Linux Scheduling

CSCI 3753 Programming Assignment 3

Conrad Hougen
Spring 2014
Professor Shivakant Mishra

Abstract

Historically, the Linux kernel has used several different algorithms for scheduling processes on the CPU, including an O(n) scheduler in Linux 2.4, an O(1) scheduler from Linux 2.6 to 2.6.22, and now the Completely Fair Scheduler, abbreviated CFS [6]. Current mainline distributions of Linux implement the CFS scheduling policy which is used to schedule all non-real-time processes running on the machine. As the name suggests, CFS is optimized for scheduling processes on time-sharing systems to allocate CPU time to processes in a fair way, where no single process may dominate use of the CPU. In addition, Linux provides two soft real-time scheduling policies, known as First-In-First-Out, or FIFO, and Round-Robin, abbreviated RR. Processes scheduled under these two real-time policies are given higher priority than those scheduled under the CFS scheduling class, but hard real-time is not implemented, meaning that there can be no guarantee of process deadlines being met. As laid out in this paper, I seek to investigate the behaviors of each of the three scheduling policies mentioned above by designing and running benchmarks that will reveal the effectiveness of each policy while running CPU-bound, I/O-bound, and mixed-type processes, as measured by variables including turnaround time and overhead efficiency [8]. I find that CFS performs surprisingly efficiently for CPU-bound processes while RR performs best for mixed-type processes. For I/O bound processes, the most efficient policy seems to depend on the number of processes running on the machine.

1. Introduction

CFS, known as SCHED_OTHER in Linux, makes scheduling decisions based on a red-black tree of waiting processes which are sorted based on a variable called the virtual runtime, or vruntime [2]. Vruntime is associated with each individual process and increases for a process when the process executes on the CPU. Specifically, the vruntime statistic is the total number of nanoseconds that a task has spent running on a CPU, weighted by its niceness. Note that a nice process is one which does not hog the CPU, implying that a nice process is more likely to perform an involuntary context switch and block on I/O rather than perform arithmetic computations on the CPU until it is preempted. Therefore, when using the CFS scheduler, I/O-bound processes will be given higher priority than CPU-bound processes.

The FIFO scheduling policy, also referred to as SCHED_FIFO, implements a queue per priority for scheduling processes based on arrival time in the system. In other words, processes follow the following rules [7]. First, a process which is preempted by a higher priority task will stay at the head of the list for its priority to execute again when all higher-priority processes are blocked. Second, processes entering the system at a specific priority will be placed at the end of the queue for that priority. Finally, a process scheduled under the FIFO policy will run until it is either blocked by an I/O request, preempted by a higher-priority process, or yields the CPU voluntarily through a sleep or yield system call.

Round-Robin, or the SCHED_RR scheduling policy, could be described as a generalization or simple enhancement of the FIFO policy, except that each process is only allowed to run for a limited time quantum [7]. Each priority under the RR scheduling class contains a queue of waiting processes which are initially ordered by arrival time. The process at the front of the list at the highest priority is scheduled next. However, unlike in FIFO scheduling, where processes hog as much CPU time as possible before they block on an I/O request or are preempted by a higher-priority process, processes under RR scheduling move to the end of the queue after running for a maximum time quantum. Of course, processes under RR may also be preempted by higher-priority processes or may block on an I/O request before exhausting the allotted time quantum which also causes the process to relinquish the CPU. The effects of these algorithmic decisions will be made clear in the ensuing analysis.

2. Experimental Method

As explained in section 1, I am interested in running benchmarks to compare three different Linux scheduler policies:

- SCHED_OTHER (aka SCHED_NORM)
- SCHED_FIFO
- SCHED_RR

In order to isolate behaviors of each algorithm with respect to different operating conditions, I design three simple C programs, each of which will serve as one of the following three types of processes:

- CPU-Bound
- Mixed (CPU and I/O)
- I/O-Bound

My objective here is to determine how the behavior of each scheduling policy changes based on the number of CPU operations performed versus the number of I/O operations performed by programs running on the machine. In an average PC computing environment, most processes are a mix of CPU and I/O operations, but the ratio of CPU to I/O operations changes depending on the specific execution state of a machine. By breaking processes into these three categories, I hope to isolate the advantages or disadvantages of each policy in an environment where as many variables are controlled as possible.

Finally, I also would like to investigate the behavior of each scheduling policy and process type with respect to system utilization, or the number of processes running on the machine. This data will be important in determining how well each scheduling algorithm scales under increasing process loads. I define the following three load categories and will refer to these definitions in the sequel:

- Low (10 processes)
- Medium (60 processes)
- High (200 processes)

Three policies combined with three process types and three load categories produce a total of 27 different benchmarks that I designed and executed. The next step is to determine how the different benchmarks will be evaluated.

I have chosen to concentrate on the following properties of the three scheduling algorithms, which vary depending on process type and load:

1. Turnaround time: the time between when a process enters the system and completes its execution
2. Context switches: the number of voluntary and involuntary context switches made during execution as a function of process type and load for each policy
3. CPU Usage: a percentage of CPU time used by each benchmark process group

I determined that the variables above were most enlightening for answering the questions addressed in this paper which I clarify here (See [8]):

1. Which scheduling policy is best suited for each process type in terms of run-time and overhead efficiency?

2. How does each scheduling policy scale?
3. What are some pros and cons of each scheduling policy?

Linux provides a tool called "time" which allows a user to collect statistics on processes including: elapsed real time in seconds which I will call turnaround time, time spent in user mode, time spent in kernel mode, percentage of CPU used by the job, number of involuntary context switches, and number of voluntary context switches.

The testing environment is on a virtual machine running Ubuntu 12.04 LTS provided by the Computer Science department at the University of Colorado – Boulder [9]. Specifics about the physical machine and virtual machine are listed below.

Physical Machine: Dell Inspiron N5010

- CPU: Intel® Core™ i3-380M @ 2.53GHz (2 Cores/4 Threads)
- Memory: 4.00 GB DDR3 (3.80 GB usable)
- OS: Windows 7 Home Premium (64-bit)
- Secondary Storage: 500 GB SATA - 3 Gb/s HDD

Virtual Machine: Virtual Box CU-CS-VM [9]

- Processor(s): 4 CPUs, Execution Cap: 100%
- Base Memory: 2.00 GB
- OS: Ubuntu 12.04 LTS (Precise Pangolin)
- Secondary Storage: 30 GB SATA Port 0 HDD

I implemented three simple programs in C, modified from pi-sched.c and rw.c written by Andy Sayler. pi-sched.c implements a CPU-bound program which calculates digits of pi using a statistical method with increasing precision under higher numbers of loop iterations. rw.c implements an I/O-bound program which transfers bytes from an input file to an output file in blocks. The third program, contained in mixed.c, provides a simple program that mixes the operations from pi-sched.c and rw.c in such a way that the resulting processes use around 200% CPU time out of a maximum of 400% (4 cores on the VM). I reasoned that a mixed program should spend only half the maximum CPU time and wait for I/O the other half of the time. I adjusted the mixed program slightly after observing preliminary results and noticing that the CPU time was too high, on average.

In order to establish experimental integrity between the three types of programs, I designed each with a similar structure. Each program takes in several command line arguments including parameters such as the number of loop iterations for calculating pi or the number of bytes to transfer. Furthermore, each program uses a command line argument to determine how many processes to run, managed by a loop which forks child processes. After forking N child processes to perform the program in question, the parent waits and reaps all children, collecting information on process-specific data using the Linux wait4 system call. I concentrated on higher-level data collected by the time utility and used process-specific data mainly as a way to check for irregular or unexpected behavior.

Finally, I modified a bash script provided by Andy Sayler in order to execute all 27 tests in five trials for a total of 135 tests. I set an ITERATIONS parameter to 100,000,000 for pi-sched.c, which causes the pi calculation to run through 100,000,000 iterations in a for loop before announcing the final answer. For the mixed program, I used MIXED_ITER of 1,000,000 for mixed.c, causing the pi calculation to run through that many iterations of its for loop. Notice that this is different than pi-sched.c because I have written mixed.c to write intermediate results to a text file. Also, I set BYTESTOCOPY to 1,048,576 and BLOCKSIZE to 1,024 for rw.c, meaning that the program will transfer a total of 1MB between an input file and output file, each transfer in a block of 1KB. See Appendix B for more details on specifics of the source code. I chose each of these parameters in an attempt to create runtimes in the same ballpark for all three types of processes. In order to isolate the tests from the effects of other processes running on the machine, I closed all applications except for Virtual Box, and disconnected from the internet. On the Virtual Machine itself, I closed all applications except for a single terminal, and also disconnected virtual wired network connection. I executed the bash script overnight with a total execution time around 12 hours. As mentioned above, I utilized the time utility in my bash script with the following format:

/usr/bin/time -f "wall=%e user=%U system=%S CPU=%P i-switched=%c v-switched=%w

I also redirected the output from the time command five text files for each of the five trials.

2.1 Multi-Core Experimental Method

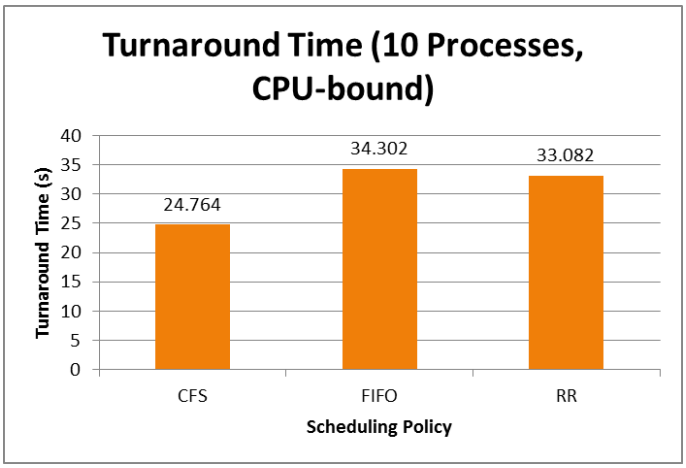
In addition to running 135 tests with the system configuration described above, I also decided to investigate the effects of core count on the three different scheduling policies. The virtual machine environment allows for easy system reconfiguration to simulate different numbers of cores on the machine, and was well suited for running these tests. I ran all 27 test configurations on core counts from one to four for an additional 108 tests. Again, see the appendix for raw data.

