518 - Assignment 1

Bharti Mehta, Victor Zhang

March 22, 2021

1 Introduction

The purpose of this assignment was to create a scheduler to expose `pthread`-like functionality via user threads. Our solution assumes that we are given one kernel thread and unlimited space within which to develop scheduling and multithreading. If the user is allowed to mix kernel and user threads, scheduling would become intractably difficult, since we have no control over how or when kernel threads run.

2 Library Operation

2.1 Timing and Scheduler Concurrency

The core scheduler relies on signals sent by an alarm `timer` to run scheduling events. `SIGALRM` invokes `schedule()`, which swaps threads and will be discussed later in this section. Immediately, interrupt signals introduce thorny concurrency issues. What happens if `schedule` is called while we're in the middle of another scheduling decision? Luckily, the signal handler blocks the same signal while executing, so we need not worry about interrupting while swapping. Further, since we assume a single-threaded process, we may "enter" the scheduler by disarming the timer and "exit" the scheduler by re-arming the timer. This gives us exclusive control of the process while scheduler-specific queues and other data are being accessed. In particular, `pthread` functions that require sequential access to important data structures call `enter_scheduler()` like a user program would acquire a lock.

The `schedule()` function handles swapping and is called by `pthread` function to finish execution. It is responsible for swapping out the currently running thread (front of the currently running queue), aging, and decay. Finally, it invokes `swapcontext` to run a new context. The particulars of swapping will be discussed in a later section.

2.2 Priority, Aging, and Decay

Priority values can be assigned from `0` to `4`, inclusive. Lower numbers indicate higher priority. Each thread is allocated runtime based on priority:

$$\texttt{runtime} = \texttt{priority} + 1$$

For instance, a thread with priority `3` will run for 4 continuous scheduling cycles.

Every thread starts with priority `0`. Whenever `schedule` is called within a thread's context, it ages according to the following algorithm:

- If `schedule` is called via `yield`, the priority does not change.
- If the thread has run for its entire time slice, the priority is incremented by 1, up to `MAX`.

A maintenance cycle is requested every second. We scan through the queues and reassign priority based formula

$$\texttt{new\_prio} = \left\lfloor (\texttt{old\_prio} + 1) \cdot e^{-x/g(\texttt{old\_prio}+1)} \right\rfloor$$

$$g(y) = \frac{\texttt{CYCLES\_SINCE\_LAST}}{\ln y} - 1$$

`old_prio` $\in [1, \texttt{MAX}]$ is the old priority of the thread. It makes no sense to decay threads with priority `0` so we don't bother. $x \in [1, \infty)$ is the number of scheduling cycles since the thread was last run. `CYCLES_SINCE_LAST` is the number of scheduling cycles since the last maintenance cycle. This is an efficient way of ensuring older threads are not starved out of resources. Decay is faster for the lowest priority threads than higher priority threads. The parameters are set up so that threads which have run recently are not decayed, regardless of priority, but every thread is guaranteed to enjoy the highest priority if it has not been run since the last maintenance cycle.

2.3 Swapping

We maintain an array of linkedlist queues and a hashmap. The hashmap is indexed by thread `id` and contains information about all threads created in the process, alive or dead. We implement multilevel priority queuing with the queue array. Upon thread creation, we generate a `tcb` for it containing a unique thread `id`, context, and metadata for priority and joining threads. We put the `tcb` onto the back of the highest priority queue and into the hashmap. We insert new threads onto the back of the queue to prevent starvation. If we were to insert to the front, an adversary could starve old threads by continually creating new threads that run for only one scheduling cycle.

When the scheduler is invoked, it first checks the priority of the currently running thread. If the thread has run for fewer contiguous cycles than its priority specifies, the scheduler exits, allowing the thread to run for longer. If instead the thread indicates it intends to `yield`, we move to the next step. The currently running thread is placed at the back of the appropriate queue determined by the aging algorithm. If a maintenance cycle is deemed necessary, it runs at this point. Finally, we pick the highest priority ready thread to run.

2.4 Priority Inversion

In order to determine if a lower priority thread has acquired a lock while there exist higher priority threads waiting on the lock creating a priority inversion, we included a variable called `hoisted_priority` for each lock. This variable is initially set to be an impossibly low priority, and is updated to match the priority of any thread which calls `my_pthread_mutex_lock()` and has a higher priority than `hoisted_priority`. In this way, we store the highest thread priority amongst those threads either waiting on or holding the lock. If a thread acquires the lock or is already holding it, we check if it has a lower priority than the `hoisted_priority`. If it does, we "hoist" the thread to the priority of the highest priority blocked thread, the `hoisted_priority`. In this way, the owner of the lock will run before any other threads waiting on that lock, therefore avoiding a priority inversion.

