# Programming Assignment 3: Investigating the Linux Scheduler

Stephen Bennett
CSCI 3753 - Operating Systems
University of Colorado at Boulder
Spring 2013

*Due Date: Sunday, March 31, 2013 11:55pm*

**Abstract**

The purpose of this project was to investigate the differences among the various Linux scheduling policies. The scheduler was tested by gathering timing and execution information from three benchmark programs written to represent the three types of processes: compute bound, I/O bound, and a mix of the two. It was determined that the ideal scheduling algorithm differs based on the type of process being scheduled. Compute bound processes perform best under the SCHED_OTHER scheduling policy, I/O bound processes perform best under a real-time scheduling policy, and mixed processes perform nearly equally under all scheduling policies.

# 1 Introduction

The purpose of this investigation into the Linux scheduler is to discover and discuss the differences among the various scheduling algorithms. To do this, I will write three test programs `pi.c`, `rw.c` and `mix.c`, each representative of a different type of real world program. The test system will then be loaded with multiple instances of these programs in order to produce various levels of system utilization. The processes will be scheduled using specified scheduling policies. Using the Linux `time` command, information about the run-time and execution of the benchmarks will be gathered. All tests will be run on a desktop computer running 64-bit Linux Mint 14 Nadia with version 3.5.0-26-generic of the Linux kernel.

# 2 Method

## 2.1 Benchmarks

Three benchmarks were written, each testing one of the three possible program types: CPU bound, I/O bound, and a mix of the two. In creating the CPU bound benchmark `pi.c`, some computationally intensive algorithm had to be chosen such that no time would be spent waiting on I/O operations or blocked on any other resource. As a result of these constraints, the statistical algorithm for calculating the value of pi based on the Monte Carlo method was chosen[1]. This algorithm is relatively slow and very CPU intensive. It creates an imaginary quarter circle of radius RAND_MAX, generates a random pair (x,y) of coordinates $\{x, y \in \mathbb{R} \mid 0 \leq x, y \leq RAND\_MAX\}$, then calculates whether the (x,y) coordinate is within the quarter circle. Additionally, there are two counters: one that keeps track of the total number of iterations and another that keeps track of the number of times the random (x,y) coordinate is found to be within the quarter circle. Finally, once all iterations are complete, all that must be done is to calculate the probability of being in the quarter circle by dividing the two counters (inside circle divided by number of iterations) and multiplying the result by 4 to create a full circle rather than a quarter circle.

The I/O bound benchmark `rw.c` was written in such a way as to "minimize the effects of filesystem buffering and maximize I/O delays"[4]. In order to do this, the low level-level `read()` and `write()` system calls were used in conjunction with files opened in O_SYNC mode. O_SYNC causes `write()` operations to block until the data is physically written to the disk rather than until the data is simply copied to a kernel buffer[5]. An input file and an output file are given to the program which then reads blocks of data from the input file and writes said data to the output file. This process occurs multiple times with minimal CPU involvement, thus creating the I/O bound benchmark. There is only a small amount of CPU use involved, primarily in setting up the input/output files and verifying the passed parameters.

The third and final benchmark program was the mixed benchmark `mix.c`. I wrote this benchmark such that it involves both computationally and I/O intensive sections. The program performs the same computation as the CPU bound process statistically calculating the value of pi, but every so many iterations it writes the current values of the counters and the estimated value of pi to a file. This `write()` operation is once again performed using O_SYNC mode to maximize I/O delays.

For each benchmark, I wrote a separate program (e.g. `rw-sched.c` for I/O benchmark, `pi-sched.c` for CPU benchmark[2], etc.) that took care of setting the correct scheduling policy and `fork()`ing the desired number of child processes. Each time a process needs to read from or write to a file, it needs its own input or output file in order to prevent additional waiting due to mutual exclusions and not actual I/O as is desired; the housekeeping programs ensure that each process has its own unique input/output file.

## 2.2 Testing and Data

I wrote a Bash script to take care of automation for running the 27 different test cases. Each test case was run three times and the results were averaged in order to get more accurate data. While compiling the source code with the Makefile, a single 1 MB file called `rwinput` is created. It is created by reading 1024 blocks of size 1024 kB of random data from `/dev/urandom` and writing the data to `rwinput`. To ensure that each instance of the I/O bound benchmark program had its own input file, the Bash script then copies `rwinput` as many times as necessary and gives each copy a unique name. The `rw-sched.c` program then passes the proper unique input file name to each child process `fork()`ed. Through the use of the `time` command and nested `for` loops iterating over both the number of child processes to `fork()` and the scheduling algorithms, I gathered results for each of the test cases. From the `time` command I gathered the following aggregate information for each set of processes:

- Wall time (turnaround time) - the real time the process took to complete from when it entered the system to when it completed
- User time - the amount of CPU-seconds the process spent executing in user mode
- System time - the amount of CPU-seconds the process spent executing in kernel mode
- The percentage of the CPU that the process got
- The number of times the process was context switched
- The number of times the process was blocked on I/O

Since the `time` command was used on the scheduler process which spawns the appropriate number of child processes, all information collected from the `time` command is the sum of the information of all child processes. As a result, in order to get more useful information on a per process basis, the aggregate results will be divided by the number of child processes spawned.

## 2.3 Test System

All of the tests were run on a desktop running 64-bit Linux Mint 14 Nadia with version `3.5.0-26-generic` of the Linux kernel. This all runs on a quad-core Intel Core i5-2500K CPU running at 3.30GHz with 8GB of RAM. Each core is capable of executing a single hardware thread; thus four processes may be running concurrently in this system. The primary disk for the system is a 1TB 7200 RPM Western Digital with 32MB of cache which uses the SATA II interface. All of the programs were compiled using `GNU C Compiler version 4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1)`. The time slice for the Round Robin scheduling algorithm for this setup is 0.1000006 seconds.

# 3 Results

Using the methodology described above, I gathered the following results. Figure 1 on page 5, Figure 2 on page 6, and Figure 3 on page 6 show the average wall time per child process averaged over three trial runs for the three different benchmarks. Each figure shows the results for a specific benchmark and compares the performance of each scheduling policy based on number of child processes.

The CPU bound benchmark results in Figure 1 show that while the number of processes remains relatively low the SCHED_OTHER scheduling policy out-paces the other policies tested; at high numbers of processes the advantage disappears for all intents and purposes. Both of the Real-time scheduling algorithms perform relatively equally regardless of number of processes. Additionally, as the number of processes increases, each process takes more time to complete.

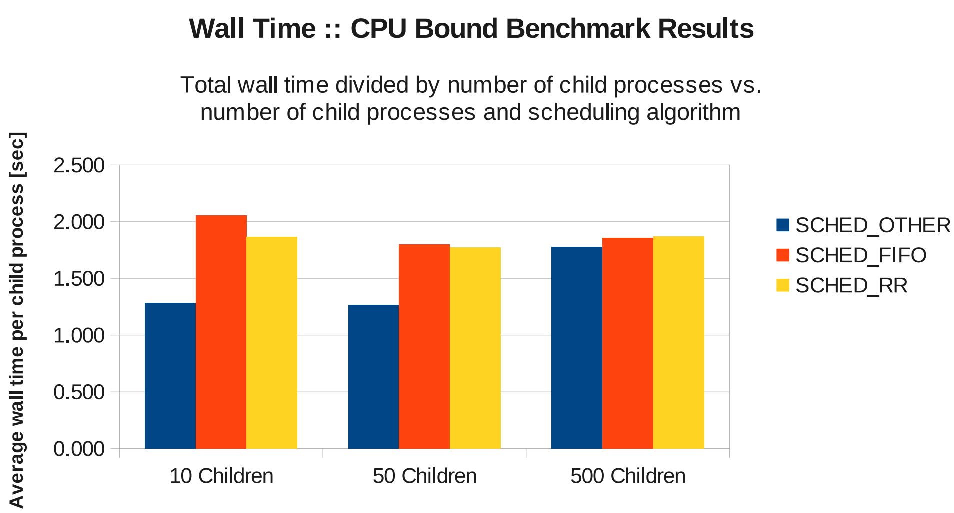


Figure 1

The I/O bound benchmark results in Figure 2 show that a higher priority scheduling algorithm improves wall time. As the number of processes increases, the performance of the different scheduling algorithms converges and the amount of time for each process to complete decreases.

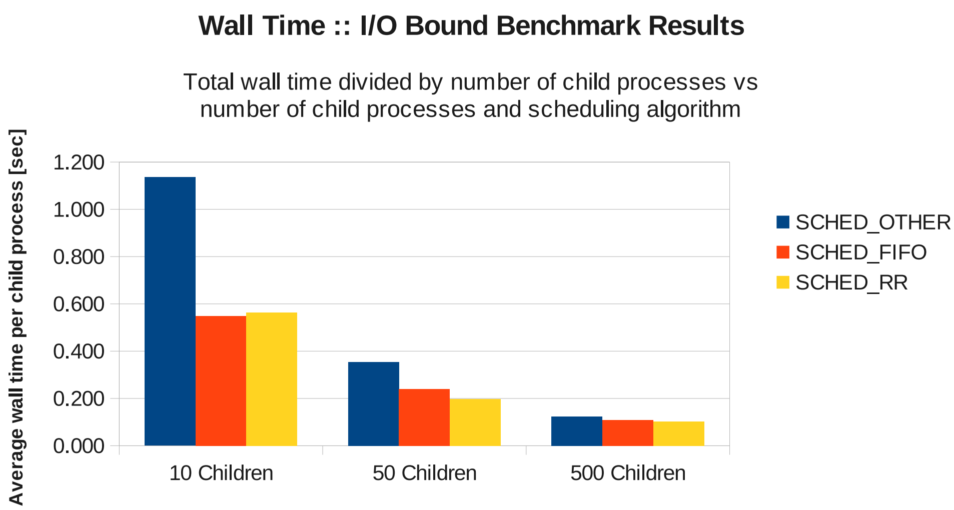


Figure 2

The mixed benchmark results in Figure 3 show that the scheduling policy does not have a significant impact on the amount of time each process takes to complete execution. As the number of processes increases, the amount of time each process takes to complete its execution decreases.

