report.md

Nicholas Clement

CSCI 3753 Operating Systems Fall 2016

Assignment Four

Abstract:

The goal of this lab is to analyze the different runtimes and efficiencies of schedulers with different types of processes. The different schedulers examined were CFS, Round Robin, and First in First Out. The different processes examined were CPU intensive, I/O intensive, and a combination of both CPU and I/O intensive tasks. All of these were then tested with changing priority (otherwise referred to as niceness) and static priority. The general results showed that CPU bound processes were much more expensive and time consuming compared to I/O processes for each scheduler. In general, I/O was by far the least time consuming and mixed I/O CPU tests were less time consuming that pure CPU test but more time consuming than pure I/O tests.

Introduction:

The purpose of this assignment is to investigate the behavior of different scheduling algorithms. To do this I was tasked with creating different testing benchmarks, all of which I ran with the different scheduling algorithms assigned. From these different benchmark tests I was able to conclude the corresponding efficiencies for each scheduling algorithm, and the pitfalls associated with that algorithm.

Method (Experimental Design):

A total of fifty-four tests were preformed to accumulate data on the different scheduling algorithms and their corresponding efficiencies. These tests were systematically divided into several subcategories including scheduler, priority, structure, and scheduling process.

Variable One: Schedulers

The goal of this lab is to investigate three different Unix scheduling algorithms and weigh the pros and cons of each scheduling algorithm. Therefore this is the primary and first variable I was interested in testing with my different test cases.

Variable Two: Benchmarks

I used three different structures of programs as benchmarks for the tests. These structures included I/O bound programs (reading/writing), CPU bound programs (computation), and a mix of I/O and CPU bound programs. These different base tests allowed me to analyze each scheduler and its performance for the basic necessities of computing.

For the CPU bound program I used the provided statistical computation of pi. I used one hundred million iterations, which seemed to be enough to thoroughly use the CPU.

```
for(i=0; i<iterations; i++){
  x = (random() % (RADIUS * 2)) - RADIUS;
```

```
  y = (random() % (RADIUS * 2)) - RADIUS;
  if(zeroDist(x,y) < RADIUS){
      inCircle++;
  }
  inSquare++;
}
pCircle = inCircle/inSquare; piCalc = pCircle * 4.0;
```

For the I/O bound program I used the provided rw file, and altered it to accept priority and create child processes. This program read 102400 bytes from a single file and then generated a new random file to write the bytes to.

```
open(outputFilename,
     O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH)) < 0)


  /* 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++;
}
```

The CPU-I/O mixed program first ran the statistical computation of pi over one hundred million iterations and then proceeded to preform the same I/O as described above.

Creating these different benchmark tests allowed me to further investigate how the three CPU schedulers handle different types of processes.

Variable Three: Number of Processes

I ran each benchmark test with a different number of child processes using the fork() systemcall. The different numbers of processes I used were five, twenty, and one hundred. My goal in changing the number of processes for each benchmark test was to see the scalability of the different schedulers. In the results section I will discuss in more detail the behavior that is required for a scheduling algorithm to scale well.

Variable Four: Priority

Along with the base benchmark tests, I included priority. When testing priority I analyzed static priority, and dynamic priority. Static priority consisted of each process preforming the same task with the same priority. When implementing dynamic priority I used the nice() systemcall to decrement the priority of each child process. The code is as follows:

```
if (niceness == 1){
  printf("niceness == 1");
  niceNumb ++;
  int test = nice(niceNumb);
    //error handling
  if (test == -1){
    fprintf(stderr, "Unable to change niceness\n");
    exit(EXIT_FAILURE);
  }
}
```

