Dennis McWherter
30 September 2011
CS 241
dmcwhe2

In MP4 I had the task of implementing a CPU scheduler for a simulated CPU. As many know, a priority queue is the very backbone of such a scheduler. I decided to naïvely implement my priority queue with a sorted doubly linked list. Although the running time is less efficient than that of a heap, the amount of time required to implement and debug the priority queue was greatly decreased.

The implementation of my priority queue, however, proved to be very useful in implementing my scheduler. I used it as the supporting structure for my entire scheduler. I had a simple structure (job_t) which held various data about my incoming jobs which was totally general to any scheduling algorithm. From there, I simply wrote different comparison functions based on the algorithm chosen for scheduling. This way, I could keep everything ordered within a single priority queue and keep my solution as general as possible with minimal code duplication. Furthermore, this implementation took care of nearly all ordering for me, the only other special cases I had to worry about were particular preemptive algorithms.

To handle multiple cores, I had a global structure (g_scheduler) which contained various data about my current scheduler information (i.e. cores, what job is running on a particular core, etc.). Using this structure, I was able to check whether a core was idle (and, therefore, immediately schedule a new job to it) or I could check information about the jobs running on a particular core for checks in preemptive algorithms. With this implementation, I was able to handle 1 core and N cores using the same method since I knew at the time of creation how many cores I was scheduling for. If the core is in use (and a new job does not preempt the current job), I simply offer the new job to my priority queue which (based on the comparison function for the scheduling algorithm) appropriately arranges the job within the queue. When a job finishes, it takes the next item from the queue and schedules the new job to be run on the core. Preemption is handled in a similar way, except jobs are appropriately moved (i.e. the running job is added to the queue and the new job is scheduled on the core).