Programming Assignment #4

Thomas Lillis
CSCI 3753

Abstract

The current mainline Linux kernel uses a CPU scheduler called “Completely Fair Scheduler” or CFS. The goal of this experiment is to test the Linux CFS against two other real time schedulers in Linux, the first-in first-out scheduler, and the round-robin scheduler. This done on with three different types of programs to see what schedulers work best in with different programs. These 3 different types of programs are CPU bound, I/O bound, or mixed between the two former types. Also, different sizes in the number of processes of each program where measured on each of the schedulers. The results came through that the Linux CFS scheduler had the best overall performance according to smallest wall time for most use cases.

Introduction

The goal of this assignment was to measure the performance of different Linux CPU schedulers running different types of programs. We tested three different schedulers: *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER*. *SCHED_FIFO* used a first-in first-out scheduling policy. *SCHED_RR* uses a round-robin policy giving everything and equal amount of time on the CPU. Last, *SCHED_OTHER* is the default Linux scheduler which is the “Completely Fair Scheduler” or CFS. The goal of the CFS scheduler is to give things equal amounts of CPU time so processes that have not gotten much time are weighted more heavily towards getting more time than processes that ave been on the processor a lot.

The main performance characteristic I follow in this experiment is the wall time or the time it takes from start to finish of the program. The amount of time spent in kernel mode vs. user mode is also looked at as well as voluntary and involuntary context switches. All of these characteristics can be looked at to analyze the how each of the schedulers handle different times of processes and loads. Different characteristics may be optimal for different types of systems so there is not necessarily one scheduler to rule them all. This is why Linux provides different schedulers for real-time vs. non-real-time jobs.

Method

Setup

I was provided with two separate C programs. The first program *pi.c* calculated the value of pi using the Monte Carlo method. This program is heavy with computation and so it is considered CPU bound. The other program we were provided was *rw.c* which copies blocks of bytes from an input file to an output file.

I extended these programs to provide the ability to chose one of three Linux CPU schedulers, *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER*. I then wrote a container program that forks a given program a given amount of times. Then, to extend it to another level I then wrote a script that completed 27 different tests using *usr/bin/time* with varying scheduler types, varying amount of forks, and different types of programs.

This script output the useful data from */usr/bin/time* into 27 different files. I then wrote a python script to combine these files into into a single space delimited file. This was then imported into Libre Office Calc for data analysis.

The script was run without any other programs running other than basic Linux utilities and the XFCE window manager environment to prevent unknown variables. There was no interaction with the computer while it was running the script.

The tests where done on a Lenovo X230-Tablet running a native installation of 64-bit Xubuntu 15.10. The computer has a solid state drive and a 3rd generation Intel CORE i7 Processor.