Overview

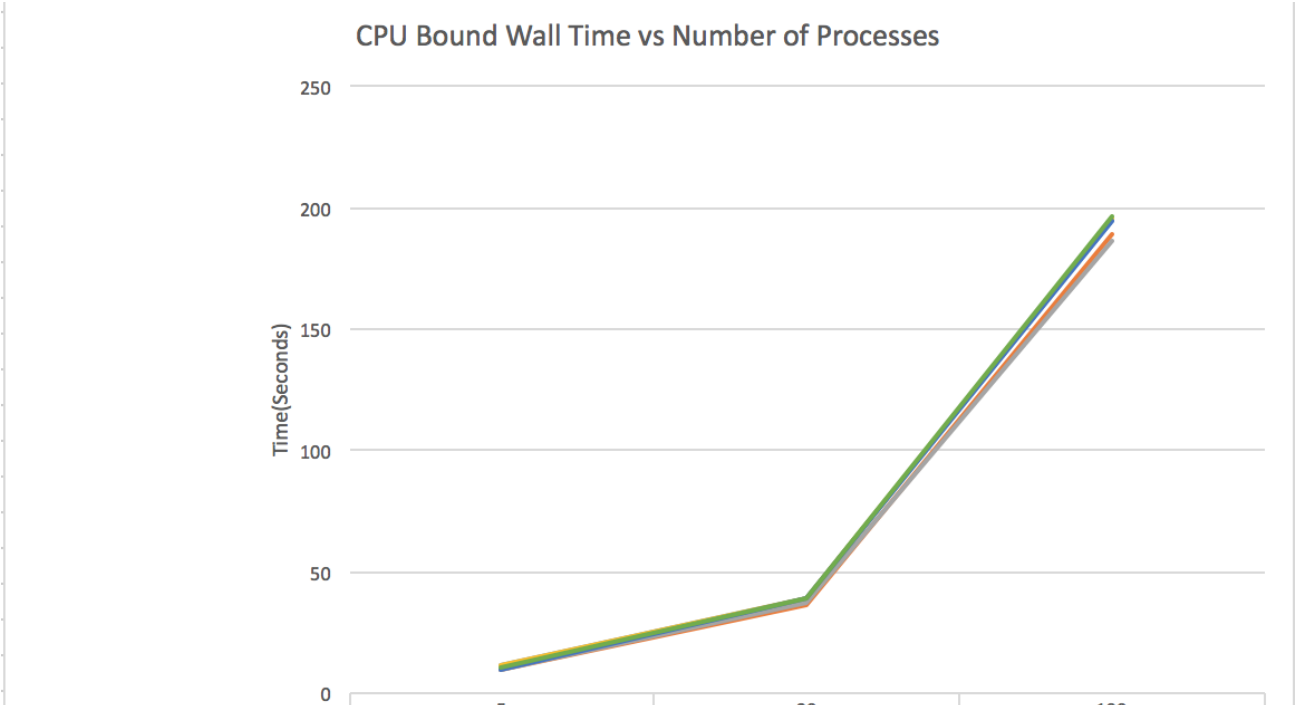
Starting with the schedulers we have three possibilities, followed by three different benchmark cases, followed by three different numbers of processes, followed by two different priorities.

3 x 3 x 3 x 2 = 54

Results and Analysis:

My primary focus was on each of the scheduling processes and how they preform on I/O, CPU, and mixed processes. The following graphs show each of the different scheduling processes and their corresponding wall times as well as the number of processes ran.

CPU Bound



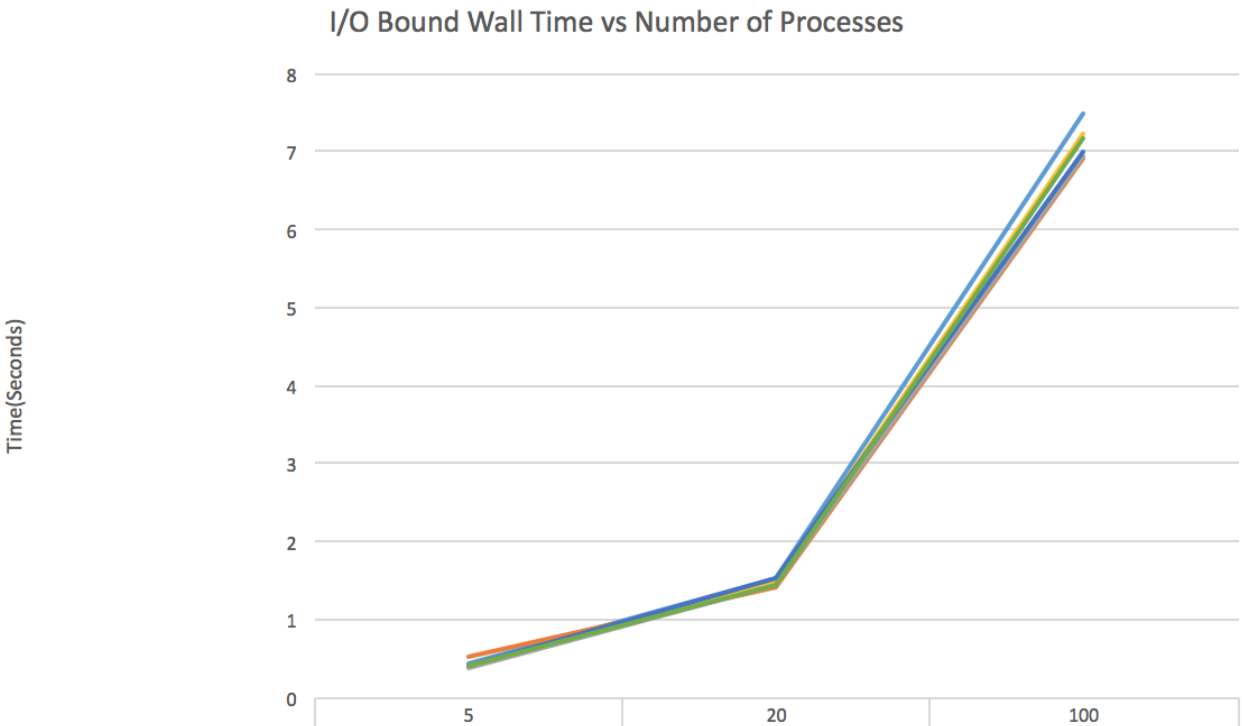
	5	20	100
Other with Priority	9.19	36.65	189.2
Other without Priority	9.21	37.16	185.82
FIFO with Priority	11.43	38.78	194.52
FIFO without Priority	11.47	39.24	195.23
Round Robin with Priority	10	39.04	194.74
Round Robin without Priority	10.15	39.04	196.36

