User-Level Thread Library and Scheduler

CS416: Operating Systems Design
Rutgers University

Introduction

The goal of this project is to get an idea of how to implement a simple user level threading library as well as get an idea of the complexities of scheduling. Although you will only be implementing a user-level thread library along with a scheduler with a simple Round Robin scheduling policy, you will learn how to use ucontexts to store and load thread execution states as well as figure out on your own on how to properly manage them. You will also have recall what you have previously done in the first project and utilize signal handling in order to support preemptive scheduling and ensure threads don't run for a more than a particular time quantum if there are other threads ready to run as well.

For this assignment you will be able to work in groups of up to 2, and in the end your group will have to submit your completed implementations along with a short writeup.

My Pthread Library Overview

Our my_pthread library will be similar to the traditional POSIX pthread library, except we will be implementing and running it all in user-space.
Provided should be the following my_pthread library template files:

- my_pthread.h
- my_pthread.c

Your task is to implement and support the following my_pthread library functions:

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

The function *my_pthread_create()* is similar to *pthread_create()* except it only takes in three arguments instead of four, leaving out the *attr* argument in the original *pthread_create()* When this function is called, a new thread will be created which will start execution at the start of *function* when it is scheduled. The thread id is also copied into the location pointed to by *thread*. For this project you can ignore the *args* argument.

```
my_pthread_t my_pthread_self();
```

The function *my_pthread_self()* is similar to *pthread_self()*. This function doesn't take arguments and simply returns the thread ID of the calling thread

```
void my_pthread_yield();
```

The function *my_pthread_yield()* is similar to *pthread_yield()* but it returns *void* instead of *int*. This function relinquishes the CPU by invoking the scheduler to context switch to the next runnable thread.

```
void my_pthread_exit();
```

The function *my_pthread_exit()* is similar to *pthread_exit()* but it doesn't take any arguments. This function marks the thread as finished as invokes the scheduler to context switch to the next runnable thread.

```
void my_pthread_join(my_pthread_t thread);
```

The function *my_pthread_join()* is similar to *pthread_join()* except it only takes one argument, the thread id of the thread being joined. This function waits until the thread being joined finishes.

1 Scheduling

Now in order to support these library functions you need to implement a framework on top of which you can run threads. In order to do this, you would need the figure out the following:

1. How to create and swap between thread execution contexts
2. What scheduler state to store
3. Where to implement the scheduler
4. When and how to invoke the scheduler

1.1 Creating and Swapping Thread Execution Contexts

In order to support threading you need a way to manage multiple execution contexts. Luckily the ucontext.h library can help us do just that. With uncontext functions you will be able to save thread execution states, create thread execution states, and swap between them during normal program execution.

There are three main ucontext functions you should be familiar with:

1. *getcontext(ucontext_t *ucp)*
   This function gets the current execution context and saves it into ucontext pointed to by *ucp*
2. *makecontext(ucontext_t *ucp, void (*func)(), int argc, ...)*
   This function makes a execution context that will start to execute the function pointed to by *func* when swapped to.
3. *swapcontext(ucontext_t *oucp, const ucontext_t *ucp)*
   This function saves the current execution context into the context pointed to by *oucp* and starts resumes the execution context of the ucontext pointed to by *ucp*

Note: *Once you create a new ucontext using makecontext(), you will need to allocate memory to serve as the new execution context's stack. Use at least 32768 bytes for a new stack.*

Here's some helpful references on how to use the ucontext functions:

- **Setcontext Wiki**
  https://en.wikipedia.org/wiki/Setcontext
- **makecontext man page**
  http://man7.org/linux/man-pages/man3/makecontext.3.html

1.2 Scheduler State

Now that you know a way to create different threads of execution and swap between them, you need to determine some state to form the basis of your scheduler. Within *my_pthread.h* template you will see the following:

```
/* Thread ID */
typedef unsigned int my_pthread_t; // Thread ID

/* Thread States */
typedef enum threadStatus {
      RUNNABLE,   // Thread can be run
      SLEEP,      // Thread is currently asleep
      FINISHED    // Thread has finished execution
} status_t;

/*  Thread Control Block */
typedef struct threadControlBlock{
      my_pthread_t tid;     // Thread ID
      status_t  status;     // Thread Status
      ucontext_t context;   // Ucontext
} my_pthread_tcb;
```

These structures will be the basis of your scheduler state. You will need to to declare static variables within *my_pthread.c* or *my_pthread.h* to hold scheduler state as you see fit. For example to keep track of multiple threads you will have to maintain some sort of list consisting of the multiple my_pthread_tcb structures of multiple threads. You may also want to keep track of what is the currently executing thread as well as a monotonically increasing unsigned int to assign thread IDs to newly created threads. Feel free to add to any of the structures if you feel like you need to.

