Lance Fletcher (laf224)
Christopher Gong (cg646)
Jeremy Banks (jpb231)

- Operating Systems [416]
- Project1

My_Pthread_T: Adventures in Scheduling

Scheduler:

For our code, we implemented a multi-level priority queue with a Round Robin algorithm. This is done by creating an array (with 15 buckets) of Thread Control Block queues (which we created as a circular linked list), making up our "running queue." Each context is given a time slice of 20 ms, and if the context has not finished, it will drop a priority level based on upon its current level. As a result, the total time a context runs will be determined by the time slice multiplied by the current priority level it is located. When selecting the next thread to run, we simply grab whichever is the lowest at that time. Maintenance is performed upon the completion of 10,000 ms total. At this time, all Thread Control Blocks that are not in priority level 0 get boosted to the end of priority level 0's queue, and the loop continues. We use this methodology because it not only prevents starvation and avoids the traps that come with having an algorithm that is strictly FIFO, but perhaps moreso because it's balanced. The time slice is small enough so that no context is running for too long over others, but also large enough in that it limits the amount of overhead the act of context switching takes in itself.

Threading [my_pthread_create, my_pthread_yield, my_pthread_exit, my_pthread_join]:

For the main threading portion of the code, we take advantage of the ucontext library. For my_pthread_create, we create a new context (along with capturing main, if it hadn't been done yet) and enqueue it into our multi-level running queue so it can be added to the scheduler. For my_pthread_yield, we simply make a call to the scheduler to swap contexts. For my_pthread_exit, we switch contexts to an auxiliary context we make (which we call "garbage_collector") that specifically frees and manages all contexts that exit (deliberately or unexpectedly, making use of ucontext's "link" feature). For my_pthread_join, we insert our thread control block into a global hash table that acts as a "waiting queue", meaning that the thread will be inactive until the thread it is waiting for exits (scheduler is also called).

Mutexes [my_pthread_mutex_init, my_pthread_mutex_lock, my_pthread_mutex_unlock, my_pthread_mutex_destroy]:

Our implementation of a mutex lock comes with a simple set of booleans (in the form of ints) and individual waiting queues (since threads wait on a particular lock, and thus shouldn't be thrown into any kind of all-purpose waiting queue). In my_pthread_mutex_init, we simply initialize all values of a fields of the mutex struct. In my_pthread_mutex_lock, we first check to see if it's available, and if it is, we use the **atomic operation test-and-set** in a loop to check the status of the locked mutex. In this loop, we enqueue each attempting thread to the mutex's waiting queue, always followed by another call to the scheduler which will switch contexts again. In my_pthread_mutex_unlock, we check to see if the thread trying to unlock a given mutex is currently holding the mutex, and if it is, we let it go and dequeue the top (if any) context in that mutex's queue (as well as enqueue it into the running queue, to account for priority

inversion). In my_pthread_mutex_destroy, we first check to see if the given mutex is still locked, and if it is, we make a call to the scheduler to wait for that lock to release.

Error Assumptions:

- IO operations (fscanf, etc), will not be performed
- Any edge case that results in undefined behavior of the normal Pthread library (destroying a locked mutex, etc) will not be attempted
- Instances of Deadlock will follow the same behavior as the normal Pthread library

Extra Credit:

We also accounted for priority inversion within our scheduling implementation. This is done by allowing priority to threads that currently holding a mutex or are trying to be joined with. In the event of a thread holding a mutex, the next thread waiting on it will be enqueued into priority level 0 so that it is not at a disadvantage after potentially waiting for so long. In the event of a thread trying to join with another, the thread that was requested to be joined with will be enqueued into priority level 0, so that the thread now being waited on will potentially get to run on its own sooner.