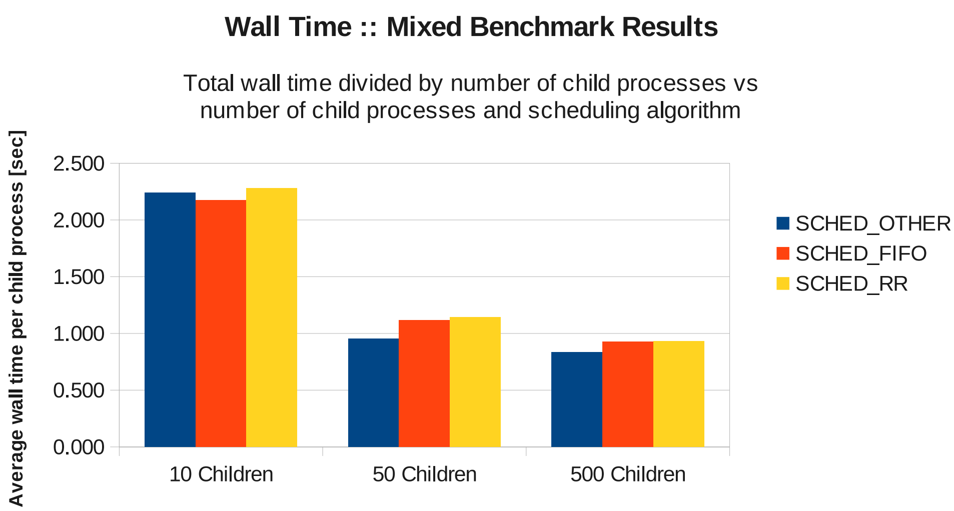


Figure 3

Figure 4 on page 7 and Figure 5 on page 7 show that the number of times a process is context switched under a Real-time scheduling policy remains constant as the number of processes increases. On the other hand, with the SCHED_OTHER scheduling policy as the number of child processes increases, so does the amount of context switches per process.

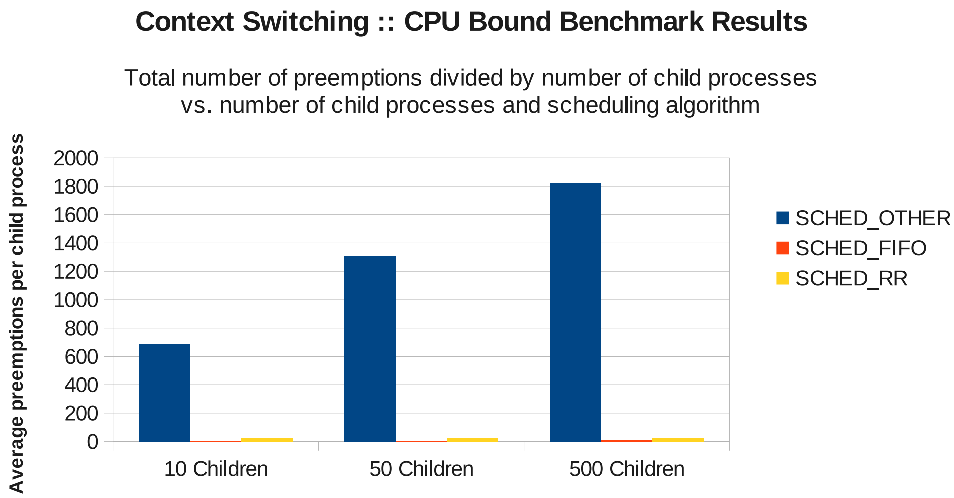


Figure 4

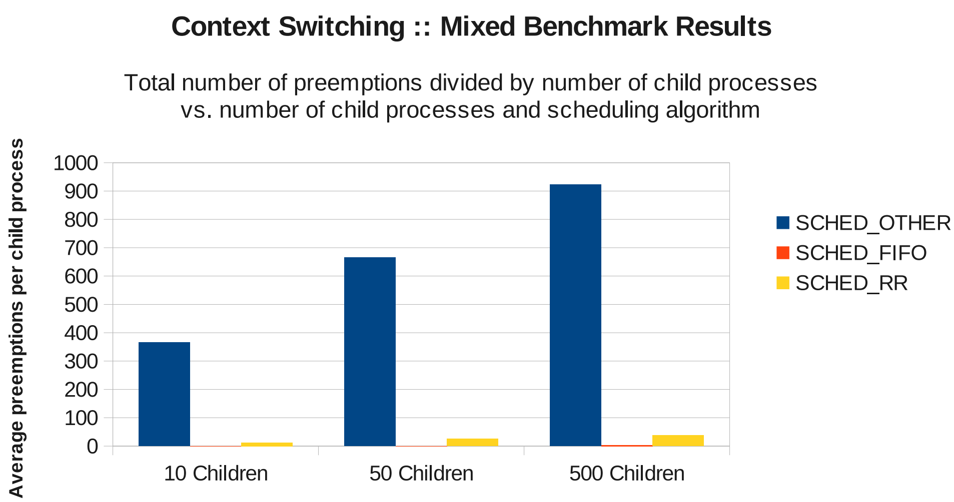


Figure 5

As can be seen in Figure 6 on page 8 showing the throughput of the CPU bound benchmark, the highest throughput is achieved using the SCHED_OTHER scheduling algorithm at relatively low quantities of child processes. As the number of processes increases, all of the scheduling algorithms converge on a throughput of roughly one process completed every two seconds.

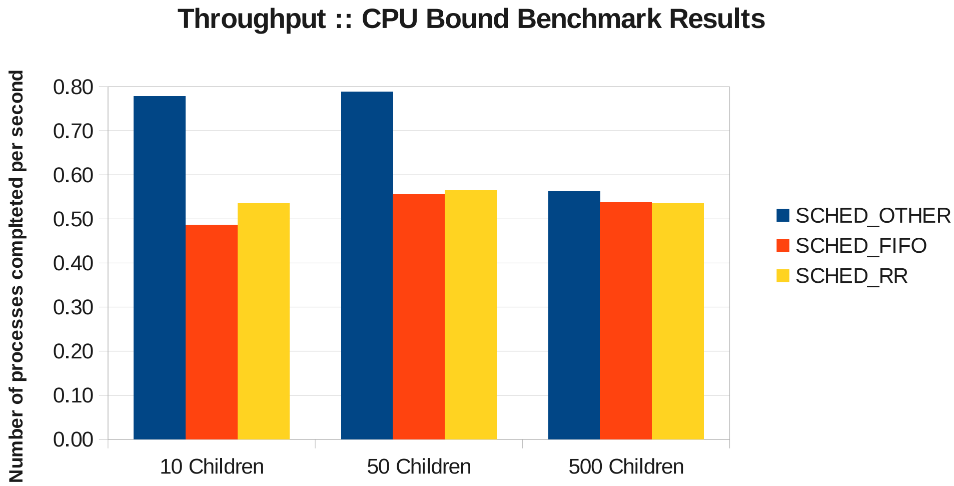


Figure 6

The throughput results in Figure 7 on page 8 show that increasing the number of concurrently executing I/O bound processes increases the overall throughput of the system. Additionally, the higher the priority of the scheduling algorithm, the higher the throughput.

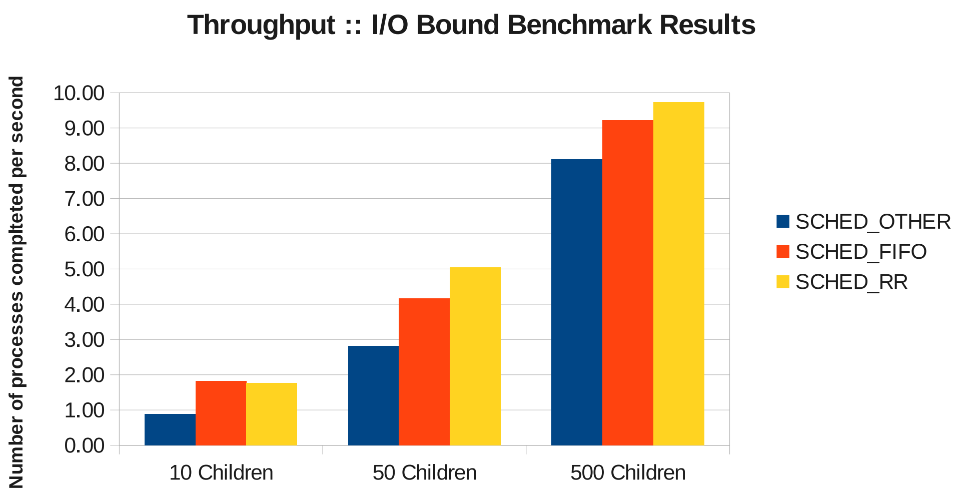


Figure 7

The throughput results in Figure 8 on page 9 show a similar overall positive correlation between number of processes and throughput to the I/O throughput results. However, at higher numbers of processes, the SCHED_OTHER scheduling algorithm seems to show a slight lead, similar to what is seen at lower system utilization in Figure 6 on page 8.