From the data collected we can see that most efficient scheduler for CPU bound processes was the CFS (other) with priority. We can also infer that priority had minimal effect on the wall times across all schedulers. Each scheduler scales linearly across the different number of processes.

We can see that the FIFO scheduler had the least number of context switches, as well as the lowest CPU usage. The CFS scheduler ran with the shortest wall time out of all the schedulers, it however had a non-linear number of i-switched with a final number of nearly 99,000 when using priority. I-switched represents involuntary context switches, meaning that a process was preempted out of its execution time. Context switches are very resource demanding, so this non-linear scaling of context switches is something to take note of.

See the CPU chart in the data section for context switch results.

I/O Bound

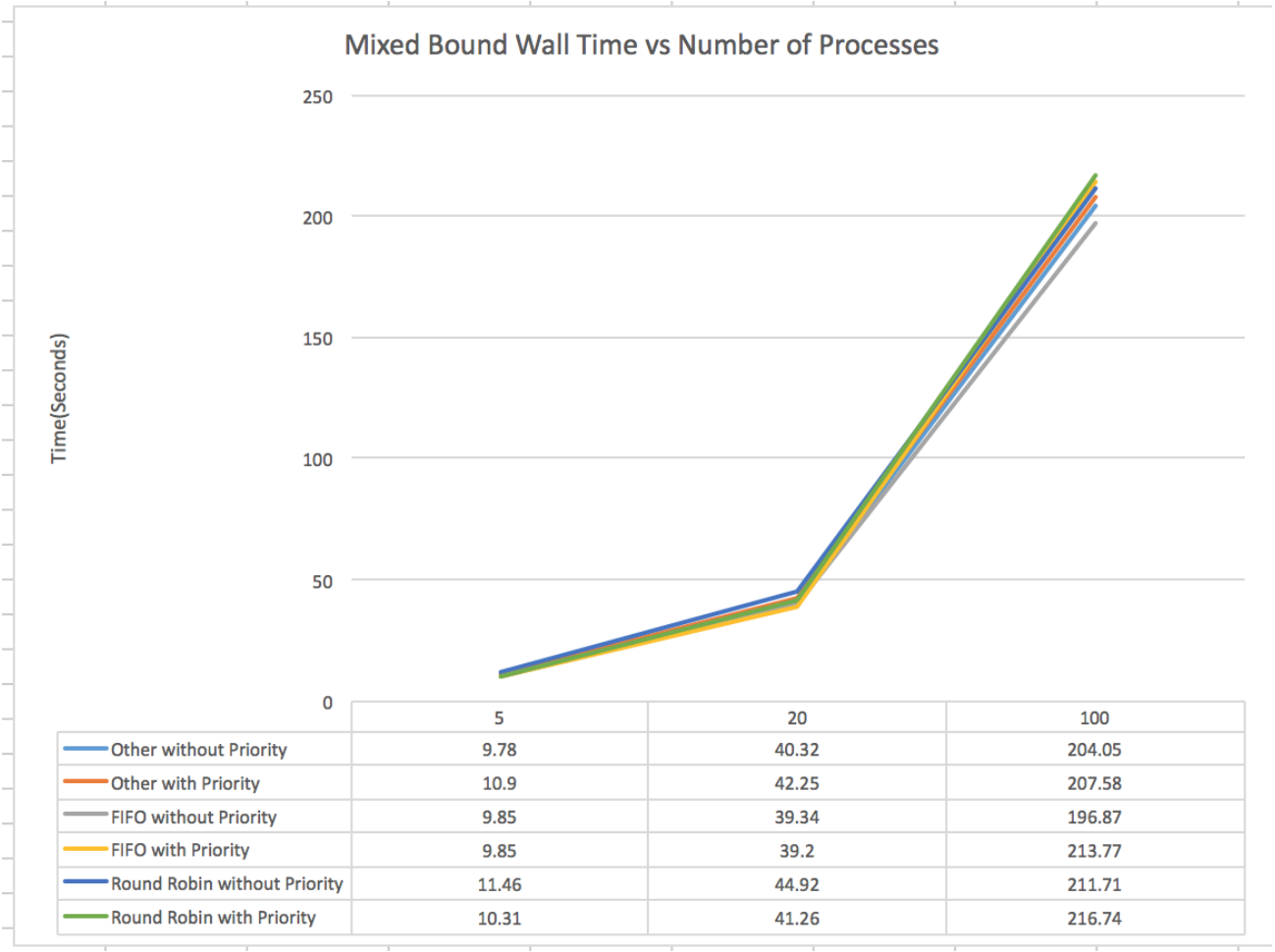


	5	20	100
Other without Priority	0.44	1.52	7.49
Other with Priority	0.53	1.43	6.92
FIFO without Priority	0.39	1.45	6.93
FIFO with Priority	0.42	1.47	7.24
Round Robin without Priority	0.42	1.52	7.01
Round Robin with Priority	0.42	1.44	7.16

