Mark Yang

Lab 2: Monitoring and Manipulating the Run-time State of Processes and Fair CPU scheduling

3)

The first process prints myglobalvictim as 0 however, after myattacker runs, myglobalvictim prints as 1 from myattackermalware. Since myattacker changes the return address of myvictim, myattackmalware is invoked so myattackermalware is run by the same process of myvictim.

4)

To monitor CPU usage of processes, I created an additional global variable timestamp to keep track of the starting time of the context switch for the processes. Then when the process uses up its quantum time (according to priority), I append the amount of time used (clktimefine – timestamp) to the prcpuused entry which holds the total amount of CPU time for the respective process.

Observations:

If all of the processes have the same priority, then they all share the CPU equally and are context switched in and out. Once all of the processes have finished and since they're all the same, their CPU time were equal.

Once I changed the priority of two of the processes to higher priority, each process finished running before moving onto the next process. Because we called sleepms(4000), there wasn't enough time for the other two processes who had the same priority as the main process to finish executing before we called kprintf and printed out all of the CPU times for each process. So the CPU time for the two processes of higher but equal priority was 6272 milliseconds while the two processes of equal priority to main were 2054 and 2040 milliseconds (because they were not finished running yet and continued to run after kprintf was called).

5)

1. For the null process, every time I context switched, I would set the null process' prcpuused to the prcpuused of the process that is being context switched and increment it by 1. Therefore, the null process would always be put at the end of the queue because of its prcpuused value.

Even though some of the CPU times are off by just a little bit, I feel that they indicate fair sharing of the CPU because their times are so close together. I feel that the offsets are just because of very minor errors that we can't control.

2. Running the iointensive processes, all four processes use roughly the same amount of CPU time again. The CPU times should be fair because all four processes are running the same thing with the same starting priorities. The very small difference in times again I feel are because of very minor errors that we can't control.

3. For the half-and-half tests, the two cpuintensive processes are being allocated fair CPU time even though they are off by a very little amount. The same is with the iointensive processes. The iointensive processes are also being allocated fair CPU time too.

Bonus)

When a new process is created, one way to prevent starvation is by setting the prcpuused of the new process equal to the prcpuused of the next process that is about to be context switched in (maybe with a bit of an offset). That way, the new process will be able to join into the queue and be given about the same amount of priority as the other processes.

I feel that this is a decent way of controlling CPU processing times as it will take the next available process' prcpuused and give that value to the new process. That way, it will just enter the queue and continue running along with the other processes in the queue with roughly the same priority without starving anyone with a really small prcpuused time (which would give it a really high priority).