Output of *uname -a:*

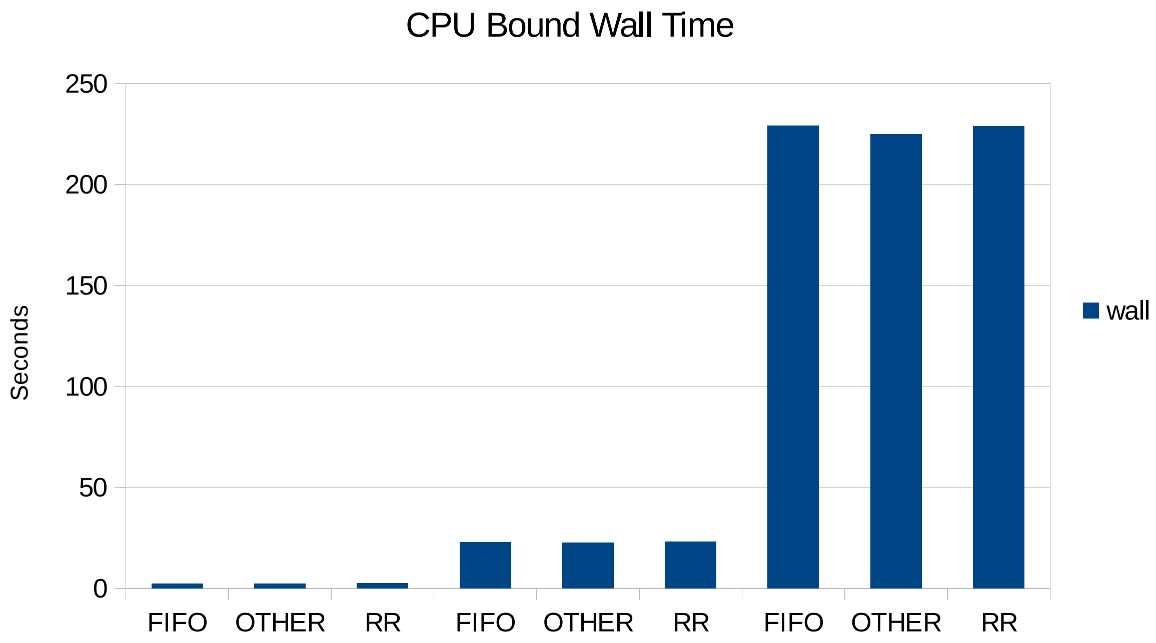
```
Linux thinkpad 3.16.0-45-generic #60~14.04.1-Ubuntu SMP Fri Jul 24 21:16:23
UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
```

