Design Doc

Our main motivation was to create an alternative scheduler in the OS161 operating system that performs better and is more involved. The current scheduler is a simple round robin scheduler, making use of the queue functionality in the kernel to keep a single list of runnable tasks. The scheduler function simply returns the first item in the queue, and making a thread runnable simply adds it to the queue. In the thread file, when a context switch happens, the scheduler is called to return the next thread to run.

Our implementation aims to make the scheduler faster, while also adding some functionality in the form of thread priorities. We created a multilevel feedback queue (MLFQ) scheduler. It contains three priorities and three corresponding queues: high priority, medium priority, and low priority. It also includes a current queue variable to keep track of which queue the scheduler is currently choosing from. In our code, we initialize the thread priority to 2 (high) on creation, but this can be easily changed in the thread.c file. All of the queues are created and allocated in the same way the singular queue was created in the original scheduler. The scheduler will return the thread removed from the high priority queue first if there are any. If not, it will return from the medium priority queue if there are any and from the low priority queue otherwise.

Changes also had to be made to the mi_switch (machine-independent context switch) function in the thread.c file. If the next state of the thread is S_READY, then the thread was switched out due to using its complete time slice, so we decrement the priority of the thread as long as it is not already 0. If the next state is S_SLEEP, then the thread has given up the CPU before its time is up and its priority stays the same.

Our hardclock was also changed to improve performance. We changed the hardclock so that the threads on the high priority scheduler run for 50 lbolt_counter ticks, the medium priority threads run for 100, and the low priority tasks run for 1000. This is because we want the high

tasks, which most likely have short CPU bursts (since using the full CPU time lowers priority), to not idle and waste their time. We also want the lower priority threads, which do not get run as often and are likely CPU intensive, to get a longer time to run.

We have two implementations of aging. One of them boosts all threads to the highest priority queue every time the scheduler is called. This gave our scheduler good performance, but realistically it is the same as using a round robin scheduler, since all tasks will always be moved to the highest priority. The other implementation keeps track of the number of context switches and only ages every 5 context switches. This number could obviously be greatly increased, as the time slices for high priority threads is very short. This other implementation also only ages 5 medium priority threads and 3 low priority threads. The logic is that it boosts the threads at the front of the medium priority queue and the low priority queue, as these threads have probably not been run for a while. It does not boost all the threads because then every thread would be in the high priority queue.

In the slides, we also present some results that we got from using the built in thread tests. We modified these tests to monitor the response and turnaround time for the threads, instead of just the total completion time of the executable. This allows us to more accurately measure the performance of our scheduler and see how it is better and worse than the original.

One of the challenges we faced was working with the clock. The hardclock file includes important information for our schedule, but we found it hard to monitor the number of context switches and clock cycles. We ended up using a lot of external integers, which are not typically supposed to be used in this way. Overall, our scheduler was able to perform much better than the original scheduler on tt1, and only slightly worse than the original on tt3. We believe that on more complex operating systems or with more complex tests, our scheduler would perform better than the original round robin scheduler.