3.1/3.2. We test these two parts together for simpler presentation. We make the CPU run one loop in two tests, for 100000000 times and 200000000 times respectively. We can observe that our implementation recorded the total CPU usage correctly. See the test results in Figure 1. Then, we run two processes in one execution. We can see that the total CPU usages were recorded correctly as well. See the results in Figure 2 (since these two processes were executed simultaneously, their test outputs are mixed together. However, even then we could observe that the CPU usages were recorded properly).

Figure 1



Figure 2

3.3. We resume a starved process (with low priority 1) first, and run a loop in the main process immediately. We can see that the starved process did not start running until the main process was no longer occupying the processor after about 2 seconds. See the test results in Figure 3.

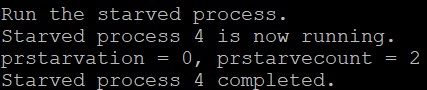


Figure 3

3.4. We resume a process with lower priority than the main process like before, and run a loop for each process. We can see that the mean response time for the lower-priority process is exactly how long the lower-priority process has been waiting. See the test results in Figure 4.



Figure 4

4.5.2A. We run the tests following the requirements of benchmark A. To generate more readable results (preventing them from mixing together), I added different small delays for each process to proceed to output (so they actually have the same clkcounterfine value if we consider that). As predicted, they have similar CPU usages and average response time, and no starvation has occurred. See the test results in Figure 5.

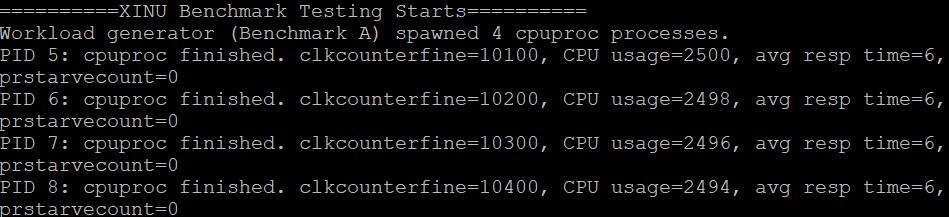


Figure 5

4.5.2B. We run the tests following the requirements of benchmark B. As we can see, they also have similar CPU usages (0) and average response time, and no starvation has occurred. See the test results in Figure 6.

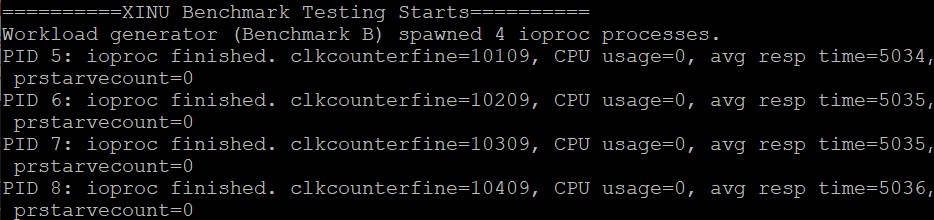


Figure 6

4.5.2C. We run the tests following the requirements of benchmark C. As we can see, the CPU usage was mainly occupied by cpuproc as expected. The average response time for cpuproc is short, while the average response time for ioproc is long. This is also expected since ioproc does minimal computations and has comparatively long sleep time. No starvation has occurred either. See the test results in Figure 7.

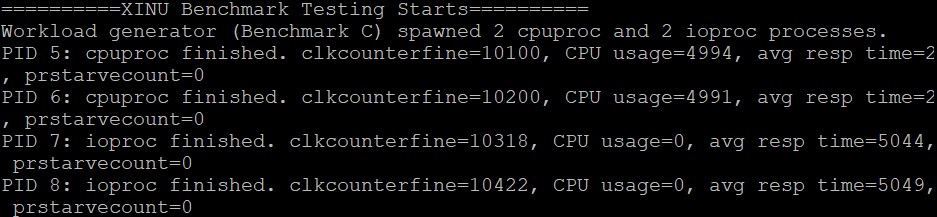


Figure 7

4.5.2D. We run the tests following the requirements of benchmark D. However, the starvation prevention mechanism in 4.4 did not seem to be effective. When it was enabled (Figure 9), we got more starvation counts than disabled (Figure 8). See the test results in Figure 8 and 9 next page.