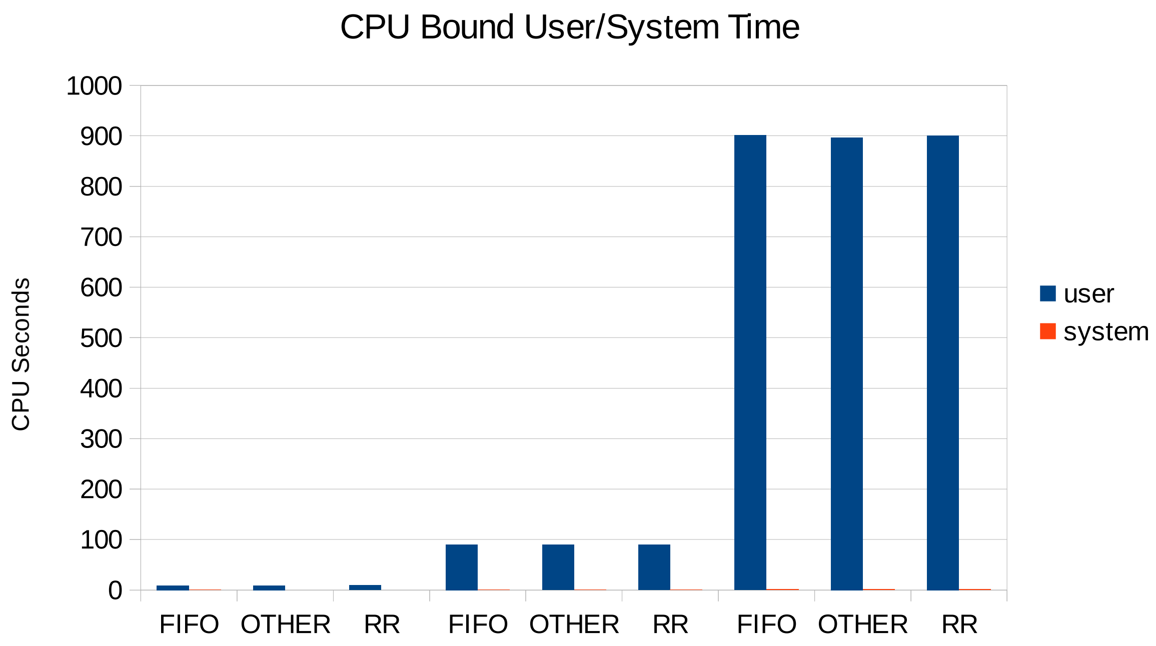
Benchmarks

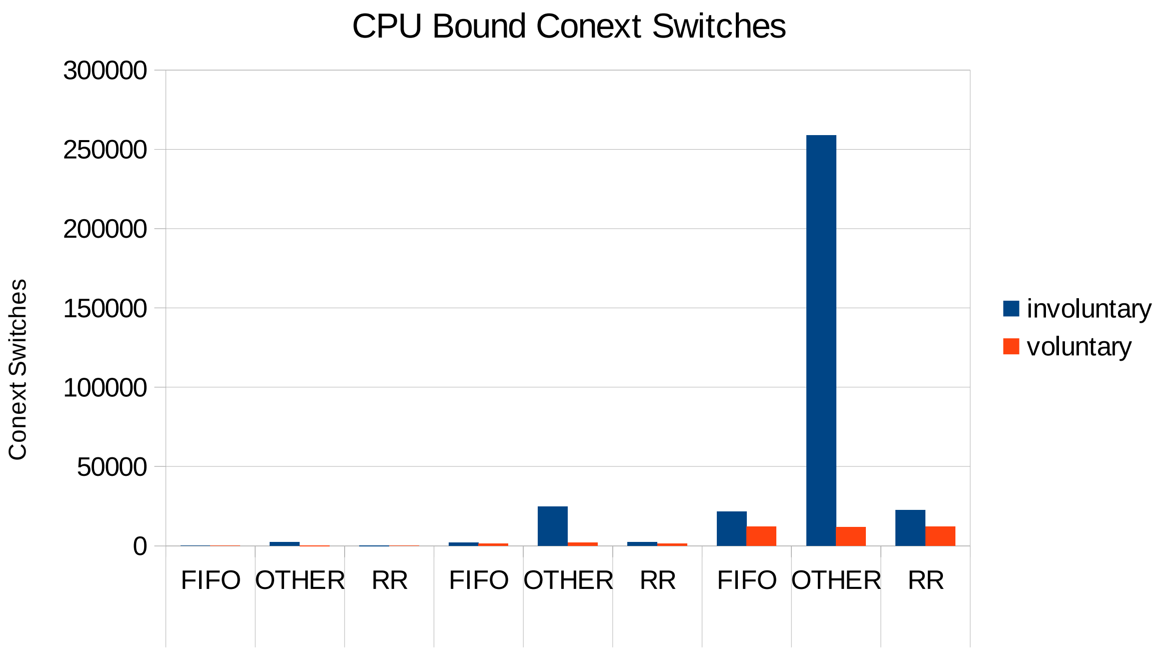
Several benchmarks were recorded with */usr/bin/time.*