Immediatly we can notice the drastic change in runtime between CPU bound and I/O bound processes. This is to be expected when putting the CPU through millions of iterations of calculations in comparison to transferring several bytes of data from one file to another.

CFS with priority was the fastest wall time out of all the schedulers. Priority had a large effect on the CFS scheduler, CFS without priorty was the slowest in the group while CFS with priority was the fastest. This demonstrates the necessity of priority for the CFS when dealing with I/O bound processes, and also shows that it is extremley I/O efficient when priority is involved.

Mixed Bound



	5	20	100
Other without Priority	9.78	40.32	204.05
Other with Priority	10.9	42.25	207.58
FIFO without Priority	9.85	39.34	196.87
FIFO with Priority	9.85	39.2	213.77
Round Robin without Priority	11.46	44.92	211.71
Round Robin with Priority	10.31	41.26	216.74

In my CPU-I/O mixed test I found that FIFO without priority had the fastest execution time out of all of the scheduling algorithms. It can also be seen in my data that this scheduling method had the highest CPU utilization out of all of the schedulers, with 193% for mixed CPU-I/O processes. This was the key to the minimal wall time, perhaps with more trials the order of processes wouldn't line up as well as they did for the FIFO scheduler.

When comparing FIFO without priority to FIFO with priority, there was nearly a 20 second gap in performance (wall) and a 4,000 voluntary context switch gap between the two. This shows us that in a situation with mixed I/O FIFO without priority will likely outperform FIFO with priority, with more context switches comes a greater overhead as reflected in the execution time of FIFO with priority.

In contrast priority had little effect on the CFS and Round Robin, both of these schedulers didn't deviate more than 500 context switches with or without priority. This leads me to believe that CFS and Round Robin although not as fast as FIFO in this case are reliable, and natural more fair than an algorithm like FIFO. CFS did outperform the Round Robin in this final test with an average of about 8 seconds faster wall time.

Conclusion:

The method I used to change priority changes the nice value of the current process. The nice() systemcall changes the priority in the user space, not the kernel space. This is something to take into account when observing my results, and may contribute to the minimal difference between the schedulers without priority and the schedulers with priority.

After running each benchmark test with a different number of processes and priority, I can now see that although the CFS is not always the best preforming, it is reliable and consistent. SCHED_RR and SCHED_FIFO are referred to as "real-time" policies. Tasks with these policies preempt all other processes and so starvation is more common in these than in SCHED_OTHER. SCHED_RR is Round Robin (preemptive) while FIFO (First in First Out) needs the executing task to yield the processor. For this reason even though FIFO had some of the better numbers in my data, it is

unreliable and unrealistic to schedule an entire system with FIFO. CFS is the best scheduling algorithm, with its self balancing nature it ensures that no processes gets starved while still implementing priority.

References:

http://stackoverflow.com -> Different scheduling policies

http://www.tutorialspoint.com/unix_system_calls/nice.htm -> nice() systemcall

http://www.linuxforums.org/forum/ -> Linux information

Appendix A: Data

CPU

Priority	Scheduler	Number of Processes	wall	CPU Usage	iswitched	v-switched
TRUE	SCHED_OTHER	5	9.19	CPU=199%	i-switched=2670	v-switched=14
TRUE	SCHED_OTHER	20	36.65	CPU=199%	i-switched=17962	v-switched=44
TRUE	SCHED_OTHER	100	189.2	CPU=197%	i-switched=98692	v-switched=204
FALSE	SCHED_OTHER	5	9.21	CPU=198%	i-switched=2741	v-switched=13
FALSE	SCHED_OTHER	20	37.16	CPU=198%	i-switched=18594	v-switched=43
FALSE	SCHED_OTHER	100	185.82	CPU=198%	i-switched=93078	v-switched=203
TRUE	SCHED_FIFO	5	11.43	CPU=161%	i-switched=26	v-switched=12
TRUE	SCHED_FIFO	20	38.78	CPU=190%	i-switched=115	v-switched=35
TRUE	SCHED_FIFO	100	194.52	CPU=190%	i-switched=593	v-switched=163
FALSE	SCHED_FIFO	5	11.47	CPU=161%	i-switched=25	v-switched=12
FALSE	SCHED_FIFO	20	39.24	CPU=189%	i-switched=98	v-switched=36
FALSE	SCHED_FIFO	100	195.23	CPU=190%	i-switched=497	v-switched=163
TRUE	SCHED_RR	5	10	CPU=186%	i-switched=97	v-switched=14
TRUE	SCHED_RR	20	39.04	CPU=190%	i-switched=813	v-switched=27

