Garrett Long
Programming Assignment 4
Operating Systems
4/6/15

Investigation of the Linux Scheduler

Abstract:

The Linux scheduler includes several different scheduling policies which users can run their applications with. The most popular of them are SCHED_OTHER, SCHED_FIFO, and SCHED_RR. Each one of these scheduling policies are tailored to a certain type of computer application. After investigation it became apparent that SCHED_OTHER is the superior policy for personal computing while SCHED_FIFO might be suited to supercomputers and running large algorithms. Unfortunately, the SCHED_RR policy did not provide much advantage over SCHED_OTHER and was tailored to a very rare type of computing. This investigation proves through careful analysis of data that SCHED_OTHER is the most versatile and reliable scheduling policy.

Introduction:

The goal of this report is to analyze the performance of three types of Linux scheduling policies SCHED_OTHER, SCHED_FIFO, and SCHED_RR. These scheduling policies were applied to three program types CPU bound, IO bound, and mixed (CPU and IO bound). Under each of these types of programs three scales of processes were also investigated small (10 processes), medium (100 processes), and large (1000 processes). After each test scenario the test program produced benchmark data which shows the different attributes of each scheduling policy. This data is analyzed in this report and allowed for me to create accurate conclusions about which scheduling policies are desired under each scenario.

Method:

Types of Benchmarks:

My program produces eight different benchmarks including total runtime, runtime per process, average turnaround time, lowest turnaround time, largest turnaround time, average wait time, lowest wait time, and largest wait time. I found that each of these benchmarks were key in analyzing the pro's and con's of each scheduling policy.

Total runtime is the simplest of the benchmarks recording only the total time for a test case to run. Total runtime is computed using two timespecs, one set before the test case runs and one set after the test case finishes. The difference between these times is the total runtime of the program.

Runtime per process, is a benchmark which takes the total runtime of the program acquired in the step above and divides it by the number of processes run during the test case. This benchmark is vital because it shows the productivity of each scheduling policy when applied to the same program type and also shows the performance between scales.

Average turnaround time, is a benchmark which shows the average time it took for a process to be completed after scheduling. A timespec was recorded before each process ran and after each process finished. The difference between these two times were added to a total turnaround time and then divided by the number of processes.

Average wait time, is a benchmark which shows the average time a process was waiting for CPU time in the queue during the test scenario. As each process finishes running the user CPU time and system CPU time are recorded using getrusage. This time is subtracted from the processes turnaround time to create the processes wait time. Each wait time is added together and then divided by the number of processes at the test case to produce the average wait time.

Types of Programs:

Three types of program are used to create benchmarks for each scheduling policy including CPU bound, IO bound, and mixed (CPU and IO bound). The CPU bound program used is pi.c which is a statistically-based pi calculator. The IO bound program used is rw.c which copies data from an input file to an output file. It should be noted that each process reads from the same input file and writes to its own output file. The mixed program used is mix.c which reads data from an input file shared by all processes, calculates pi statistically, and then writes to a file owned by each process. Three different types of programs are used in this experiment in order to investigate which scheduling policies are best suited for each program type.

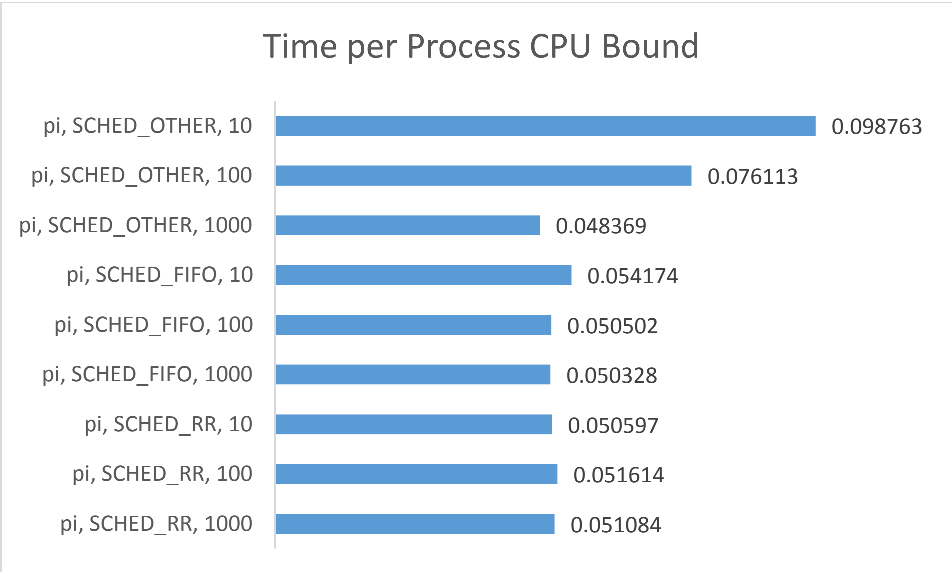
Types of Scales:

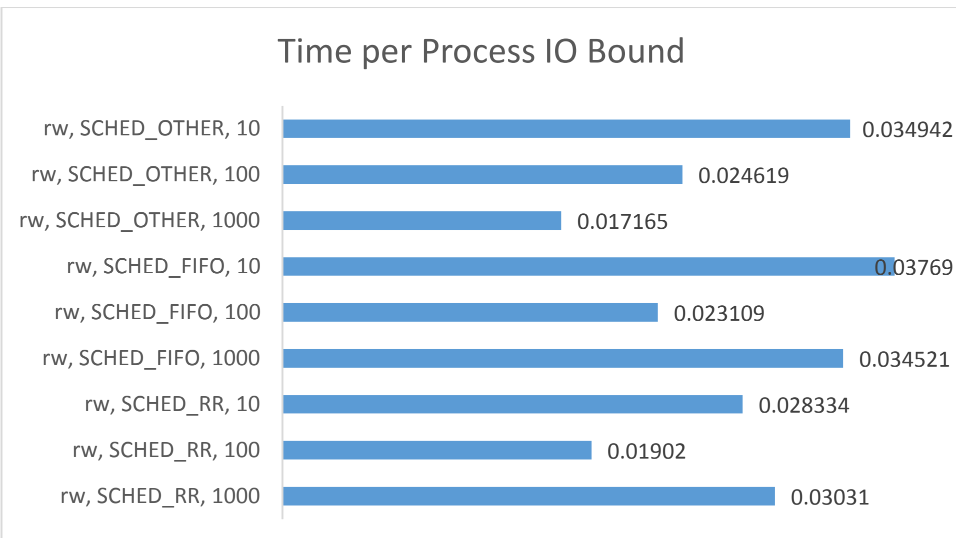
Three sizes of scheduling scales were used in this experiment small, medium, and large. The small scale consisted of 10 simultaneous processes, the medium scale consisted of 100 simultaneous processes, and the large scale consisted of 1000 processes.