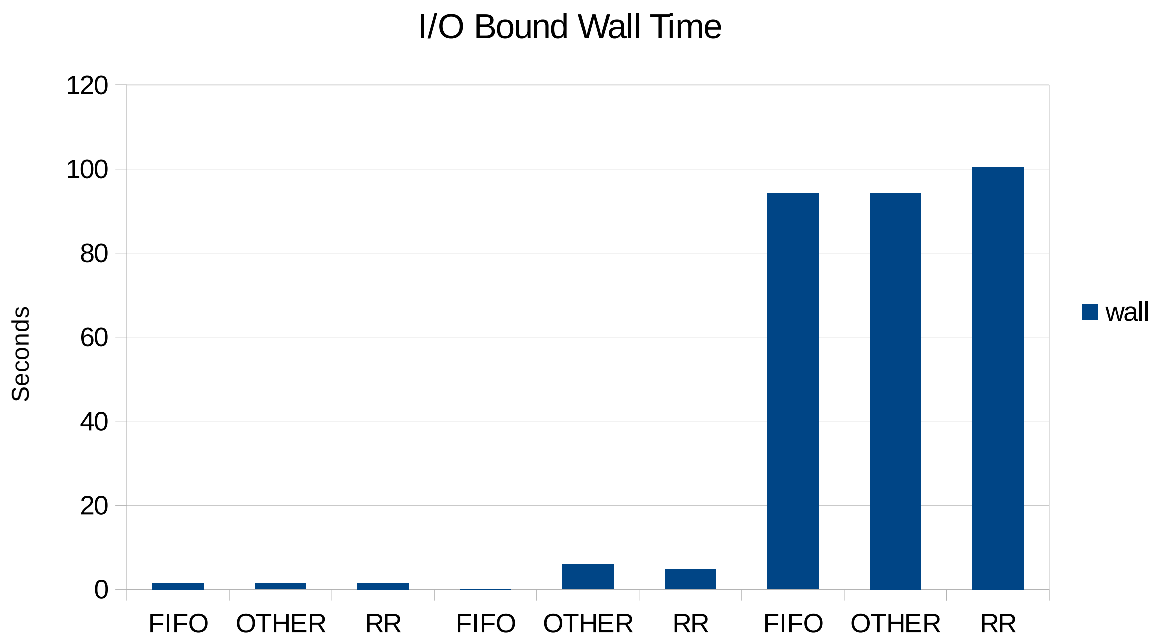
- Wall time: This is the real time start to finish of the program.
- User time: This is the time actually spent on the code.
- System time: This is the time spent in the kernel (swaps/context switches).
- CPU%: The percentage of the CPU used.
- I-switched: This is an involuntary context switch. This is when the systems makes the program get off the CPU. An example of this would be for a round robin time quanta. Once it is up the process is moved off the CPU.
- V-switched: This is a voluntary context switch. This is when the system gives up the CPU usually because it is waiting for I/O.