Priority	Scheduler	Number of Processes	wall	CPU Usage	iswitched	v-switched
TRUE	SCHED_RR	100	194.74	CPU=190%	i-switched=4105	v-switched=123
FALSE	SCHED_RR	5	10.15	CPU=184%	i-switched=112	v-switched=13
FALSE	SCHED_RR	20	39.04	CPU=190%	i-switched=722	v-switched=35
FALSE	SCHED_RR	100	196.36	CPU=190%	i-switched=4038	v-switched=124

I/O

IO						
Priority	Scheduler	Number of Processes	wall	CPU Usage	iswitched	v-switched
FALSE	SCHED_OTHER	5	0.44	CPU=13%	i-switched=180	v-switched=1331
FALSE	SCHED_OTHER	20	1.52	CPU=6%	i-switched=1337	v-switched=4076
FALSE	SCHED_OTHER	100	7.49	CPU=5%	i-switched=6387	v-switched=19428
TRUE	SCHED_OTHER	5	0.53	CPU=11%	i-switched=79	v-switched=1339
TRUE	SCHED_OTHER	20	1.43	CPU=8%	i-switched=1381	v-switched=4970
TRUE	SCHED_OTHER	100	6.92	CPU=12%	i-switched=7460	v-switched=27448
FALSE	SCHED_FIFO	5	0.39	CPU=15%	i-switched=295	v-switched=1513
FALSE	SCHED_FIFO	20	1.45	CPU=13%	i-switched=1221	v-switched=4745
FALSE	SCHED_FIFO	100	6.93	CPU=13%	i-switched=5608	v-switched=22485
TRUE	SCHED_FIFO	5	0.42	CPU=8%	i-switched=348	v-switched=1239
TRUE	SCHED_FIFO	20	1.47	CPU=12%	i-switched=1022	v-switched=4481
TRUE	SCHED_FIFO	100	7.24	CPU=12%	i-switched=7624	v-switched=30529
FALSE	SCHED_RR	5	0.42	CPU=15%	i-switched=195	v-switched=1358
FALSE	SCHED_RR	20	1.52	CPU=12%	i-switched=1024	v-switched=4390
FALSE	SCHED_RR	100	7.01	CPU=13%	i-switched=6464	v-switched=23061
TRUE	SCHED_RR	5	0.42	CPU=13%	i-switched=49	v-switched=1336

IO						
TRUE	SCHED_RR	20	1.44	CPU=8%	i-switched=1108	v-switched=4352
TRUE	SCHED_RR	100	7.16	CPU=12%	i-switched=6905	v-switched=28043

Mixed I/O and CPU

Mixed						
Priority	Scheduler	Number of Processes	wall	CPU Usage	iswitched	v-switched
FALSE	SCHED_OTHER	5	9.78	CPU=191%	i-switched=5581	v-switched=1810
FALSE	SCHED_OTHER	20	40.32	CPU=192%	i-switched=22682	v-switched=4951
FALSE	SCHED_OTHER	100	204.05	CPU=192%	i-switched=107346	v-switched=21647
TRUE	SCHED_OTHER	5	10.9	CPU=192%	i-switched=5439	v-switched=1429
TRUE	SCHED_OTHER	20	42.25	CPU=192%	i-switched=22119	v-switched=4642
TRUE	SCHED_OTHER	100	207.58	CPU=193%	i-switched=108068	v-switched=21463
FALSE	SCHED_FIFO	5	9.85	CPU=191%	i-switched=4735	v-switched=1648
FALSE	SCHED_FIFO	20	39.34	CPU=192%	i-switched=22796	v-switched=5699
FALSE	SCHED_FIFO	100	196.87	CPU=193%	i-switched=101533	v-switched=21193
TRUE	SCHED_FIFO	5	9.85	CPU=191%	i-switched=4989	v-switched=1483
TRUE	SCHED_FIFO	20	39.2	CPU=193%	i-switched=20316	v-switched=4851
TRUE	SCHED_FIFO	100	213.77	CPU=192%	i-switched=111313	v-switched=26744
FALSE	SCHED_RR	5	11.46	CPU=192%	i-switched=5457	v-switched=1904
FALSE	SCHED_RR	20	44.92	CPU=192%	i-switched=24064	v-switched=4900
FALSE	SCHED_RR	100	211.71	CPU=193%	i-switched=114657 v-switched=22698	
TRUE	SCHED_RR	5	10.31	CPU=192%	i-switched=5037	v-switched=1481

Mixed						
TRUE	SCHED_RR	20	41.26	CPU=192%	i-switched=22413	v-switched=5361
TRUE	SCHED_RR	100	216.74	CPU=192%	i-switched=113637	v-switched=23007

Appendix B:

Usage:

Download all files in the repo. Run testscript using ./ from the terminal. testscript runs all of the CPU intensive tests and depending on the machine can take several minutes to run.

