Overview

In this assignment you will implement the Linux pthread library, which will require that you also write a scheduler to manaully switch between your various threads. Since you will be dealing with multiple threads, you will also need to implement pthread mutexes, mutual exclusion devices that keep a thread locked if it is waiting for a particular mutex. This assignment is intended to illustrate the mechanics and difficulties of scheduling taks within an operating system.

You will need to do a fair amount of investigation and reading in order to accomplish this assignment. If you are unfamiliar or uncomfortable with threading and synchronization in C, it will take longer.

This assignment is a group project. Your group may consist of up to 3 students. You should inform a course TA of your group members as soon as possible.

Specifics of Operation

Build a library named "my_pthread_t.h" that contains implementations of:

Pthread note: Your internal implementation of pthreads should have a running and waiting queue. Pthreads that are waiting for a mutex should be moved to the waiting queue. Threads that can be scheduled to run should be in the running queue.

int my_pthread_create( pthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg);
Creates a pthread that executes function. Attributes are ignored.

void my_pthread_yield();
Explicit call to the my_pthread_t scheduler requesting that the current context be swapped out and another be scheduled.

void pthread_exit(void *value_ptr);
Explicit call to the my_pthread_t 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(pthread_t thread, void **value_ptr);
Call to the my_pthread_t library ensuring that the calling thread will not execute until the one it references exits. If value_ptr is not null, the return value of the exiting thread will be passed back.

Mutex note: Both the unlock and lock functions should be very fast. If there are any threads that are meant to compete for these functions, my_pthread_yield should be called immediately after running the function in question. Relying on the internal timing will make the function run slower than using yield.

int my_pthread_mutex_init(my_pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
Initializes a my_pthread_mutex_t created by the calling thread. Attributes are 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.

You should also investigate the system library ucontext.h. It has a series of commands to make, swap and get the currently running context. When a context is running, it will continue running until it completes. In order to interrupt the current context you should set an interrupt time (setitimer) in 50ms quanta. You should implement and register a signal handler to run any time the interrupt fires. In your signal handler you should schedule and swap to the next context. You can freely modify context and runtime data in your signal handler since, if you are running the signal handler, the signal must have fired, pausing the current context.

Your scheduler should implement a multilevel priority queue, granting highest priority to new threads. When a thread runs for its entire time slice without terminating, its priority should be decreased. As threads decrease in priority they should run less often, but for longer. One of the things your scheduler should get the current system time and check if the current context has run for its entire time slice. If it has run its entire time slice, it should be swapped out and its priority decreased. If it has not yet run its entire time slice, no change should be made and the current context should resume computing. If a thread explicitly yields, a new thread should be scheduled without decreasing the priority of the previous thread.

You should establish a maintenance cycle. After some amount of time, when your scheduler runs, you should scan through all threads in your queue and determine how long they have been running. The oldest threads should have their priorities scaled up based on how long they have been running. This is so that the oldest threads do not get starved out of run time. The number of different priority values, how often you update them, how long the time slices are, how large the time slices are per priority value, and other details are entirely yours to specify. You should write a suite of benchmarking functions to test your scheduler with to determine how well your scheduler works. The details of your scheduler and your testing results should be written up in a readme.pdf

Extra (25pts)
Add instrumentation to your scheduler to detect and intelligently react to priority inversion. You should detail your detection method(s) and remedies in your readme.

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

Submission

Submit your code through Sakai before the due date.

Your code must operate on one of the ilab machines.

You should additionally create a readme.pdf that details how your

Submit a tarred, gzipped file named Asst1.tgz that contains:

- my_pthread_t.h
- my_pthread.c
- readme.pdf

Note your name and the ilab machine you wrote and tested your code on as comment at the top of the file.