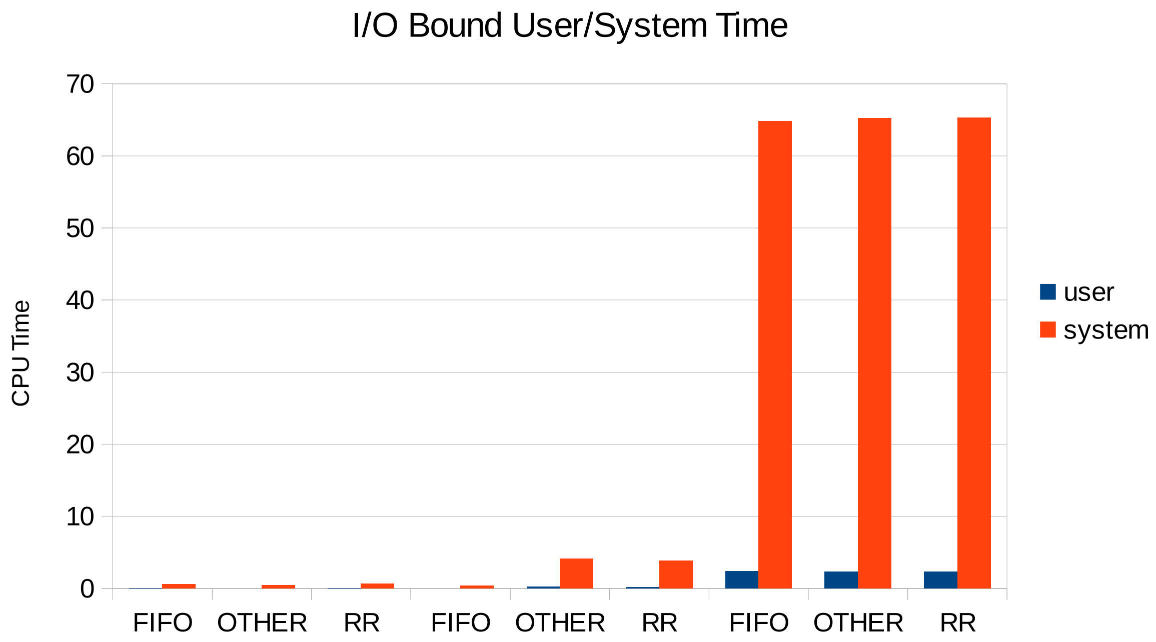
Results

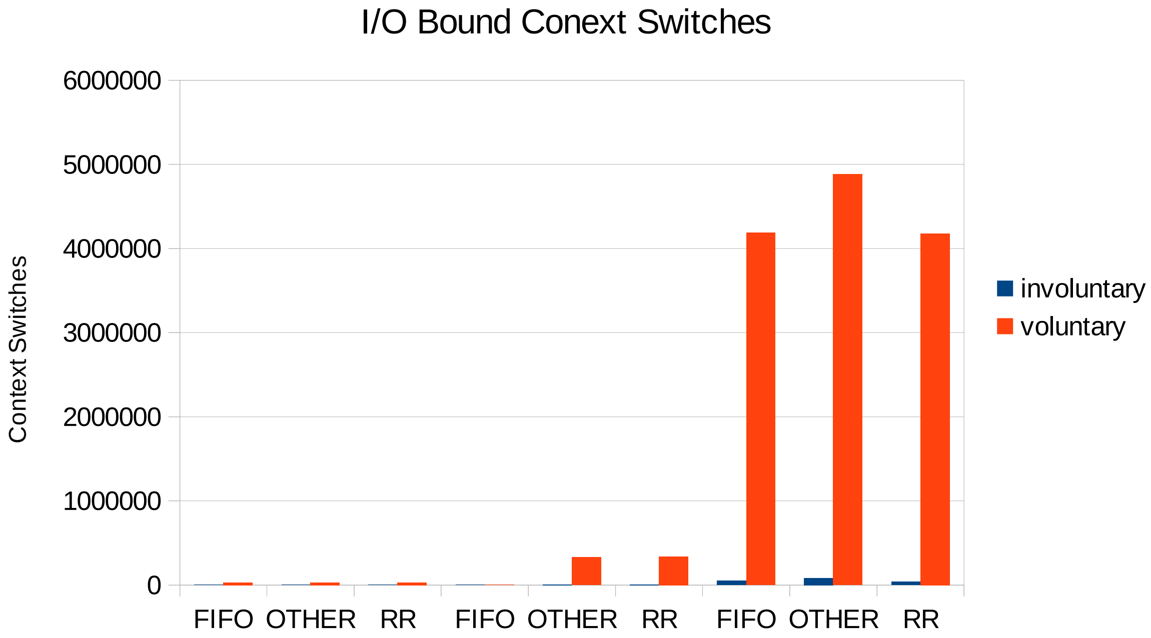


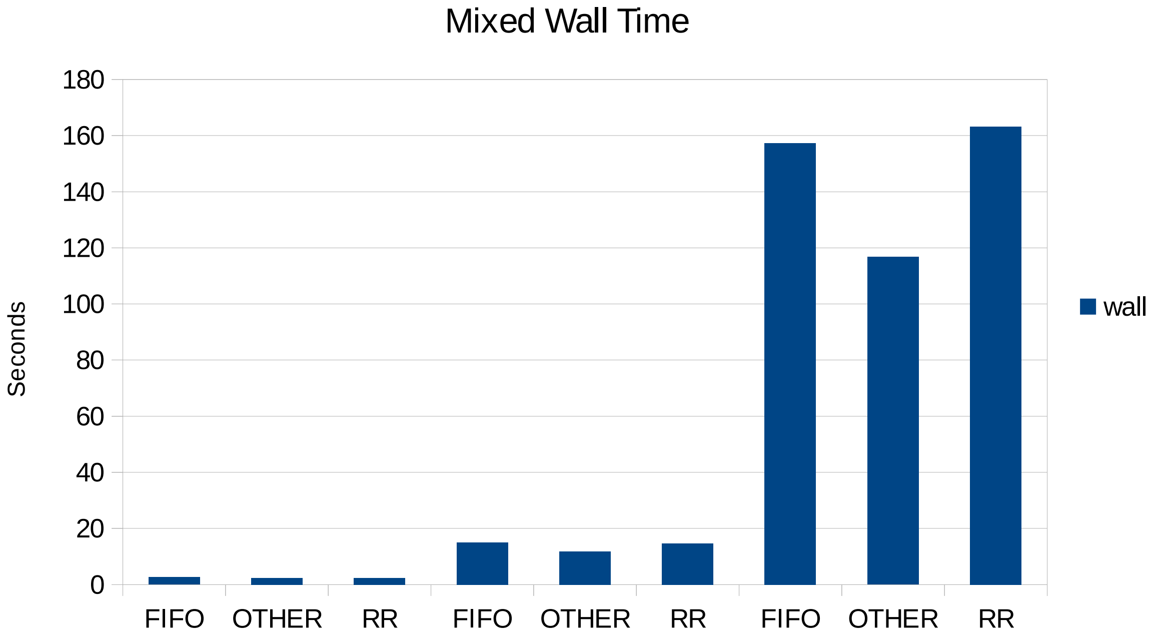

CPU Bound User/System Time
CPU Seconds
1000
900
800
700
600
500
400
300
200
100
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