If a thread which acquires the lock needs no hoisting, in other words when an acquiring thread's priority is the same as `hoisted_priority`, we know there are no blocked threads with a higher priority than the holding thread and so there cannot be a priority inversion at the moment. As a result, we reset `hoisted_priority` to the impossibly low priority to ensure accurate priority inversion detection hereafter.

3 Testing and Analysis

3.1 Testing

To test our thread library against the POSIX thread library, we measured the CPU time it took for each library to run the same program. For each library, the context spawning the threads called `clock()` right before creating any attributes, locks, and threads, and right after joining with spawned threads and destroying any locks or attributes. In this way, we obtained the number of clock ticks at the end and start of the run, which when subtracted and divided by `CLOCKS_PER_SEC` resulted in the elapsed CPU time in seconds for each run. Additionally, by `join()`ing each thread, the context spawning the threads only "stopped its timer" once all other threads were done ensuring an accurate time measurement. In testing the two libraries, we made sure to test them using the same function and same parameters.

3.2 Parameters

Before the start of each timed run, certain parameters were set to ensure scalability. Every combination of the following parameters was executed on each library 10 times to obtain an average time.

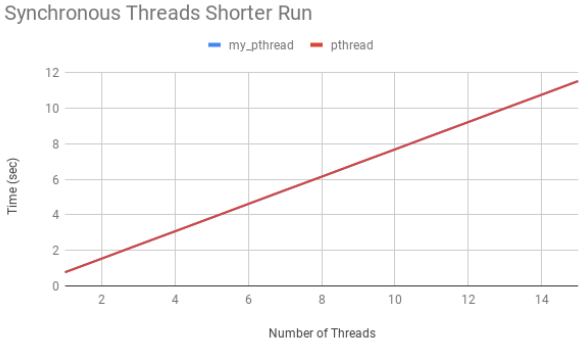
1. Number of threads: this ranged from 1 to 15.
2. Duration of threads: each thread ran a function called `thread_func`, which contained a `while` which looped either 500 million times, `shorter loop` or 1 billion times, `longer loop`.
3. Asynchronous or Synchronous: before running the while loop, we either required the threads to acquire a lock or not.

If the parameters were set to `5 threads`, `shorter loop`, `asynchronous`, then 5 threads would be spawned all running the shorter loop asynchronously. Likewise, if the parameters were set to `5 threads`, `longer loop`, `synchronous`, then all 5 threads would run the longer loop after obtaining the lock. In this way, we ran $15 * 2 * 2 = 60$ combinations. Each library ran each of the 60 combinations 10 times to calculate an average time.

3.3 Results

(a) Compares the time elapsed for each thread library to asynchronously run 1 to 15 threads, where each thread ran a shorter loop.

(b) Compares the time elapsed for each thread library to asynchronously run 1 to 15 threads, where each thread ran a longer loop.



(c) Compares the time elapsed for each thread library to synchronously run 1 to 15 threads, where each thread ran a shorter loop.

(d) Compares the time elapsed for each thread library to synchronously run 1 to 15 threads, where each thread ran a longer loop.

Comparing the libraries when threads were run asynchronously (1b and 1a), we see that when the number of threads spawned is smaller the two libraries take the same amount of time, however as the number of threads increases, the POSIX library takes more time than our library. This trend is the same whether

we ran the threads for a shorter duration or a longer duration. On the other hand, the libraries are quite comparable in terms of efficiency with synchronous thread execution (1c and 1d). This is true for both threads that ran for a longer duration or a shorter duration.

4 Further Work

In terms of further work, we could further optimize the scheduling mechanism. For example, each thread runs for 25 ms or a multiple of it before being swapped. We could play around with this value in an effort to ensure we find a sweet spot between first come first serve scheduling system and high overhead of too many context switches. Additionally, we decided to schedule a maintenance every second, however we might have been able to wait longer between maintaining our priority queue without starving any long running processes. By playing with how often we schedule maintenances, we could find a better balance of reducing the time spent maintaining the priority queue without starving long running threads. Likewise, other scheduling parameters such as number of queues, how much to decay and age a queue by could all be further fine tuned.