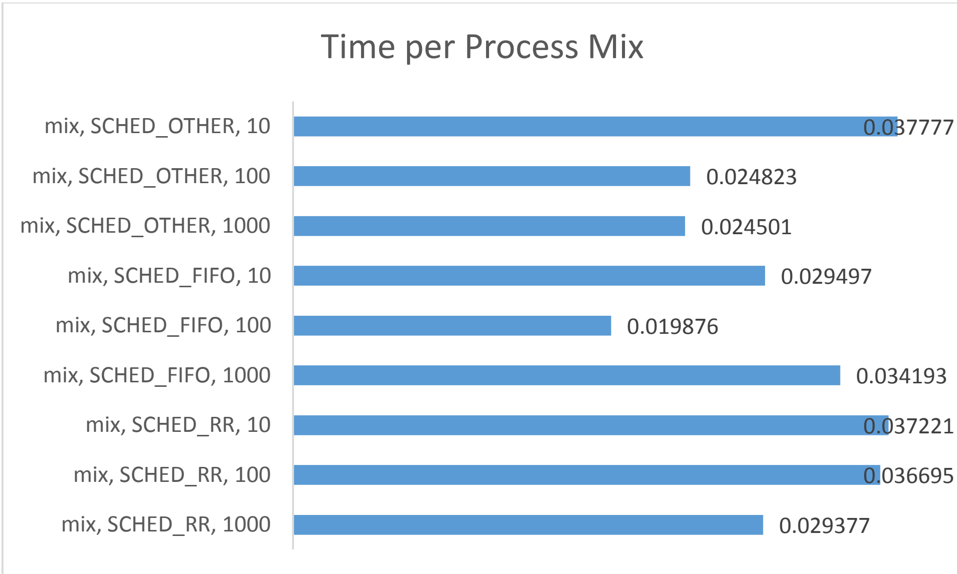
Test System and Setup:

The testing environment for this project is a 32 bit Ubuntu system with 8872 MB of memory and 4 processors. This platform was run using Oracle VM VirtualBox on a Lenovo Y500 laptop running Windows 8.1 x64 with 8 GB of memory and an Intel Core i7-3630QM processor.

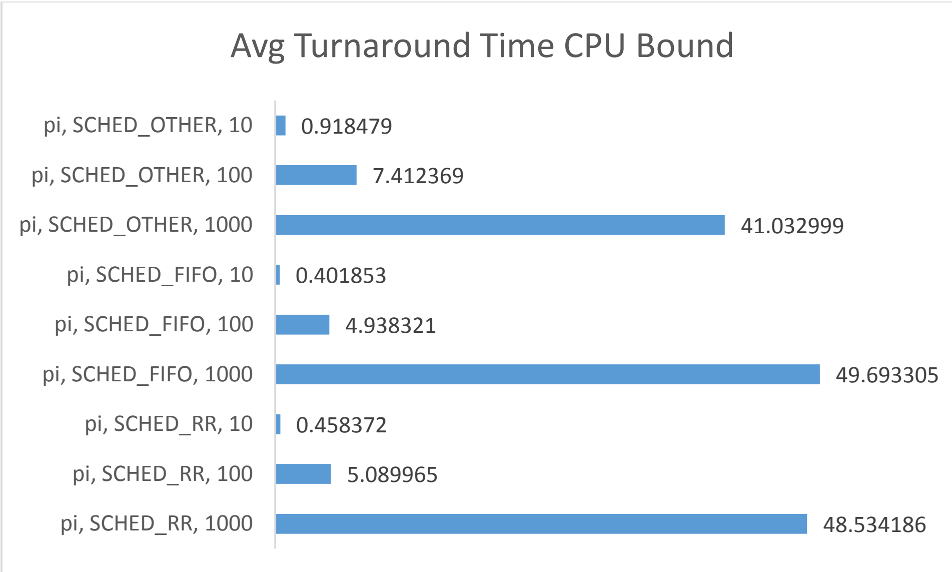
Results:


Time per Process CPU Bound
pi, SCHED_OTHER, 10
0.098763
pi, SCHED_OTHER, 100
0.076113
pi, SCHED_OTHER, 1000
0.048369
pi, SCHED_FIFO, 10
0.054174
pi, SCHED_FIFO, 100
0.050502
pi, SCHED_FIFO, 1000
0.050328
pi, SCHED_RR, 10
0.050597
pi, SCHED_RR, 100
0.051614
pi, SCHED_RR, 1000
0.051084