■ user
■ system

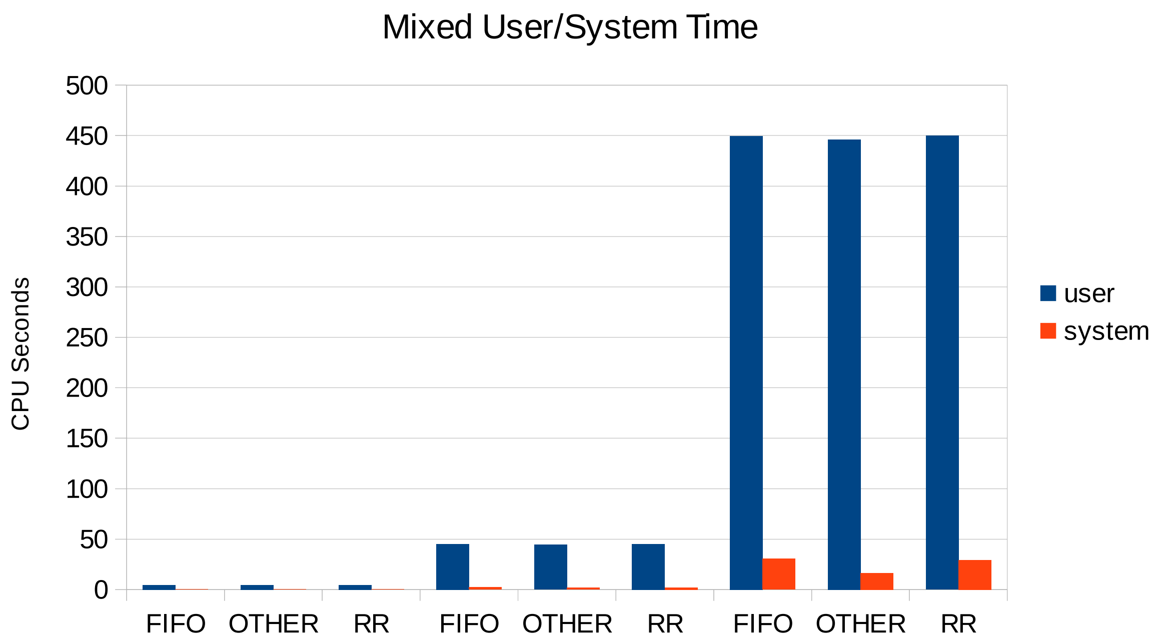

CPU Bound Conext Switches
Conext Switches
300000
250000
200000
150000
100000
50000
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
■ involuntary
■ voluntary