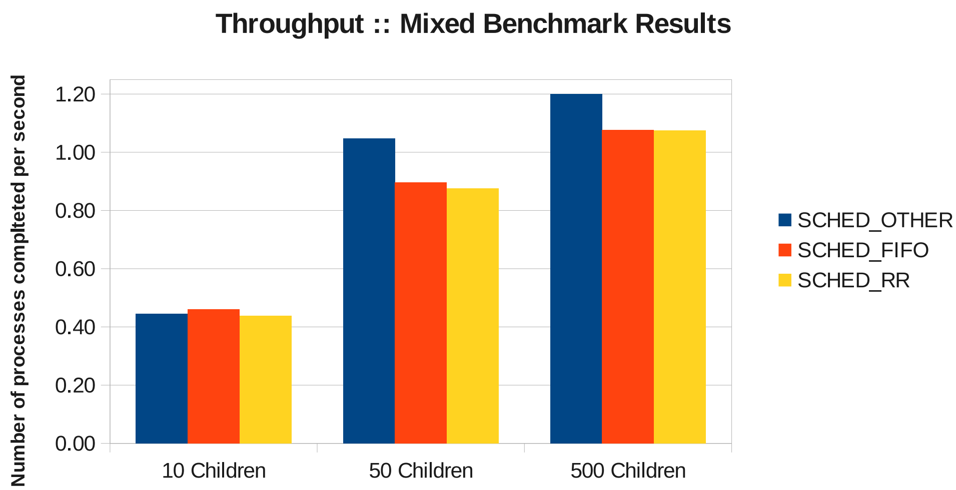


Figure 8

Figure 9 on page 9 and Figure 10 on page 10 show that as the system utilization increases, so does the CPU utilization; the mixed benchmark shows this more drastically. The CPU bound benchmark results show that the SCHED_OTHER scheduling algorithm makes the most efficiency use of the CPU, nearly maxing out all four cores regardless of system utilization.

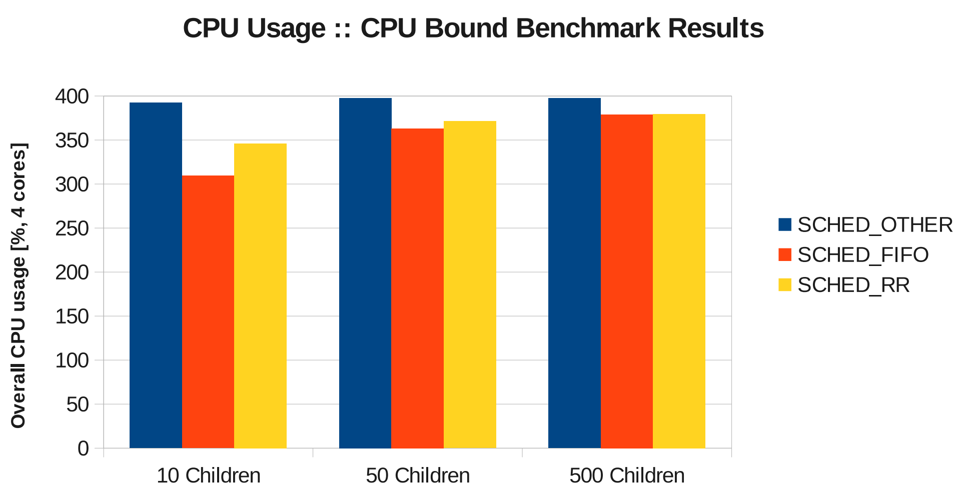


Figure 9

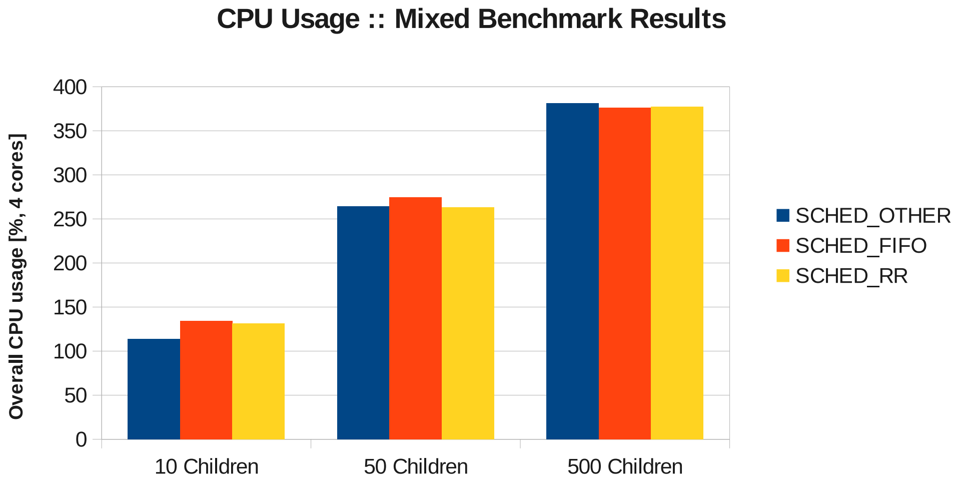


Figure 10

# 4 Analysis

The ideal scheduling policy in terms of minimizing turnaround time varies depending on the type of process. For CPU bound processes, it is clear the SCHED_OTHER algorithm reduced overall run-time, regardless of the system load and in spite of the greater amount of context switching that occurred with this scheduling policy. I believe this was a result of the scheduler's ability to maximize CPU usage at all levels of system utilization, unlike what was done with either SCHED_FIFO or SCHED_RR. There does appear to be a trend where increasing the number of processes reduces SCHED_OTHER's advantage. In fact, it is quite possible that if the system utilization was increased further, SCHED_OTHER might eventually lose out to the real-time policies.

For I/O bound processes, there is no best scheduler, but SCHED_OTHER is clearly outperformed by both of the real-time scheduling algorithms. As system utilization was increased though, the gap between the CFS and real-time algorithms was reduced. This can most likely be attributed to a more efficient use of the I/O subsystem. Since SCHED_RR and SCHED_FIFO processes have higher priorities than SCHED_OTHER processes, they are more likely to be given CPU time as soon as they need it which allows them to more quickly make their next I/O request. As more processes are added to the system, the I/O subsystem spends less time sitting idle. This idea can also be seen in Figure 7; the I/O subsystem is able to support a large amount of data transfer which is not efficiently put to use until enough I/O bound processes are present in the system. Using the information from the throughput figure does seem to indicate that the SCHED_RR algorithm is actually the best for completely I/O bound processes.

As for mixed processes, there is simply no best scheduling policy directly in terms of wall time. The combination of computation and I/O counteract each other and as a result each scheduling algorithm performs essentially identically, although a slight edge may be given to SCHED_OTHER. This advantage can also be seen in the throughput results at higher levels of system utilization. It is entirely likely that when I wrote the mixed benchmark, I unknowingly gave the program more of a CPU bound tendency. As described above, this would cause the processes to perform best under the SCHED_OTHER scheduling policy. I would think that mixed processes would lend themselves to SCHED_FIFO as the best scheduling policy since that is somewhere between SCHED_OTHER and SCHED_RR in terms of priority, but that does not seem to be entirely the case.

If minimizing time wasted on context switching is desired, SCHED_FIFO is the ideal scheduling algorithm. Since a SCHED_FIFO process can only be preempted by a higher priority process and all processes were given the same priority, the SCHED_FIFO were almost never preempted (any preemptions were outside the control of my test system setup).

# 5 Conclusion

The purpose of this investigation into the Linux Scheduler was to determine how each scheduling algorithm performed under various circumstances. The real-time scheduling algorithms SCHED_RR and SCHED_FIFO proved to be best suited for processes that involved performing a lot of I/O operations. SCHED_RR performed the best or equally as well for I/O bound processes under all levels of system utilization. For CPU bound processes, SCHED_OTHER was clearly the optimal scheduling algorithm. For low to medium levels of system utilization, the wall time was significantly less, and for high system utilization the wall time was still slightly lower. Additionally, the SCHED_OTHER scheduling algorithm was able to maximize throughput and CPU usage. Lastly, for mixed processes that involved both computationally and I/O intensive sections there proved to be no clear best scheduling policy. The results support SCHED_OTHER slightly more than SCHED_FIFO and least of all SCHED_RR. These results lend themselves to the theory that the mixed benchmark was written to be more CPU bound than I/O bound.

# References

[1] Sayler, Andy. *pi.c*. 03/09/2012. Forked 03/17/2013. https://github.com/asayler/CU-CS3753-PA3/blob/master/pi.c.

[2] Sayler, Andy. *pi-sched.c*. 03/09/2012. Forked 03/17/2013. https://github.com/asayler/CU-CS3753-PA3/blob/master/pi-sched.c.

[3] Sayler, Andy. *rw.c*. 03/09/2012. Forked 03/17/2013. https://github.com/asayler/CU-CS3753-PA3/blob/master/rw.c.

