Steven Conflenti

CSCI 3753 – Operating Systems

6 April 2015

Programming Assignment 4: Investigating the Linux Scheduler

Abstract:

The purpose of this lab is to analyze the performance a few of Linux's scheduling policies. We accomplished this by collecting benchmark data for each of the policies. Data was collected from multiple trials and then averaged to reduce inconsistencies and the probability of anomalies. Overall, SCHED_OTHER is the best suited policy all around. In some situations, specifically with small and medium amounts of simultaneous processes, SCHED_FIFO comes close or outperforms SCHED_OTHER. To my surprise, SCHED_RR was not the best suited for any scenario. SCHED_FIFO performed similarly to SCHED_RR but was almost always slightly more efficiently. Additionally, there are big performances differences when you change CPU core utilization to an amount greater than one core, but there were not as pronounced differences when you change from a number of cores already greater than one to a higher number. In completing this lab I learned both about each individual scheduling policy and a lot about how they perform in niche scenarios.

Introduction:

Our goal was benchmark and analyze the Linux scheduler. To accomplish this we examined three different scheduling policies, SCHED_OTHER (Other or Normal), SCHED_RR (Round Robin or RR), and SCHED_FIFO (First in First Out or FIFO). We evaluated each policy within the context of variously bounded processes, different quantities of simultaneous processes and different amounts of utilized CPU cores.

In order to compare and contrast the scheduler policies and see why they had performance differences we kept track of a few statistics from our tests. The first value we kept track of was the wall time. This is essentially the time from beginning to end of the task, and it is analogous to execution time. The wall time allows us to see the order in which the schedulers actually finish the tasks all other things included. The next three values we kept track of were user time, system time and wait time. User time is the time spent executing user code, system time is the time spent executing system code (primarily system calls) and wait time is the time spent waiting for the CPU. The total wall time is the sum of these three values (wall = user + system + wait). The last two values we benchmarked were the numbers of voluntary and involuntary context switches. These help us analyze how impactful the context switch overhead is for the various scheduling policies. Together, measuring all of this data will allow us to see which policies are well and poorly suited for each scenario as well as how the policies scale with processes and CPU cores.

Method:

We conducted all of our benchmarks within the same testing environment to ensure consistency throughout the data. The tests were performed on a machine running the virtual machine image distributed by the CU computer science department. Furthermore, the CPU we ran our benchmarks on was an Intel i7-4770k 3.5 GHz, and the VM was utilizing 4 GB of ram. This CPU has four cores, but (for the extra credit portion of the lab) we gathered different sets of data by limiting the cores being utilized to 1, 2 and 4. We made sure that the system had the same conditions for all tests and that no extra processes running threw off the data for one or more of the tests.

Our benchmarks work by keeping track of the three timers (wall, user and system) and counters for the numbers of voluntary and involuntary context switches. We kept track of these values in all tests and performed multiple tests to see how they varied between different process types, scheduling policies, quantities of simultaneous processes and CPU cores.

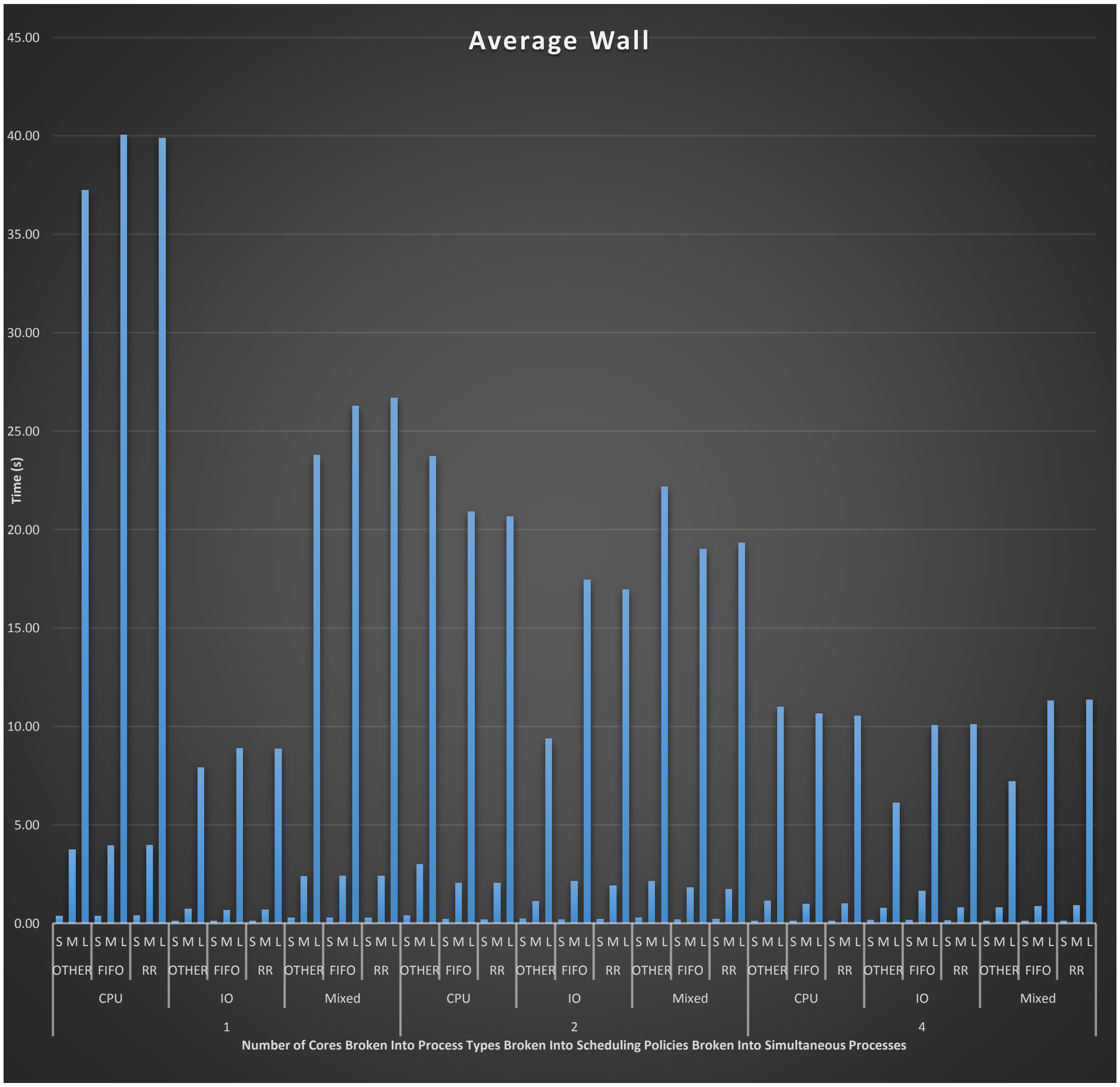
We used these three programs in our tests to compare different ways programs can be limited in their execution times. The three different types of processes we were CPU bounded, I/O bounded and mixed. The CPU bounded program calculates pi using the statistical method across a certain amount of iterations. The completion of this program is limited by how fast the CPU can make the necessary calculations for the given iterations. Conversely, the I/O bounded program reads some information from an input file and writes it to an output file in discrete blocks. This program is limited by how fast it can read and write to the disk rather than how fast it can make CPU calculations. The mixed program is just a combination of the previous two. It calculates pi using the same method as above while also reading the contents of an input file and writing them to an output file.