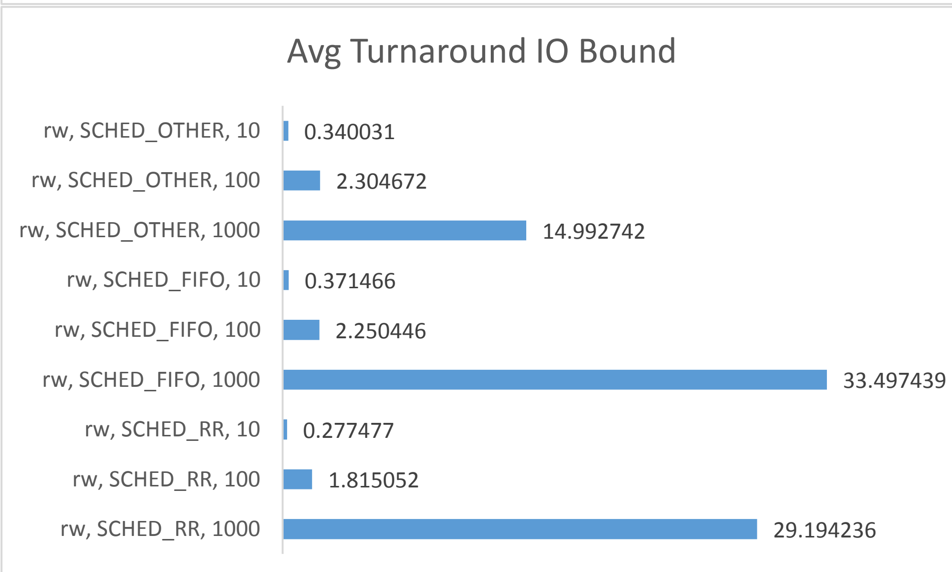

Time per Process IO Bound
rw, SCHED_OTHER, 10
0.034942
rw, SCHED_OTHER, 100
0.024619
rw, SCHED_OTHER, 1000
0.017165
rw, SCHED_FIFO, 10
0.03769
rw, SCHED_FIFO, 100
0.023109
rw, SCHED_FIFO, 1000
0.034521
rw, SCHED_RR, 10
0.028334
rw, SCHED_RR, 100
0.01902
rw, SCHED_RR, 1000
0.03031



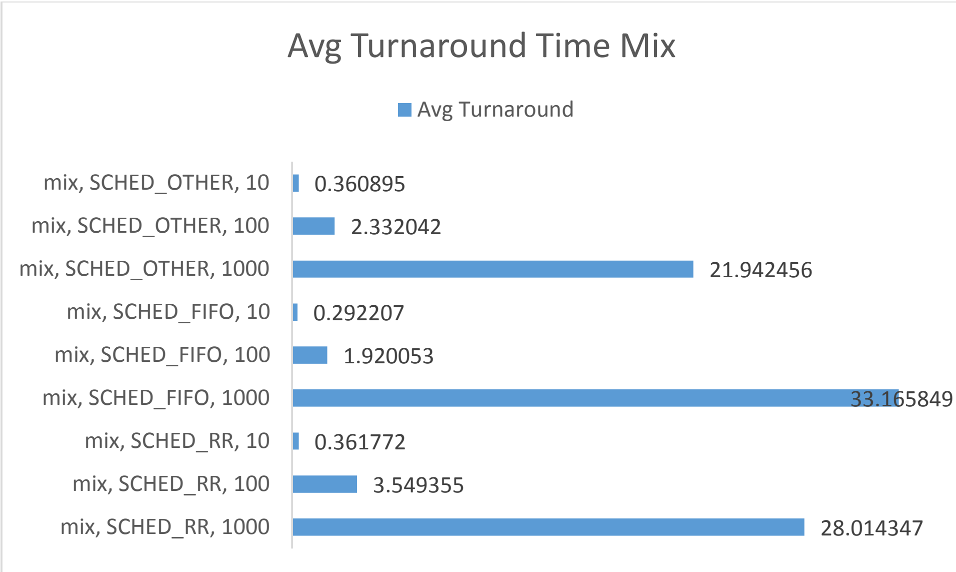
Time per Process Mix
mix, SCHED_OTHER, 10
0.037777
mix, SCHED_OTHER, 100
0.024823
mix, SCHED_OTHER, 1000
0.024501
mix, SCHED_FIFO, 10
0.029497
mix, SCHED_FIFO, 100
0.019876
mix, SCHED_FIFO, 1000
0.034193
mix, SCHED_RR, 10
0.037221
mix, SCHED_RR, 100
0.036695
mix, SCHED_RR, 1000
0.029377