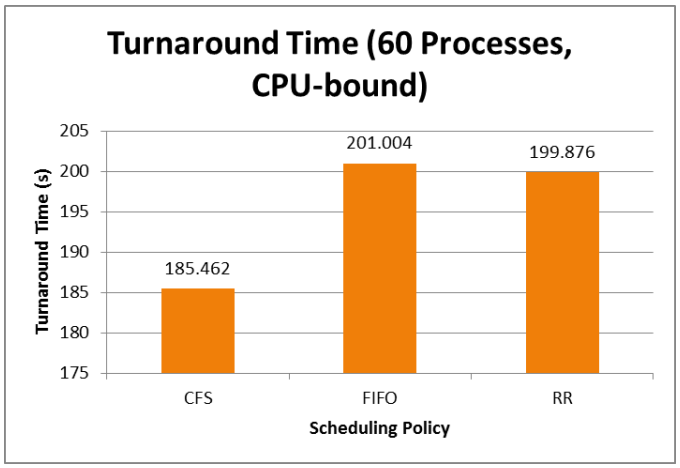
3. Results

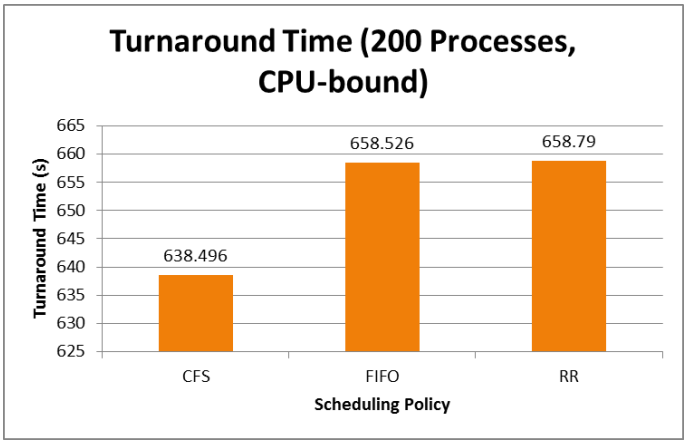
After completing all trials, I ported the collected data to Excel for analysis (see Appendix A for raw data). I first averaged the resulting data over the five trials for each of the 27 scenarios in order to mitigate any random effects that may have skewed the data over a single trial. This means that all data displayed in the following graphs are the results after averaging.

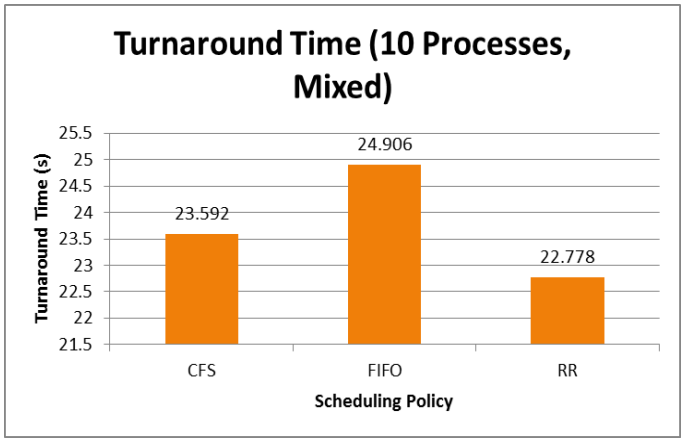
Referring to question 1 in section 2, I wanted to make a direct comparison between each of the three policies in order to determine if any of the policies were well-suited to a particular type of process. I also wanted to answer question 2 which asks how well each of the algorithms scale for larger numbers of processes. Therefore, I created nine different column charts, one for each process type and process load combination, and plotted the turnaround times for each policy.

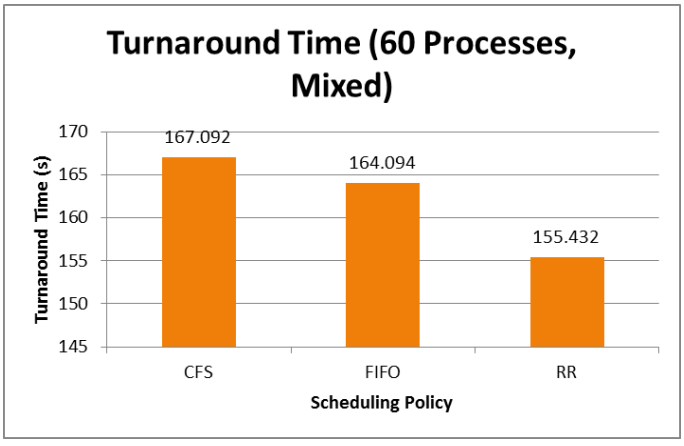
First, I noticed an intriguing trend between CPU-bound processes and CFS turnaround time as shown in the first three graphs below. Interestingly, the CFS algorithm averaged the lowest turnaround time for all three process loads under the CPU-bound process type. This result appears counterintuitive at first because the CFS scheduler is O(log(n)) for selecting a process to run (recall that CFS uses a red-black tree to store waiting processes). However, looking at the raw data, this result does not seem to be a fluke since the turnaround time was nearly always least for CFS throughout all trials.

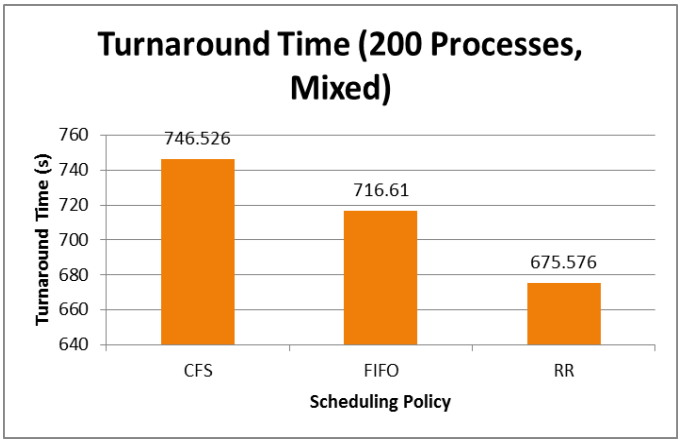








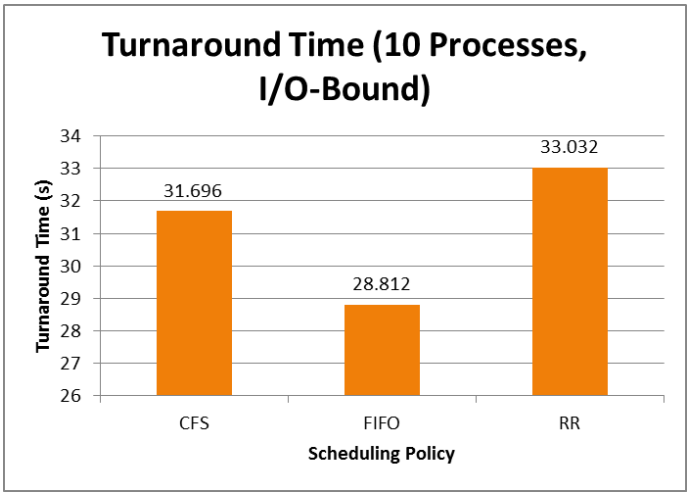


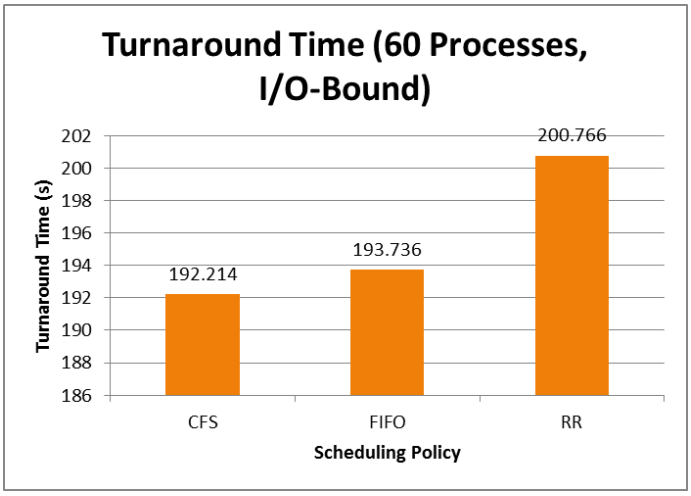


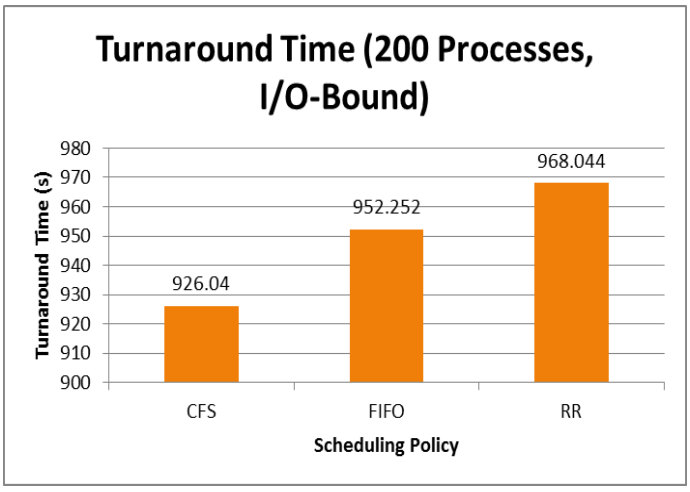
Throughout all three process loads under the CPU-bound process type, the FIFO and RR policies appear to nearly match, with RR edging out FIFO for 10 processes and 60 processes, but taking slightly longer than FIFO when running 200 CPU-bound processes. This may indicate that FIFO scales better than RR for CPU-bound processes, but on the other hand, the differences are small enough that the data appears inconclusive, and I believe more testing must be done to come to a consensus.