Within the context of each individual process type we benchmarked the performances of three different types of schedulers with different amounts of simultaneous processes. The three schedulers were made up of two real-time schedulers, Round Robin and First In First Out, and one completely fair scheduler, Other. The tasks scheduled via real-time policies have the quality that they preempt tasks with all other priorities. The third scheduler, Other, uses a round-robin time-sharing policy that gives a task a certain timeslice depending on the other tasks concurrently being scheduled. In order to benchmark how each scheduler scales we tested them with three different amounts of simultaneous processes. Our small test consisted of 10 simultaneous processes, our medium had 100 and finally our large had 1000. Collecting this data will allow us to examine how the schedulers scale with system utilization. This is important because even though one scheduler may handle a few processes better than another it may be surpassed when it has to handle many processes.

One final note about our methodology is that we tried to eliminate as many inconsistencies and outliers in our data as possible. The main way we tried to accomplish this was by performing the above tests multiple times and using the average values for comparison. Although we made an effort to keep the testing environment controlled throughout all of the tests there are many different factors that could influence any individual trial. Averaging values across multiple trials should greatly improve the consistency of our data by adding some reliability and reproducibility to it. Thus, the values we will use in comparison and analysis will hopefully be more accurate and not as random.

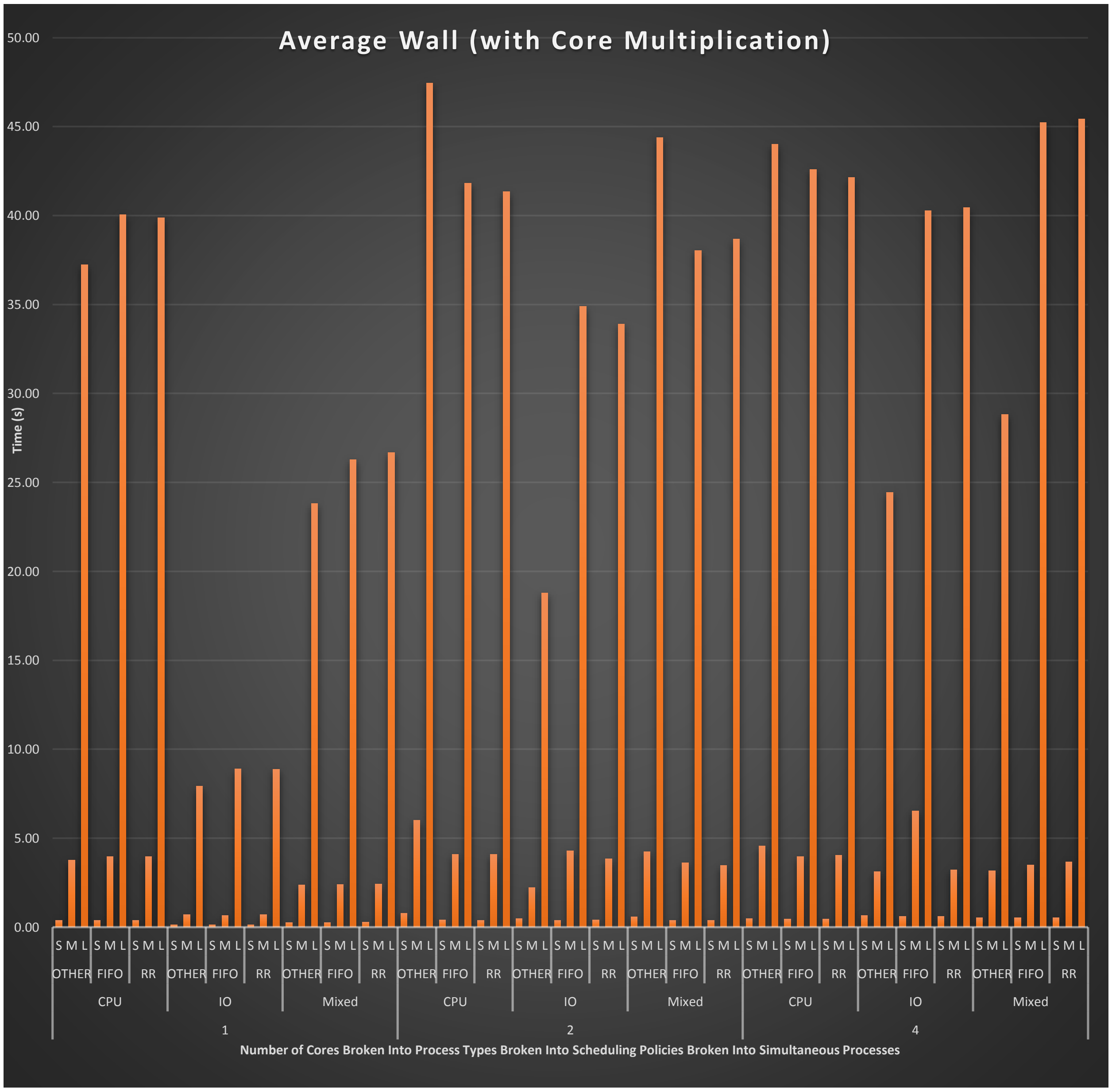
Results:

Figure 1: Average Wall Times without Core Multiplication



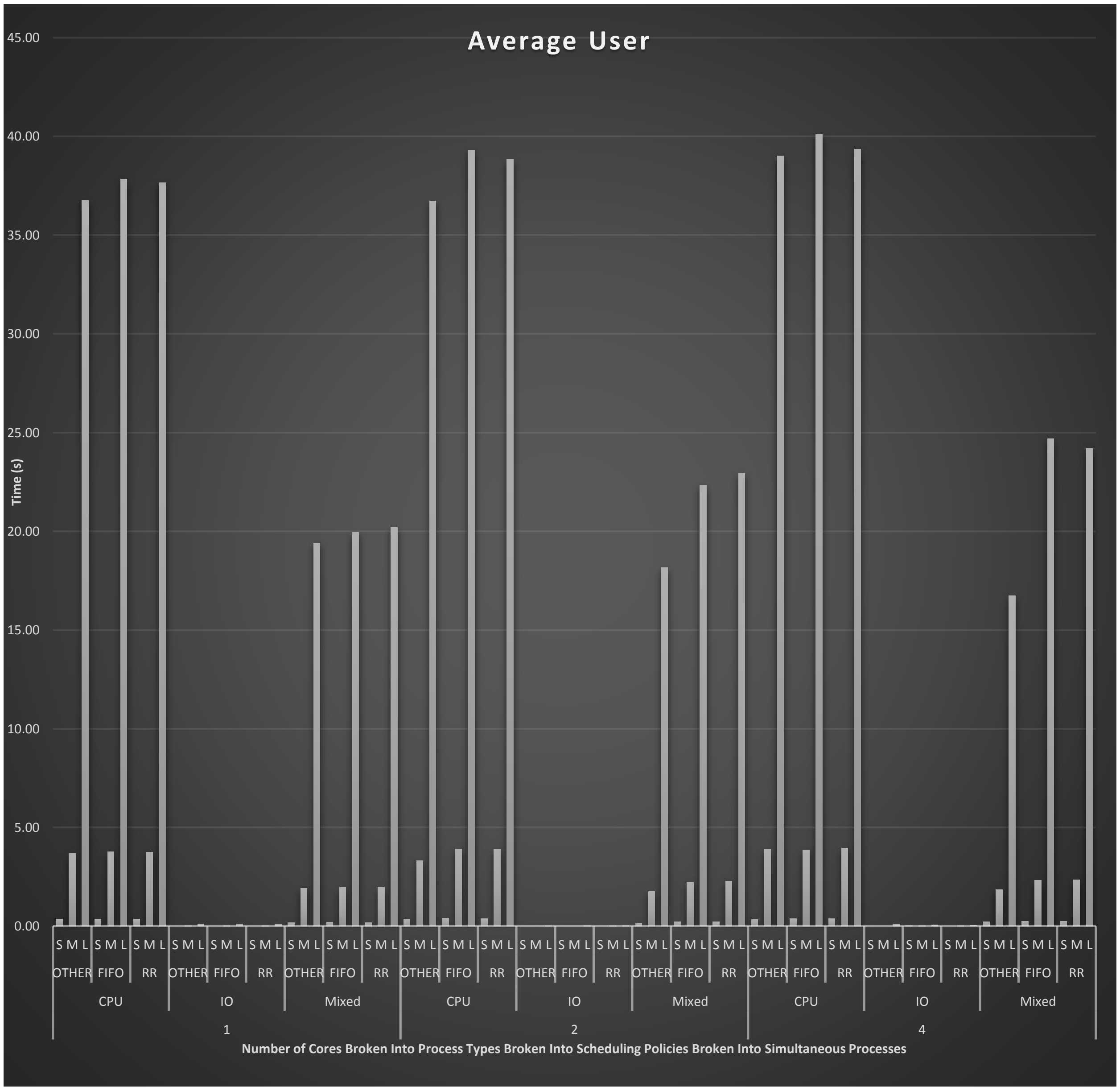
This graph shows benchmarks for average wait times without multiplication for the number of cores. This allows us to see how the actual time required scales down as the number of cores utilized increases. Running simultaneous processes across multiple cores essentially lets us utilize more than 100% of the CPU which decreases the wall times. We can compare how different schedulers handled the same process type with the same number of cores, but comparisons across program types or number of cores aren't useful in this chart.

Figure 2: Average Wall Times with Core Multiplication



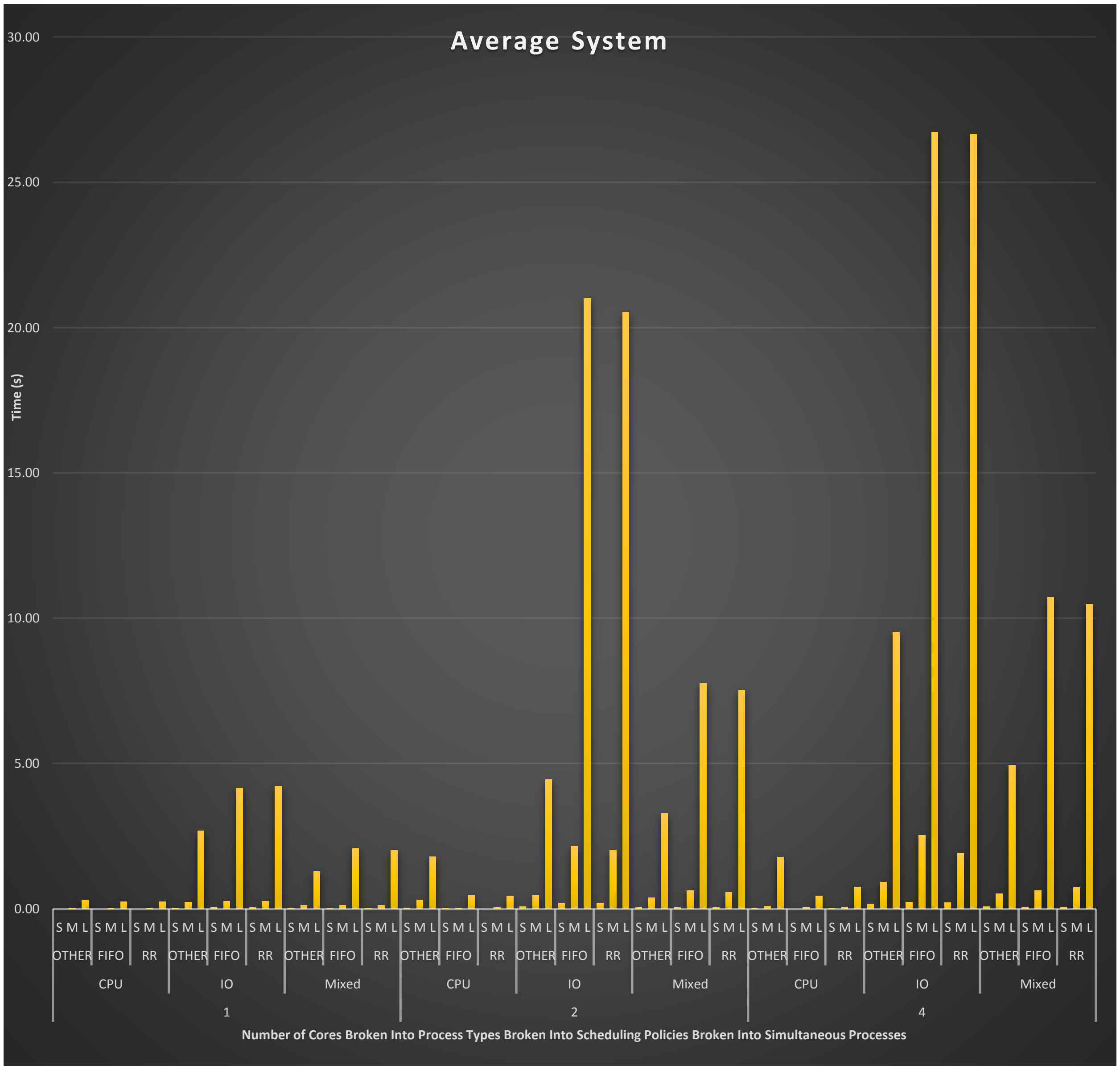
This graph is similar to the previous one except the times for two and four cores were multiplied by two and four, respectively. Although these aren't the true wall times, they allow us to better analyze the performance differences between the schedulers as opposed to the core utilization. Thus, you can make comparisons across different amounts of cores, but comparisons between different process types still won't be useful.