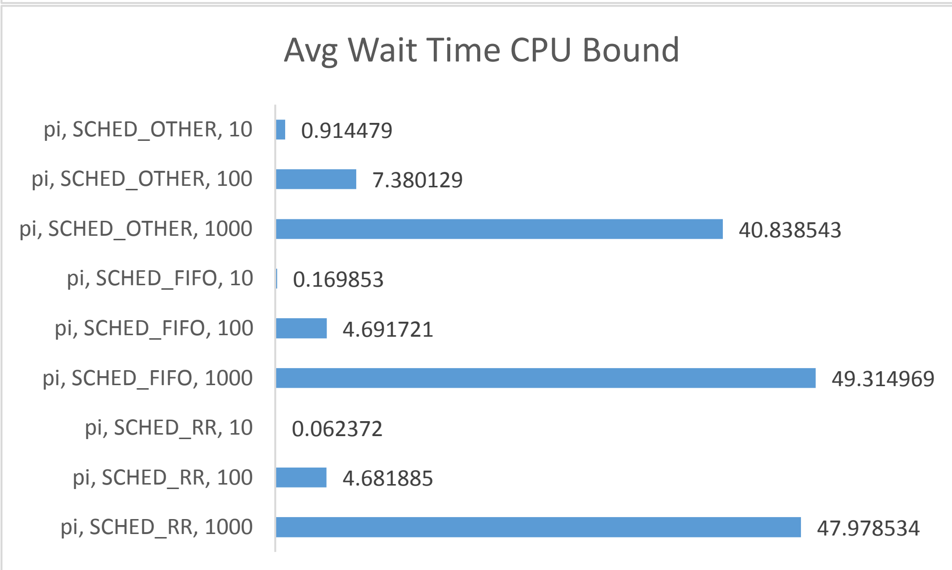


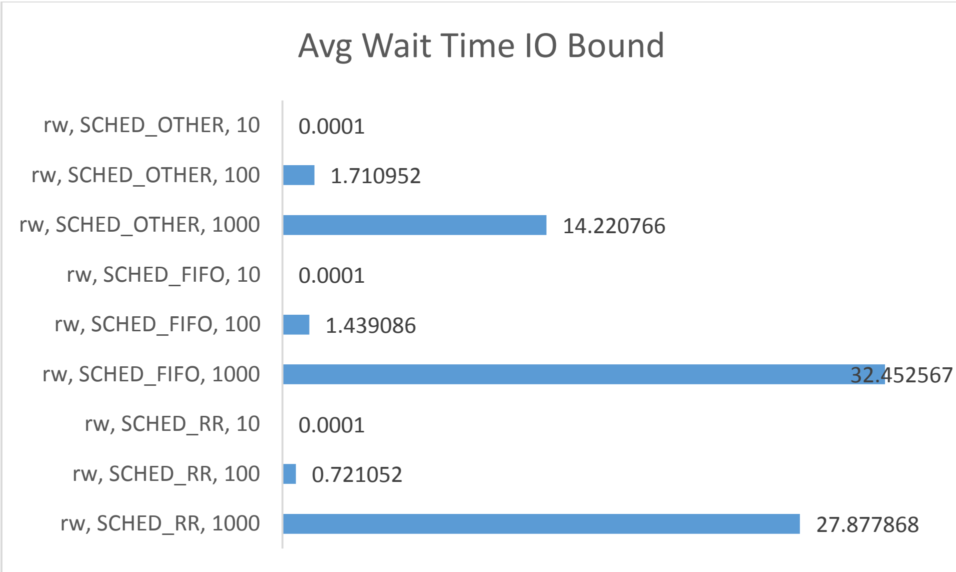
Avg Turnaround Time CPU Bound
pi, SCHED_OTHER, 10
0.918479
pi, SCHED_OTHER, 100
7.412369
pi, SCHED_OTHER, 1000
41.032999
pi, SCHED_FIFO, 10
0.401853
pi, SCHED_FIFO, 100
4.938321
pi, SCHED_FIFO, 1000
49.693305
pi, SCHED_RR, 10
0.458372
pi, SCHED_RR, 100
5.089965
pi, SCHED_RR, 1000
48.534186

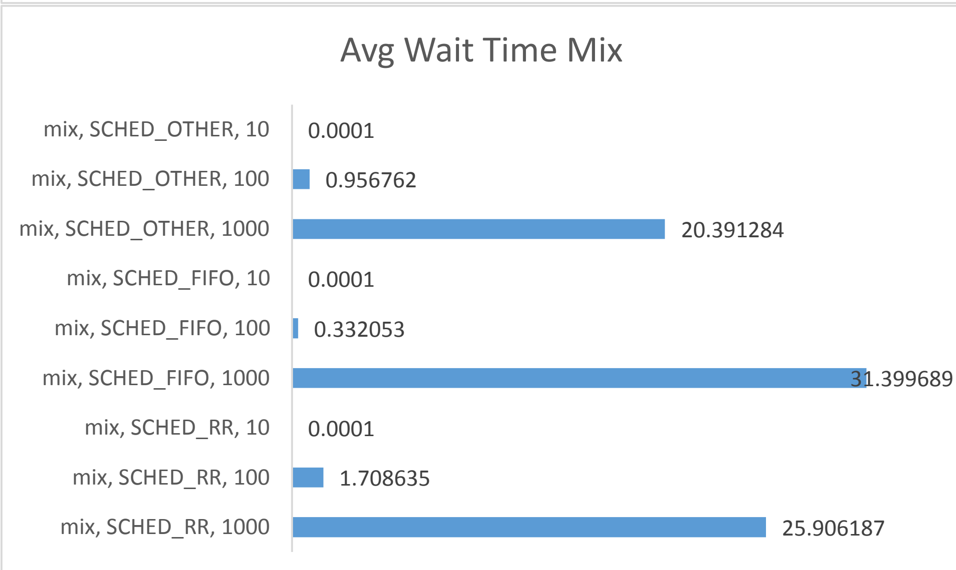


Avg Turnaround IO Bound
rw, SCHED_OTHER, 10
0.340031
rw, SCHED_OTHER, 100
2.304672
rw, SCHED_OTHER, 1000
14.992742
rw, SCHED_FIFO, 10
0.371466
rw, SCHED_FIFO, 100
2.250446
rw, SCHED_FIFO, 1000
33.497439
rw, SCHED_RR, 10
0.277477
rw, SCHED_RR, 100
1.815052
rw, SCHED_RR, 1000
29.194236


Avg Turnaround Time Mix
Avg Turnaround
mix, SCHED_OTHER, 10
0.360895
mix, SCHED_OTHER, 100
2.332042
mix, SCHED_OTHER, 1000
21.942456
mix, SCHED_FIFO, 10
0.292207
mix, SCHED_FIFO, 100
1.920053
mix, SCHED_FIFO, 1000
33.165849
mix, SCHED_RR, 10
0.361772
mix, SCHED_RR, 100
3.549355
mix, SCHED_RR, 1000
28.014347


Avg Wait Time CPU Bound
pi, SCHED_OTHER, 10
0.914479
pi, SCHED_OTHER, 100
7.380129
pi, SCHED_OTHER, 1000
40.838543
pi, SCHED_FIFO, 10
0.169853
pi, SCHED_FIFO, 100
4.691721
pi, SCHED_FIFO, 1000
49.314969
pi, SCHED_RR, 10
0.062372
pi, SCHED_RR, 100
4.681885
pi, SCHED_RR, 1000
47.978534





Analysis:

The three benchmarks which provide relevant information are time per process, average wait time, and average turnaround time. Since total runtime does not take into account the number of processes it does not provide much insight into how each scheduling scheme performed.

Time per Process:

This benchmark is important because it provides insight to which scheduling scheme should be used when the user does not care about turnaround time but instead only cares about speed of computation. The best application for this benchmark is for computer application which do not require user input but perform large batch processes. From this benchmark it is apparent that CPU bound applications run the quickest with the SCHED_FIFO policy. IO bound and mixed applications in general run the quickest with a SCHED_OTHER policy. An interesting observation is that the SCHED_OTHER policy does not perform well with a small number of processes. This most likely occurs because there is a computational overhead with this policy which will waste CPU time. Another interesting observation is that the SCHED_OTHER policy performs extremely well with a large number of processes. The SCHED_RR policy is most effective with IO bound programs with roughly 100 processes, but other than this rare instance is no better than SCHED_OTHER.