For the mixed process type (charts shown above), the RR policy seems to trump the other two in all three process loads. Meanwhile, it appears that CFS may not scale well with larger numbers of mixed processes since the difference in turnaround times between CFS and FIFO seems to increase disproportionately with increasing numbers of processes. However, these results cannot yet be definitive due to large variations in turnaround time per trial (see Appendix A).

For the I/O process type, none of the three policies ran with definitely lower turnaround time, but the data indicates some interesting scaling effects with increasing process loads. Also, the RR policy had the longest turnaround time of the three policies over all loads.



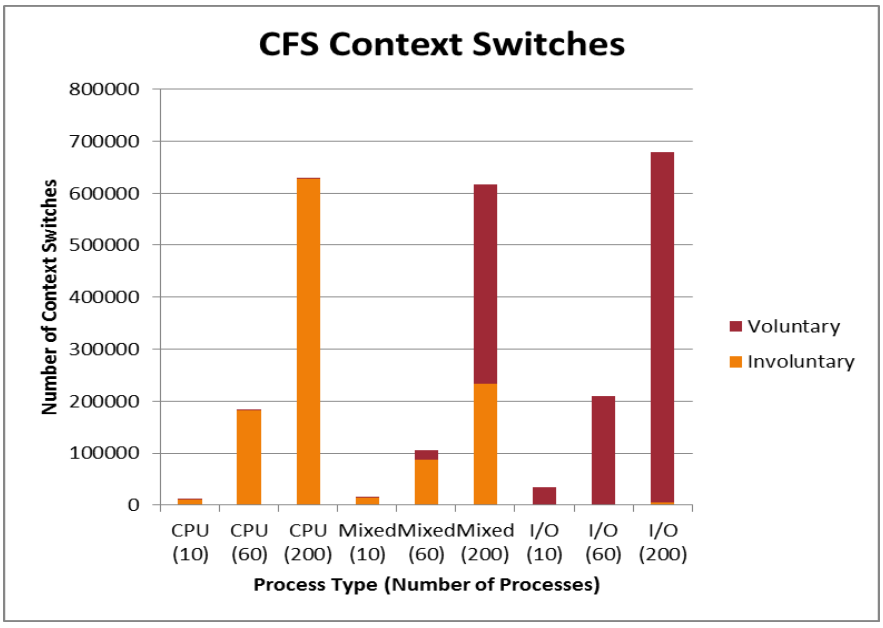


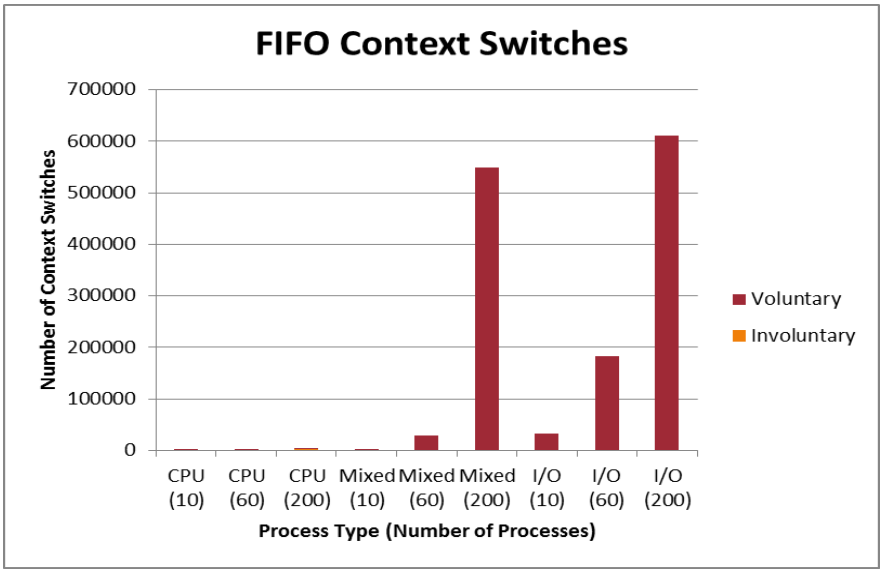


For low numbers of processes, FIFO appears best-suited to I/O-bound processes, but it soon loses its edge to CFS by the time we increase to 60 processes. At a high process load of 200 processes, CFS only increases its advantage over both FIFO and RR, seeming to indicate that CFS scales much better for I/O than does the FIFO algorithm. RR does not seem to perform well regardless of process load. This result is surprising initially since RR performed best while executing mixed-type processes, therefore implying that RR minimizes turnaround

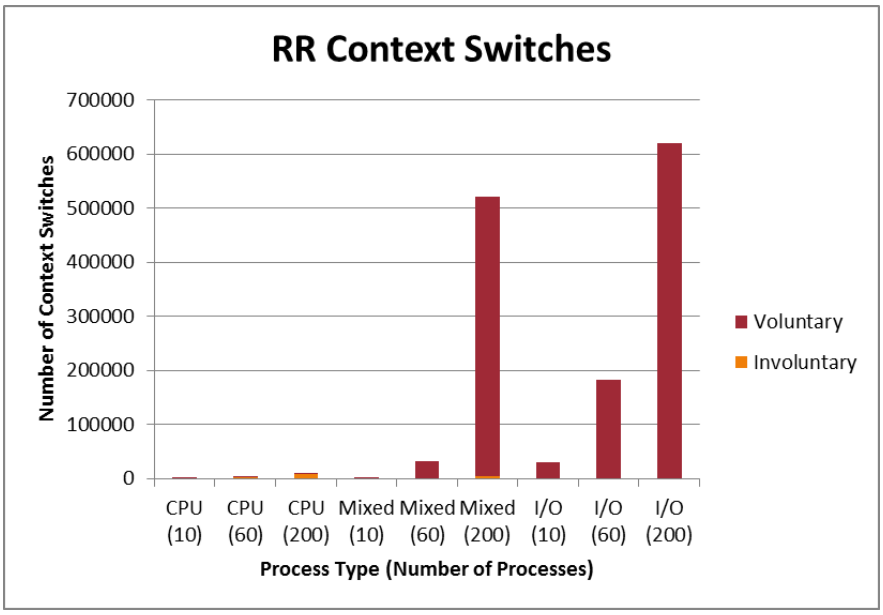
time for an average amount of I/O but cannot handle processes which perform too large a number of I/O operations.

Next, I looked at overhead by observing context switch behavior. The time utility collects integer numbers of both voluntary switches (from blocking on I/O) and involuntary switches (preemption). In general, one would expect larger numbers of voluntary context switches for mixed and I/O-bound processes, and one would also expect small numbers of involuntary context switches for FIFO scheduling in general due to the fact that the FIFO policy is non-preemptive. Indeed the following charts seem to assert the validity of these assumptions.



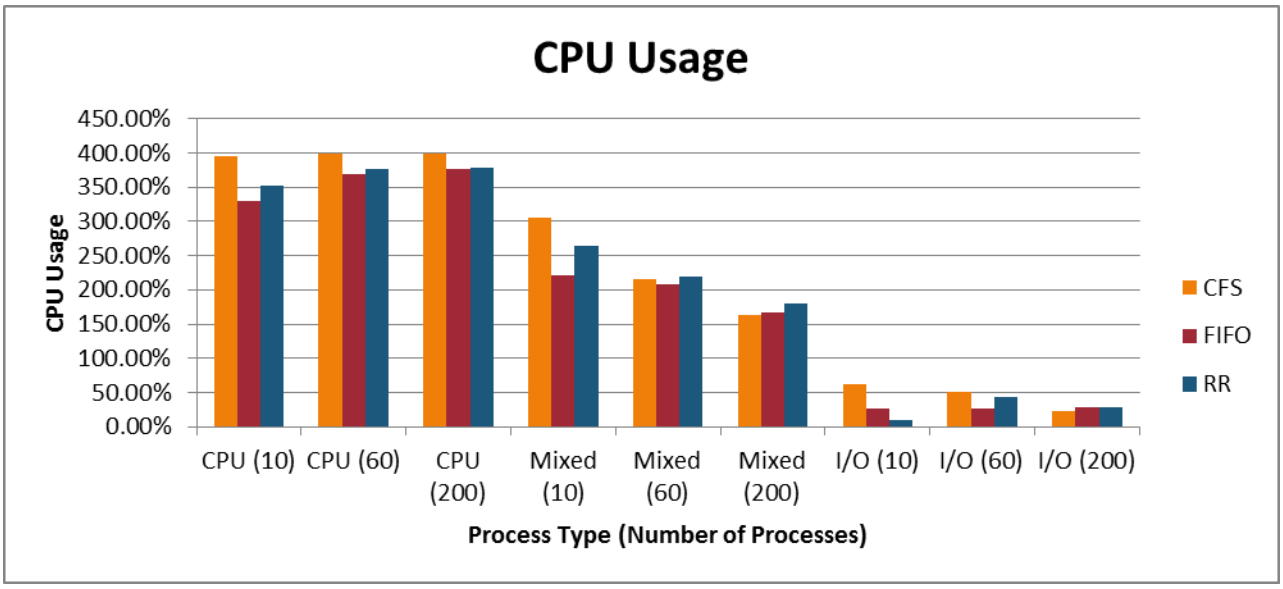


Perhaps unsurprisingly, I observed that all three scheduler policies resulted in nearly identical context switches for I/O-bound processes. I/O-bound processes, by definition, perform high numbers of I/O operations which require the processes to block until their I/O requests have been satisfied. Thus, I/O-bound processes block on I/O devices and tend to voluntarily give up control of the CPU for the scheduler to choose the next process. Seen in maroon, I observe that context switches in the I/O-bound benchmarks are dominated by voluntary switches rather than involuntary, preemptive context switches. Observing mixed and CPU-bound process types, differences begin to emerge since the number of involuntary switches is seen to be much higher for CFS than for the other two policies. In fact, FIFO and RR have almost no context switches at all for CPU-bound processes.