To run the I/O tests use sh ./io.sh, beware that this will create around 100 new files in the directory that this is run in.

To run the mixed CPU/IO use sh ./mixed.sh, again beware that this will create around 100 new files in the directory that you run this command in. This test will likely take several minutes to complete as it is CPU intensive.

Makefile:

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

INPUTFILESIZEMEGABYTES = 1

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

.PHONY: all clean

all: pi pi-sched rw cpu-io

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

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

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

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

cpu-io.o: cpu-io.c
    $(CC) $(CFLAGS) $<

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

pi-sched.o: pi-sched.c
    $(CC) $(CFLAGS) $<

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

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

clean: testclean
```

```
    rm -f pi pi-sched rw
    rm -f rwinput
    rm -f *.o
    rm -f *~
    rm -f handout/*~
    rm -f handout/*.log
    rm -f handout/*.aux

testclean:
    rm -f rwoutput*
```

Testscript

```
#/!/bin/bash

#File: testscript
#Author: Andy Sayler
#Revised: Shivakant Mishra
#Project: CSCI 3753 Programming Assignment 4
#Create Date: 2016/30/10
#Modify Date: 2016/30/10
#Description:
#   A simple bash script to run a signle copy of each test case
#   and gather the relevent data.

ITERATIONS=100000000
FORKA=5
FORKB=20
FORKC=100
PRIORITY=1
BYTESTOCOPY=102400
BLOCKSIZE=1024
TIMEFORMAT="wall=%e user=%U system=%S CPU=%P i-switched=%c v-switched=%w"
MAKE="make -s"

echo Building code...
$MAKE clean
$MAKE

echo Starting test runs...

echo ------------------------------------------------------------------------------1
echo CPU BOUND WITH SCHED_OTHER AND PRIORITY

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKA $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKB $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKC $PRIORITY > /dev/null


echo ------------------------------------------------------------------------------2
echo CPU BOUND WITH SCHED_OTHER AND NO PRIORITY

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKA > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKB  > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_OTHER $FORKC > /dev/null

echo ------------------------------------------------------------------------------3
echo CPU BOUND WITH SCHED_FIFO AND PRIORITY
```

```
echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKA $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKB $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKC $PRIORITY > /dev/null

echo ------------------------------------------------------------------------------4
echo CPU BOUND WITH SCHED_FIFO AND NO PRIORITY


echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKA > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKB > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_FIFO $FORKC > /dev/null

echo ------------------------------------------------------------------------------5
# CPU BOUND WITH SCHED_RR AND PRIORITY

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKA $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKB $PRIORITY > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKC $PRIORITY > /dev/null

echo ------------------------------------------------------------------------------6
# CPU BOUND WITH SCHED_RR AND NO PRIORITY

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 5 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKA > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 20 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKB > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with 100 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./pi-sched $ITERATIONS SCHED_RR $FORKC > /dev/null

echo ------------------------------------------------------------------------------6


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_OTHER with 1 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE > /dev/null
```

io.sh

```
#File: io.sh
#Author: Nicholas Clement
#Project: CSCI 3753 Programming Assignment 4
#Create Date: 2016/11/11
#Description:
# Bash script to test file I/O times with different scheduling processes

FORKA=5
FORKB=20
FORKC=100
BYTESTOCOPY=102400
BLOCKSIZE=1024
TIMEFORMAT="wall=%e user=%U system=%S CPU=%P i-switched=%c v-switched=%w"
```

```
MAKE="make -s"

echo Building code...
$MAKE clean
$MAKE

echo ------------------------------------------------------------------------------1
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_OTHER with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_OTHER 0
echo -
echo using SCHED_OTHER with 10 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_OTHER 0
echo -
echo using SCHED_OTHER with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_OTHER 0
echo ------------------------------------------------------------------------------2
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_OTHER with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_OTHER 1
echo -
echo using SCHED_OTHER with 10 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_OTHER 1
echo -
echo using SCHED_OTHER with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_OTHER 1
echo ------------------------------------------------------------------------------3
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_FIFO with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_FIFO 0
echo  -
echo using SCHED_FIFO with 10 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_FIFO 0
echo -
echo using SCHED_FIFO with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_FIFO 0
echo ------------------------------------------------------------------------------4
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_FIFO with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_FIFO 1
echo -
echo using SCHED_FIFO with 20 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_FIFO 1
echo -
echo using SCHED_FIFO with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_FIFO 1
echo ------------------------------------------------------------------------------5
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_RR with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_RR 0
echo -
echo using SCHED_RR with 20 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_RR 0
echo -
echo using SCHED_RR with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_RR 0
echo ------------------------------------------------------------------------------6
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_RR with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_RR 1
echo -
echo using SCHED_RR with 20 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_RR 1
echo -
echo using SCHED_RR with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./rw $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_RR 1
```

mixed.sh

