Michael North
CS 354
Lab 3 Answers

Note: each and every modification I made for this lab is preceded by a comment of the format /* LAB 3 – Some notes about the code below */
As such, to find all of my work for this class, you can just run:
grep –rn "LAB 3" *
From the root directory of the project.

3.1: To prevent the null process from stealing CPU time from real processes, I implemented the simplest solution I could think of – setting the priority of every process to the value dictated by the Solaris time-share table *plus one*, unless the process would be set to priority 59, which which case it is still set to 59. While simple, this solution is not particularly elegant. For one thing, the zero-th queue in the multi-level feedback queue is essentially wasted, as it never contains anything other than the null process. This additionally means that there is one less priority level, practically speaking. None the less, this solution works well for this academic exercise, and does not disrupt any other functionality in XINU.

A run (with useful looking results) occurs with LOOP1 = 10, LOOP2 = 1000000, and the contents of the loop computing ALU operations only. This does *not* lead to the quantum being exhausted for each outer loop iteration. It seems that in this case, each outer loop runs 3 times before the quantum expires and the priority is lowered. What *is* seen is fair CPU time sharing among these four, initially identical processes. Each process of the same tier gets to run until it expires it's quantum exactly once.

3.2: It was more difficult to get interesting looking results here because, with the Solaris time-share table, IO-intensive processes escalate their priority *very* quickly. This is not an indication of un-fairness by any means, as once they are all at maximum priority, they will be handled in round-robin fashion. In any case, I was able to get interesting results with LOOP1 = LOOP2 = 2 (anything higher and they all reach maximum priority before anything is even printed). In this case, they were still handled in roughly round-robin order, which for identical processes is very fair.

Recreating the suggested parameters of LOOP1 = 10, no inner loop, and sleepms(500) also indicates fairness in execution and completion order among the processes, but due to some quirk of how console output is buffered, the output is very difficult to decipher.

3.3: With a mix of half CPU-intensive processes and half IO-intensive processes, things went roughly as expected.

Among the CPU-Intensive processes: again, mostly fairness, just as in 3.1. With some variation due to the priority of the main process, it was mostly the case that each CPU-intensive process got its turn until its quantum expired and then the next one ran in roughly round-robin fashion.

Among the IO-Intensive processes: roughly perfect fairness. These alternated in order perfectly.

Total fairness: For this case, it *seemed* approximately fair. Once the IO intensive processes were created, they took control away from the other processes only briefly, and then mostly allowed the CPU-intensive processes to finish their work in between the sleep periods. In turn, the CPU-intensive processes were fair to the IO-intensive processes because their "long" quantum time was still shorter than the sleep time of the IO-intensive processes.

That said, it seems clear to me that it would be easy for this fairness to collapse. If, for example, there were significantly more IO-intensive processes (say, at least 10 times more) and/or if the IO-intensive processes had a much, much shorter sleep time, then the CPU-intensive processes may be completely starved out because the first IO-intensive process to sleep will be ready again before the last one relinquishes control, meaning the lower priority CPU-intensive processes would never be reached.

4: In my implementation of the process timing code, I inserted the commands to store the current value of clktimeaccru for the new process and update prtotalcpu for the old process immediately before the call to ctxsw in resched.c.

This appears to be the *only* place (based on a grep search) in the kernel where ctxsw is called, so I felt safe including the code here.

To do the aforementioned computations, I added an additional field in the procent struct called clktimeold, where the current value of clktimeaccru is stored for the new process.

Kernel Measurements:
Each CPU intensive process used approximately 280-300ms of CPU time to completely execute.
Each IO Intensive process used approximately 9-10ms of CPU time to completely execute.
Fairness between identical processes still seems great, and it is again worth noting that the IO intensive processes grab control as soon as they are created and again whenever they become available (after sleeping). As such, fairness between all processes seems to be excellent now.

Bonus: I implemented the hybrid process to use up close to its entire quantum and then sleep for one millisecond. I then created one instance of cpuintensive and one instance of hybridprocess, both with the same initial priority of 20.

The cpuintensive process took control from the main process (as often happens) after being created and ran until its quantum expired. Then, the hybrid process was created, and it took control. It ran nearly uninterrupted until completion. The low priority cpuintensive process did manage to take control once, presumably while hybridprocess was sleeping for that single millisecond.

None the less, hybridproccess managed to hold on to and use the cpu significantly faster than the cpuintensive process.

If there are any additional clarifications I can make regarding my work, please let me know. I feel that I have accomplished everything asked in this lab and then some, but I'm not sure I have fully understood what information is being looked for in this pdf. Thank you.