Average Turnaround Time:

The average turnaround time benchmark is important because programs which require constant user input or need to give quick responses to other devices need to have a superior turnaround time. If a job is left on the queue too long than other devices may fail or the user may become frustrated with a slow GUI. In this experiment the focus of the turnaround time benchmark was on the medium and large scale tests because the small scale tests were quick enough were turnaround time was irrelevant. In every type of program (IO, CPU, or mix) the SCHED_OTHER policy provided the fastest average turnaround time when there 1000 processes. This makes sense since this scheduler is regarded as the fair scheduler. As expected the SCHED_FIFO policy had the worst turnaround time and SCHED_RR was in between

SCHED_FIFO and SCHED_OTHER. An interesting observation is that SCHED_OTHER was extremely effective at lowering turnaround time for the IO bound program. On medium sized tests scales SCHED_FIFO provided the best turnaround times for CPU bound and mixed programs while SCHED_RR provided the best turnaround time for IO bound programs.

Average Wait Time:

The average wait time benchmark is very similar to the average turnaround time benchmark and did not provide much more insight into the different policies but confirmed the previous results. On large scale tests SCHED_OTHER provided the lowest wait times and was most effective with IO bound programs. On medium scales tests SCHED_RR provided the lowest average wait time for IO bound programs and SCHED_FIFO provided the lowest average wait time for CPU bound and mixed programs.

Conclusion:

After analyzing the data provided by the benchmark test we see that different scheduling policies are better tailored toward different applications. The SCHED_OTHER policy provides the best running time and turnaround time for every type of application as long as there is a large scale of processes. SCHED_OTHER also produces the lowest runtime for any scale if processes are IO bound but is least effective with CPU bound scheduling. The SCHED_FIFO policy is best suited for systems which perform CPU bound processes at any scale and is also very effective with medium scale mixed programs. The SCHED_RR policy was rarely as effective as the other policies but performed well with medium scale IO bound test cases.

In the larger picture the SCHED_OTHER policy is very well suited to personal computing applications. It performs well on every type of application and is effective when 100 processes are running simultaneously. Since most personal computer run between 75-125 processes during use I would consider SCHED_OTHER to be the best scheduling policy for most computers. The SCHED_OTHER policy also provided very good turnaround times which would be useful for computers running GUIs.

The SCHED_FIFO policy is best suited to applications which do not require IO input but are simply performing calculations. I would consider using SCHED_FIFO if I was doing large scale computing or running simulations. The SCHED_RR policy did not provide much benefit over the other two scheduling policies. It was only effective with IO bound medium size programs but not much faster than SCHED_OTHER. I do not think there is a benefit to using SCHED_RR over SCHED_OTHER.

References:

`//http://www.cs.fsu.edu/~baker/realtime/restricted/notes/utils.c`

For information on subtracting timestructs

Appendix A – Raw data:

**		
Current Scheduling Policy: 0		
Finished Running ./pi SCHED_OTHER of scale 10.		
Total Real Time: 0.987630s		
Real Time Per Process: 0.098763s		
Average Turnaround Time: 0.918479s		
Lowest Turnaround Time: 0.825671s		
Largest Turnaround Time: 0.985669s		
Average Wait Time: 0.914479s		
Lowest Wait Time: 0.821671s		
Largest Wait Time: 0.981669s		
**		
Current Scheduling Policy: 0		
Finished Running ./pi SCHED_OTHER of scale 100.		
Total Real Time: 7.611311s		
Real Time Per Process: 0.076113s		
Average Turnaround Time: 7.412369s		
Lowest Turnaround Time: 7.113299s		
Largest Turnaround Time: 7.600272s		
Average Wait Time: 7.380129s		
Lowest Wait Time: 7.085299s		
Largest Wait Time: 7.564272s		
**		
Current Scheduling Policy: 0		
Finished Running ./pi SCHED_OTHER of scale 1000.		
Total Real Time: 48.369207s		
Real Time Per Process: 0.048369s		

Average Turnaround Time: 41.032999s		
Lowest Turnaround Time: 29.925393s		
Largest Turnaround Time: 45.242601s		
Average Wait Time: 40.838543s		
Lowest Wait Time: 29.781393s		
Largest Wait Time: 45.030601s		
**		
Current Scheduling Policy: 1		
Finished Running ./pi SCHED_FIFO of scale 10.		
Total Real Time: 0.541735s		
Real Time Per Process: 0.054174s		
Average Turnaround Time: 0.401853s		
Lowest Turnaround Time: 0.202363s		
Largest Turnaround Time: 0.540184s		
Average Wait Time: 0.169853s		
Lowest Wait Time: -0.029637s		
Largest Wait Time: 0.308184s		
**		
Current Scheduling Policy: 1		
Finished Running ./pi SCHED_FIFO of scale 100.		
Total Real Time: 5.050219s		
Real Time Per Process: 0.050502s		
Average Turnaround Time: 4.938321s		
Lowest Turnaround Time: 4.922197s		
Largest Turnaround Time: 5.039325s		
Average Wait Time: 4.691721s		
Lowest Wait Time: 4.678197s		

Largest Wait Time: 4.791325s		
**		
Current Scheduling Policy: 1		
Finished Running ./pi SCHED_FIFO of scale 1000.		
Total Real Time: 50.327797s		
Real Time Per Process: 0.050328s		
Average Turnaround Time: 49.693305s		
Lowest Turnaround Time: 0.677698s		
Largest Turnaround Time: 50.320939s		
Average Wait Time: 49.314969s		
Lowest Wait Time: 0.321698s		
Largest Wait Time: 49.928939s		
**		
Current Scheduling Policy: 2		
Finished Running ./pi SCHED_RR of scale 10.		
Total Real Time: 0.505974s		
Real Time Per Process: 0.050597s		
Average Turnaround Time: 0.458372s		
Lowest Turnaround Time: 0.391548s		
Largest Turnaround Time: 0.504988s		
Average Wait Time: 0.062372s		
Lowest Wait Time: -0.004452s		
Largest Wait Time: 0.108988s		
**		
Current Scheduling Policy: 2		
Finished Running ./pi SCHED_RR of scale 100.		
Total Real Time: 5.161414s		
Real Time Per Process: 0.051614s		