For the FIFO policy, I expected this result, but I was surprised at the low number of preemptions under RR scheduling. As for CFS, the results match intuition since I expect CPU-bound processes to be more likely to be preempted than I/O-bound processes. I also noticed a subtle trend where the ratio of voluntary to involuntary switches increases for CFS running mixed-type processes under increasing loads.

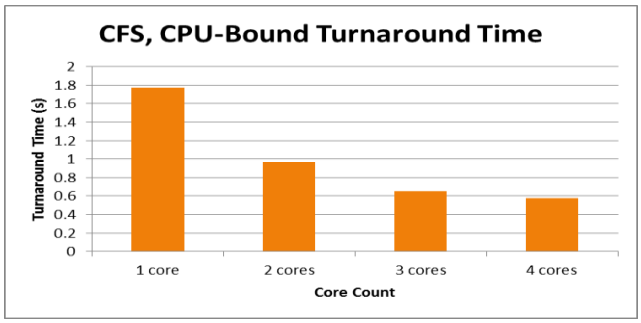
Lastly, I analyzed CPU usage as a percentage value. Notice that the percentage values may be greater than 100% and in fact are maximized at 400% due to the four processor cores allocated to the virtual machine.

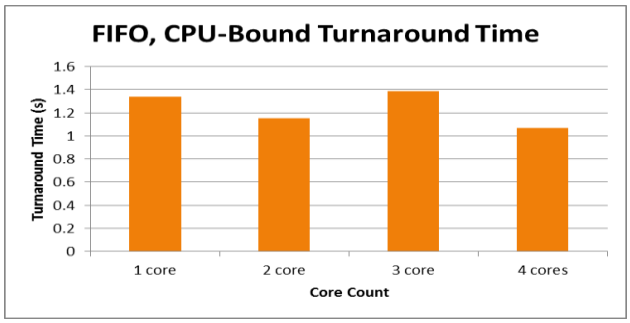


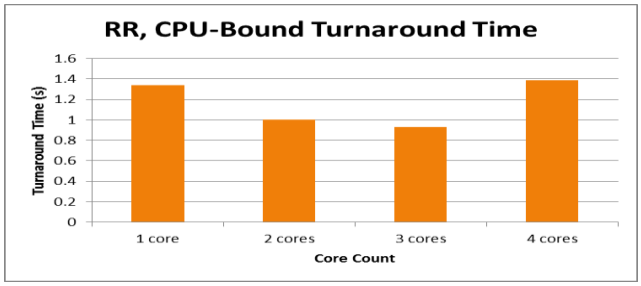
Correlating the chart above with the raw data, I noticed that CFS did a better job maximizing CPU usage for CPU-bound processes which matches the lower turnaround time observed for CFS and CPU-bound processes. Of course, CPU usage decreases as the process type changes from CPU heavy to I/O heavy as processes spend more time waiting and less time utilizing the CPU.

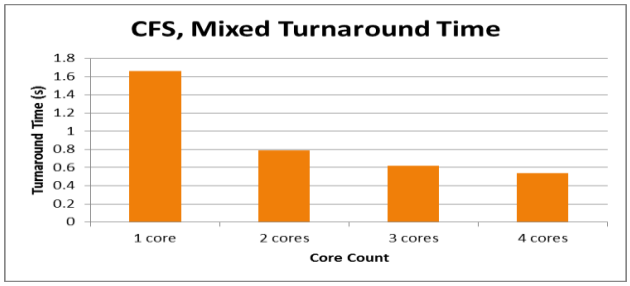
3.1 Multi-Core Results

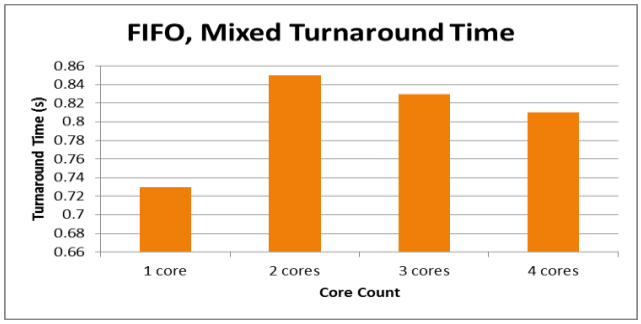
I created charts for each of nine different test configurations, each of which forked 200 processes in order to compare all configurations under identical process load. See the charts below.

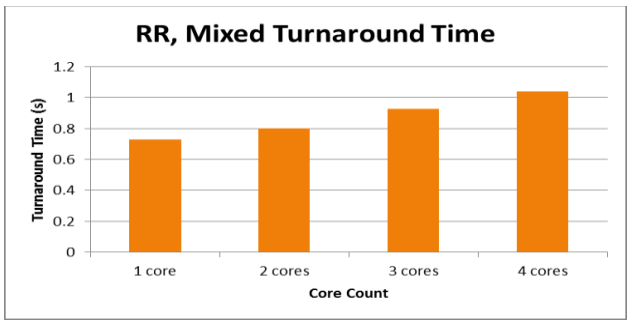


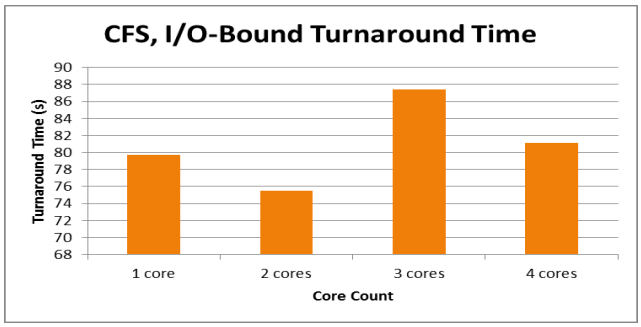


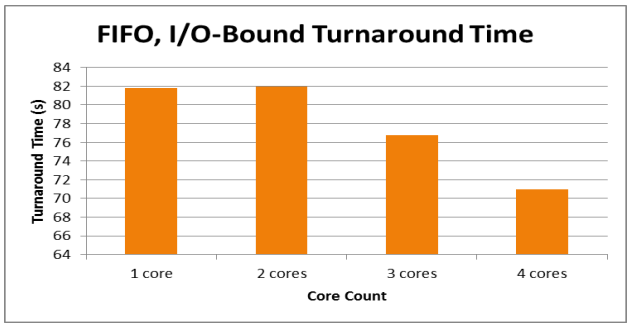


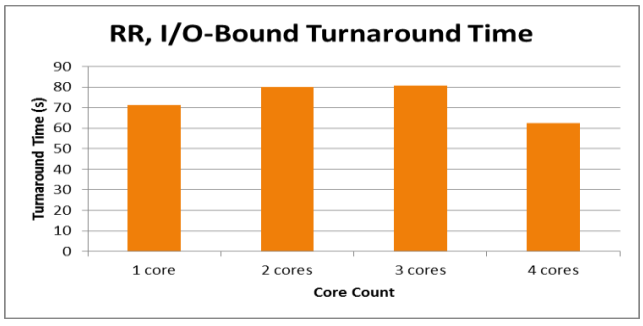












4. Analysis

Considering the results mentioned in section 3, I have drawn several conclusions about the different Linux scheduling policies and process types. First, it appears that CFS is most effective for executing CPU-bound processes, RR is most effective for executing mixed-type processes, and either FIFO or CFS is most effective for I/O-bound processes. Effectiveness, in this context, refers to the success of a particular algorithm in minimizing turnaround time. I note also that FIFO does not scale well with larger numbers of I/O-bound processes, and CFS does not scale well with increasing numbers of mixed-type processes. Furthermore, both FIFO and RR are rarely preempted and perform few context switches while running CPU-bound processes compared to CFS. More I/O operations corresponds to more voluntary context switches as can be seen by the types of context switches performed by CFS going left to right across the chart. Finally, CPU usage is maximized by CPU-bound processes, particularly when under the CFS scheduling policy.

While these conclusions follow readily from the benchmark data collected, I would also point out that some of these results may have been affected by the specifics of the testing environment. For example, I believe that the type of secondary storage used may have a large effect on results in that classic hard disk drives (HDDs) like that in my machine are much slower for read and write access than solid-state drives (SSDs). I would predict that an SSD may improve the scalability of FIFO for I/O-bound processes since I/O operations would be generally faster, thus allowing the scheduler to run through the entire ready queue more quickly since all processes under test performed I/O operations. This would also mean increased efficiency in terms of context-switch overhead. An SSD might also increase the efficiency of RR for I/O operations for similar reasons. Also, because I used a virtual machine instead of a native Linux installation, some effects from Window 7 scheduling could have adversely affected results in unpredictable ways.

Keeping the variables introduced by my testing environment in mind, I would like to now consider why my benchmark tests behaved as they did and assert what we can deduce conclusively from these results. First, CFS is the most effective of the three algorithms considered for scheduling CPU-bound processes. I was perhaps most surprised by this result initially because CFS was designed to maximize interactive performance [4], which implies overhead for a large number of context switches. Indeed, CFS performed by far the most context switches out of the three policies for the CPU-bound process benchmarks. However, CFS also was designed to maximize CPU utilization [4], and succeeded in doing so, averaging 394.80% for 10 processes and an incredible 398% for both 60 and 200 processes! I believe the success of CFS can be directly attributed to its success in utilizing CPU time more effectively than FIFO and RR, demonstrating that CFS is best –suited for maximizing overhead efficiency. CFS also performs well at higher process loads under I/O-bound tests, demonstrating that it is an effective strategy for highly interactive applications. The results indicate that CFS does not scale well for mixed-type processes, which is surprising in that CFS is the default Linux scheduler, and the majority of processes on an average workstation perform a mix of CPU and I/O operations [1]. Thus, my results would seem to indicate that the default Linux scheduler is poor for everyday use. Due to

high variability in the mixed-type data, I believe more work is necessary before settling on this conclusion.