```
#File: mixed.sh
#Author: Nicholas Clement
#Project: CSCI 3753 Programming Assignment 4
#Create Date: 2016/11/11
#Description:
# Bash script to test mixed CPU-I/O times with different scheduling processes

FORKA=5
FORKB=20
FORKC=100
BYTESTOCOPY=102400
BLOCKSIZE=1024
TIMEFORMAT="wall=%e user=%U system=%S CPU=%P i-switched=%c v-switched=%w"
MAKE="make -s"

echo Building code...
$MAKE clean
$MAKE

echo ------------------------------------------------------------------------------1
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_OTHER with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_OTHER 0
echo -
echo using SCHED_OTHER with 10 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_OTHER 0
echo -
echo using SCHED_OTHER with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_OTHER 0
echo ------------------------------------------------------------------------------2
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_OTHER with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_OTHER 1
echo -
echo using SCHED_OTHER with 10 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_OTHER 1
echo -
echo using SCHED_OTHER with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_OTHER 1
echo ------------------------------------------------------------------------------3
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_FIFO with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_FIFO 0
echo  -
echo using SCHED_FIFO with 10 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_FIFO 0
echo -
echo using SCHED_FIFO with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_FIFO 0
echo ------------------------------------------------------------------------------4
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_FIFO with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_FIFO 1
echo -
echo using SCHED_FIFO with 20 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_FIFO 1
echo -
echo using SCHED_FIFO with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_FIFO 1
echo ------------------------------------------------------------------------------5
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_RR with 5 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_RR 0
echo -
echo using SCHED_RR with 20 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_RR 0
echo -
echo using SCHED_RR with 100 simultaneous process PRIORITY 0
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_RR 0
echo ------------------------------------------------------------------------------6
```

```
# echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rwinput to rwoutput
echo using SCHED_RR with 5 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKA SCHED_RR 1
echo -
echo using SCHED_RR with 20 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKB SCHED_RR 1
echo -
echo using SCHED_RR with 100 simultaneous process PRIORITY 1
/usr/bin/time -f "$TIMEFORMAT" ./cpu-io $BYTESTOCOPY $BLOCKSIZE $FORKC SCHED_RR 1
```

pi-sched.c

```
/*
 * File: pi-sched.c
 * Author: Andy Sayler
 * Revised: Dhivakant Mishra
 * Project: CSCI 3753 Programming Assignment 4
 * Create Date: 2012/03/07
 * Modify Date: 2012/03/09
 * Modify Date: 2016/31/10
 * 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>

#define FORKS 1

#define DEFAULT_ITERATIONS 100000000
#define RADIUS (RAND_MAX / 2)

static double dist(double x0, double y0, double x1, double y1){
    return sqrt(pow((x1-x0),2) + pow((y1-y0),2));
}

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


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

    long i;
    long iterations;
    struct sched_param param;
    int policy;
    int forks;
    int niceness;
    int niceNumb;
    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 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){
      forks = FORKS;
      printf("Using default number of forks\n");
    }
    else{
      forks = atoi(argv[3]);
    //  printf("Using user specified forks\n");
      if(forks <= 1){
        fprintf(stderr, "Bad fork value\n");
        exit(EXIT_FAILURE);
      }
    }

    if (argc < 5){
      niceness = 0;
      printf("Not changing priority");
    }
    else{
      niceness = atoi(argv[4]);
      if (niceness != 1){
        printf("Not changing priority");
        niceness = 0;
      }
    }


    /* 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;
    }

    }

  param.sched_priority = sched_get_priority_max(policy);
    for ( int i = 0; i < forks; i++ ){
      if ( fork() == 0 )
        {
          if (niceness == 1){
            printf("niceness == 1");
            niceNumb ++;
            int test = nice(niceNumb);
            //error handling after decreasing priority
            if (test == -1){
              fprintf(stderr, "Unable to change niceness\n");
                exit(EXIT_FAILURE);
            }
          }

        if(sched_setscheduler(0, policy, &param)){
              perror("Error setting scheduler policy");
              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 * /
          fprintf(stdout, "pi = %f\n", piCalc);