Hint: *Think about using an array instead of a linked list so you can easily index TCBs with thread IDs. This could help in situations such as my_pthread_join()*

1.3 Scheduler Logic

Now you need to implement the scheduler. If you notice, within *my_pthread.c* there will be the following function template,

```
void schedule(int signum){

     /* Implement Here */

}
```

In this function template you will implement your scheduler logic. For the sake of simplicity you should implement a simple Round Robin scheduler. Basically the scheduler should simply do the following:

1. Find the next thread execution context to switch to.
2. Swap the current thread execution context to the next one.

1.4 Scheduler Activation

Last but not least, you need to figure out **when** to run the scheduler. If you haven't noticed, you'll see that the *schedule()* function declartion is not in the *my_pthread.h* header file, meaning this can not be called by user applications that include *my_pthread.h*. The scheduler function, in this case can only be called by other *my_pthread_** functions or by other means. If we have multiple threads ready to be scheduled, then we need something to activate the scheduler to start sharing execution time between each thread. When we are running multiple threads, each thread should run for some short period of time or *quantum*. When that time quantum has passed, it's time for the next thread to run. However a thread also may choose to give up the cpu when it either calls *my_pthread_yield()* or *my_pthread_exit()*. So ultimately the scheduler should run when a thread has (1) used up its time quantum or (2) gives up the CPU with *my_pthread_yield()* or *my_pthread_exit()*.

We can do this two ways:

1. By setting a timer interrupt, registering the scheduler as a signal handler and having the resulting interrupt invoke the scheduler. You'll want to set a timer interrupt using *setitimer()*
2. Call as a regular function if thread decides to call my_pthread_yield() or my_pthread_exit()

When setting up the timer interrupt, set the timer to the value specified by *TIME_QUANTUM_MS* within *my_pthread.h*. This value represents is how many microseconds each thread should be able to run for at a time. It is set to 500000, which equates to a time quantum of half a second which is slow enough to allow you to observe the correctness of you scheduler with the provided test programs.

Another Important Note: *If the scheduler is invoked explicitly, some things may go bad if the scheduler is triggered by a timer interrupt, causing the scheduler to preempt a currently running scheduler. You might want to mitigate this from happening by ignoring signal interrupts from the start of the scheduler and reregistering the scheduler as a signal handler before the scheduler swaps to the next thread context that is going to run.*

Here's some helpful references on how to use the *setitimer()* functions:

- **Setitimer man page**
  http://man7.org/linux/man-pages/man2/setitimer.2.html
- **Setting a timer with signal handler**
  http://man7.org/linux/man-pages/man2/setitimer.2.html

2 Testing your implementation

Provided should be 5 basic programs, each using the my_pthread library to an extent. Use these programs to test your pthread library and observe whether or not your scheduler works.

2.1 ThreadRun.c

This program will simply create a thread using *my_pthread_create()* that will print its tid forever within a while(1) loop. After the main thread created the thread, itself will print out within a while(1) loop. If properly done, you should see the created thread and the main thread switch printing out statements every time quantum. Since this program will loop forever, in order to terminate the program use the key combo ctrl+c.

Example Run

Command Line

```
$ make clean
$ make ThreadRun
$ ./ThreadRun
Main Thread Running
Main Thread Running
Main Thread Running
Thread 1 Running
Thread 1 Running
Thread 1 Running
Main Thread Running
Main Thread Running
Main Thread Running
Thread 1 Running
Thread 1 Running
Thread 1 Running
.....
```

2.2 ThreadJoin.c

This program will simply create a thread using *my_pthread_create()* that does busy work within a for loop. The main thread will simply join the thread that was just created. If properly done, you should see only the created thread running and printing for a while, until it finishes. When it finishes the main thread would continue to execute and until it finishes.