Second, I found that RR is best-suited for mixed-type processes but behaved poorly for pure I/O-bound processes. The context switches performed under RR scheduling were nearly all voluntary, perhaps indicating that the time quantum was too long for preemption to occur before blocking on I/O in either my mixed or I/O-bound programs. Considering the source code for mixed.c, I am unsurprised by this result due to the fact that I interleaved I/O operations with CPU operations rather than creating longer bursts. This may be one change which could reveal more about RR scheduling, but it appears that RR was more efficient at maximizing CPU usage, thereby resulting in shorter turnaround times.

Third, I would conclude that FIFO, while comparable to RR under most benchmarks, did not scale well for I/O-bound processes. This result seems fairly intuitive; when a process blocks on I/O under the FIFO scheduler, it moves to the wait queue and the process at the front of the list is allocated the CPU. When the I/O operation completes, the waiting process is pushed to the rear of the ready queue [10], and if the currently process is not preempted. Therefore, the secondary storage is not used effectively. At lower numbers of processes, this effect is less pronounced, as expected.

4.1 Multi-Core Analysis

Immediately, I noticed that turnaround time decreases with core count under the CFS policy and under CPU-bound and mixed-type processes. This makes intuitive sense since more of the workload gets pushed off to other processors. Multiple CPUs allow CPU operations to be performed in parallel, especially since there are no dependencies between processes under my test benchmarks. Interestingly, both FIFO and RR policies seemed to be unaffected by core count at all. From this result, I would surmise that neither of these policies is well-suited for multi-core environments. I believe this is due to the fact that all processes are placed in the same priority under my benchmarks, so each process must wait until it is at the front of the queue for both FIFO and RR scheduling.

5. Conclusion

To summarize, I have designed three C benchmark programs for testing the CFS, FIFO and RR scheduling policies. Each program implements a different type of test based on different categories of processes such as CPU-bound, I/O-bound, and a mix of the two. The objectives of my investigation were to determine how effective each policy was in terms minimizing turnaround time of processes, minimizing context-switch overhead, and maximizing CPU utilization. Using the Linux time utility, I was able to fork different numbers of processes to simulate different process load scenarios and gather data on several different variables. Over five trials on 27 tests each, I was able to analyze the raw data and form several conclusions, some stronger than others, about the behaviors of the three current Linux scheduling policies.

On average, CFS performed best at both ends of the process type spectrum, minimizing turnaround time for CPU-bound and I/O-bound processes while performing poorly for mixed-type processes. CFS scaled better than FIFO for I/O-bound processes and performed very well in terms of maximizing CPU utilization, especially for CPU-bound processes. FIFO performed similarly to RR as expected, but slightly worse due to lack of preemption which led to poor utilization of the secondary storage device, which in my case was a hard disk drive. RR minimized turnaround time for mixed-type processes and also appeared more efficient at utilizing CPU time in that category but did not perform well under high I/O load.

By analyzing these results, I believe it can be concluded that some advantages of CFS include the fact that it is very effective at maximizing CPU usage overall. However, CFS does not scale well for mixed type processes, probably due to the way in which it sorts processes by vruntime. It may be that for my particular mixed-type program, the immediate interleaving of CPU and I/O operations may have adversely affected CFS by causing a large number of red-black tree rebalancing operations. In the future, I would like to re-design my mixed-type benchmark to include longer bursts of CPU and I/O activity. I would also conclude that RR performs better overall than FIFO for minimizing turnaround time and that RR is particularly well-suited for running large numbers of mixed-type processes. However, I would advise against using RR for CPU-bound or I/O-bound processes due to its apparent inability to maximize CPU usage or effectively retrieve I/O results, respectively. In the future, I believe it would be enlightening to run these benchmark tests on a native Linux installation and with different CPU and I/O burst algorithms.

5.1 Multi-Core Conclusion

For multi-core tests, I believe more investigation is necessary before determining the effects of increasing core count on RR and FIFO policies. However, the data demonstrate that CFS is well-suited to multi-core, parallel-processing environments. This is an intuitive result and fits well with the result that CFS is optimized for maximizing CPU utilization.

6. References

[1] Silberschatz, Abraham, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. 9th ed. Hoboken, NJ: Wiley, 2012. Print.

[2] Groves, Taylor, Jeff Knockel, and Eric Schulte. "BFS vs. CFS - Scheduler Comparison." Thesis. University of New Mexico, 2009. Web. <http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf>.

[3] Trung Thang, Le. "Comparing Real-time Scheduling on the Linux Kernel and an RTOS."*Embedded*. N.p., n.d. Web. 27 Mar. 2014. <http://www.embedded.com/design/operating-systems/4371651/Comparing-the-real-time-scheduling-policies-of-the-Linux-kernel-and-an-RTOS->.

[4] "Completely Fair Scheduler." *Wikipedia*. Wikimedia Foundation, 26 Mar. 2014. Web. 27 Mar. 2014. <http://en.wikipedia.org/wiki/Completely_Fair_Scheduler>.

[5] "Linux." *Wikipedia*. Wikimedia Foundation, 27 Mar. 2014. Web. 27 Mar. 2014. <http://en.wikipedia.org/wiki/Linux>.

[6] "Scheduling (computing)." *Wikipedia*. Wikimedia Foundation, n.d. Web. 27 Mar. 2014. <http://en.wikipedia.org/wiki/Scheduling_%28computing%29#Linux>.

[7] "Sched_setscheduler(2) - Linux Manual Page." *Sched_setscheduler(2) - Linux Manual Page*. N.p., n.d. Web. 27 Mar. 2014. <http://man7.org/linux/man-pages/man2/sched_setscheduler.2.html>.

[8] *Programming Assignment 3: Investigating the Linux Scheduler*. Boulder, CO: University of Colorado, 2014. Print. CSCI 3753 - Operating Systems.

[9] Sayler, Andy. "CU CS Standard Development Environment." *CU CS SDE*. University of Colorado - Boulder, n.d. Web. 27 Mar. 2014. <http://foundation.cs.colorado.edu/sde/>.

[10] Krzyzanowski, Paul. "Process Scheduling." *Process Scheduling*. Rutgers University, 19 Feb. 2014. Web. 27 Mar. 2014. <http://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html>.

Appendix A – Raw Data

T0 (low, CPU, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	27.37	107.98	0.18	395%	11645	21
Trial2	30.74	120.32	0.18	391%	13383	23
Trial3	22.44	89.04	0.05	396%	9392	23
Trial4	20.88	82.34	0.16	395%	8786	23
Trial5	22.39	89.06	0.04	397%	9266	23
Averages	24.764	97.748	0.122	394.80%	10494.4	22.6
T1 (med, CPU, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	188.28	748.9	0.7	398%	184735	123
Trial2	192.78	767.74	0.58	398%	189507	130
Trial3	182.85	728.56	0.55	398%	179297	127
Trial4	181.2	722.27	0.46	398%	178404	128
Trial5	182.2	725.6	0.53	398%	178037	131
Averages	185.462	738.614	0.564	398.00%	181996	127.8
T2 (high, CPU, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	649.86	2587.21	1.77	398%	638632	461
Trial2	647.81	2577.49	1.93	398%	636202	456
Trial3	629.81	2510.47	1.63	398%	618929	474
Trial4	633.61	2526.38	1.62	398%	621552	489
Trial5	631.39	2516.95	1.71	398%	620133	462
Averages	638.496	2543.7	1.732	398.00%	627089.6	468.4
T3 (low, CPU, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	35.62	118.64	0.08	333%	86	17
Trial2	36.68	116.28	0.04	317%	103	17
Trial3	31.67	105.9	0.04	334%	88	17
Trial4	33.47	111.61	0.07	333%	100	17
Trial5	34.07	112.4	0.06	330%	81	17
Averages	34.302	112.966	0.058	329.40%	91.6	17
T4 (med, CPU, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	207.04	764.76	0.27	369%	714	67
Trial2	202.84	746.48	0.27	368%	578	67
Trial3	196.81	735.35	0.14	373%	409	67
Trial4	199.3	732.8	0.18	367%	696	67
Trial5	199.03	739.18	0.16	371%	513	67
Averages	201.004	743.714	0.204	369.60%	582	67
T5 (high, CPU, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	663.5	2501.85	0.65	377%	1566	207
Trial2	663.53	2486.71	0.51	374%	1682	207