[4] Sayler, Andy. *Programming Assignment 3: Investigating the Linux Scheduler*. 03/06/2012 https://github.com/cgartrel/CU-CS3753-PA3/blob/master/handout/pa3.pdf.

[5] Linux man-pages project. *open(2)*. 2013/02/18. Accessed 03/23/2013. http://man7.org/linux/man-pages/man2/open.2.html.

# A Appendix - Raw Data

Figure 11: Results gathered from the first test run

| Process Type | Scheduler | Children | wall [sec] | user [CPU-sec] | system [CPU-sec] | CPU [%] | preempted [count] | blocked [count] |
|---|---|---|---|---|---|---|---|---|
| CPU Bound | SCHED_OTHER | 10 | 12.73 | 50.40 | 0.03 | 396 | 6841 | 23 |
| | | 50 | 63.38 | 251.92 | 0.14 | 397 | 65188 | 103 |
| | | 500 | 873.94 | 3480.89 | 0.81 | 398 | 890523 | 1013 |
| | SCHED_FIFO | 10 | 16.17 | 50.01 | 0.01 | 309 | 37 | 15 |
| | | 50 | 75.11 | 275.70 | 0.06 | 367 | 273 | 55 |
| | | 500 | 896.43 | 3389.46 | 0.57 | 378 | 3526 | 506 |
| | SCHED_RR | 10 | 14.83 | 49.99 | 0.00 | 337 | 171 | 19 |
| | | 50 | 75.39 | 277.29 | 0.06 | 367 | 1047 | 77 |
| | | 500 | 913.14 | 3474.32 | 0.60 | 380 | 13310 | 796 |
| I/O Bound | SCHED_OTHER | 10 | 11.59 | 0.00 | 0.03 | 0 | 17 | 2021 |
| | | 50 | 16.37 | 0.01 | 0.05 | 0 | 142 | 16785 |
| | | 500 | 62.96 | 0.00 | 0.07 | 0 | 589 | 162039 |
| | SCHED_FIFO | 10 | 5.39 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 15.89 | 0.06 | 0.07 | 0 | 1 | 17379 |
| | | 500 | 57.84 | 0.04 | 0.12 | 0 | 1 | 162372 |
| | SCHED_RR | 10 | 5.85 | 0.00 | 0.00 | 0 | 2 | 2012 |
| | | 50 | 10.04 | 0.00 | 0.00 | 0 | 1 | 10052 |
| | | 500 | 47.90 | 0.02 | 0.02 | 0 | 1 | 117478 |
| Mixed | SCHED_OTHER | 10 | 19.61 | 25.31 | 0.06 | 129 | 3845 | 4966 |
| | | 50 | 48.83 | 126.43 | 0.27 | 259 | 33109 | 25207 |
| | | 500 | 391.28 | 1489.02 | 2.49 | 381 | 430702 | 237800 |
| | SCHED_FIFO | 10 | 21.07 | 27.26 | 0.04 | 129 | 1 | 4434 |
| | | 50 | 55.94 | 150.94 | 0.24 | 270 | 1 | 21302 |
| | | 500 | 450.46 | 1695.02 | 2.51 | 376 | 1648 | 202745 |
| | SCHED_RR | 10 | 22.21 | 30.08 | 0.04 | 135 | 125 | 4340 |
| | | 50 | 58.78 | 151.95 | 0.20 | 258 | 1244 | 21720 |
| | | 500 | 453.42 | 1704.89 | 2.27 | 376 | 17887 | 190474 |

Figure 12: Results gathered from the second test run

| Process Type | Scheduler | Children | wall [sec] | user [CPU-sec] | system [CPU-sec] | CPU [%] | preempted [count] | blocked [count] |
|---|---|---|---|---|---|---|---|---|
| CPU Bound | SCHED_OTHER | 10 | 12.95 | 50.46 | 0.02 | 389 | 6848 | 21 |
| | | 50 | 63.45 | 252.54 | 0.11 | 398 | 65261 | 103 |
| | | 500 | 898.79 | 3572.34 | 1.33 | 397 | 924232 | 1014 |
| | SCHED_FIFO | 10 | 22.04 | 71.51 | 0.04 | 324 | 57 | 15 |
| | | 50 | 97.17 | 356.24 | 0.07 | 366 | 360 | 55 |
| | | 500 | 951.87 | 3617.49 | 1.23 | 380 | 3797 | 505 |
| | SCHED_RR | 10 | 19.89 | 72.53 | 0.02 | 364 | 227 | 17 |
| | | 50 | 96.23 | 359.64 | 0.08 | 373 | 1346 | 72 |
| | | 500 | 952.80 | 3623.47 | 0.88 | 380 | 13707 | 983 |
| I/O Bound | SCHED_OTHER | 10 | 11.51 | 0.03 | 0.00 | 0 | 13 | 2028 |
| | | 50 | 16.94 | 0.03 | 0.06 | 0 | 98 | 17272 |
| | | 500 | 58.82 | 0.04 | 0.06 | 0 | 503 | 162071 |
| | SCHED_FIFO | 10 | 5.57 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 10.09 | 0.00 | 0.00 | 0 | 1 | 10052 |
| | | 500 | 51.67 | 0.00 | 0.02 | 0 | 1 | 118768 |
| | SCHED_RR | 10 | 5.70 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 9.90 | 0.00 | 0.00 | 0 | 1 | 10052 |
| | | 500 | 51.16 | 0.00 | 0.03 | 0 | 2 | 117443 |
| Mixed | SCHED_OTHER | 10 | 25.86 | 25.01 | 0.06 | 96 | 3450 | 5059 |
| | | 50 | 49.13 | 125.83 | 0.18 | 256 | 32865 | 25361 |
| | | 500 | 431.74 | 1610.59 | 3.48 | 373 | 462777 | 236914 |
| | SCHED_FIFO | 10 | 21.26 | 30.34 | 0.06 | 142 | 1 | 4164 |
| | | 50 | 58.43 | 153.67 | 0.24 | 263 | 5 | 21544 |
| | | 500 | 474.63 | 1786.46 | 2.83 | 376 | 1792 | 187504 |
| | SCHED_RR | 10 | 23.97 | 29.86 | 0.05 | 124 | 93 | 4353 |
| | | 50 | 55.53 | 147.36 | 0.24 | 265 | 1216 | 21618 |
| | | 500 | 474.85 | 1790.48 | 2.23 | 377 | 19045 | 166890 |

Figure 13: Results gathered from the third test run

| Process Type | Scheduler | Children | wall [sec] | user [CPU-sec] | system [CPU-sec] | CPU [%] | preempted [count] | blocked [count] |
|---|---|---|---|---|---|---|---|---|
| CPU Bound | SCHED_OTHER | 10 | 12.83 | 50.33 | 0.05 | 392 | 6925 | 21 |
| | | 50 | 63.35 | 252.13 | 0.05 | 398 | 65678 | 105 |
| | | 500 | 892.69 | 3544.59 | 1.30 | 397 | 921181 | 1014 |
| | SCHED_FIFO | 10 | 23.44 | 69.17 | 0.00 | 295 | 55 | 15 |
| | | 50 | 97.68 | 348.50 | 0.12 | 356 | 345 | 55 |
| | | 500 | 938.54 | 3564.14 | 1.13 | 379 | 3733 | 505 |
| | SCHED_RR | 10 | 21.32 | 71.94 | 0.02 | 337 | 246 | 17 |
| | | 50 | 94.10 | 352.11 | 0.07 | 374 | 1327 | 78 |
| | | 500 | 937.95 | 3561.11 | 0.81 | 379 | 13487 | 593 |
| I/O Bound | SCHED_OTHER | 10 | 10.98 | 0.03 | 0.05 | 0 | 27 | 4012 |
| | | 50 | 19.86 | 0.02 | 0.06 | 0 | 102 | 16814 |
| | | 500 | 63.02 | 0.09 | 0.08 | 0 | 541 | 166311 |
| | SCHED_FIFO | 10 | 5.53 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 10.05 | 0.00 | 0.00 | 0 | 2 | 10052 |
| | | 500 | 53.22 | 0.01 | 0.05 | 0 | 1 | 147790 |
| | SCHED_RR | 10 | 5.38 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 9.74 | 0.00 | 0.00 | 0 | 1 | 10052 |
| | | 500 | 55.14 | 0.03 | 0.08 | 0 | 1 | 165565 |
| Mixed | SCHED_OTHER | 10 | 21.78 | 25.57 | 0.05 | 117 | 3675 | 4881 |
| | | 50 | 45.31 | 125.82 | 0.20 | 278 | 33757 | 25191 |
| | | 500 | 426.77 | 1664.64 | 2.89 | 390 | 490777 | 235389 |
| | SCHED_FIFO | 10 | 22.86 | 30.13 | 0.06 | 132 | 1 | 4189 |
| | | 50 | 53.02 | 153.78 | 0.25 | 290 | 2 | 21428 |
| | | 500 | 467.74 | 1760.98 | 2.67 | 377 | 1756 | 169198 |
| | SCHED_RR | 10 | 22.20 | 30.07 | 0.06 | 135 | 150 | 4336 |
| | | 50 | 57.06 | 152.13 | 0.12 | 266 | 1315 | 21765 |
| | | 500 | 466.12 | 1766.21 | 2.30 | 379 | 19033 | 161616 |

