Kernel.h

Change QUANTUM from default 2 to 10

multlevelfq.h

created and added externs to mlfqinsert, mlfqdequeue, and mlfprocqueue

queue.h

changed NQENT from (NPROC+4+NSEM+NSEM) to (NPROC+4+NSEM+NSEM+120)

ts_disptb.h

added ts_disptb.h

create.c

changed prptr->prprio = priority to prptr->prprio = 59

initialize.c

added initTSTable function, added tsdtab declaration and mlfprocqueue, change nullproc priority to 0,

initialize mlfprocqueue in sysinit

multilevelfq.c

added multilevelfq.c, declares mlfqinsert and mlfqdequeue

ready.c

change insert(pid, readylist, prptr->prprio) to insert(pid, mlfprocqueue[prptr->prprio], prptr->prprio)

resched.c

rewrote much of function, checks if IOINTENSIVE or CPUINTENSIVE process, then reassigns its priority and requeues it.

Problem 3.2

My decision for the initial priority of a newly created process is the highest level, or 59. The null process is handled by assigning it an initial priority of 0, the lowest priority, and then when it is dequeued, checking the queue again to ensure that there are no other processes in the queue, if there is insert the null process and dequeue the next process. The null process is never promoted because it is uses the entirety of its time slice to run, so it stays at the lowest level.

Problem 3.3

My data structure for the multilevel feedback queue is an array of qid16 of size DISPTBSIZE, or 60. Each index of the queue represents a different level of priority, where the quantum corresponds to the quantum in tsdtab[priority].

To meet the time complexity requirement of O(1) for mlfqinsert I just enqueue the process into the level of the mlfprocqueue corresponding to the process priority. Enqueue has a complexity of O(1) since it just sets the tail of the queue to the pid passed.

For mlfqdequeue I obtain a complexity of O(c), where c is the number of levels of the priority, by creating a for loop that goes from c-1 to 0, total c times, and checking if the mlfqueue is empty at each level, if it isn't then dequeue the process at that level since it is the highest priority process at the time.

Problem 4

When running cpuintensive(), all process used the same amount of cpu time(20 milliseconds) in the order which they were resumed, this shows the fairness for cpu intensive processes. For iointensive, all processes used 13-14 milliseconds of cpu time, and were completed a little out of order. Similar to lab2 the processes did not stay in their initial order, due to the prcpuused not being set precisely on the beginning/end of a millisecond.

In my initial testing of the mix between the two, the cpuintensive and iointensive process were both fair to themselves, taking up 20miliiseconds each for the cpuintensive, and 13 and 14 milliseconds for the io intensive. They were completed with cpu processes first and io processes last. This is because the cpu intensive processes are resumed first and the main process is a cpu intensive process and gets demoted in priority. It always has a lower priority compared to the first 2, so is the io intensive process are not resumed until after the cpu intensive processes are completed. When switching the resume order so that the io intensive processes are resumed first, yields something different.

With io intensive resumed first, io intensive use 13-14 milliseconds again, and cpu intensive use 25-26 milliseconds of cpu time, thus both are fair compared to themselves. Though the cpu intensive used more cpu time, they completed before the io intensive processes. However, the io intensive processes had a higher priority so that when they received input they would be given priority of the cpu intensive process, but while waiting the cpu intensive processes were given larger quantums to spend more time running while the io intensive were waiting.

Compared to the dynamic scheduler from lab 2, the time share scheduler is "better" in that cpu intensive processes are given larger time slices so less time is spent switching between them, and io

intensive process are actually given priority over cpu intensive so that when they receive input they can execute. Whereas with dynamic scheduler, all process are given the same time slice so a lot of time is spent switching between processes.