Trial3	654.67	2470.65	0.53	377%	1586	207
Trial4	648.98	2459.85	0.42	379%	1392	207
Trial5	661.95	2503.02	0.43	378%	1455	207
Averages	658.526	2484.416	0.508	377.00%	1536.2	207
T6 (low, CPU, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	33.03	115.94	0.02	351%	366	20
Trial2	32.13	119.29	0.06	371%	356	22
Trial3	32.71	117.51	0.01	359%	333	20
Trial4	34.04	117.52	0.04	345%	387	22
Trial5	33.5	112.86	0.05	337%	367	21
Averages	33.082	116.624	0.036	352.60%	361.8	21
T7 (med, CPU, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	200.32	751.73	0.2	375%	2637	106
Trial2	199.74	752.21	0.18	376%	2466	110
Trial3	198	746.82	0.16	377%	2564	99
Trial4	198.82	752.4	0.23	378%	2643	104
Trial5	202.5	759.62	0.37	375%	2710	104
Averages	199.876	752.556	0.228	376.20%	2604	104.6
T8 (high, CPU, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	668.96	2531.63	0.59	378%	8824	288
Trial2	651.84	2476.72	0.51	380%	8469	341
Trial3	653.85	2458.58	0.64	376%	8240	274
Trial4	657.33	2491.44	0.59	379%	8583	278
Trial5	661.97	2495.47	0.5	377%	8394	357
Averages	658.79	2490.768	0.566	378.00%	8502	307.6
T9 (low, mixed, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	27.89	72.73	5.48	280%	32161	251
Trial2	27.55	73.75	5.36	287%	22849	181
Trial3	20.84	61.65	2.7	308%	6070	224
Trial4	19.98	65.48	3.14	343%	6874	53
Trial5	21.7	64.28	2.7	308%	6664	155
Averages	23.592	67.578	3.876	305.20%	14923.6	172.8
T10 (med, mixed, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	179.55	365.78	28.67	219%	149310	13581
Trial2	194.19	341.99	29.12	191%	116530	21894
Trial3	159.56	318.61	18.42	211%	55713	19209
Trial4	152.58	325.57	19.16	225%	54893	21093
Trial5	149.58	330.23	18.71	233%	57480	20348
Averages	167.092	336.436	22.816	215.80%	86785.2	19225
T11 (high, mixed, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	800.3	1060.15	128.69	148%	392440	394161

Trial2	1052.31	971.03	182.92	109%	287708	730380
Trial3	621.72	1088.82	85.3	188%	162127	245890
Trial4	626.53	1081.08	90.03	186%	163473	274208
Trial5	631.77	1089.69	88.73	186%	164714	269438
Averages	746.526	1058.154	115.134	163.40%	234092.4	382815.4
T12 (low, mixed, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	29.62	58.79	3.25	209%	19	602
Trial2	26.59	48.9	2.48	193%	3	732
Trial3	21.29	59.84	2.58	293%	37	68
Trial4	24.29	48.48	1.8	207%	19	242
Trial5	22.74	45.19	1.91	207%	10	795
Averages	24.906	52.24	2.404	221.80%	17.6	487.8
T13 (med, mixed, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	180.19	354.43	23.88	209%	107	25780
Trial2	206.12	297.8	26	157%	25	41696
Trial3	148.28	294.82	20.03	212%	16	30407
Trial4	140.25	300.74	18.08	227%	36	25788
Trial5	145.63	331.43	16.16	238%	132	17215
Averages	164.094	315.844	20.83	208.60%	63.2	28177.2
T14 (high, mixed, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	841.84	997.17	134.45	134%	18	632188
Trial2	985.41	948.02	164.66	112%	32	794470
Trial3	587.12	1025.05	126.18	196%	33	457683
Trial4	580.85	1021.48	124.65	197%	68	423628
Trial5	587.83	1017.14	124.86	194%	49	439233
Averages	716.61	1001.772	134.96	166.60%	40	549440.4
T15 (low, mixed, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	27.94	54.64	3.4	207%	261	1759
Trial2	24.35	44.46	2.38	192%	149	1420
Trial3	18.37	62.55	2.73	355%	228	37
Trial4	20.4	56.78	2.51	290%	228	177
Trial5	22.83	60.85	2.35	276%	339	60
Averages	22.778	55.856	2.674	264.00%	241	690.6
T16 (med, mixed, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	189.05	325.62	24.22	185%	1885	36144
Trial2	142.74	299.54	18.76	222%	1446	28929
Trial3	151.47	309.23	21.57	218%	1419	33123
Trial4	146.95	323.61	18.8	232%	1732	28068
Trial5	146.95	335.5	20.25	242%	1812	27636
Averages	155.432	318.7	20.72	219.80%	1658.8	30780
T17 (high, mixed, RR)	wall	user	system	CPU	i-switched	v-switched

Trial1	1031.28	1007.44	172.68	114%	5092	829140
Trial2	576.37	1033.66	113.87	199%	5076	397819
Trial3	589.45	1028.74	124.19	195%	4723	463146
Trial4	591.45	1024.43	123.98	194%	4583	458398
Trial5	589.33	1024.7	121.96	194%	4659	437939
Averages	675.576	1023.794	131.336	179.20%	4826.6	517288.4
T18 (low, I/O, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	27.78	0.07	7.21	26%	358	36536
Trial2	30.53	0.08	20.98	68%	470	34078
Trial3	35.85	0	25.29	70%	445	34980
Trial4	32.35	0.05	24.97	77%	482	33916
Trial5	31.97	0.03	21.47	67%	470	33967
Averages	31.696	0.046	19.984	61.60%	445	34695.4
T19 (med, I/O, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	194.65	0.21	40.87	21%	1855	214154
Trial2	184.56	0.42	113.18	61%	2444	206558
Trial3	173.8	0.24	105.28	60%	2290	205489
Trial4	199.86	0.25	107.16	53%	2263	205848
Trial5	208.2	0.29	115.26	55%	2306	206448
Averages	192.214	0.282	96.35	50.00%	2231.6	207699.4
T20 (high, I/O, CFS)	wall	user	system	CPU	i-switched	v-switched
Trial1	890.65	0.74	139.95	15%	5318	692122
Trial2	948.46	1.04	233.07	24%	5645	675615
Trial3	927.62	1.3	239.7	25%	5998	649414
Trial4	966.54	1.28	237.68	24%	6130	674965
Trial5	896.93	1.21	233.16	26%	5549	671568
Averages	926.04	1.114	216.712	22.80%	5728	672736.8
T21 (low, I/O, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	29.26	0	2.89	9%	7	29823
Trial2	27.91	0.01	4.16	14%	6	35720
Trial3	27.53	0	3.82	13%	6	33154
Trial4	29.69	0.03	8.49	28%	6	31732
Trial5	29.67	0.01	19.96	67%	6	30846
Averages	28.812	0.01	7.864	26.20%	6.2	32255
T22 (med, I/O, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	170.69	0.29	49.8	29%	5	184460
Trial2	189.54	0.11	55.27	29%	6	180835
Trial3	246.3	0.28	66.93	27%	6	189382
Trial4	201.49	0.19	84.34	41%	6	182286
Trial5	160.66	0.08	49.35	30%	6	179642
Averages	193.736	0.19	61.138	31.20%	5.8	183321

T23 (high, I/O, FIFO)	wall	user	system	CPU	i-switched	v-switched
Trial1	910.92	0.82	223.72	24%	6	627882
Trial2	930.55	0.92	308.89	33%	6	612468
Trial3	1029.39	1.08	304.7	29%	6	615555
Trial4	942.32	0.49	306.2	32%	6	610156
Trial5	948.08	0.57	262.87	27%	6	590054
Averages	952.252	0.776	281.276	29.00%	6	611223
T24 (low, I/O, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	29.15	0.01	2.46	8%	17	26128
Trial2	31	0.01	2.37	7%	15	38445
Trial3	40.72	0.01	6.02	14%	17	29875
Trial4	33.15	0	4.23	12%	13	29281
Trial5	31.14	0	3.87	12%	12	32071
Averages	33.032	0.006	3.79	10.60%	14.8	31160
T25 (med, I/O, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	201.89	0.22	78.7	39%	33	180790
Trial2	179.42	0.16	48.36	27%	75	182289
Trial3	274.3	0.42	98.91	36%	82	176430
Trial4	193.3	0.2	94.92	49%	93	182777
Trial5	154.92	0.28	102.75	66%	70	187399
Averages	200.766	0.256	84.728	43.40%	70.6	181937
T26 (high, I/O, RR)	wall	user	system	CPU	i-switched	v-switched
Trial1	929.89	0.74	205.44	22%	127	627206
Trial2	915.11	0.91	291.2	31%	203	619437
Trial3	1127.76	0.8	336.16	29%	198	619027
Trial4	1028.7	0.75	295.84	28%	202	614042
Trial5	838.76	0.44	306.49	36%	228	616287
Averages	968.044	0.728	287.026	29.20%	191.6	619199.8

Multi-Core Raw Data

1 core	wall	user	system	CPU	i-switched	v-switched
T0	0.14	0.05	0	39%	53	27
T1	0.51	0.22	0.03	48%	267	174
T2	1.77	0.77	0.08	48%	856	452
T3	0.86	0.05	0.01	8%	10	26
T4	0.43	0.24	0.02	61%	11	64
T5	1.34	0.76	0.07	61%	9	204
T6	0.08	0.04	0	49%	8	14
T7	0.39	0.23	0.01	62%	8	64
T8	1.34	0.75	0.07	61%	8	205
T9	0.09	0	0	8%	40	29
T10	0.5	0.02	0	7%	203	244
T11	1.66	0.2	0.06	16%	628	565
T12	0.05	0	0	14%	8	14
T13	0.24	0.02	0.01	16%	8	64
T14	0.73	0.07	0.03	14%	10	204
T15	0.05	0	0	7%	8	14
T16	0.23	0.01	0	8%	11	64
T17	0.73	0.05	0.01	8%	11	204
T18	5.23	0.01	0.03	0%	32	3941
T19	25.05	0.1	0.28	1%	541	23454
T20	79.72	0.21	1.47	2%	1973	79088
T21	5.52	0	0.26	4%	7	1694
T22	23.61	0	1.78	7%	9	12953
T23	81.82	0.03	4.81	5%	9	52170
T24	4.29	0	0.21	4%	7	1474
T25	19.98	0	1.7	8%	7	12505
T26	71.4	0.06	4.77	6%	8	51684
2 cores	wall	user	system	CPU	i-switched	v-switched
T0	0.07	0.08	0	123%	63	29
T1	0.32	0.32	0.03	113%	312	144
T2	0.97	0.91	0.13	106%	1152	594
T3	0.73	0.04	0.01	8%	8	26
T4	0.41	0.26	0	64%	7	64
T5	1.15	0.86	0.04	79%	5	205
T6	0.09	0.04	0.01	62%	8	14
T7	0.47	0.25	0.06	67%	10	64
T8	1	0.97	0.06	103%	8	412
T9	0.06	0.01	0	29%	38	31