Bonus. To achieve this goal, we add brief sleeps in between the computations done by rogue, in order to “cheat” the scheduler with an apparent CPU usage lower than actual usage. See the test results in Figure 10.

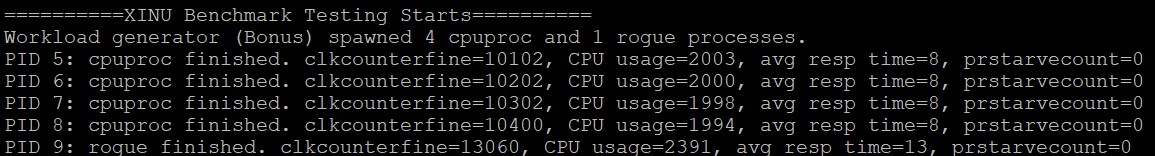


Figure 10

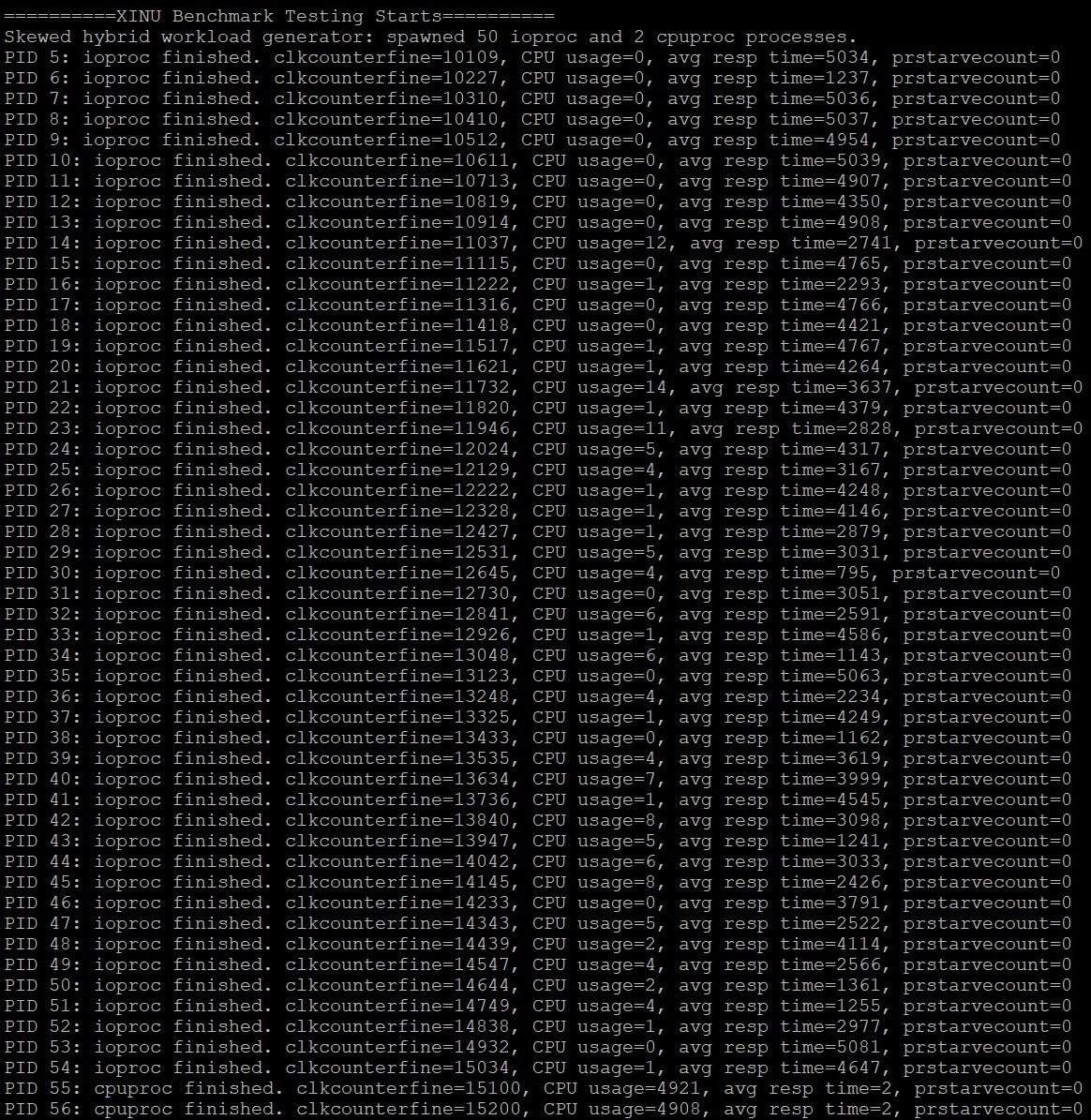


Figure 8

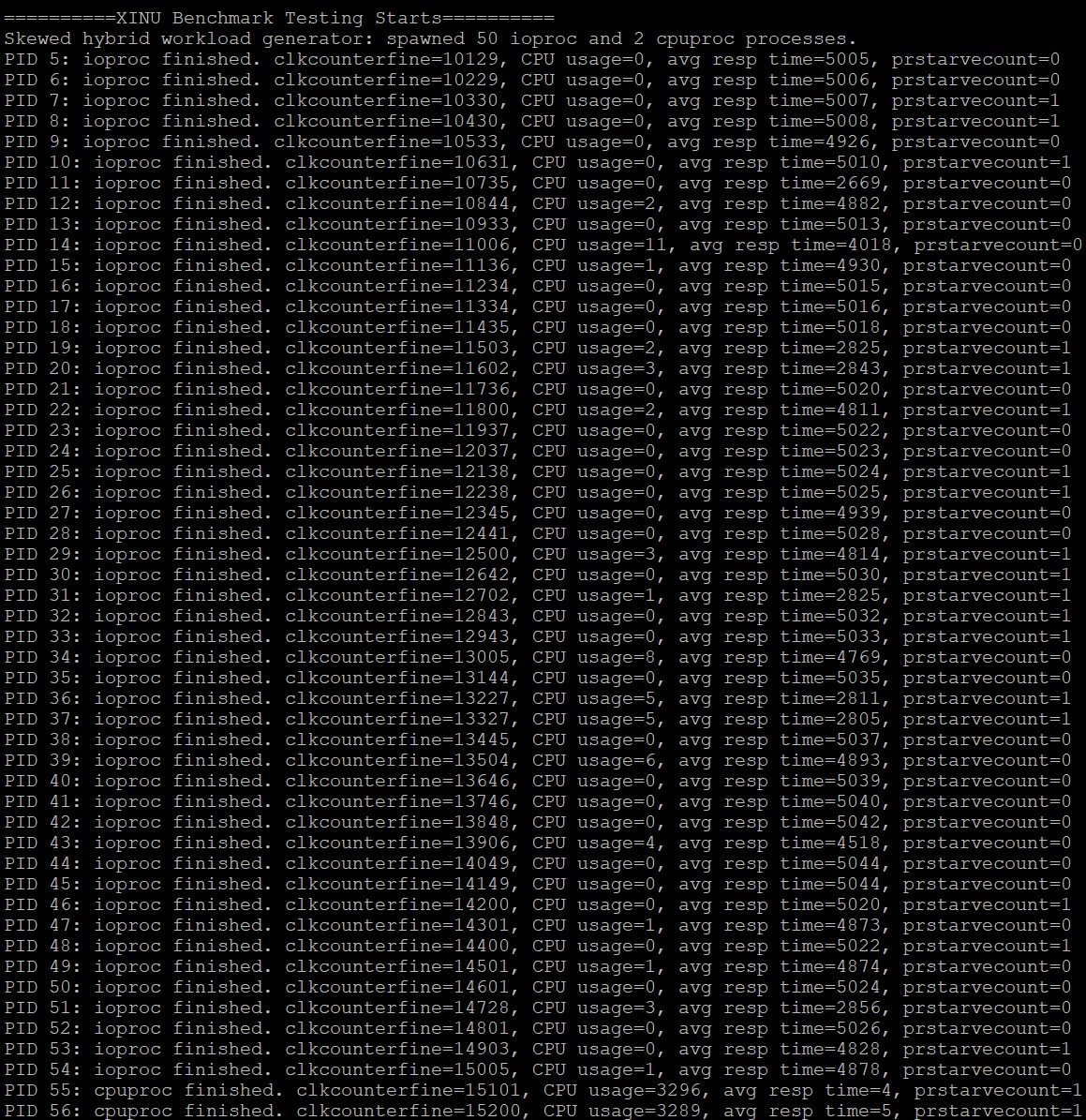


Figure 9