Figure 14: Averaged results over all three test runs

| Process Type | Scheduler | Children | wall [sec] | user [CPU-sec] | system [CPU-sec] | CPU [%] | preempted [count] | blocked [count] |
|---|---|---|---|---|---|---|---|---|
| CPU Bound | SCHED_OTHER | 10 | 12.84 | 50.40 | 0.04 | 391 | 6887 | 21 |
| | | 50 | 63.39 | 252.20 | 0.08 | 398 | 65470 | 104 |
| | | 500 | 888.47 | 3532.61 | 1.32 | 397 | 922707 | 1014 |
| | SCHED_FIFO | 10 | 20.55 | 63.56 | 0.02 | 310 | 56 | 15 |
| | | 50 | 89.99 | 326.81 | 0.10 | 361 | 353 | 55 |
| | | 500 | 928.95 | 3523.70 | 1.18 | 380 | 3765 | 505 |
| | SCHED_RR | 10 | 18.68 | 64.82 | 0.02 | 351 | 237 | 17 |
| | | 50 | 88.57 | 329.68 | 0.08 | 374 | 1337 | 75 |
| | | 500 | 934.63 | 3552.97 | 0.85 | 380 | 13597 | 788 |
| I/O Bound | SCHED_OTHER | 10 | 11.36 | 0.02 | 0.03 | 0 | 20 | 3020 |
| | | 50 | 17.72 | 0.02 | 0.06 | 0 | 100 | 17043 |
| | | 500 | 61.60 | 0.04 | 0.07 | 0 | 522 | 164191 |
| | SCHED_FIFO | 10 | 5.50 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 12.01 | 0.02 | 0.00 | 0 | 2 | 10052 |
| | | 500 | 54.24 | 0.02 | 0.04 | 0 | 1 | 133279 |
| | SCHED_RR | 10 | 5.64 | 0.00 | 0.00 | 0 | 1 | 2012 |
| | | 50 | 9.89 | 0.00 | 0.00 | 0 | 1 | 10052 |
| | | 500 | 51.40 | 0.02 | 0.06 | 0 | 2 | 141504 |
| Mixed | SCHED_OTHER | 10 | 22.42 | 25.30 | 0.06 | 107 | 3563 | 4970 |
| | | 50 | 47.76 | 126.03 | 0.19 | 267 | 33311 | 25276 |
| | | 500 | 416.60 | 1588.08 | 3.19 | 382 | 476777 | 236152 |
| | SCHED_FIFO | 10 | 21.73 | 29.24 | 0.06 | 137 | 1 | 4177 |
| | | 50 | 55.80 | 152.80 | 0.25 | 277 | 4 | 21486 |
| | | 500 | 464.28 | 1747.49 | 2.75 | 377 | 1774 | 178351 |
| | SCHED_RR | 10 | 22.79 | 30.00 | 0.06 | 130 | 122 | 4345 |
| | | 50 | 57.12 | 150.48 | 0.18 | 266 | 1266 | 21692 |
| | | 500 | 464.80 | 1753.86 | 2.27 | 378 | 19039 | 164253 |

Figure 15: Averaged results over all three test runs, divided by the respective number of child processes

| Process Type | Scheduler | wall/child [sec] | user/child [CPU-sec] | system/child [CPU-sec] | preempted/child [count] | blocked/child [count] |
|---|---|---|---|---|---|---|
| CPU Bound | SCHED_OTHER | 1.284 | 5.040 | 0.00350 | 689 | 2 |
| | | 1.268 | 5.044 | 0.00160 | 1309 | 2 |
| | | 1.777 | 7.065 | 0.00263 | 1845 | 2 |
| | SCHED_FIFO | 2.055 | 6.356 | 0.00200 | 6 | 2 |
| | | 1.800 | 6.536 | 0.00190 | 7 | 1 |
| | | 1.858 | 7.047 | 0.00236 | 8 | 1 |
| | SCHED_RR | 1.868 | 6.482 | 0.00200 | 24 | 2 |
| | | 1.771 | 6.594 | 0.00150 | 27 | 2 |
| | | 1.869 | 7.106 | 0.00169 | 27 | 2 |
| I/O Bound | SCHED_OTHER | 1.136 | 0.002 | 0.00250 | 2 | 302 |
| | | 0.354 | 0.000 | 0.00120 | 2 | 341 |
| | | 0.123 | 0.000 | 0.00014 | 1 | 328 |
| | SCHED_FIFO | 0.550 | 0.000 | 0.00000 | 0 | 201 |
| | | 0.240 | 0.000 | 0.00000 | 0 | 201 |
| | | 0.108 | 0.000 | 0.00007 | 0 | 267 |
| | SCHED_RR | 0.564 | 0.000 | 0.00000 | 0 | 201 |
| | | 0.198 | 0.000 | 0.00000 | 0 | 201 |
| | | 0.103 | 0.000 | 0.00011 | 0 | 283 |
| Mixed | SCHED_OTHER | 2.242 | 2.530 | 0.00550 | 356 | 497 |
| | | 0.955 | 2.521 | 0.00380 | 666 | 506 |
| | | 0.833 | 3.176 | 0.00637 | 954 | 472 |
| | SCHED_FIFO | 2.173 | 2.924 | 0.00600 | 0 | 418 |
| | | 1.116 | 3.056 | 0.00490 | 0 | 430 |
| | | 0.929 | 3.495 | 0.00550 | 4 | 357 |
| | SCHED_RR | 2.279 | 3.000 | 0.00550 | 12 | 434 |
| | | 1.142 | 3.010 | 0.00360 | 25 | 434 |
| | | 0.930 | 3.508 | 0.00453 | 38 | 329 |

# B Appendix - All Code

Listing 1: CPU Bound Benchmark `pi.c`

```
/*
 * File:        pi.c
 * Author:      Andy Sayler
 * Project:     CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/07
 * Modify Date: 2012/03/09
 * Description:
 *  This file contains a simple program for statistically
 *      calculating pi.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <errno.h>

/* Local Defines */
#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)

/* Local Functions */
inline double dist(double x0, double y0, double x1, double y1){
    return sqrt(pow((x1-x0),2) + pow((y1-y0),2));
}

inline double zeroDist(double x, double y){
    return dist(0, 0, x, y);
}

int main(int argc, char* argv[]){

    long i;
    long iterations;
    double x, y;
    double inCircle = 0.0;
    double inSquare = 0.0;
    double pCircle  = 0.0;
    double piCalc   = 0.0;

    /* Process program arguments to select iterations */
    /* Set default iterations if not supplied */
    if (argc < 2) {
        iterations = DEFAULT_ITERATIONS;
    }
```

```c
    /* Set iterations if supplied */
    else {
        iterations = atol(argv[1]);
        if (iterations < 1) {
            fprintf(stderr, "Bad iterations value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Calculate pi using statistical method across all iterations*/
    for (i = 0; i < iterations; ++i) {
        x = (random() % (RADIUS * 2)) - RADIUS;
        y = (random() % (RADIUS * 2)) - RADIUS;
        if (zeroDist(x, y) < RADIUS) {
            inCircle++;
        }
        inSquare++;
    }

    /* Finish calculation */
    pCircle = inCircle/inSquare;
    piCalc  = pCircle * 4.0;

    /* Print result */
    fprintf(stdout, "pi = %f\n", piCalc);

    return 0;
}
```

Listing 2: CPU Bound Benchmark Scheduler `pi-sched.c`

```c
/*
 * File:        pi-sched.c
 * Author:      Andy Sayler
 * Modified by: Stephen Bennett
 * Project:     CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/07
 * Modify Date: 2013/03/21
 * Description:
 *  A program for setting the desired scheduling policy and creating
 *      the desired number of child processes of pi.c
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>  // Used for fork
#include <sys/wait.h>   // Used for fork
#include <unistd.h>     // Used for fork

/* Local Defines */
#define DEFAULT_ITERATIONS 1000000
#define DEFAULT_CHILDREN   10

int main(int argc, char* argv[]){

    int     i;
    long    iterations;
    struct sched_param param;
    int     policy;
    int     children;
    pid_t   pid;

    /* Process program arguments to select iterations and policy */
    /* Set default iterations if not supplied */
    if (argc < 2) {
        iterations = DEFAULT_ITERATIONS;
    }

    /* Set default policy if not supplied */
    if (argc < 3) {
        policy = SCHED_OTHER;
    }

    /* Set default number of child processes to spawn */
    if (argc < 4) {
        children = DEFAULT_CHILDREN;
    }
```

```
    /* Set iterations if supplied */
    if (argc > 1) {
        iterations = atol(argv[1]);
        if (iterations == 0) {
            /* Set default iterations */
            iterations = DEFAULT_ITERATIONS;
        }
        else if (iterations < 0) {
            fprintf(stderr, "Bad iterations value [%li]\n", iterations);
            fprintf(stderr, "  0:  Default iterations [%d]\n", DEFAULT_ITERATIONS);
            fprintf(stderr, "  <0: Bad iterations value\n");
            fprintf(stderr, "  >0: Good iterations value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set policy if supplied */
    if (argc > 2) {
        if (!strcmp(argv[2], "SCHED_OTHER")) {
            policy = SCHED_OTHER;
        }
        else if (!strcmp(argv[2], "SCHED_FIFO")) {
            policy = SCHED_FIFO;
        }
        else if (!strcmp(argv[2], "SCHED_RR")) {
            policy = SCHED_RR;
        }
        else {
            fprintf(stderr, "Unhandeled scheduling policy [%s]\n", argv[2]);
            fprintf(stderr, "Available scheduling policies:\n");
            fprintf(stderr, "  SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set children if supplied */
    if (argc > 3) {
        children = atol(argv[3]);
        if (children == 0) {
            /* Set default iterations */
            children = DEFAULT_CHILDREN;
        }
        else if (children < 0) {
            fprintf(stderr, "Bad children value [%d]\n", children);
            fprintf(stderr, "  0:  Default children [%d]\n", DEFAULT_CHILDREN);
            fprintf(stderr, "  <0: Bad children value\n");
            fprintf(stderr, "  >0: Good children value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set process to max priority for given scheduler */
    param.sched_priority = sched_get_priority_max(policy);
```

```c
    /* Set new scheduler policy */
    fprintf(stdout, "Current Scheduling Policy: %d\n", sched_getscheduler(0));
    fprintf(stdout, "Setting Scheduling Policy to: %d\n", policy);
    if (sched_setscheduler(0, policy, &param)) {
        perror("Error setting scheduler policy");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "New Scheduling Policy: %d\n", sched_getscheduler(0));

    /* Fork children child processes */
    for (i = 0; i < children; ++i) {
        if ((pid = fork()) == -1) exit(EXIT_FAILURE);   /* Fork Failed */

        if (pid == 0) { /* Child process */
            // execl(exe, argv[0], argv[1], argv[2], ..., NULL)
            execl("pi", "pi", argv[1], NULL);
            exit(EXIT_SUCCESS);
        } else {        /* Parent process */
            printf("Forked %d pid = %d\n", i, pid);
        }
    }

    /* Wait for children child processes to finish */
    for (i = 0; i < children; ++i) {
        pid = wait(NULL);
        printf("Waited %d pid = %d\n", i+1, pid);
    }

    return 0;
}
```