T10	0.24	0.07	0.04	46%	497	189
T11	0.79	0.33	0.08	52%	890	643
T12	0.05	0	0	7%	7	14
T13	0.25	0.08	0.03	45%	8	64
T14	0.85	0.27	0.12	46%	11	204
T15	0.05	0.01	0	27%	9	14
T16	0.25	0.06	0.02	34%	6	118
T17	0.8	0.22	0.05	34%	10	204
T18	5.43	0.01	0.12	2%	45	4666
T19	22.69	0.04	0.69	3%	243	25124
T20	75.47	0.12	2.76	3%	1674	84288
T21	3.56	0.02	0.15	5%	7	4041
T22	24.96	0	2.04	8%	7	20350
T23	81.91	0.08	2.8	3%	6	73616
T24	2.33	0.03	0.16	8%	8	3390
T25	22.16	0.02	1.14	5%	8	22901
T26	80.13	0.06	4.7	5%	8	70211
3 cores	wall	user	system	CPU	i-switched	v-switched
T0	0.06	0.08	0	135%	139	23
T1	0.22	0.38	0.04	185%	453	271
T2	0.65	1.31	0.08	213%	986	481
T3	0.47	0.06	0.01	16%	6	26
T4	0.41	0.24	0.02	65%	6	64
T5	1.39	0.78	0.12	65%	6	204
T6	0.09	0.06	0.01	79%	12	14
T7	0.46	0.28	0.03	68%	6	64
T8	0.93	1.1	0.08	126%	5	207
T9	0.06	0.02	0.01	56%	48	31
T10	0.17	0.12	0.06	106%	754	168
T11	0.62	0.66	0.17	133%	1047	875
T12	0.07	0.04	0.01	70%	8	14
T13	0.29	0.14	0.04	61%	7	105
T14	0.83	0.29	0.06	42%	8	204
T15	0.04	0.01	0	41%	6	14
T16	0.26	0.08	0.02	39%	7	64
T17	0.93	0.43	0.11	58%	7	204
T18	4.01	0	0.3	7%	52	4483
T19	28.88	0.02	1.69	5%	529	25888
T20	87.44	0.14	4.63	5%	1949	85951
T21	3.12	0	0.22	7%	8	3443
T22	21.77	0.05	2.25	10%	7	21586

T23	76.72	0.22	5.73	7%	7	78072
T24	4.29	0	0.32	7%	6	3575
T25	25.77	0.07	1.98	7%	7	23914
T26	80.58	0.07	6.51	8%	12	71668
4 cores	wall	user	system	CPU	i-switched	v-switched
T0	0.03	0.08	0	237%	43	24
T1	0.18	0.45	0.06	286%	304	174
T2	0.58	1.64	0.12	299%	1002	508
T3	0.9	0.08	0.02	12%	3	26
T4	0.44	0.26	0.03	65%	5	64
T5	1.07	1.08	0.09	110%	6	206
T6	0.11	0.06	0.02	83%	8	14
T7	0.45	0.29	0.01	68%	7	64
T8	1.39	0.78	0.1	64%	7	204
T9	0.03	0.03	0.02	153%	46	23
T10	0.15	0.21	0.06	182%	246	362
T11	0.54	0.67	0.23	168%	1324	707
T12	0.05	0.02	0	57%	2	14
T13	0.27	0.12	0.03	56%	7	64
T14	0.81	0.64	0.13	95%	6	305
T15	0.06	0.03	0.01	70%	6	14
T16	0.3	0.17	0.04	69%	6	64
T17	1.04	0.5	0.17	65%	5	515
T18	3.66	0	0.66	17%	54	4627
T19	25.7	0.01	3.25	12%	550	26863
T20	81.14	0.32	10.58	13%	1157	87990
T21	3.75	0	0.25	6%	5	2918
T22	16.72	0.04	2.2	13%	7	26420
T23	70.94	0.26	14.84	21%	6	75647
T24	4.65	0	0.62	13%	8	3619
T25	20.62	0.18	0.97	5%	7	23851
T26	62.35	0.2	16.19	26%	6	77484

Appendix B – Source Code

Figure 1: pi-sched.c


```
/*
 * File: pi-sched.c
 * Author: Conrad Hougen adapted from pi-sched.c by Andy Sayler
 * Project: CSCI 3753 Programming Assignment 3
 * Description:
 *  This file contains a simple program for statistically
 *      calculating pi using a specific scheduling policy.
 *      The program is used to collect data on Linux
 *      scheduling for a CPU-bound process. The program
 *      takes the number of iterations, scheduling policy,
 *      and number of child processes to fork as command-line
 *      arguments, in that order.
 *
 * E.g.
 *  ./pi-sched 100000000 SCHED_OTHER 90 > /dev/null
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

#define RADIUS (RAND_MAX / 2)
#define MAX_CHILDREN 200
#define MAXFILENAMELENGTH 80
char STATOUT_FILE[MAXFILENAMELENGTH];


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;
    int j;
    long iterations;
    struct sched_param param;
```

```
    int policy;
    double x, y;
    double inCircle = 0.0;
    double inSquare = 0.0;
    double pCircle = 0.0;
    double piCalc = 0.0;
    int num_children;
    pid_t pid;
    int status;
    FILE *statout_fp = NULL;
    struct rusage r_usage; // struct to collect statistics for each forked process

    // check that the correct number of cmdline args are used
    if(argc != 5){
        printf("argc = %d\n", argc);
        printf("Needs 4 args: <num iterations> <policy> <num processes> <test number>\n");
        exit(EXIT_FAILURE);
    }
    // Set iterations
    iterations = atol(argv[1]);
    if(iterations < 1){
        printf("Bad iterations value\n");
        exit(EXIT_FAILURE);
    }
    // Set policy
    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{
        printf("Unhandled scheduling policy\n");
        exit(EXIT_FAILURE);
    }

    // Set number of child processes to fork
    num_children = atol(argv[3]);
    if(num_children < 1 || num_children > MAX_CHILDREN)
    {
        printf("The number of child processes must be in the range [1,200]\n");
        exit(EXIT_FAILURE);
    }

    // Get the output file name corresponding to trial number
    j = atol(argv[4]);
    sprintf(STATOUT_FILE, "test_results/test%d.txt", j);
```

```
// 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 num_children child processes to test CPU bound process scheduling
for(j=0; j<num_children; ++j)
{
    if((pid = fork()) == -1)
    {
        printf("Fork failed with j = %d\n", j);
        exit(EXIT_FAILURE);
    }
    else if(pid == 0)
    {
        // Calculate pi using statistical method
        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);

        // child process done, so exit
        exit(EXIT_SUCCESS);
    }
}

statout_fp = fopen(STATOUT_FILE, "a"); // append to file
if(statout_fp == NULL)
```

```
    {
        printf("Error Opening Output File\n");
    }
    else
    {
        fprintf(statout_fp,
            "Statistics for CPU bound, policy:%s, num_children:%d\n",
            argv[2], num_children);
    }
    // done forking in parent process...now wait and reap children
    while (1) {
        pid = wait4(-1, &status, 0, &r_usage);
        if(pid == -1)
        {
            if(errno == ECHILD)
            {
                printf("No more children\n");
                break; // no more child processes
            }
        }
        else
        {
            if(statout_fp != NULL)
            {
                // write the output file
                fprintf(statout_fp, "PID[%d]: Utime: %ld(s), %ld(us)", pid,
                    r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec);
                fprintf(statout_fp, " Stime: %ld(s), %ld(us)",
                    r_usage.ru_stime.tv_sec, r_usage.ru_stime.tv_usec);
                fprintf(statout_fp,
                    " inblockops:%ld, outblockops:%ld, volcsw:%ld, involcsw:%ld\n",
                    r_usage.ru_inblock, r_usage.ru_oublock, r_usage.ru_nvcsw,
                    r_usage.ru_nivcsw);
            }
            if(!WIFEXITED(status) || WEXITSTATUS(status) != 0)
            {
                printf("pid %d failed\n", pid);
                exit(EXIT_FAILURE);
            }
        }
    }


    if(statout_fp != NULL)
    {
        fclose(statout_fp);
    }

    exit(EXIT_SUCCESS);
}
```

