CS 341 Operating Systems, Spring 2018
Programming Assignment 2: Real-Time Event Scheduler

1 Introduction

This assignment requires the design and implementation of a multithreaded event scheduler. This will give you an opportunity to write a multithreaded program that reads in and stores event requests in any order and then executes the requests in order in real time. You will gain experience in writing a multithreaded program that uses timers, signals, signal handlers and semaphores to manage the use of a shared data structure.

Your program will have two threads. The first thread will read event requests from stdin. An event request will consist of a time span and an event string. Tie spans are expressed in a non-negative number of seconds. The incoming event requests may be entered in any order. The first thread will reject any event request with a negative time span and add any future request to a *MinHeap* shared with the second thread.

The second thread will respond to timer events ("timeouts"). The second thread will block until a timeout and then respond to the timeout by updating *now* from the current time with clock_gettime(2). It will then extract and perform all events prior to the updated *now*.

2 Event Requests

An event request has two parts: A time span and the request string. The time span is a non-negative integer number of seconds untl the event is to be performed.

The request string is just a simple string. The string contents take the place of an encoded request that might be arbitrarily complicated. As far as this program is concerned, "performing a request" is nothing more than printing out the request string and the associated time. In a more realistic implementation, performing the request would involve various forms of application-specific activities.

3 The MinHeap

The data structure of choice for string and retrieving ordered event requests is the MinHeap. The MinHeap allows you to add data elements in any order and later extract them in increasing order. The lowest element in the minheap is always at the top of the heap structure, and thus readily and inexpensively available.

The MinHeap data structure will be extensible and be able to hold any number of event requests, ordered by absolute time.

4 Starting Now

This is not just an admonition to start working on the program immediately, it is also something your program needs to do its job.

The first thread reads incoming event requests and rejects any request whose time span is less than (i.e. earlier than) zero. The scheduler cannot perform actions in the past. Extra credit if you can come up with a working time-travel mechanism to support performing actions before they are scheduled. When the timer expires, the program will have to update its idea of the current time.

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. Note that timer delays are expressed in terms of a relative time span rather than absolute times. Your program will calculate a a future absolute time as current time plus relative time span.

Both the first and second threads have to reset the timer to the nearest scheduled future event. The first thread resets the timer when it adds a new event request to the MinHeap. Keep in mind that an event request scheduled to happen at an earlier absolute time may have been read after the earlier entry of an event request scheduled to happen at a later absolute time. The second thread will have to reset the timer after the extraction of some event requests. There may be more events in the MinHeap scheduled to happen in the future.

6 Sanity

Any test data you submit and any test data we use to grade your program should encompass a fairly small span of time, not more than a minute or two.

Your program should deal with multiple event requests scheduled for the same time. Multiple event requests scheduled for the same time can be executed in any order.

7 Extra Credit

We've greatly oversimplified the execution of a request, merely printing out the descriptive string. Let's assume that the actual execution of a request takes some time. Input to your program would now be [time span, duration, event string]. The duration, expressed in seconds, specifies how long it would take to "execute" the request. If the second thread pretended to do some action (faked by sleep()ing) for the duration while holding a mutex, the first thread would get blocked from adding new event requests. We could solve this by having the second thread spawn a *worker thread* to do time-expensive requests. The worker thread would pretend to do some action, again by sleep()ing, leaving the original second thread free to continue to extract and execute requests.

8 Deadlocks and Race Conditions

There should be NO DEADLOCKS and NO RACE CONDITIONS in your code.

9 Program Termination

Your program should shut down gracefully after all requests have been performed.

10 What to turn in

A tarred gzipped file named `pa1.tar` that contains a directory called pa1 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).