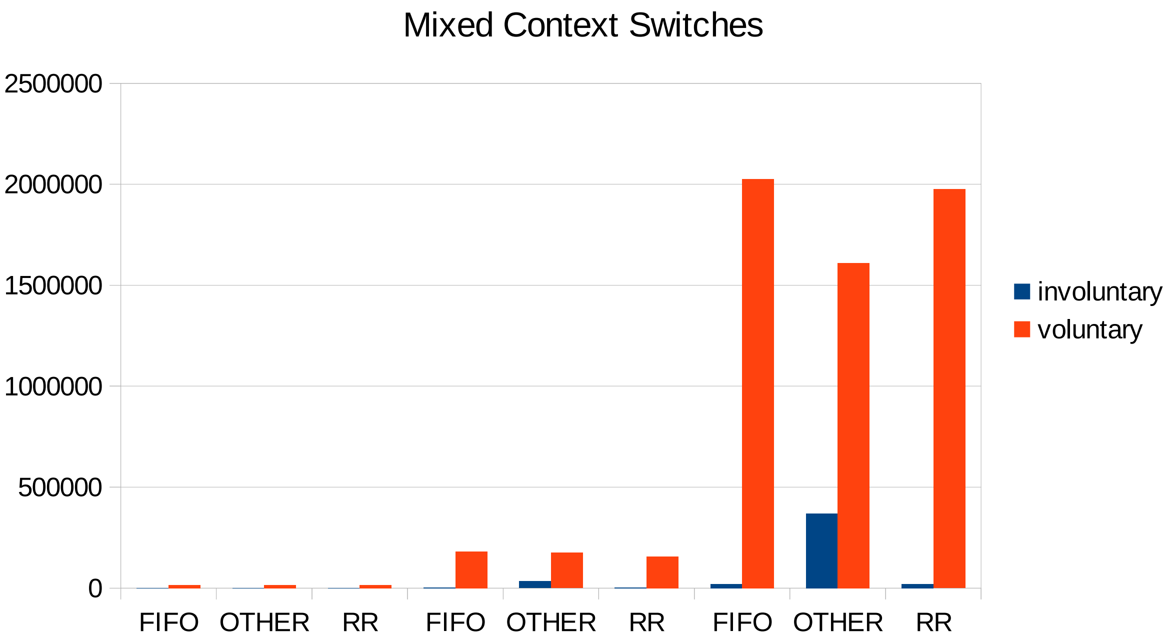

I/O Bound Wall Time
Seconds
120
100
80
60
40
20
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
wall


I/O Bound User/System Time
CPU Time
70
60
50
40
30
20
10
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
user
system


I/O Bound Conext Switches
Context Switches
6000000
5000000
4000000
3000000
2000000
1000000
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
involuntary
voluntary


Mixed Wall Time
Seconds
180
160
140
120
100
80
60
40
20
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
wall


Mixed User/System Time
CPU Seconds
500
450
400
350
300
250
200
150
100
50
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
user
system


Mixed Context Switches
2500000
2000000
1500000
1000000
500000
0
FIFO
OTHER
RR
FIFO
OTHER
RR
FIFO
OTHER
RR
involuntary
voluntary

Analysis

CPU-Bound

For CPU-Bound programs, no matter the size, the Linux CFS slightly outperformed each of the other two schedulers. The CFS was about 2% faster for all sizes of the forks of the program. On the other hand CFS had the most involuntary context switches. Both the FIFO and RR schedulers had about %8 of the involuntary context switches that CFS had. With voluntary context switches on the other hand all three schedulers performed about the same.

CFS had so many involuntary context switches because it is trying to be fair to all of the processes. When a process has not gotten much of the CPU it gets weighted more heavily to be run next and get more time. I would expect more involuntary context switches from the RR scheduler than the FIFO scheduler but that did not seem to occur.