Figure 2: mixed.c


```
/* mixed.c
 * Author: Conrad Hougen
 * Project: CSCI 3753 Programming Assignment 3
 * Description:
 *  This file implements a mix of CPU and I/O operations
 *  for benchmarking different Linux scheduler policies.
 *  Specifically, it computes and approximation of pi using
 *  a statistical method, then writes intermediate results
 *  to data files.  The pi computation is CPU intensive while
 *  the file writes are I/O intensive.  The data files can be
 *  used as input to the I/O intensive rw.c program.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>

#define MAXFILENAMELENGTH 80
#define RADIUS (RAND_MAX / 2)
#define MAX_CHILDREN 200
#define DEFAULT_OUTPUTFILENAMEBASE "output/dat"
char STATOUT_FILE[MAXFILENAMELENGTH];

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;
    int j;
    int rv;
    long iterations;
    struct sched_param param;
    int policy;
    double x, y;
    double inCircle = 0.0;
```

```
    double inSquare = 0.0;
    double pCircle = 0.0;
    double piCalc = 0.0;
    int num_children;
    pid_t pid;
    int status;
    FILE *statout_fp = NULL;
    struct rusage r_usage; // struct to collect statistics for each forked process
    FILE *output_fp;
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];


    // check that the correct number of cmdline args are used
    if(argc != 5){
        printf("argc = %d\n", argc);
        printf("Needs 4 args: <num iterations> <policy> <num processes> <trial number>\n"
        );
        exit(EXIT_FAILURE);
    }
    // Set iterations
    iterations = atol(argv[1]);
    if(iterations < 1){
        printf("Bad iterations value\n");
        exit(EXIT_FAILURE);
    }
    // Set policy
    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{
        printf("Unhandled scheduling policy\n");
        exit(EXIT_FAILURE);
    }

    // Set number of child processes to fork
    num_children = atol(argv[3]);
    if(num_children < 1 || num_children > MAX_CHILDREN)
    {
        printf("The number of child processes must be in the range [1,999]\n");
        exit(EXIT_FAILURE);
    }

    // Get the output file name corresponding to trial number
```

```
    j = atol(argv[4]);
    sprintf(STATOUT_FILE, "test_results/test%d.txt", j);

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

    if(strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
        fprintf(stderr, "Default output filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);

    // 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 num_children child processes to test mixed process scheduling
    for(j=0; j<num_children; ++j)
    {
        if((pid = fork()) == -1)
        {
            printf("Fork failed with j = %d\n", j);
            exit(EXIT_FAILURE);
        }
        else if(pid == 0)
        {
            // Open output data file for writing with standard permissions
            rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d",
                outputFilenameBase, j+1);
            if(rv > MAXFILENAMELENGTH)
            {
                printf("Output filename length exceeds limit of %d characters.\n",
                    MAXFILENAMELENGTH);
                exit(EXIT_FAILURE);
            }
            else if(rv < 0){
                printf("Failed to generate output filename\n");
                exit(EXIT_FAILURE);
            }
            if((output_fp = fopen(outputFilename, "w")) == NULL)
            {
                printf("Failed to open output file\n");
                exit(EXIT_FAILURE);
            }
```

```
            // Calculate pi using statistical method
            for(i=0; i<iterations; ++i)
            {
                // CPU operation
                x = (random() % (RADIUS * 2)) - RADIUS;
                fprintf(output_fp, "%8.7f", x);
                y = (random() % (RADIUS * 2)) - RADIUS;
                fprintf(output_fp, "%8.7f", y);
                if(zeroDist(x,y) < RADIUS)
                {
                    inCircle++;
                    fprintf(output_fp, "%8.7f", inCircle);
                }
                inSquare++;
                pCircle = inCircle/inSquare;
                piCalc = pCircle * 4.0;

                // write result to file (I/O operation)
                if(fprintf(output_fp, "%8.7f", piCalc) < 0){
                    printf("Error writing output file");
                    exit(EXIT_FAILURE);
                }
            }

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

            // Close Output File Descriptor
            fclose(output_fp);

            // child process done, so exit
            exit(EXIT_SUCCESS);
        }
    }

    statout_fp = fopen(STATOUT_FILE, "a"); // append to file
    if(statout_fp == NULL)
    {
        printf("Error Opening Output File\n");
    }
    else
    {
        fprintf(statout_fp, "Statistics for Mixed, policy:%s, num_children:%d\n",
            argv[2], num_children);
    }
    // done forking in parent process...now wait and reap children
    while (1) {
        pid = wait4(-1, &status, 0, &r_usage);
        if(pid == -1)
```

```
        {
            if(errno == ECHILD)
            {
                printf("No more children\n");
                break; // no more child processes
            }
        }
        else
        {
            if(statout_fp != NULL)
            {
                // write the output file
                fprintf(statout_fp, "PID[%d]: Utime: %ld(s), %ld(us)", pid,
                    r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec);
                fprintf(statout_fp, " Stime: %ld(s), %ld(us)", r_usage.ru_stime.tv_sec,
                    r_usage.ru_stime.tv_usec);
                fprintf(statout_fp,
                    " inblockops:%ld, outblockops:%ld, volcsw:%ld, involcsw:%ld\n",
                    r_usage.ru_inblock, r_usage.ru_oublock, r_usage.ru_nvcsw,
                    r_usage.ru_nivcsw);
            }
            if(!WIFEXITED(status) || WEXITSTATUS(status) != 0)
            {
                printf("pid %d failed\n", pid);
                exit(EXIT_FAILURE);
            }
        }
    }


    if(statout_fp != NULL)
    {
        fclose(statout_fp);
    }

    exit(EXIT_SUCCESS);
}
```

Figure 3: rw.c


```
/*
 * File: rw.c
 * Author: Conrad Hougen adapted from rw.c by Andy Sayler
 * Project: CSCI 3753 Programming Assignment 3
 * 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.
 *              Forks multiple child processes and changes the
 *              scheduling policy based on command-line arguments.
 */

/* 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>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <sched.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAMEBASE "input/rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "output/dat"
#define MAX_CHILDREN 200
char STATOUT_FILE[MAXFILENAMELENGTH];


int main(int argc, char* argv[])
{
    FILE *statout_fp = NULL;
    int num_children;
    pid_t pid;
    int status;
    int policy;
    int i;
    int rv;
    int inputFD;
    int outputFD;
    char inputFilename[MAXFILENAMELENGTH];
    char inputFilenameBase[MAXFILENAMELENGTH];
```

```
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];
    struct sched_param param;
    struct rusage r_usage; // struct to collect statistics

    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
    if(argc != 6)
    {
        printf("%d args received...expected 6\n", argc);
        exit(EXIT_FAILURE);
    }
    transfersize = atol(argv[1]);
    if(transfersize < 1)
    {
        printf("Bad transfersize value\n");
        exit(EXIT_FAILURE);
    }

    // Set supplied block size
    blocksize = atol(argv[2]);
    if(blocksize < 1)
    {
        printf("Bad blocksize value\n");
        exit(EXIT_FAILURE);
    }

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

```
    // Get scheduling policy from command line
    if(!strcmp(argv[3], "SCHED_OTHER")){
        policy = SCHED_OTHER;
    }
    else if(!strcmp(argv[3], "SCHED_FIFO")){
        policy = SCHED_FIFO;
    }
    else if(!strcmp(argv[3], "SCHED_RR")){
        policy = SCHED_RR;
    }
    else{
        printf("Unhandled scheduling policy\n");
        exit(EXIT_FAILURE);
    }

    // Get number of child processes from command line
    num_children = atol(argv[4]);
    if(num_children < 1 || num_children > MAX_CHILDREN)
    {
        printf("Incorrect number of children to fork: %d\n", num_children);
        exit(EXIT_FAILURE);
    }

    // Get the output file name corresponding to trial number
    i = atol(argv[5]);
    sprintf(STATOUT_FILE, "test_results/test%d.txt", i);

    //
    if(strnlen(DEFAULT_INPUTFILENAMEBASE, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
        fprintf(stderr, "Default input filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(inputFilenameBase, DEFAULT_INPUTFILENAMEBASE, MAXFILENAMELENGTH);

    //
    if(strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
        fprintf(stderr, "Default output filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);

    // 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 num_children io-bound processes
    for(i = 0; i < num_children; ++i)
    {
        if((pid = fork()) == -1)
        {
            printf("Fork failed with i = %d\n", i);
            exit(EXIT_FAILURE);
        }
        else if(pid == 0)
        {
            // Allocate buffer space
            buffersize = blocksize;
            if(!(transferBuffer = malloc(buffersize*sizeof(*transferBuffer))))
            {
                printf("Failed to allocate transfer buffer\n");
                exit(EXIT_FAILURE);
            }

            // Open Input File Descriptor in Read Only mode
            // Files 1 to 200 must be created for this to work
            // Names are rwinput-1 to rwinput-200 in input folder
            // i should be unique in each child process depending on when
            // the child was forked
            rv = snprintf(inputFilename, MAXFILENAMELENGTH, "%s-%d",
                    inputFilenameBase, i+1);
            if(rv > MAXFILENAMELENGTH)
            {
                printf("Input filename length exceeds limit of %d characters.\n",
                    MAXFILENAMELENGTH);
                exit(EXIT_FAILURE);
            }
            else if(rv < 0){
                printf("Failed to generate input filename\n");
                exit(EXIT_FAILURE);
            }
            if((inputFD = open(inputFilename, O_RDONLY | O_SYNC)) < 0)
            {
                printf("Failed to open input file\n");
                exit(EXIT_FAILURE);
            }


            // Open Output File Descriptor in Write Only mode with standard permissions
            rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d",
                outputFilenameBase, i+1);
```

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

        // Print Status
        printf("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){
                printf("Error reading input file\n");
                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))
                    {
                        printf("Error resetting to beginning of file\n");
                        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))
            {
                printf("Failed to close output file\n");
                exit(EXIT_FAILURE);
            }

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

            exit(EXIT_SUCCESS);
        }
    }

    statout_fp = fopen(STATOUT_FILE, "a"); // append to file
    if(statout_fp == NULL)
    {
        printf("Error Opening Output File\n");
    }
    else
    {
        fprintf(statout_fp, "Statistics for I/O bound, policy:%s, num_children:%d\n",
            argv[3], num_children);
    }
```

```
    // done forking in parent process...now wait and reap children
    while (1) {
        // wait and collect statistics
        pid = wait4(-1, &status, 0, &r_usage);

        if(pid == -1)
        {
            if(errno == ECHILD)
            {
                printf("No more children\n");
                break; // no more child processes
            }
        }
        else
        {
            if(statout_fp != NULL)
            {
                // write the output file
                fprintf(statout_fp, "PID[%d]: Utime: %ld(s), %ld(us)", pid,
                    r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec);
                fprintf(statout_fp, " Stime: %ld(s), %ld(us)", r_usage.ru_stime.tv_sec,
                    r_usage.ru_stime.tv_usec);
                fprintf(statout_fp,
                    " inblockops:%ld, outblockops:%ld, volcsw:%ld, involcsw:%ld\n",
                    r_usage.ru_inblock, r_usage.ru_oublock, r_usage.ru_nvcsw,
                    r_usage.ru_nivcsw);
            }
            if(!WIFEXITED(status) || WEXITSTATUS(status) != 0)
            {
                printf("pid %d failed\n", pid);
                exit(EXIT_FAILURE);
            }
        }
    }


    if(statout_fp != NULL)
    {
        fclose(statout_fp);
    }

    exit(EXIT_SUCCESS);

}
```