX thread library tutorial: http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/5c-pthreads/pthreads-tut2.html Some notes on implementing thread libraries in Linux: http://www.evanjones.ca/software/threading.html

4. Submission

You need to submit your thread library and report in a compressed tar file on Sakai. You could use the following command to archive your thread library folder named code.

```
> tar -zcvf <your_net_id>.tar.gz code
```