          exit( 0 );
        }
      }

      for ( int i = 0; i < forks; i++ ){
          wait( NULL );
      }

      return 0;
  }
```

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>
  #include <sched.h>

  /* Local Defines */
  #define MAXFILENAMELENGTH 80
  #define DEFAULT_INPUTFILENAME "rwinput"
  #define DEFAULT_OUTPUTFILENAMEBASE "testOut/"
  #define DEFAULT_BLOCKSIZE 1024
  #define DEFAULT_TRANSFERSIZE 1024*100
  #define FORKS 1

  int main(int argc, char* argv[]){
      int testn = 0;
      struct sched_param param;
      int forks;
      int policy;
      int niceness;
      int niceNumb;
      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);
      }
  }


if (argc < 4){
  forks = FORKS;
  printf("Using default number of forks\n");
}
else{
  forks = atoi(argv[3]);
//  printf("Using user specified forks\n");
  if(forks <= 1){
    fprintf(stderr, "Bad fork value\n");
    exit(EXIT_FAILURE);
  }
}
/* add in scheduling policy*/
if(argc < 5){
     policy = SCHED_OTHER;


}
else{
  if(!strcmp(argv[4], "SCHED_OTHER")){
        policy = SCHED_OTHER;
      //printf("schedule = SCHED_OTHER");
    }
    else if(!strcmp(argv[4], "SCHED_FIFO")){
        policy = SCHED_FIFO;
      // printf("schedule = SCHED_FIFO");
    }
    else if(!strcmp(argv[4], "SCHED_RR")){
        policy = SCHED_RR;
      // printf("schedule = SCHED_RR");
    }
}

if (argc < 6){
```

```
  niceness = 0;
  printf("Not changing priority");
}
else{
  niceness = atoi(argv[5]);
  if (niceness != 1){
    printf("Not changing priority");
    niceness = 0;
  }
}

if(argc < 7){
     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[6], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
        fprintf(stderr, "Input filename too long\n");
         exit(EXIT_FAILURE);
        }
     strncpy(inputFilename, argv[6], 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);
}

//Fork here
param.sched_priority = sched_get_priority_max(policy);
for ( int i = 0; i < forks; i++ ){
  if ( fork() == 0 )
    {
      if (niceness == 1){
        printf("niceness == 1");
        niceNumb ++;
        testn ++;
        int test = nice(niceNumb);
          //error handling
        if (test == -1){
          fprintf(stderr, "Unable to change niceness\n");
          exit(EXIT_FAILURE);
        }
      }

      /* make a new random file*/
      strncpy(outputFilename, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);

      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 =

/*read and write occurs here */
  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);
  }

    /* 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);

    /* 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);
    }
      printf('CHILD \n');
```

```
                exit( 0 );
              }
            }

  for ( int i = 0; i < forks; i++ ){
      wait( NULL );
  }

    return EXIT_SUCCESS;
}
```

cpu-io.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 <sched.h>
#include <math.h>


/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "testOut/"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define FORKS 1
/* for calc pi */
#define DEFAULT_ITERATIONS 100000000
#define RADIUS (RAND_MAX / 2)

static double dist(double x0, double y0, double x1, double y1){
    return sqrt(pow((x1-x0),2) + pow((y1-y0),2));
}

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



int main(int argc, char* argv[]){
    int testn = 0;
    struct sched_param param;
    int forks;
    int policy;
    int niceness;
    int niceNumb;
    int rv;
    int inputFD;
    int outputFD;
    char inputFilename[MAXFILENAMELENGTH];
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];


    long iterations;
    double x, y;
    double inCircle = 0.0;
    double inSquare = 0.0;
```

```
double pCircle = 0.0;
double piCalc = 0.0;

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);
      }
  }


if (argc < 4){
  forks = FORKS;
  printf("Using default number of forks\n");
}
else{
  forks = atoi(argv[3]);
//  printf("Using user specified forks\n");
  if(forks <= 1){
    fprintf(stderr, "Bad fork value\n");
    exit(EXIT_FAILURE);
  }
}
/* add in scheduling policy*/
if(argc < 5){
     policy = SCHED_OTHER;
}
else{
  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;
    }
}
if (argc < 6){
  niceness = 0;
//  printf("Not changing priority");
```

```
}
else{
  niceness = atoi(argv[5]);
  if (niceness != 1){
    printf("Not changing priority");
    niceness = 0;
  }
}


if (argc < 7){
  iterations = DEFAULT_ITERATIONS;
}



if(argc < 8){
     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[6], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH){
        fprintf(stderr, "Input filename too long\n");
         exit(EXIT_FAILURE);
        }
     strncpy(inputFilename, argv[6], 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);
}

//Fork here
param.sched_priority = sched_get_priority_max(policy);
for ( int i = 0; i < forks; i++ ){
  if ( fork() == 0 )
    {
      if (niceness == 1){
        printf("niceness == 1");
        niceNumb ++;
        testn ++;
        int test = nice(niceNumb);
          //error handling
        if (test == -1){
          fprintf(stderr, "Unable to change niceness\n");
          exit(EXIT_FAILURE);
        }
      }

/* First run CPU intesive task */

/* 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;

        /* Next run I/O*/


        /* make a new random file in the testOut folder */
        strncpy(outputFilename, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);

        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 =

    /*read and write occurs here */
        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);
        }

        /* 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);

        /* 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);
        }
          printf('CHILD \n');

          exit( 0 );
        }
      }

      for ( int i = 0; i < forks; i++ ){
          wait( NULL );
      }

        return EXIT_SUCCESS;
    }
```