Figure 3: Average User Times



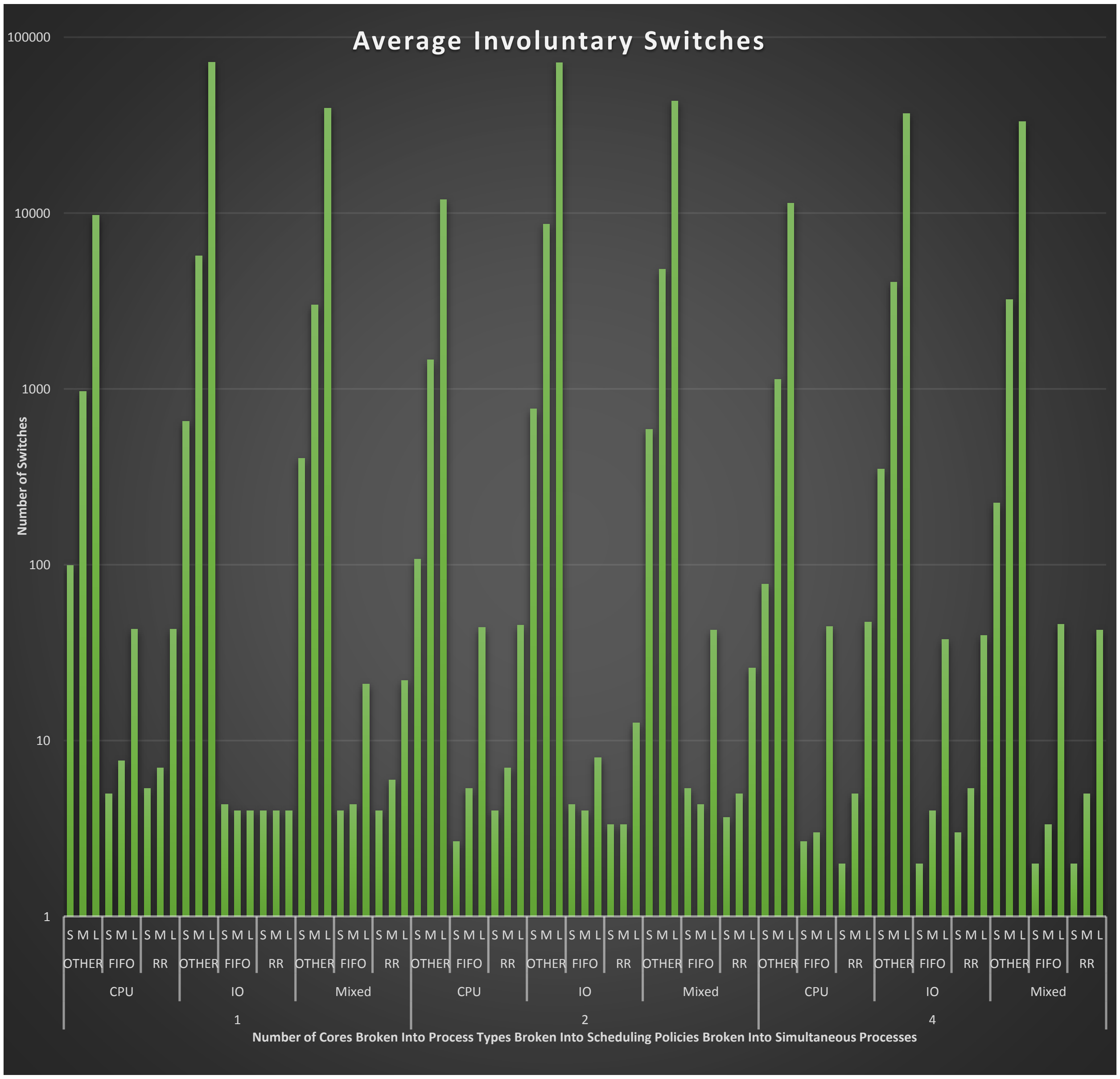
This chart shows us the time average times spent executing user code. The I/O processes hardly show up on this scale because they spent much more time reading and writing than executing the code and making computations which is what we would expect. Also, the mixed processes appear to be somewhere in the middle between the CPU and I/O bounded processes which we would also expect.