Listing 3: I/O Bound Benchmark `rw.c`

```
/*
 * File:        rw.c
 * Author:      Andy Sayler
 * Project:     CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/19
 * Modify Date: 2013/03/21
 * Description:
 *  A small I/O bound program to copy N bytes from an input
 *      file to an output file. May read the input file multiple
 *      times if N is larger than the size of the input file.
 */

/* Include Flags */
#define _GNU_SOURCE

/* System Includes */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100


int main(int argc, char* argv[]){

    int  rv;
    int  inputFD;
    int  outputFD;
    char inputFilename[MAXFILENAMELENGTH];
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];

    ssize_t transfersize   = 0;
    ssize_t blocksize      = 0;
    char*   transferBuffer = NULL;
    ssize_t buffersize;

    ssize_t bytesRead         = 0;
    ssize_t totalBytesRead    = 0;
    int     totalReads        = 0;
    ssize_t bytesWritten      = 0;
```

```
    ssize_t totalBytesWritten = 0;
    int     totalWrites       = 0;
    int     inputFileResets   = 0;

    /* Process program arguments to select run-time parameters */
    /* Set supplied transfer size or default if not supplied */
    if (argc < 2) {
        transfersize = DEFAULT_TRANSFERSIZE;
    }
    else {
        transfersize = atol(argv[1]);
        if (transfersize == 0) {
            transfersize = DEFAULT_TRANSFERSIZE;
        }
        else if (transfersize < 0) {
            fprintf(stderr, "Bad transfersize value\n");
            fprintf(stderr, "  Default value = %d\n", DEFAULT_TRANSFERSIZE);
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied block size or default if not supplied */
    if (argc < 3) {
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else {
        blocksize = atol(argv[2]);
        if (blocksize == 0) {
            blocksize = DEFAULT_BLOCKSIZE;
        }
        else if (blocksize < 0) {
            fprintf(stderr, "Bad blocksize value\n");
            fprintf(stderr, "  Default value = %d\n", DEFAULT_BLOCKSIZE);
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied input filename or default if not supplied */
    if (argc < 4) {
        if (strnlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
            {
            fprintf(stderr, "Default input filename too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(inputFilename, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else {
        if (!strcmp(argv[3], "default")) {
            strncpy(inputFilename, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
        }
        else {
            if (strnlen(argv[3], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
                fprintf(stderr, "Input filename too long\n");
                exit(EXIT_FAILURE);
```

```
            }
            strncpy(inputFilename, argv[3], MAXFILENAMELENGTH);
        }
    }

    /* Set supplied output filename base or default if not supplied */
    if (argc < 5) {
        if (strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >=
           MAXFILENAMELENGTH) {
            fprintf(stderr, "Default output filename base too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
    }
    else {
        if (!strcmp(argv[4], "default")) {
            strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
                MAXFILENAMELENGTH);
        }
        else {
            if (strnlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
                fprintf(stderr, "Output filename base is too long\n");
                exit(EXIT_FAILURE);
            }
            strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
        }
    }

    /* Confirm blocksize is multiple of and less than transfersize */
    if (blocksize > transfersize) {
        fprintf(stderr, "blocksize can not exceed transfersize\n");
        exit(EXIT_FAILURE);
    }
    if (transfersize % blocksize) {
        fprintf(stderr, "blocksize must be multiple of transfersize\n");
        exit(EXIT_FAILURE);
    }

    /* Allocate buffer space */
    buffersize = blocksize;
    if (!(transferBuffer = malloc(buffersize * sizeof(*transferBuffer)))) {
        perror("Failed to allocate transfer buffer");
        exit(EXIT_FAILURE);
    }

    /* Open Input File Descriptor in Read Only mode */
    if ((inputFD = open(inputFilename, O_RDONLY | O_SYNC)) < 0) {
        perror("Failed to open input file");
        exit(EXIT_FAILURE);
    }

    /* Open Output File Descriptor in Write Only mode with standard permissions */
    rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d",
            outputFilenameBase, getpid());
```

```c
    if (rv > MAXFILENAMELENGTH) {
        fprintf(stderr, "Output filename length exceeds limit of %d characters.\n",
                MAXFILENAMELENGTH);
        exit(EXIT_FAILURE);
    }
    else if (rv < 0) {
        perror("Failed to generate output filename");
        exit(EXIT_FAILURE);
    }

    if ((outputFD =
                open(outputFilename,
                    O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH)) < 0) {
        perror("Failed to open output file");
        exit(EXIT_FAILURE);
    }

    /* Print Status */
    fprintf(stdout, "Reading from %s and writing to %s\n",
            inputFilename, outputFilename);

    /* Read from input file and write to output file*/
    do {
        /* Read transfersize bytes from input file*/
        bytesRead = read(inputFD, transferBuffer, buffersize);
        if (bytesRead < 0) {
            perror("Error reading input file");
            exit(EXIT_FAILURE);
        }
        else {
            totalBytesRead += bytesRead;
            totalReads++;
        }

        /* If all bytes were read, write to output file*/
        if (bytesRead == blocksize) {
            bytesWritten = write(outputFD, transferBuffer, bytesRead);
            if (bytesWritten < 0) {
                perror("Error writing output file");
                exit(EXIT_FAILURE);
            }
            else {
                totalBytesWritten += bytesWritten;
                totalWrites++;
            }
        }
        /* Otherwise assume we have reached the end of the input file and reset */
        else {
            if (lseek(inputFD, 0, SEEK_SET)) {
                perror("Error resetting to beginning of file");
                exit(EXIT_FAILURE);
            }
            inputFileResets++;
```

```
        }

    } while (totalBytesWritten < transfersize);

    /* Output some possibly helpful info to make it seem like we were doing stuff
       */
    fprintf(stdout, "Read:    %zd bytes in %d reads\n",
            totalBytesRead, totalReads);
    fprintf(stdout, "Written: %zd bytes in %d writes\n",
            totalBytesWritten, totalWrites);
    fprintf(stdout, "Read input file in %d pass%s\n",
            (inputFileResets + 1), (inputFileResets ? "es" : ""));
    fprintf(stdout, "Processed %zd bytes in blocks of %zd bytes\n",
            transfersize, blocksize);

    /* Free Buffer */
    free(transferBuffer);

    /* Close Output File Descriptor */
    if (close(outputFD)) {
        perror("Failed to close output file");
        exit(EXIT_FAILURE);
    }

    /* Close Input File Descriptor */
    if (close(inputFD)) {
        perror("Failed to close input file");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Listing 4: I/O Bound Benchmark Scheduler `rw-sched.c`

```c
/*
 * File:        rw-sched.c
 * Author:      Stephen Bennett
 * Project:     CSCI 3753 Programming Assignment 3
 * Create Date: 2013/03/21
 * Modify Date: 2013/03/21
 * Description:
 *  A program for setting the desired scheduling policy and creating
 *      the desired number of child processes of rw.c.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>  // Used for fork
#include <sys/wait.h>   // Used for fork
#include <unistd.h>     // Used for fork

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEFAULT_CHILDREN 10

int main(int argc, char* argv[]){

    ssize_t transfersize = 0;
    ssize_t blocksize    = 0;
    char    inputFilenameBase[MAXFILENAMELENGTH];
    char    inputFilename[MAXFILENAMELENGTH];
    char    outputFilenameBase[MAXFILENAMELENGTH];
    int     policy;
    int     children;

    struct sched_param param;
    int    i;
    int    rv;
    pid_t  pid;