Average Turnaround Time: 5.089965s		
Lowest Turnaround Time: 3.960247s		
Largest Turnaround Time: 5.150047s		
Average Wait Time: 4.681885s		
Lowest Wait Time: 3.552247s		
Largest Wait Time: 4.738047s		
***		
Current Scheduling Policy: 2		
Finished Running ./pi SCHED_RR of scale 1000.		
Total Real Time: 51.083575s		
Real Time Per Process: 0.051084s		
Average Turnaround Time: 48.534186s		
Lowest Turnaround Time: 0.319736s		
Largest Turnaround Time: 51.071417s		
Average Wait Time: 47.978534s		
Lowest Wait Time: -0.212264s		
Largest Wait Time: 50.503417s		
***		
Current Scheduling Policy: 0		
Finished Running ./rw SCHED_OTHER of scale 10.		
Total Real Time: 0.349420s		
Real Time Per Process: 0.034942s		
Average Turnaround Time: 0.340031s		
Lowest Turnaround Time: 0.321772s		
Largest Turnaround Time: 0.348557s		
Average Wait Time: -0.235969s		
Lowest Wait Time: -0.254228s		
Largest Wait Time: 0.000000s		

**		
Current Scheduling Policy: 0		
Finished Running ./rw SCHED_OTHER of scale 100.		
Total Real Time: 2.461938s		
Real Time Per Process: 0.024619s		
Average Turnaround Time: 2.304672s		
Lowest Turnaround Time: 1.780104s		
Largest Turnaround Time: 2.459183s		
Average Wait Time: 1.710952s		
Lowest Wait Time: 1.188104s		
Largest Wait Time: 1.863183s		
**		
Current Scheduling Policy: 0		
Finished Running ./rw SCHED_OTHER of scale 1000.		
Total Real Time: 17.164963s		
Real Time Per Process: 0.017165s		
Average Turnaround Time: 14.992742s		
Lowest Turnaround Time: 10.238509s		
Largest Turnaround Time: 16.549408s		
Average Wait Time: 14.220766s		
Lowest Wait Time: 9.498509s		
Largest Wait Time: 15.761408s		
**		
Current Scheduling Policy: 1		
Finished Running ./rw SCHED_FIFO of scale 10.		
Total Real Time: 0.376898s		
Real Time Per Process: 0.037690s		
Average Turnaround Time: 0.371466s		

Lowest Turnaround Time: 0.366492s		
Largest Turnaround Time: 0.376854s		
Average Wait Time: -0.424534s		
Lowest Wait Time: -0.429508s		
Largest Wait Time: 0.000000s		
***		
Current Scheduling Policy: 1		
Finished Running ./rw SCHED_FIFO of scale 100.		
Total Real Time: 2.310915s		
Real Time Per Process: 0.023109s		
Average Turnaround Time: 2.250446s		
Lowest Turnaround Time: 2.068901s		
Largest Turnaround Time: 2.309391s		
Average Wait Time: 1.439086s		
Lowest Wait Time: 1.260901s		
Largest Wait Time: 1.493391s		
***		
Current Scheduling Policy: 1		
Finished Running ./rw SCHED_FIFO of scale 1000.		
Total Real Time: 34.521483s		
Real Time Per Process: 0.034521s		
Average Turnaround Time: 33.497439s		
Lowest Turnaround Time: 28.567237s		
Largest Turnaround Time: 34.483186s		
Average Wait Time: 32.452567s		
Lowest Wait Time: 27.595237s		
Largest Wait Time: 33.406114s		
***		

Current Scheduling Policy: 2		
Finished Running ./rw SCHED_RR of scale 10.		
Total Real Time: 0.283338s		
Real Time Per Process: 0.028334s		
Average Turnaround Time: 0.277477s		
Lowest Turnaround Time: 0.262888s		
Largest Turnaround Time: 0.282652s		
Average Wait Time: -0.802523s		
Lowest Wait Time: -0.817112s		
Largest Wait Time: 0.000000s		
**		
Current Scheduling Policy: 2		
Finished Running ./rw SCHED_RR of scale 100.		
Total Real Time: 1.902028s		
Real Time Per Process: 0.019020s		
Average Turnaround Time: 1.815052s		
Lowest Turnaround Time: 1.646944s		
Largest Turnaround Time: 1.895788s		
Average Wait Time: 0.721052s		
Lowest Wait Time: 0.554944s		
Largest Wait Time: 0.799788s		
**		
Current Scheduling Policy: 2		
Finished Running ./rw SCHED_RR of scale 1000.		
Total Real Time: 30.309704s		
Real Time Per Process: 0.030310s		
Average Turnaround Time: 29.194236s		
Lowest Turnaround Time: 26.240941s		

Largest Turnaround Time: 30.260548s		
Average Wait Time: 27.877868s		
Lowest Wait Time: 25.000941s		
Largest Wait Time: 28.908548s		
**		
Current Scheduling Policy: 0		
Finished Running ./mix SCHED_OTHER of scale 10.		
Total Real Time: 0.377768s		
Real Time Per Process: 0.037777s		
Average Turnaround Time: 0.360895s		
Lowest Turnaround Time: 0.336540s		
Largest Turnaround Time: 0.376778s		
Average Wait Time: -0.997105s		
Lowest Wait Time: -1.019460s		
Largest Wait Time: 0.000000s		
**		
Current Scheduling Policy: 0		
Finished Running ./mix SCHED_OTHER of scale 100.		
Total Real Time: 2.482326s		
Real Time Per Process: 0.024823s		
Average Turnaround Time: 2.332042s		
Lowest Turnaround Time: 1.953371s		
Largest Turnaround Time: 2.480534s		
Average Wait Time: 0.956762s		
Lowest Wait Time: 0.581371s		
Largest Wait Time: 1.100534s		
**		
Current Scheduling Policy: 0		