Figure 4: Average System Times



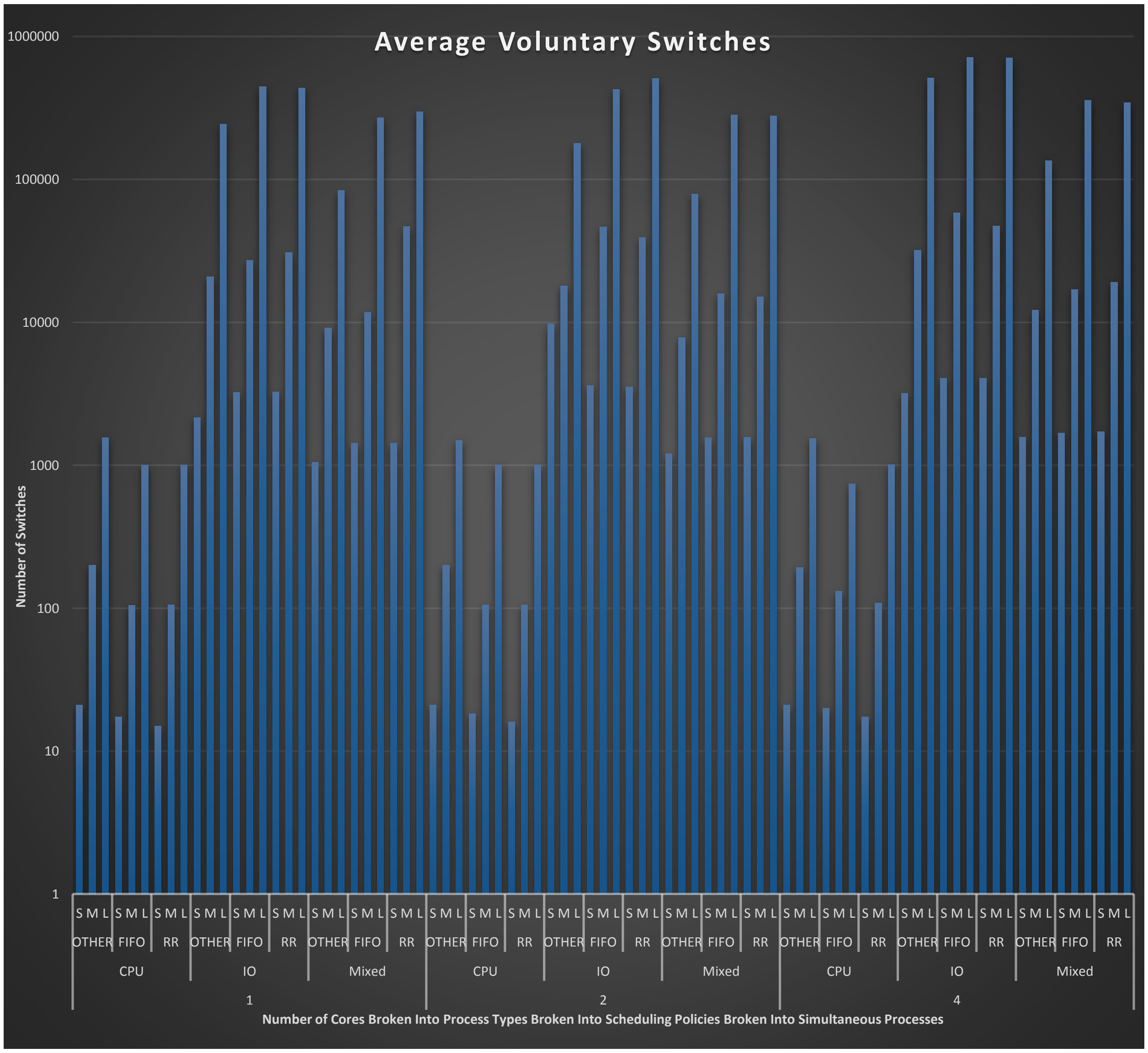
This chart shows the average times spent executing system (kernel) code. We can see that the system time scales up substantially with CPU utilization. This is probably because more system code is required to execute simultaneous processes and load balance with more cores. Additionally, the round robin scheduler (almost always) has higher system times than the other schedulers. This is because round robin has more context switches and need more kernel intervention when scheduling the processes.

Figure 5: Average Involuntary Switches



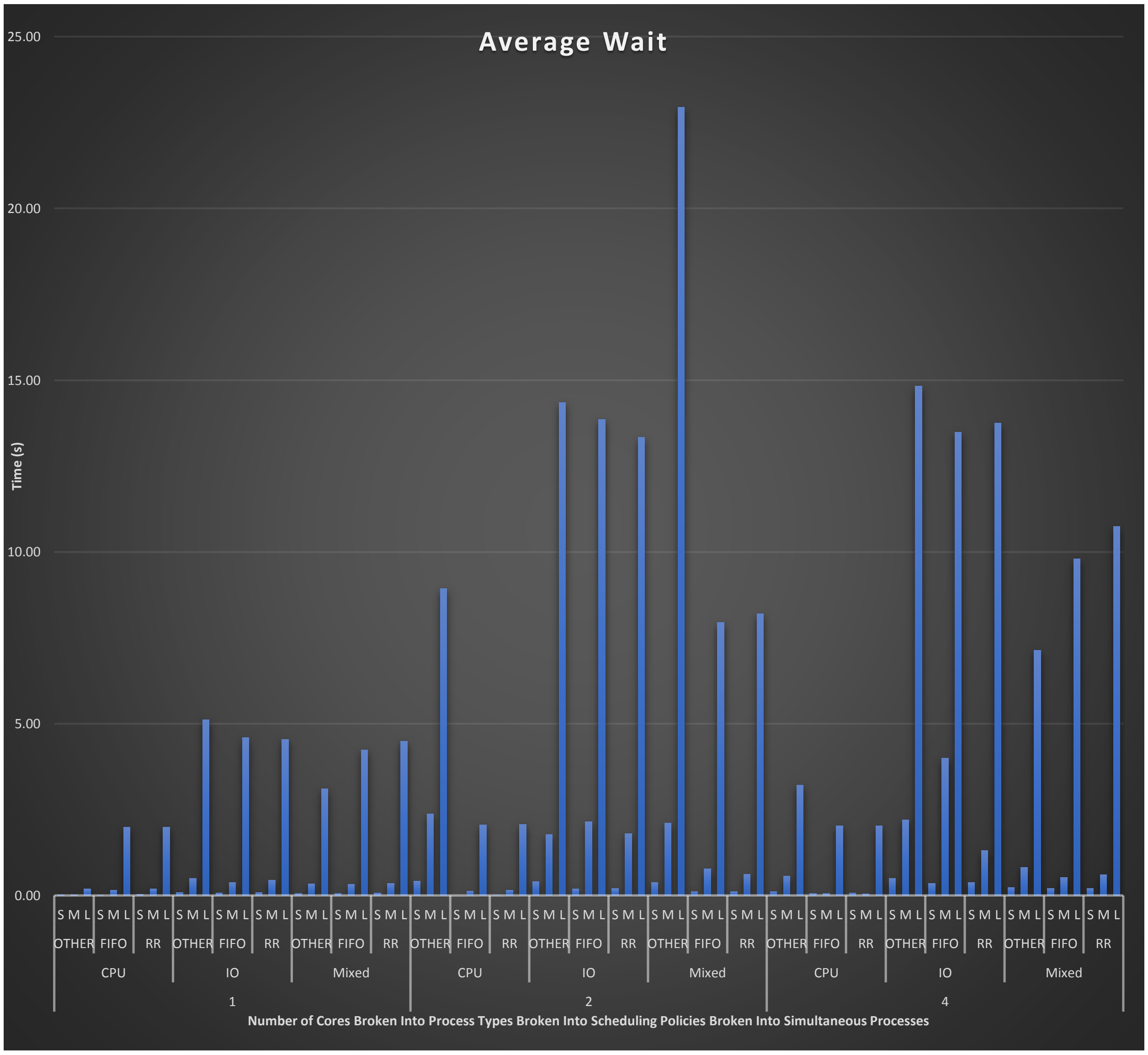
This graph shows the average involuntary switches that happened during each test. Note that the quantity of switches on the vertical axis has a logarithmic scale. This makes the data easier to read because the upper quantities of involuntary switches were extremely high compared to the lower values. We can see that the Round Robin and Other scheduling policies have significantly more involuntary context switches than the FIFO scheduler. This makes sense because these two policies schedule processes via timeslices whereas FIFO does not. All three process types had similar amounts of involuntary switches (although CPU had slightly less) and it stayed pretty constant as the amount of cores increased.

