Eric Grounds
C435
2-26-13
Homework 2 – Priority based user process scheduling

Running and Testing

Once logged into Minix, there is a folder in /usr/src called hw2test. Inside that folder are two precompiled C programs called cpu and io. The source for each is also in that folder for reading or modification purposes. cpu is a CPU-bound program that contains a nested for loop that multiplies two numbers together for a large amount of iterations. The purpose of this program is to waste CPU time On the other hand, io is an IO-bound program that writes clock ticks to a file. The second component needed for testing the scheduling algorithm is the use of the F9 key, which dumps the scheduling queues. This routine was modified to display more information. The information it displays is the queue number (Q_nr), queue position (Q_pos), process number (P_nr), process name (P_name), process priority (P_pri), and process recent cpu time (P_rec_cpu). The two C programs can be run in the background from the terminal.

For example:

```
  # ./cpu & ./io & ./cpu & ./cpu & ./io &
```

Can be entered at the terminal, once the processes are running, the F9 key can be pressed one or more times to see the scheduling queue information.

Several test runs are shown at the end of this write up.

Changes/Observations/Assumptions/Correctness

The goal of this assignment was to modify the scheduler to keep track of how much CPU time each user process has had recently. When no task or server wants to run, pick the user process that has had the smallest share of the CPU. So, to start, the proc.h file located in /usr/src/kernel was modified.

Eric Grounds
C435
2-26-13
Homework 2 – Priority based user process scheduling

The modification was to add an additional clock_t variable called rec_cpu_time. This is what will be used to keep track of a processes' recent CPU time. For rec_cpu_time to actually keep track of CPU time, /usr/src/kernel/clock.c was modified. In this file, the clock_handler function was modified to increment rec_cpu_time on every clock tick. Now that rec_cpu_time is keeping track of clock ticks, these have to be decayed frequently. This was taken care in the balance_queues function located in /usr/src/kernel/proc.c. balance_queues is called at least every 100 clock ticks. This is a fair enough length of time to decay the recent CPU time of a process. The way it is decayed is the recent CPU time of a process is divided by 2 and then added to the base. The value chosen for the base is 60.

So now that the recent CPU time is being kept track of, the task of picking the user process that has the smallest share of the CPU when no tasks or server wants to run needs to be taken care of. The default scheduling method in the version of Minix being used in this class uses round robin based scheduling. There are a total of 16 scheduling queues used in Minix. Processes are assigned an initial queue at system boot. The lowest queue number, 0, is the highest priority queue. This is used for kernel processes. The default queue for user processes is queue 7. And the lowest priority queue, 15, is used for the IDLE process. Besides the IDLE process, processes may be moved to a different priority queue by the system or in some circumstances, the user. The second piece of Minix's scheduling algorithm is the time quantum. This is the time interval allowed before a process is preempted. When a process uses its time quantum, it's placed at the end of its queue and may also have its priority lowered. So as whole, the Minix scheduler is a modified round robin. When the scheduler picks a process to run, the head of highest priority queue is checked for a ready process. If there is not one ready, the next lower priority queue is checked and so on. Since tasks normally have the highest priority, followed by drivers, then servers and user processes last, then user processes should not run unless all system processes have nothing to do. But since processes can change queues, there could be a situation in which a user process is in queue 0 and a server process is in queue 7. To prevent this from happening, the MAX_USER_Q, USER_Q, and MIN_USER_Q values in /usr/src/kernel/proc.h were all changed to 14. This guarantees that all user process will be in queue 14. The next modification was to the balance_queues routine in /usr/src/kernel/proc.c. The change made here is to assure that the priority of user processes is never changed, thus keeping them in queue 14. Similarly, in the sched routine also located in the same file, a change was made here to ensure that the priorities of user processes aren't changed, either. And also to make sure non-user processes are never put in the user queue. The last change to the Minix scheduler resides in the enqueue function contained in /usr/src/kernel/proc.c. This routine was changed to distinguish between user processes and non-user processes. When a user process is being queued, the enqueue routine determines which position in USER_Q the process must go. The position is determined by the recent CPU time of the process. User processes are placed in the queue based on ascending order of recent CPU time. So, the process with the smallest amount of recent CPU time is at the head of the queue, while the process with the most amount of recent CPU time is at the tail of the queue. Since the user process with the least amount of recent CPU time is at the head of the user queue, it's guaranteed that the user process with the least amount of recent CPU time will be run when the servers and tasks have nothing to do. No changes were made to the way non-user processes are enqueued.

With all the modifications that were made, non-user processes are scheduled the exact same way as before. The only difference is the way that user processes are scheduled. This was a very interesting assignment because it allowed me to gain a deeper insight into how each function fits together to make the scheduler work. There are plenty of different ways this problem could have been solved. This approach is believed to be correct. A lot of testing was done and the results are confirming.

Eric Grounds
C435
2-26-13
Homework 2 – Priority based user process scheduling

Below are the results of some test cases.

```
Dumping scheduling queues.
Q_nr--Q_pos--P_nr--P_name--P_pri--P_rec_cpu
  0     0    -2     system   0        1

 14     0    93        cpu  14        0
 14     1    92         io  14        1
 14     2    94         io  14        1
 14     3    91        cpu  14        8
 14     4    87        cpu  14        8

 15     0    -4       idle  15        0

Dumping scheduling queues.
Q_nr--Q_pos--P_nr--P_name--P_pri--P_rec_cpu
  0     0    -2     system   0        4

 14     0    92         io  14        0
 14     1    48        mfs  14        0
 14     2    94         io  14        1
 14     3    93        cpu  14        8
 14     4    91        cpu  14        8
 14     5    87        cpu  14        8

 15     0    -4       idle  15        0

Dumping scheduling queues.
Q_nr--Q_pos--P_nr--P_name--P_pri--P_rec_cpu
  0     0    -2     system   0        4

 14     0    92         io  14        1
 14     1    94         io  14        2
 14     2    93        cpu  14        7
 14     3    87        cpu  14        8
 14     4    91        cpu  14        8

 15     0    -4       idle  15        0

Dumping scheduling queues.
Q_nr--Q_pos--P_nr--P_name--P_pri--P_rec_cpu
  0     0    -2     system   0        4

 14     0    59     random  14        0
 14     1    94         io  14        2
 14     2    92         io  14        3
 14     3    93        cpu  14        8
 14     4    91        cpu  14        8
 14     5    87        cpu  14        8

 15     0    -4       idle  15        0
```