    /* Process program arguments to select run-time parameters */
    /* Set supplied transfer size or default if not supplied */
    if (argc < 2) {
        transfersize = DEFAULT_TRANSFERSIZE;
    }
    else {
        transfersize = atol(argv[1]);
```

```
        if (transfersize == 0) {
            transfersize = DEFAULT_TRANSFERSIZE;
        }
        else if (transfersize < 0) {
            fprintf(stderr, "Bad transfersize value\n");
            fprintf(stderr, "  Default value = %d\n", DEFAULT_TRANSFERSIZE);
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied block size or default if not supplied */
    if (argc < 3) {
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else {
        blocksize = atol(argv[2]);
        if (blocksize == 0) {
            blocksize = DEFAULT_BLOCKSIZE;
        }
        else if (blocksize < 0) {
            fprintf(stderr, "Bad blocksize value\n");
            fprintf(stderr, "  Default value = %d\n", DEFAULT_BLOCKSIZE);
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied input filename or default if not supplied */
    if (argc < 4) {
        if (strnlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
            {
            fprintf(stderr, "Default input filename too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(inputFilenameBase, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else {
        if (!strcmp(argv[3], "default")) {
            strncpy(inputFilenameBase, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
        }
        else {
            if (strnlen(argv[3], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
                fprintf(stderr, "Input filename too long\n");
                exit(EXIT_FAILURE);
            }
            strncpy(inputFilenameBase, argv[3], MAXFILENAMELENGTH);
        }
    }

    /* Set supplied output filename base or default if not supplied */
    if (argc < 5) {
        if (strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >=
           MAXFILENAMELENGTH) {
            fprintf(stderr, "Default output filename base too long\n");
            exit(EXIT_FAILURE);
```

```c
        }
        strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
    }
    else {
        if (!strcmp(argv[4], "default")) {
            strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
                MAXFILENAMELENGTH);
        }
        else {
            if (strnlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
                fprintf(stderr, "Output filename base is too long\n");
                exit(EXIT_FAILURE);
            }
            strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
        }
    }

    /* Set policy or default if not supplied */
    if (argc < 6) {
        policy = SCHED_OTHER;
    }
    else {
        if (!strcmp(argv[5], "SCHED_OTHER")) {
            policy = SCHED_OTHER;
        }
        else if (!strcmp(argv[5], "SCHED_FIFO")) {
            policy = SCHED_FIFO;
        }
        else if (!strcmp(argv[5], "SCHED_RR")) {
            policy = SCHED_RR;
        }
        else {
            fprintf(stderr, "Unhandeled scheduling policy [%s]\n", argv[5]);
            fprintf(stderr, "Available scheduling policies:\n");
            fprintf(stderr, "  SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set number of child processes to spawn or default if not supplied */
    if (argc < 7) {
        children = DEFAULT_CHILDREN;
    }
    else {
        children = atol(argv[6]);
        if (children == 0) {
            /* Set default iterations */
            children = DEFAULT_CHILDREN;
        }
        else if (children < 0) {
            fprintf(stderr, "Bad children value [%d]\n", children);
            fprintf(stderr, "  0:  Default children [%d]\n", DEFAULT_CHILDREN);
            fprintf(stderr, "  <0: Bad children value\n");
            fprintf(stderr, "  >0: Good children value\n");
```

```
            exit(EXIT_FAILURE);
        }
    }

    /* Set process to max priority for given scheduler */
    param.sched_priority = sched_get_priority_max(policy);

    /* Set new scheduler policy */
    fprintf(stdout, "Current Scheduling Policy: %d\n", sched_getscheduler(0));
    fprintf(stdout, "Setting Scheduling Policy to: %d\n", policy);
    if (sched_setscheduler(0, policy, &param)) {
        perror("Error setting scheduler policy");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "New Scheduling Policy: %d\n", sched_getscheduler(0));

    /* Fork children child processes */
    for (i = 0; i < children; ++i) {
        rv = snprintf(inputFilename, MAXFILENAMELENGTH, "%s-%d",
                inputFilenameBase, i+1);
        if (rv > MAXFILENAMELENGTH) {
            fprintf(stderr, "Output filename length exceeds limit of %d characters
                .\n",
                    MAXFILENAMELENGTH);
            exit(EXIT_FAILURE);
        }
        else if (rv < 0) {
            perror("Failed to generate output filename");
            exit(EXIT_FAILURE);
        }

        if ((pid = fork()) == -1) exit(EXIT_FAILURE);   /* Fork Failed */

        if (pid == 0) { /* Child process */
            // execl(exe, argv[0], argv[1], argv[2], ..., NULL)
            execl("rw", "rw", argv[1], argv[2], inputFilename, argv[4], NULL);
            exit(EXIT_SUCCESS);
        } else {        /* Parent process */
            printf("Forked %d pid = %d\n", i, pid);
        }
    }

    /* Wait for children child processes to finish */
    for (i = 0; i < children; ++i) {
        pid = wait(NULL);
        printf("Waited %d pid = %d\n", i+1, pid);
    }

    return 0;
}
```

Listing 5: Mixed Benchmark `mix.c`

```
/*
 * File:        mix.c
 * Author:      Stephen Bennett
 * Project:     CSCI 3753 Programming Assignment 3
 * Create Date: 2013/03/21
 * Modify Date: 2013/03/21
 * Description:
 *  This file contains a small program for statistically calculating pi
 *      as well as writing the intermediate results of this calculation
 *      to a file.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>     // getpid()
#include <errno.h>
#include <fcntl.h>      // open()
#include <string.h>
#include <math.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_OUTPUTFILENAMEBASE "mixoutput"
#define DEFAULT_ITERATIONS 10000
#define DEFAULT_INTERMEDIATERESULTS 100
#define RADIUS (RAND_MAX / 2)

/* Local Functions */
inline double dist(double x0, double y0, double x1, double y1){
    return sqrt( pow((x1 - x0), 2) + pow((y1 - y0), 2) );
}

inline double zeroDist(double x, double y){
    return dist(0, 0, x, y);
}

int main(int argc, char* argv[]) {

    /* Parameters */
    long intermediateResults;
    long iterations;
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];

    /* I/O Bound Variables */
    int  rv;
    int  outputFD;
    char writeBuffer[78];

    /* CPU Bound Variables */
```

```
long   i, j;
double x, y;
double inCircle = 0.0;
double inSquare = 0.0;
double pCircle  = 0.0;
double piCalc   = 0.0;

/* Process program arguments to run-time parameters */
/* Set intermediate results or default if not supplied */
if (argc < 2) {
    intermediateResults = DEFAULT_INTERMEDIATERESULTS;
}
else {
    intermediateResults = atol(argv[1]);
    if (intermediateResults== 0) {
        intermediateResults = DEFAULT_INTERMEDIATERESULTS;
    }
    else if (intermediateResults < 0) {
        fprintf(stderr, "Bad intermediate results value [%li]\n",
           intermediateResults);
        fprintf(stderr, "   0:  Default intermediate results [%d]\n",
           DEFAULT_INTERMEDIATERESULTS);
        fprintf(stderr, "  <0: Bad intermediate results value\n");
        fprintf(stderr, "  >0: Good intermediate results value\n");
        exit(EXIT_FAILURE);
    }
}

/* Set iterations or default if not supplied */
if (argc < 3) {
    iterations = DEFAULT_ITERATIONS;
}
else {
    iterations = atol(argv[2]);
    if (iterations == 0) {
        iterations = DEFAULT_ITERATIONS;
    }
    else if (iterations < 0) {
        fprintf(stderr, "Bad iterations value [%li]\n", iterations);
        fprintf(stderr, "   0:  Default iterations [%d]\n", DEFAULT_ITERATIONS);
        fprintf(stderr, "  <0: Bad iterations value\n");
        fprintf(stderr, "  >0: Good iterations value\n");
        exit(EXIT_FAILURE);
    }
}

/* Set supplied output filename base or default if not supplied */
if (argc < 4) {
    if (strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >=
        MAXFILENAMELENGTH) {
        fprintf(stderr, "Default output filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
```

```
    }
    else {
        if (!strcmp(argv[3], "default")) {
            strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
                MAXFILENAMELENGTH);
        }
        else {
            if (strnlen(argv[3], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
                fprintf(stderr, "Output filename base is too long\n");
                exit(EXIT_FAILURE);
            }
            strncpy(outputFilenameBase, argv[3], MAXFILENAMELENGTH);
        }
    }

    /* Open Output File Descriptor in Write Only mode with standard permissions */
    rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d",
            outputFilenameBase, getpid());
    if (rv > MAXFILENAMELENGTH) {
        fprintf(stderr, "Output filename length exceeds limit of %d characters.\n",
                MAXFILENAMELENGTH);
        exit(EXIT_FAILURE);
    }
    else if (rv < 0) {
        perror("Failed to generate output filename");
        exit(EXIT_FAILURE);
    }

    if ((outputFD =
                open(outputFilename,
                    O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH)) < 0) {
        perror("Failed to open output file");
        exit(EXIT_FAILURE);
    }

    for (j = 0; j < intermediateResults; ++j) {
        /* Calculate pi using statistical method across all iterations*/
        for (i = 0; i < iterations; ++i) {
            x = (random() % (RADIUS * 2)) - RADIUS;
            y = (random() % (RADIUS * 2)) - RADIUS;
            if (zeroDist(x, y) < RADIUS) {
                inCircle++;
            }
            inSquare++;
        }
        /* Finish calculation */
        pCircle = inCircle/inSquare;
        piCalc  = pCircle * 4.0;

        snprintf(writeBuffer, 78, "In Circle Count = %-58f\n", inCircle);
        write(outputFD, writeBuffer, 77);
        snprintf(writeBuffer, 78, "In Square Count = %-58f\n", inSquare);
        write(outputFD, writeBuffer, 77);
```