Figure 6: Average Voluntary Switches



This chart shows the average amount of voluntary switches that happened in the tests. Similarly to the last graph, the quantity of switches on the vertical axis is displayed on a logarithmic scale. This set of data is very uniform, and it hardly changes at all with varying numbers of cores. Additionally, the CPU processes consistently had less voluntary switches than I/O or mixed. This makes sense because the CPU bounded program wouldn't want to give up many of its timeslices whereas the I/O and mixed programs had more opportunities to when it was blocked on I/O.

Figure 7: Average Wait Times



This graph shows the average wait times for each test. The I/O programs generally had the highest wait times because they had to wait for the reading/writing to finish. This is different from the CPU programs which were limited by CPU calculations and didn't have to spend as much time waiting for something to happen. The mixed programs mostly fall somewhere in-between the other two which is exactly what we would expect.

Analysis:

Gathering all of this data allows to see many of the nuanced differences between the three different schedulers. First, let's examine which scheduler is best suited for each process type in terms of run-time. The Other scheduler almost always better suited for CPU limited processes than Round Robin and FIFO schedulers on one core. This is somewhat unintuitive because it is also the most inefficient scheduler as far as overhead efficiency is concerned with the same constraints. The reason why Other is the best scheduler for CPU bounded processes is because it has significantly less wait times than both FIFO and RR (one order of magnitude less). For example, for the large amount of simultaneous processes, Other had a wait time of .19 seconds compared to FIFO's 1.98 RR's 1.99. Thus, even though it had slightly higher system times Other still outperforms. The one exception is for very small inputs FIFO's extremely low amount of context switch overhead actually outperforms Other, but the difference of .01 seconds is essentially negligible.

For I/O limited processes the best scheduler is a tossup between FIFO and Other. For small quantities of simultaneous processes, the two schedulers are about the same. The interesting thing is that the circumstances are almost exactly the opposite as CPU bounded processes. In this case, FIFO actually has almost double the amount of voluntary context switches. This results in FIFO having more overhead and more system time than Other, but this is balanced out by Other having a much higher wait time than

FIFO. Overall, the two schedulers are nearly identical for small amounts of simultaneous processes, FIFO has a slight edge for medium amounts of processes and Other goes back to being better with large amounts. Round Robin is not well suited for I/O processes because it has a high amount of switch overhead as well as high wait times. Therefore, FIFO would be the best scheduler for smaller amounts of processes, but Other scales more efficiently.

For mixed processes the best suited scheduler becomes even more ambiguous. It seems like the I/O portion of the process may have had a slightly largely impact on the results than the CPU portion because they are very similar to what we saw with the I/O process. Other loses less time to overhead than both FIFO and RR, but it also has a higher average wait time that almost balances it out. FIFO outperforms RR for every amount of simultaneous processes. Similarly, we see that Other scales better into larger process quantities than FIFO. This is probably because Other's higher wait time is eventually outweighed by FIFO's greater amount of overhead. Overall, I would say that Other is the best suited scheduler for mixed processes because it is very close to FIFO for small and medium amounts of processes (even though FIFO performs better) while having a more pronounced performance edge for large amounts.

Scaling up the amount of cores being utilized makes the differences between the schedulers more clear-cut. For both two and four cores (as compared to one previously) and all sizes of simultaneous processes there are two obvious winners. The FIFO scheduler is clearly the best suited for CPU limited processes. It outperforms Other by having both significantly less wait times and context switching overhead. On the other hand, despite still having a higher average wait time than FIFO, Other has less than half of FIFO's system time for I/O and mixed processes. This results in it having a huge performance advantage for these two process types. One more important thing to note is that FIFO consistently outperforms RR under all categories for two and four cores. Therefore, for a CPU with more than one core FIFO is the best suited scheduler for CPU limited processes, Other is the best for I/O bounded and mixed processes and RR is the worst suited for all three.

Conclusion:

In taking these benchmarks I learned a lot about the various schedulers because the results were not at all what I expected. The part of the data that surprised me the most was that the FIFO scheduler consistently outperformed Round Robin. Going into this lab I didn't really know what the Other scheduler was and thus didn't really know how it would compare to the other two. On the other hand, since they were both real-time schedulers (same priority) I thought that RR would definitely be more efficient than FIFO especially with I/O and mixed processes. My reasoning for this is because I thought that RR's timeslices would be more efficient (and could yield voluntarily more) when a process was waiting for a read or write. I was wrong was because I didn't anticipate that RR often had a higher wait time than FIFO. Additionally, I didn't think that the overhead from context switches would be so large especially as the number of simultaneous processes scaled up.

Even though it was only a small portion of the lab I also learned a great deal about CPU core utilization. I was surprised to see that the difference between one and two cores was huge while the difference between two and four cores was more subtle. Additionally, I was surprised to see that even though having more cores generally meant less user and system times it also meant far greater wait times. This negated some of the performance advantages of having more cores although it was obviously much more efficient solely in terms of execution times. In conclusion, this lab hugely improved my understanding of Linux's scheduler while also allowing me to inherently learn about how a CPU is impacted by its number of cores.

References:

Sayler, Andy (9 March 2012) *Pi-Sched.c* [Computer program] (Accessed 5 April 2015)

Sayler, Andy (9 March 2012) *testscript* [Computer program] (Accessed 5 April 2015)

Sayler, Andy (20 March 2012) *Rw.c* [Computer program] (Accessed 5 April 2015)

Appendix A:

