1 Overview

A thread library provides an API for creating and managing threads. Support for threads is provided either at user-level or by the kernel, depending on the goals of the end-product. Kernel level threads are managed directly by the operating system, where each thread is viewed as an independent task, managed via system calls from user space, scheduled by the kernel, and good for applications that frequently block. User level threads, on the other hand, are managed without kernel support, are defined by the user level thread library, can be used on systems that have no kernel-level thread support, thread switching is as efficient as function calls, but the kernel knows nothing about them so any blocking affects all threads.

2 Assignment

You will implement a simplified version of many-to-one user level threads. (Credit for the assignment's inspiration is due to the Operating Systems Course at Uppsala University.*)

In the many-to-one model for user level threads, all threads execute on the same kernel thread. As there is only one kernel-level thread associated with the process (the process containing threads is represented by a single context within the kernel), only one user-level thread may run at a time.

Your thread manager will include a preemptive round-robin scheduler. If a thread does not yield during its time-slice, it will be preempted and one of the other ready threads will be resumed. The preempted and resumed threads should change state accordingly.

Important: You will NOT use pthreads anywhere in your implementation. The intent is to create similar functionality entirely in a user-space program.

You will also implement the lottery scheduler.

How you test your schedulers is up to you, but they ought to show correct and concurrent execution of several threads. Ensure that your threads do something that will allow you to see your schedulers in action, i.e. make sure to make your threads busy for long enough to see preemption and rescheduling at a later time. In your design document, describe the tests you devised to convince yourself that your schedulers work.

Required: Choose one of Waldspurger's experiments and reconstruct it using your threads. Include a graph that shows your threads' performance, as Waldspurger did. Compare the outcome with the round-robin scheduler.

*https://www2.it.uu.se/education/course/homepage/os/vt18/module-4/simple-threads/

Optional: For extra credit, devise a new experiment demonstrating some aspect of the lottery scheduler. Again, compare the lottery scheduler's performance with the round-robin scheduler.

2.1 Preliminaries

To complete this assignment, there are two necessary concepts you must master: managing execution contexts and signal handlers. Examples are given for both for you to examine and adapt. Get the files from */pub/cs514/p0* on the OS server.

2.1.1 Execution Contexts

The example code that is the essential starting point for grasping execution context management is in the context.c source file. To fully understand the code, play with it while reading the following manual pages (or ask ChatGPT for explanations and examples):

- getcontext
- setcontext
- makecontext
- swapcontext

2.1.2 Timers

In timer.c you will find an example of how to register a signal handler and set a timer repeatedly (as a scheduler might do in order to call itself when a thread's time expires). See chapter 10 of Advanced Programming in the UNIX Environment for an authoritative treatment of signals.

In order to implement preemptive scheduling, you will need to set a timer, and register a timer handler that will act as the thread scheduler for your library. You will figure out how to suspend and resume your threads.

3 Grading

Hard requirements (no points without these):

- Assignment is to be done in C, without the use of pthreads, and is to compile and run on os.cs.siue.edu.

- In your design file, describe what you did and how it convinces you that your solution works.
- Place all of your thread code in uthreads.h and uthreads.c. Include a Makefile that compiles your files into an executable program.
- If you use ChatGPT to produce examples and synthesize parts of the code, you must include all and complete transcripts in your design file.

Rubric

- 10 points if there is a reasonable attempt at a solution conforming to the original problem statement.
- 10 points if a single thread is initialized and runs
- 10 points if more than one thread successfully runs by yielding
- 20 points if many threads are being scheduled preemptively, the round-robin scheduler works to select each thread in turn, and all threads run to completion.
- 30 points if many threads are being scheduled preemptively, the lottery scheduler works to select each thread, and all threads run to completion.
- 20 points if you reconstruct one of Waldspurger's experiments that demonstrates some process behaviors using the lottery scheduler and compare it to the round-robin scheduler.
- (optional) 20 points if you construct a novel experiment that demonstrates some feature of the lottery scheduler and compare its performance to the round-robin scheduler.

4 What to Turn In

A .tgz of your solution files. Design and usage notes should be included in a design.txt, also to be turned in via the .tgz archive. Design.txt should clearly state how and what you've done to convince yourself that your code works. Include a Makefile that builds your solution on os.cs.siue.edu. Use of ChatGPT is not prohibited, nor is it required, but if used, include any and all transcripts in their entirety (include all prompts and responses) in your design.txt.