CS 341 Operating Systems, Spring 2018
Programming Assignment 4: User-Level Thread Implementation

1 Introduction

This assignment requires the design and implementation of a user-level thread scheduler. This gives you an opportunity to implement a preemptive thread scheduler. You will gain insight into how preemptive schedulers work for multithreaded code and by extension, how to Linux kernel preemptively schedules the execution of multiple processes on a computer.

You will be implementing code for a thread scheduling library, similar to the POSIX pthread library. You will also write one or more driver programs that demonstrate the complete and correct functioning of your thread scheduling code.

2 Thread Scheduling

Each thread had an *execution context*, which is the execution state of the specific thread. The execution state includes the value of the registers and instruction pointer for the thread as well as the stack information for the thread. Using the Linux operations getcontext(), makecontext(), swapcontext() and setcontext(), it is possible to capture the context of a thread, save it and later restart the thread by using the previously captured thread context. The restarted thread resumes execution right where it left off.

The thread scheduler runs as part of user-level thread execution. A *preemptive scheduler* can interrupt an executing thread, save the execution context for a thread and (re)start the execution of another thread by transferring execution to another thread context. The preemptive scheduler uses a repeating timer and attempts to switch context to another thread on each timeout.

A *Round-robin* scheduling discipline is a simple thread-scheduling algorithm. The runnable thread queue is a circular queue. Each thread is given a time slice, which is the maximum time the thread can execute. The scheduler selects a thread to execute and (re)starts execution for that thread. At the end of the time slice, the scheduler places the context for the most recently executed thread is placed on the back of the queue and the next thread to execute is taken from the front of the circular queue.

3 User-level Thread Operations

You will have to implement the following user-level thread functions:

- `thread_create( thread_t *, void (*func)( void *), void * );` This function creates a new thread to execute.

- `thread_yield( void * );` This function ends the time slice for the calling thread. Process execution should continue with another thread. If no other thread exists, then execution should continue with the same thread.
- `thread_exit( void * );` This function exits the calling thread. Process execution should continue with another thread. If there is no other thread to execute, the process should exit.

4 Starting Now

This is my admonition to start working on this immediately. It is also an admonition not to join a group and let someone else do all the work. You will be asked what you contributed to the overall project as part of your overall grade.

5 Timers and Timeouts

The UNIX/Linux timer is a fairly primitive mechanism. One sets a timer to go off some number of seconds and nanoseconds in the future. Your process continues execution after setting the timer. After the specified span has elapsed (a "timeout"), your process gets a SIGALRM signal. In a multithreaded program, some thread should be executing at all times. Library calls that just make the process sleep will not do the job, since they shut down the entire process and no thread can execute. I recommend the use of setitimer() as a solution.

6 Extra Credit

This assignment is open-ended, meaning that the more you do, the more points you get. You can get extra points by implementing thread joining, mutex locks, condition variables.

7 Program Termination

Your driver program(s) and thread scheduling code can be augmented with printf() and/or write() calls that allow the code to show what it's doing as it executes.

8 Completion of Assignment

You are allowed to work in groups of up to four people. Group members are free to collaborate among themselves and produce a single body of code to hand in. Your group may collaborate with other groups, but you are not allowed to share code with other groups.

At the end of the assignment, your grade will be determined by a presentation where you will demonstrate your working code, any extra credit features and answer questions about how your code does what it does.

9 What to turn in

A tarred gzipped file named `pa4.tar` that contains a directory called pa4 with the following files in it:

- A `readme.pdf` file documenting your design **paying particular attention to the thread synchronization requirements of your application**.
- A file called scheduler-testcases.txt that contains a thorough set of test cases for your code, including inputs and expected outputs.
- All source code including both implementation (.c) and interface(.h) files.
- A test plan documented in `testplan.txt` and code to exercise the test cases in your code.
- A makefile for producing the executable program files.

Your grade will be based on:

- Correctness (how well your code is working, including avoidance of deadlocks.
- Efficiency (avoidance of recomputation of the same results).
- Good design (how well written your design document and code are, including modularity and comments).