Figure 8: Raw Data

Cores	TYPE	Scheduler	Size	Avg Wall	Avg Wall (Core Multiplication)	Avg User	Avg System	Avg Wait	Avg I-Switches	Avg V-Switches
1	CPU	OTHER	S	0.38	0.38	0.36	0.00	0.02	99	21
			M	3.75	3.75	3.69	0.03	0.03	968	201
			L	37.24	37.24	36.75	0.30	0.19	9706	1566
		FIFO	S	0.37	0.37	0.36	0.00	0.01	5	17
			M	3.96	3.96	3.76	0.04	0.16	8	105
			L	40.06	40.06	37.83	0.24	1.98	43	1005
		RR	S	0.40	0.40	0.36	0.00	0.03	5	15
			M	3.97	3.97	3.74	0.03	0.20	7	105
			L	39.89	39.89	37.66	0.24	1.99	43	1006
	IO	OTHER	S	0.13	0.13	0.00	0.03	0.10	654	2159
			M	0.72	0.72	0.01	0.22	0.50	5738	20878
			L	7.91	7.91	0.11	2.69	5.12	72345	243130
		FIFO	S	0.12	0.12	0.00	0.05	0.08	4	3237
			M	0.66	0.66	0.01	0.27	0.39	4	27292
			L	8.88	8.88	0.12	4.16	4.60	4	448046
		RR	S	0.13	0.13	0.00	0.04	0.09	4	3253
			M	0.70	0.70	0.00	0.26	0.44	4	30770
			L	8.88	8.88	0.11	4.22	4.55	4	435987
	Mixed	OTHER	S	0.27	0.27	0.19	0.02	0.07	404	1051
			M	2.39	2.39	1.92	0.12	0.34	3021	9139
			L	23.80	23.80	19.41	1.29	3.11	39610	83430
		FIFO	S	0.27	0.27	0.20	0.01	0.06	4	1427
			M	2.41	2.41	1.96	0.12	0.33	4	11696
			L	26.28	26.28	19.95	2.09	4.24	21	270392
		RR	S	0.28	0.28	0.20	0.01	0.07	4	1427
			M	2.43	2.43	1.95	0.12	0.36	6	46914
			L	26.70	26.70	20.20	2.01	4.50	22	297838
2	CPU	OTHER	S	0.40	0.79	0.36	0.01	0.42	108	21
			M	3.00	6.00	3.32	0.30	2.38	1469	200
			L	23.73	47.46	36.73	1.79	8.94	11919	1495
		FIFO	S	0.20	0.41	0.40	0.00	0.00	3	18
			M	2.04	4.09	3.91	0.04	0.13	5	106
			L	20.92	41.84	39.31	0.46	2.07	44	1008

		RR	S	0.19	0.39	0.38	0.00	0.01	4	16
			M	2.05	4.09	3.89	0.05	0.16	7	106
			L	20.67	41.35	38.83	0.44	2.07	45	1006
	IO	OTHER	S	0.24	0.48	0.00	0.07	0.41	772	9751
			M	1.12	2.23	0.00	0.46	1.77	8648	18000
			L	9.40	18.79	0.00	4.44	14.35	71747	178739
		FIFO	S	0.19	0.38	0.00	0.19	0.19	4	3620
			M	2.15	4.30	0.00	2.15	2.15	4	46580
			L	17.45	34.91	0.04	21.00	13.87	8	426090
		RR	S	0.20	0.41	0.00	0.20	0.21	3	3543
			M	1.92	3.84	0.01	2.02	1.81	3	39407
			L	16.95	33.91	0.04	20.53	13.34	13	507263
	Mixed	OTHER	S	0.30	0.59	0.17	0.05	0.37	592	1206
			M	2.13	4.25	1.75	0.38	2.12	4793	7831
			L	22.19	44.38	18.16	3.28	22.94	43579	79023
		FIFO	S	0.19	0.39	0.22	0.05	0.12	5	1562
			M	1.82	3.64	2.22	0.64	0.78	4	15902
			L	19.02	38.04	22.33	7.76	7.95	43	283825
		RR	S	0.20	0.39	0.24	0.04	0.11	4	1570
			M	1.73	3.47	2.29	0.56	0.62	5	15129
			L	19.34	38.67	22.94	7.52	8.21	26	278791
4	CPU	OTHER	S	0.12	0.49	0.36	0.01	0.12	78	21
			M	1.14	4.56	3.89	0.10	0.57	1132	193
			L	11.00	44.00	39.00	1.77	3.22	11400	1546
		FIFO	S	0.12	0.47	0.40	0.00	0.07	3	20
			M	0.99	3.97	3.87	0.04	0.06	3	132
			L	10.65	42.60	40.11	0.45	2.04	45	741
		RR	S	0.12	0.47	0.38	0.01	0.08	2	17
			M	1.01	4.05	3.95	0.05	0.05	5	109
			L	10.54	42.15	39.36	0.75	2.03	47	1009
	IO	OTHER	S	0.17	0.67	0.00	0.16	0.51	350	3188
			M	0.78	3.12	0.00	0.92	2.20	4059	31902
			L	6.11	24.45	0.11	9.52	14.83	36971	511708
		FIFO	S	0.15	0.61	0.03	0.22	0.36	2	4080
			M	1.63	6.53	0.00	2.53	4.00	4	58322
			L	10.07	40.27	0.06	26.72	13.48	38	714783
		RR	S	0.15	0.60	0.00	0.22	0.38	3	4075
			M	0.81	3.24	0.01	1.92	1.31	5	47229
			L	10.11	40.45	0.05	26.65	13.75	40	711003
	Mixed	OTHER	S	0.13	0.53	0.22	0.07	0.24	224	1576
			M	0.80	3.19	1.86	0.51	0.82	3221	12143
			L	7.21	28.83	16.73	4.94	7.15	33237	135793
		FIFO	S	0.13	0.52	0.25	0.06	0.21	2	1678
			M	0.87	3.49	2.33	0.64	0.53	3	16947
			L	11.31	45.23	24.70	10.72	9.81	46	356531
		RR	S	0.13	0.53	0.25	0.06	0.22	2	1718
			M	0.92	3.68	2.34	0.73	0.60	5	19096
			L	11.36	45.44	24.21	10.48	10.75	43	343973

Appendix B:

Figure 9: pi-sched.c – CPU Bounded Program

```
/*
 * File: pi-sched.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 using a specific scheduling policy.
 */

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

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)

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 calculatePi(long iter){
            long iterations = iter;
            double x, y;
   double inCircle = 0.0;
   double inSquare = 0.0;
   double pCircle = 0.0;
   double piCalc = 0.0;

   /* Calculate pi using statistical methode across all iterations*/
```

```
    long i;
    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;
}

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

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


    /* 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;
    }
    if(argc < 4){
                    i = 1;
            }
    /* Set iterations if supplied */
    if(argc > 1){
                    iterations = atol(argv[1]);
                    if(iterations < 1){
                            fprintf(stderr, "Bad 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\n");
			exit(EXIT_FAILURE);
		}
}

/* Set process to max prioty 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));

	if(argc > 3){
		i = atoi(argv[3]);
	}

	int j = 0;
while(j < i){
		pid = fork();
		if(pid == 0){
			calculatePi(iterations);
			break;
			exit(EXIT_SUCCESS);
		}
		else if(pid < 0){
			perror("Forking error");
			exit(EXIT_FAILURE);
		}
		else{
			j++;
		}
		}

		if(pid != 0){
			while(waitpid(-1, NULL, 0) > 0){}
		}

   return 0;
}
```

Figure 10: rw-sched.c – I/O Bounded Program

```
/*
 * File: rw.c
 * Author: Andy Sayler
 * Project: CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/19
 * Modify Date: 2012/03/20
 * 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>
#include <sched.h>

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

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