```
        snprintf(writeBuffer, 78, "Calculated pi  = %-58f\n\n", piCalc);
        write(outputFD, writeBuffer, 77);
    }

    /* Close Output File Descriptor */
    if (close(outputFD)) {
        perror("Failed to close output file");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Listing 6: Mixed Benchmark Scheduler `mix-sched.c`

```
/*
 * File:         mix-sched.c
 * Author:       Stephen Bennett
 * Project:      CSCI 3753 Programming Assignment 3
 * Create Date: 2013/03/21
 * Modify Date: 2013/03/21
 * Description:
 *  A program for setting the desired scheduling policy and creating
 *      the desired number of child processes of mix.c.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>  // Used for fork
#include <sys/wait.h>   // Used for fork
#include <unistd.h>     // Used for fork

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEFAULT_CHILDREN 10

int main(int argc, char* argv[]){

    /* Parameters */
    int policy;
    int children;

    /* Scheduler/Fork Variables */
    struct sched_param param;
    int    i;
    pid_t  pid;

    /* Process program arguments to select run-time parameters */
    /* Set policy or default if not supplied */
    if (argc < 5) {
        policy = SCHED_OTHER;
    }
    else {
        if (!strcmp(argv[4], "SCHED_OTHER")) {
            policy = SCHED_OTHER;
        }
        else if (!strcmp(argv[4], "SCHED_FIFO")) {
            policy = SCHED_FIFO;
```

```
        }
        else if (!strcmp(argv[4], "SCHED_RR")) {
            policy = SCHED_RR;
        }
        else {
            fprintf(stderr, "Unhandeled scheduling policy [%s]\n", argv[4]);
            fprintf(stderr, "Available scheduling policies:\n");
            fprintf(stderr, "  SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set number of child processes to spawn or default if not supplied */
    if (argc < 6) {
        children = DEFAULT_CHILDREN;
    }
    else {
        children = atol(argv[5]);
        if (children == 0) {
            /* Set default iterations */
            children = DEFAULT_CHILDREN;
        }
        else if (children < 0) {
            fprintf(stderr, "Bad children value [%d]\n", children);
            fprintf(stderr, "  0:  Default children [%d]\n", DEFAULT_CHILDREN);
            fprintf(stderr, "  <0: Bad children value\n");
            fprintf(stderr, "  >0: Good children value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set process to max priority for given scheduler */
    param.sched_priority = sched_get_priority_max(policy);

    /* Set new scheduler policy */
    fprintf(stdout, "Current Scheduling Policy: %d\n", sched_getscheduler(0));
    fprintf(stdout, "Setting Scheduling Policy to: %d\n", policy);
    if (sched_setscheduler(0, policy, &param)) {
        perror("Error setting scheduler policy");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "New Scheduling Policy: %d\n", sched_getscheduler(0));

    /* Fork children child processes */
    for (i = 0; i < children; ++i) {
        if ((pid = fork()) == -1) exit(EXIT_FAILURE);   /* Fork Failed */

        if (pid == 0) { /* Child process */
            execv("mix", argv);
            exit(EXIT_SUCCESS);
        } else {        /* Parent process */
            printf("Forked %d pid = %d\n", i, pid);
        }
    }
```

```c
    /* Wait for children child processes to finish */
    for (i = 0; i < children; ++i) {
        pid = wait(NULL);
        printf("Waited %d pid = %d\n", i, pid);
    }

    return 0;
}
```

Listing 7: Bash script to run tests `testscript`

```
#!/bin/bash

# File:          testscript
# Author:        Stephen Bennett
# Project:       CSCI 3753 Programming Assignment 3
# Create Date:   2013/03/21
# Modify Date:   2013/03/21
# Description:
#   A bash script to run each test case and gather the relevant data.

techo()
{
    echo "$1" | tee -a "$RESFILE"
}

LOW=10
MEDIUM=50
HIGH=500
RESFILE="results.txt"
ITERATIONS=100000000
BYTESTOCOPY=102400
BLOCKSIZE=1024
INTERMEDIATERESULTS=50
ITERATIONSPERWRITE=1000000
TIMEFORMAT="wall=%e, user=%U, system=%S, CPU=%P, preempted=%c, blocked=%w"
MAKE="make -s"

printf "Building code...\n"
$MAKE clean
$MAKE

printf "Making $HIGH copies of rwinput so each process has its own identical input
    file...\n"
for i in $(seq 1 $HIGH); do cp rwinput rwinput-$i; done
echo

printf "Starting test runs...\n\n"

for SCHED in SCHED_OTHER SCHED_FIFO SCHED_RR; do
    techo "
        ==============================================================================
        "
    techo "TEST :: CPU BOUND"
    techo "SCHEDULER       = $SCHED"
    techo "ITERATIONS      = $ITERATIONS"
    techo ""
    for CHILDREN in $LOW $MEDIUM $HIGH; do
        techo "CHILD PROCESSES = $CHILDREN"
        /usr/bin/time -o "$RESFILE" -a -f "$TIMEFORMAT" ./pi-sched $ITERATIONS
            $SCHED $CHILDREN > /dev/null
        #{ /usr/bin/time -f "$TIMEFORMAT" ./pi-sched $ITERATIONS $SCHED $CHILDREN >
```

```
            /dev/null ; } 2>> temp.txt
        techo ""
    done
done

for SCHED in SCHED_OTHER SCHED_FIFO SCHED_RR; do
    techo "
       ==============================================================================
       "
    techo "TEST :: I/O BOUND"
    techo "SCHEDULER         = $SCHED"
    techo "BYTES TO COPY     = $BYTESTOCOPY"
    techo "BLOCK SIZE        = $BLOCKSIZE"
    techo ""
    for CHILDREN in $LOW $MEDIUM $HIGH; do
        techo "CHILD PROCESSES = $CHILDREN"
        /usr/bin/time -o "$RESFILE" -a -f "$TIMEFORMAT" ./rw-sched $BYTESTOCOPY
           $BLOCKSIZE default default $SCHED $CHILDREN > /dev/null
        techo ""
    done
done

for SCHED in SCHED_OTHER SCHED_FIFO SCHED_RR; do
    techo "
       ==============================================================================
       "
    techo "TEST :: MIXED"
    techo "SCHEDULER         = $SCHED"
    techo "INTERMED WRITES = $INTERMEDIATERESULTS"
    techo "ITERATIONS/WRITE= $ITERATIONSPERWRITE"
    techo ""
    for CHILDREN in $LOW $MEDIUM $HIGH; do
        techo "CHILD PROCESSES = $CHILDREN"
        /usr/bin/time -o "$RESFILE" -a -f "$TIMEFORMAT" ./mix-sched
           $INTERMEDIATERESULTS $ITERATIONSPERWRITE default $SCHED $CHILDREN > /dev
           /null
        techo ""
    done
done
```

Listing 8: Build file `Makefile`

```
CC = gcc
CFLAGS = -c -g -Wall -Wextra
LFLAGS = -g -Wall -Wextra

INPUTFILESIZEMEGABYTES = 1

KILO = 1024
MEGA = $(shell echo $(KILO)\*$(KILO) | bc)
INPUTFILESIZEBYTES = $(shell echo $(MEGA)\*$(INPUTFILESIZEMEGABYTES) | bc)
INPUTBLOCKSIZEBYTES = $(KILO)
INPUTBLOCKS = $(shell echo $(INPUTFILESIZEBYTES)\/$(INPUTBLOCKSIZEBYTES) | bc)

EXECUTABLES = pi pi-sched rw rw-sched mix mix-sched rr_quantum
OBJECTS = $(EXECUTABLES:%=%.o)
#OBJECTS = $(foreach exe,$(EXECUTABLES),$(exe).o)

.PHONY: all clean

all: $(EXECUTABLES)

pi: pi.o
	$(CC) $(LFLAGS) $^ -o $@ -lm

pi-sched: pi-sched.o pi
	$(CC) $(LFLAGS) $@.o -o $@ -lm

rw: rw.o rwinput
	$(CC) $(LFLAGS) $@.o -o $@ -lm

rw-sched: rw-sched.o rw
	$(CC) $(LFLAGS) $@.o -o $@ -lm

mix: mix.o
	$(CC) $(LFLAGS) $^ -o $@ -lm

mix-sched: mix-sched.o mix
	$(CC) $(LFLAGS) $@.o -o $@ -lm

rr_quantum: rr_quantum.o
	$(CC) $(LFLAGS) $^ -o $@ -lm

rwinput: Makefile
	dd if=/dev/urandom of=./rwinput bs=$(INPUTBLOCKSIZEBYTES) count=$(
	   INPUTBLOCKS)

%.o: %.c
	$(CC) $(CFLAGS) $<

clean: testclean
	rm -f $(EXECUTABLES)
	rm -f *.o
```

```
	rm -f *~
	rm -f handout/*~
	rm -f handout/*.log
	rm -f handout/*.aux

testclean:
	rm -f mixoutput*
	rm -f rwoutput*
	rm -f rwinput*
```