Finished Running ./mix SCHED_OTHER of scale 1000.		
Total Real Time: 24.501185s		
Real Time Per Process: 0.024501s		
Average Turnaround Time: 21.942456s		
Lowest Turnaround Time: 15.794464s		
Largest Turnaround Time: 23.927283s		
Average Wait Time: 20.391284s		
Lowest Wait Time: 14.274464s		
Largest Wait Time: 22.363283s		
***		
Current Scheduling Policy: 1		
Finished Running ./mix SCHED_FIFO of scale 10.		
Total Real Time: 0.294967s		
Real Time Per Process: 0.029497s		
Average Turnaround Time: 0.292207s		
Lowest Turnaround Time: 0.288847s		
Largest Turnaround Time: 0.294579s		
Average Wait Time: -1.287793s		
Lowest Wait Time: -1.291153s		
Largest Wait Time: 0.000000s		
***		
Current Scheduling Policy: 1		
Finished Running ./mix SCHED_FIFO of scale 100.		
Total Real Time: 1.987621s		
Real Time Per Process: 0.019876s		
Average Turnaround Time: 1.920053s		
Lowest Turnaround Time: 1.865111s		
Largest Turnaround Time: 1.985779s		

Average Wait Time: 0.332053s		
Lowest Wait Time: 0.277111s		
Largest Wait Time: 0.397779s		
**		
Current Scheduling Policy: 1		
Finished Running ./mix SCHED_FIFO of scale 1000.		
Total Real Time: 34.193293s		
Real Time Per Process: 0.034193s		
Average Turnaround Time: 33.165849s		
Lowest Turnaround Time: 29.525238s		
Largest Turnaround Time: 34.185580s		
Average Wait Time: 31.399689s		
Lowest Wait Time: 27.821238s		
Largest Wait Time: 32.365580s		
**		
Current Scheduling Policy: 2		
Finished Running ./mix SCHED_RR of scale 10.		
Total Real Time: 0.372214s		
Real Time Per Process: 0.037221s		
Average Turnaround Time: 0.361772s		
Lowest Turnaround Time: 0.352129s		
Largest Turnaround Time: 0.371046s		
Average Wait Time: -1.458228s		
Lowest Wait Time: -1.467871s		
Largest Wait Time: 0.000000s		
**		
Current Scheduling Policy: 2		
Finished Running ./mix SCHED_RR of scale 100.		

Total Real Time: 3.669509s		
Real Time Per Process: 0.036695s		
Average Turnaround Time: 3.549355s		
Lowest Turnaround Time: 3.175117s		
Largest Turnaround Time: 3.668034s		
Average Wait Time: 1.708635s		
Lowest Wait Time: 1.339117s		
Largest Wait Time: 1.820034s		
**		
Current Scheduling Policy: 2		
Finished Running ./mix SCHED_RR of scale 1000.		
Total Real Time: 29.377452s		
Real Time Per Process: 0.029377s		
Average Turnaround Time: 28.014347s		
Lowest Turnaround Time: 22.509966s		
Largest Turnaround Time: 29.363221s		
Average Wait Time: 25.906187s		
Lowest Wait Time: 20.465966s		
Largest Wait Time: 27.207221s		

Appendix B – All Code:

Benchmarks.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <string.h>
#include <sys/wait.h>
#include <time.h>
#include <stdint.h>
```

```
#include <sys/time.h>
#include <sys/resource.h>

#define LARGE_SCALE 1000
#define MEDIUM_SCALE 100
#define SMALL_SCALE 10

#define DEBUG 0

int processScales[3] = { SMALL_SCALE, MEDIUM_SCALE, LARGE_SCALE };
char schedulingPolicies[3][12] = { "SCHED_OTHER", "SCHED_FIFO",
"SCHED_RR" };
char programs[3][6] = { "./pi", "./rw", "./mix" };
struct sched_param param;


//http://www.cs.fsu.edu/~baker/realtime/restricted/notes/utils.c
struct timespec timespec_sub(struct timespec timespec_1,struct
timespec timespec_2) {
	struct timespec rtn_val;
	int xsec;
	int sign = 1;

	if ( timespec_2.tv_nsec > timespec_1.tv_nsec ) {
		xsec = (int)((timespec_2.tv_nsec - timespec_1.tv_nsec) /
(1E9 + 1));
		timespec_2.tv_nsec -= (long int)(1E9 * xsec);
		timespec_2.tv_sec += xsec;
	}

	if ( (timespec_1.tv_nsec - timespec_2.tv_nsec) > 1E9 ) {
		xsec = (int)((timespec_1.tv_nsec - timespec_2.tv_nsec) /
1E9);
		timespec_2.tv_nsec += (long int)(1E9 * xsec);
		timespec_2.tv_sec -= xsec;
	}

	rtn_val.tv_sec = timespec_1.tv_sec - timespec_2.tv_sec;
	rtn_val.tv_nsec = timespec_1.tv_nsec - timespec_2.tv_nsec;

	if (timespec_1.tv_sec < timespec_2.tv_sec) {
		sign = -1;
	}

	rtn_val.tv_sec = rtn_val.tv_sec * sign;

	return rtn_val;
}

double get_secs_diff(struct timespec clock_1, struct timespec clock_2)
{
	double rtn_val;
```

```
	struct timespec diff;

	diff = timespec_sub(clock_1, clock_2);

	rtn_val = diff.tv_sec;
	rtn_val += (double)diff.tv_nsec/(double)1E9;

	return rtn_val;
}


void executeProgram(char* program)
{
	char command[50];
	snprintf(command, 50, "%s", program);
    system(command);
}

void setPolicy(char* p){
	int policy;
	if(!strcmp(p, "SCHED_OTHER")){
	    policy = SCHED_OTHER;
	}
	else if(!strcmp(p, "SCHED_FIFO")){
	    policy = SCHED_FIFO;
	}
	else if(!strcmp(p, "SCHED_RR")){
	    policy = SCHED_RR;
	}
	else{
	    fprintf(stderr, "Unhandeled scheduling policy\n");
	    exit(EXIT_FAILURE);
	}

	param.sched_priority = sched_get_priority_max(policy);
#if DEBUG
	fprintf(stdout, "Current Scheduling Policy: %d\n",
sched_getscheduler(0));
	fprintf(stdout, "Setting Scheduling Policy to: %d\n", policy);
#endif
	if(sched_setscheduler(0, policy, &param)){
	perror("Error setting scheduler policy");
	exit(EXIT_FAILURE);
	}
#if DEBUG
	fprintf(stdout, "New Scheduling Policy: %d\n",
sched_getscheduler(0));
#endif
}