```
int doRW(ssize_t trans, ssize_t block, char* input, char* output, char* base){
            int rv;
    int inputFD;
    int outputFD;
            char inputFilename[MAXFILENAMELENGTH];
            strncpy(inputFilename, input, MAXFILENAMELENGTH);
    char outputFilename[MAXFILENAMELENGTH];
    strncpy(outputFilename, output, MAXFILENAMELENGTH);
    char outputFilenameBase[MAXFILENAMELENGTH];
    strncpy(outputFilenameBase, base, MAXFILENAMELENGTH);

            ssize_t transfersize = trans;
    ssize_t blocksize = block;
    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;

            /* 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());
    if(rv > MAXFILENAMELENGTH){
                fprintf(stderr, "Output filenmae 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 helpfull 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;
}

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

        pid_t pid;
        struct sched_param param;
        int policy;
        char inputFilename[MAXFILENAMELENGTH];
        char outputFilename[MAXFILENAMELENGTH];
        char outputFilenameBase[MAXFILENAMELENGTH];

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

```
   /* 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 < 1){
                            fprintf(stderr, "Bad transfersize value\n");
                            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 < 1){
                            fprintf(stderr, "Bad blocksize value\n");
                            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(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(strnlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
			fprintf(stderr, "Output filename base is too long\n");
			exit(EXIT_FAILURE);
		}
		strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
}
if(argc < 6){
		policy = SCHED_OTHER;
}
if(argc > 5){
		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\n");
			exit(EXIT_FAILURE);
		}
}

/* Set process to max prioty 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));

if(argc > 6){
		i = atoi(argv[6]);
	}

int j = 0;
while(j < i){
		pid = fork();
		if(pid == 0){
				doRW(transfersize, blocksize, inputFilename, outputFilename,
outputFilenameBase);
				break;
				exit(EXIT_SUCCESS);
			}
			else if(pid < 0){
				perror("Forking error");
				exit(EXIT_FAILURE);
			}
			else{
				j++;
			}
		}

		if(pid != 0){
			while(waitpid(-1, NULL, 0) > 0){}
		}

    return 0;
}
```

Figure 11: mixed.c – Combination of pi-sched.c and rw-sched.c (Mixed CPU and I/O)

```
/*
 * File: mixed.c
 * Author: Cameron Taylor & Steven Conflenti
 * Project: CSCI 3753 Programming Assignment 4
 * Create Date: 2012/03/30
 * Description:
 *	This file contains a mixed IO/CPU program.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <errno.h>
#include <sched.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

// includes for forking
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
#include <signal.h>

#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)

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

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 calculatePi(long iter){
            long iterations = iter;
            double x, y;
   double inCircle = 0.0;
   double inSquare = 0.0;
   double pCircle = 0.0;
   double piCalc = 0.0;

   /* Calculate pi using statistical methode across all iterations*/
   long i;
   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;
}

int doRW(ssize_t trans, ssize_t block, char* input, char* output, char* base){
            int rv;
   int inputFD;
   int outputFD;
            char inputFilename[MAXFILENAMELENGTH];
            strncpy(inputFilename, input, MAXFILENAMELENGTH);
   char outputFilename[MAXFILENAMELENGTH];
   strncpy(outputFilename, output, MAXFILENAMELENGTH);
   char outputFilenameBase[MAXFILENAMELENGTH];
   strncpy(outputFilenameBase, base, MAXFILENAMELENGTH);

            ssize_t transfersize = trans;
   ssize_t blocksize = block;
   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;

            /* 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());
```

```
    if(rv > MAXFILENAMELENGTH){
                        fprintf(stderr, "Output filenmae 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 helpfull 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;
}

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

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

    char inputFilename[MAXFILENAMELENGTH];
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];

    ssize_t transfersize = 0;
    ssize_t blocksize = 0;


    /* Process program arguments to select iterations and policy */
    /* Set default iterations if not supplied */
    if(argc < 2){
                    iterations = DEFAULT_ITERATIONS;
    }
    /* Set iterations if supplied */
    if(argc > 1){
                    iterations = atol(argv[1]);
                    if(iterations < 1){
                            fprintf(stderr, "Bad iterations value\n");
                            exit(EXIT_FAILURE);
                    }
    }

    /* Process program arguments to select run-time parameters */
    /* Set supplied transfer size or default if not supplied */
    if(argc < 3){
                    transfersize = DEFAULT_TRANSFERSIZE;
    }
    else{
                    transfersize = atol(argv[2]);
                    if(transfersize < 1){
                            fprintf(stderr, "Bad transfersize value\n");
                            exit(EXIT_FAILURE);
                    }
    }
    /* Set supplied block size or default if not supplied */
    if(argc < 4){
                    blocksize = DEFAULT_BLOCKSIZE;
    }
    else{
                    blocksize = atol(argv[3]);
                    if(blocksize < 1){
                            fprintf(stderr, "Bad blocksize value\n");
                            exit(EXIT_FAILURE);
                    }
    }
    /* Set supplied input filename or default if not supplied */
    if(argc < 5){
                    if(strnlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >=
  MAXFILENAMELENGTH){
                            fprintf(stderr, "Default input filename too long\n");
                            exit(EXIT_FAILURE);
                    }
                    strncpy(inputFilename, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else{
                    if(strnlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
                            fprintf(stderr, "Input filename too long\n");
				exit(EXIT_FAILURE);
			}
			strncpy(inputFilename, argv[4], MAXFILENAMELENGTH);
    }
    /* Set supplied output filename base or default if not supplied */
    if(argc < 6){
			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(strnlen(argv[5], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
				fprintf(stderr, "Output filename base is too long\n");
				exit(EXIT_FAILURE);
			}
			strncpy(outputFilenameBase, argv[5], MAXFILENAMELENGTH);
    }

    if(argc < 7){
			policy = SCHED_OTHER;
    }
    else{
			if(!strcmp(argv[6], "SCHED_OTHER")){
				policy = SCHED_OTHER;
			}
			else if(!strcmp(argv[6], "SCHED_FIFO")){
				policy = SCHED_FIFO;
			}
			else if(!strcmp(argv[6], "SCHED_RR")){
				policy = SCHED_RR;
			}
			else{
				fprintf(stderr, "Unhandeled scheduling policy\n");
				exit(EXIT_FAILURE);
			}
	}

	/* Set process to max prioty 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));

            if(argc < 8){
                        i = 1;
            }
            else{
                        i = atoi(argv[7]);
            }

            int j = 0;
    while(j < i){
                        pid = fork();
                        if(pid == 0){
                                    calculatePi(iterations);
                                    doRW(transfersize, blocksize, inputFilename, outputFilename,
outputFilenameBase);
                                    break;
                                    exit(EXIT_SUCCESS);
                        }
                        else if(pid < 0){
                                    perror("Forking error");
                                    exit(EXIT_FAILURE);
                        }
                        else{
                                    j++;
                        }
            }

            if(pid != 0){
                        while(waitpid(-1, NULL, 0) > 0){}
            }

    return 0;
}
```