Example Run

Command Line

```
$ make clean
$ make ThreadJoin
$ ./ThreadJoin
Thread 1 Running, Counter at 5
Thread 1 Running, Counter at 4
Thread 1 Running, Counter at 3
Thread 1 Running, Counter at 2
Thread 1 Running, Counter at 1
Thread 1 Finished
Main Thread Resuming
Main Thread Exiting
```

2.3 MultiThreadRun.c

This program is similar to ThreadRun.c, except that there will be multiple threads created. Like before, each thread should be printing out for periods equal to around a time quantum. Again, this program will not terminate, so use the key combination ctrl+c to terminate the program.

Example Run

Command Line

```
$ make clean
$ make MultiThreadRun
$ ./MultiThreadRun
Main Thread Running
Main Thread Running
Thread 1 Running
Thread 1 Running
Thread 2 Running
Thread 2 Running
Main Thread Running
Main Thread Running
........
```

2.4 MultiJoinThread.c

This program is similar to ThreadJoin except it creates multiple threads instead of just a single thread. The main thread then joins each thread in a for loop. The amount of work needing to be done should be more than a time quantum so you should see the threads execute in round robin until each of them finish. The main thread would then continue to execute until it finishes.

Example Run

Command Line

```
$ make clean
$ make MultiJoinThread
$ ./MultiJoinThread
Thread 1 Running
Thread 1 Running
Thread 2 Running
Thread 2 Running
Thread 3 Running
Thread 3 Running
Thread 1 Running
Thread 1 Running
Thread 1 Finished
Thread 2 Running
Thread 2 Running
Thread 2 Finished
Thread 3 Running
Thread 3 Running
Thread 3 Finished
Main Thread Resuming
Main Thread Exiting
```

2.5 MultiThreadYield.c

This program is similar MultiThreadJoin except within each created thread execution, the thread will yield the cpu with *my_pthread_yield()* after a single print statement. In this case. You should see the thread a single print statement in a round robin fashion. After each thread prints out 20 times, they call *my_pthread_exit()*. After all created threads exit, the main thread should continue to execute until termination.

Example Run

Command Line

```
$ make clean
$ make MultiThreadYield
$ ./MultiThreadYield
Main Thread Running
Thread 1 Running
Thread 2 Running
Thread 3 Running
Main Thread Running
Thread 1 Running
Thread 2 Running
Thread 3 Running
Main Thread Running
Thread 1 Finished
Thread 2 Finished
Thread 3 Finished
Main Thread Resuming
Main Thread Exiting
```

Other Important Things to Note:

When completing your assignment make sure to keep the following things in mind:

- **Make sure your programs can compile and run on the iLab machines**
  We will be grading your assignments on the iLab machines, so if your programs are unable to compile or run on the iLab machines, points will be deducted. No exceptions.
- **Make sure you can compile you programs with *make***
  We will be using the Makefile to compile your programs. If for some reason we can not compile your program with *make*, points will be deducted. No exceptions.

Submissions

To submit your assignment, simply submit the following files as is, directly to sakai as is. (**Do not compress the files**):

1. **my_pthread.h**
2. **my_pthread.c**
3. **A 1 page pdf that includes the names and NetIDs of all members within the group as well as answers to the following questions:**
   (a) What kind of data structures did you use to keep track of different thread threads/TCBs? A linked list? A linear array?
   (b) What was the most challenging part of implementing the user-level thread library/scheduler?
   (c) Is there any part of your implementation where you thought you can do better?

Additional Tips and Tricks

When in doubt, Print it out: If you have an issue or your program is throwing an error that you don't know how to fix, put some print statements to see where the program may not be executing as you intended. Just be sure to remove the print statements before you submit your project.

When in more doubt, Google: If you have an issue or your program is throwing an error that you don't know how to fix, Google It. Someone, somewhere, probably faced the same issue at some point.

Frequently Asked Questions

- **Can I use helper functions in my implementations?** Yes, as long as the scheduler works as intended without extra modifications to test programs.
- **Can I use other *.c or *.h files in my implementation?** For the most part you shouldn't have to separate functionality into other files, but as long as the main files remain intact and as long as we will use the makefile to compile your programs before running.
- **Can I modify the makefile?** Yes as long as we can compile still compile the test programs via make
- **Can we implement the programs on my own computer?** Yes as long as it can compile and run on the iLab machines as that is where they will be compiled, ran, and graded.

Additional Questions

If you have any questions about the assignment or are having any issues, email me at David.Domingo@rutgers.edu