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

```
    /* void unused vars */
    (void) argc;
    (void) argv;

    /* Setup Local Vars */
    int i, a, n;

    /* Loop Through Each Program Type */
    for(a=0; a < 3; a++){
     char* program = programs[a];

            /* Loop Through Each Scheduling Type */
            for(i=0; i < 3; i++){
                  char* policy = schedulingPolicies[i];
                  setPolicy(policy);

                  /* Loop Through Each Scale */
                  for(n=0; n < 3; n++){
                        /* Create PIDS */
                        int num_pids = processScales[n];
                        pid_t pids[num_pids];
                        int i;

                        /* Measure Total Time */
                        struct timespec t_begin, t_end;
                        double t_time;
                        double p_time_total = 0, max_turn_time = 0,
max_wait_time = 0, total_wait_time = 0, min_turn_time = 100,
min_wait_time = 100;
                        double processes = (double) processScales[n];
                        clock_gettime(CLOCK_REALTIME, &t_begin);
                        struct timespec p_start_times[32768];


                        /* Start children. */
                        for (i = 0; i < num_pids; ++i) {
                              struct timespec p_begin;
                              clock_gettime(CLOCK_REALTIME, &p_begin);

                              if ((pids[i] = fork()) < 0) {
                                    perror("fork");
                                    abort();
                              } else if (pids[i] == 0) {
                                    setPolicy(policy);
                                    executeProgram(program);
                                  //printf("Current Scheduling
Policy: %d\n", sched_getscheduler(pids[i]));
                                    exit(0);
                              }
                              p_start_times[(long) pids[i]] = p_begin;
                        }
```

```
                    /* Wait for children to exit. */
                    int status;
                    pid_t pid;

                    while (num_pids > 0) {
                         struct rusage usage;
                         struct timeval utime, stime;
                         struct timespec p_end, p_start;
                         double p_time;

                         pid = wait(&status);
                         --num_pids;
#if DEBUG
                         printf("Child with PID %ld exited with
status 0x%x.\n", (long)pid, status);
#endif

                         /* Calculate Wait Time of Process */
                         getrusage(RUSAGE_SELF, &usage);
                         utime = usage.ru_utime;
                         stime = usage.ru_stime;
                         long wait_secs = utime.tv_sec +
stime.tv_sec;
                         long wait_milli = utime.tv_usec +
stime.tv_usec;

                         /* Calculate Wall Time of Process */
                         clock_gettime(CLOCK_REALTIME, &p_end);
                         p_start = p_start_times[(long) pid];
                         p_time = get_secs_diff(p_end, p_start);

                         double wait_time = p_time - (wait_secs +
(wait_milli / 1000000.0));

                         /* Add Times to Benchmark */
                         total_wait_time += wait_time;
                         p_time_total += p_time;

                         if(wait_time < min_wait_time){
                              min_wait_time = wait_time;
                         }

                         if(p_time < min_turn_time){
                              min_turn_time = p_time;
                         }

                         if(wait_time > max_wait_time){
                              max_wait_time = wait_time;
                         }

                         if(p_time > max_turn_time){
                              max_turn_time = p_time;
```

```
                              }

                        }
                        /* Create Report */

      printf("*********************************************************
*******\n");
                        printf("Current Scheduling Policy: %d\n",
sched_getscheduler(0));
                        printf("Finished Running %s %s of scale %i.\n",
program, policy, (int) processes);

                        clock_gettime(CLOCK_REALTIME, &t_end);
                        t_time = get_secs_diff(t_end, t_begin);

                        printf("Total Real Time:         %fs\n", t_time);
                        printf("Real Time Per Process:   %fs\n\n", t_time
/ processes);

                        printf("Average Turnaround Time: %fs\n",
p_time_total / processes);
                        printf("Lowest Turnaround Time:  %fs\n",
min_turn_time);
                        printf("Largest Turnaround Time: %fs\n\n",
max_turn_time);

                        printf("Average Wait Time:       %fs\n",
total_wait_time / processes);
                        printf("Lowest Wait Time:        %fs\n",
min_wait_time);
                        printf("Largest Wait Time:       %fs\n",
max_wait_time);
                  }
            }
      }

    return 0;
}
```

Pi.c

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

```
/* Local Defines */
#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)
#define DEBUG 0

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

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

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

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

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

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

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

```
    /* Print result */
#if DEBUG
    fprintf(stdout, "pi = %f\n", piCalc);
#endif

    return 0;
}
```

Mix.c

```
/* 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 <math.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEFAULT_ITERATIONS 1000
#define RADIUS (RAND_MAX / 2)
#define DEBUG 0

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[]){

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

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

ssize_t bytesRead = 0;
ssize_t totalBytesRead = 0;
int totalReads = 0;
ssize_t bytesWritten = 0;
ssize_t totalBytesWritten = 0;
int totalWrites = 0;
int inputFileResets = 0;

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

/* Set default iterations if not supplied */
if(argc < 2){
       iterations = DEFAULT_ITERATIONS;
}

/* Set supplied transfer size or default if not supplied */
if(argc < 3){
 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 < 4){
 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 < 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[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 < 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[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
          fprintf(stderr, "Output filename base is too long\n");
          exit(EXIT_FAILURE);
      }
      strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
     }

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

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

     /* Open Input File Descriptor in Read Only mode */
     if((inputFD = open(inputFilename, O_RDONLY | O_SYNC)) < 0){
```

```
      perror("Failed to open input file");
      exit(EXIT_FAILURE);
    }

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

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

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

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

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

    /* 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 */
#if DEBUG
    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);
#endif

    /* 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;
}
```

Rw.c

```
/* Include Flags */
#define _GNU_SOURCE

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

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEBUG 0

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

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

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

```
    ssize_t bytesRead = 0;
    ssize_t totalBytesRead = 0;
    int totalReads = 0;
    ssize_t bytesWritten = 0;
    ssize_t totalBytesWritten = 0;
    int totalWrites = 0;
    int inputFileResets = 0;

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

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

    /* 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);
#if DEBUG
    /* 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);
#endif
    /* 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;
}
```