For all the schedulers the user time was much greater than the system time. There was almost not system time. This makes sense because everything is just user space code running on the CPU. There are not many system calls needed so there is almost no time spent in the kernel space.

I/O-Bound

For I/O bound processes Linux CFS and FIFO performed about the same at the large scale. RR was the worst for large scale. This makes sense because RR will keep context switching while CFS and FIFO can get through more of the CPU time and then context switch away for I/O. RR performed about 10% worse than the other two schedulers when it came to large scale I/O. On a medium scale FIFO performed the best while Linux CFS performed the worst. On a small scale all of the schedulers performed about the same.

For all of the schedulers there were many more voluntary context switches than involuntary. This is because when waiting for I/O the processes all voluntarily give up the CPU.

For all of the schedulers the majority of the time was spent in kernel space. That is because I/O involves kernel level actions. There still is some time spent in user space because there is still user space code being run but not even close to the amount of CPU time spend on the system.

Another unexpected result with the I/O bound program is the FIFO scheduler for the medium sized task performed better than the FIFO scheduler for the smaller task. It is unclear why this is the case. In the medium results it also performed much better than the other two schedulers.

The I/O jobs could have taken much more time in general if a solid state drive was not used. Because of this they were pretty quick relative to how slow they could have been with a standard hard drive.

Mixed

The mixed type program produced the most interesting results. CFS significantly out performed the two other schedulers. CFS only took 73% of the time it took the two other schedulers to finish their jobs. It also had significantly less voluntary context switches than the other two schedulers. CFS had 20% less context voluntary switches compared to the other two schedulers while it had 10 fold more involuntary context switches versus the other two schedulers.

Even though there are an equal number of CPU-Bound and I/O-Bound processes running, there is still way more time spent in the system. This is because I/O take much more time than running user level code so much more of the time will be spent in I/O.

As stated before there were an equal number of processes that were CPU-Bound and I/O-Bound. Even though this was the case there were significantly more voluntary context switches than involuntary. This is because still majority of the time is spent waiting for I/O. When waiting for I/O the CPU is voluntarily given up.

Other Trends

I would expect to see more difference in in RR vs. FIFO schedulers. Mainly, I would expect to see more involuntary context switches from RR compared to FIFO. I would expect to see RR getting a similar amount of context switches to Linux CFS but that is not the case in the results for any of the types of programs.

All of the programs performed at about the same level on the small scale. It was hard to see much meaningful performance difference. As the scale increased you could see the different schedulers performances change. This makes sense. Sometimes the trend in performance changed going from medium scale to large scale. This is unusual but means that the optimal scheduler can vary based on the size of the task.

Conclusion

CFS is the best performing scheduler in almost every use case according to wall time. The only competition it had was FIFO for I/O bound scheduling. For CPU-Bound scheduling it had a slight edge over the two other schedulers and for the mixed case it performed extremely better than the other two schedulers. The mixed case is the most important case to analysis for a typical desktop user because most desktop usage is a mix between I/O and CPU heavy processes. This shows that Linux has a very good scheduler for flexible usage and FIFO and RR should only be used if a process needs more priority than the normal scheduler (soft real-time). These results are according to the wall time characteristic so nothing can be said for other means of measurement of performance.

References

Andy Saylor. *Programming Assignment 4: Investigating the Linux Scheduler-Handout.*

Abraham Silberschatz, Peter Galvin, Greg Gagne. *Operating System Concepts, 9th Edition.*

Richard Han. *Chapter 6: Advanced Scheduling Slides.*

Appendix a: Raw data

```
program,forks,sched,wall,user,system,CPU,I-switched,V-switched
CPU,100,FIFO,2.28,8.97,0.03,394%,134,122
CPU,100,OTHER,2.25,8.98,0,398%,2479,217
CPU,100,RR,2.36,9.1,0,385%,221,127
CPU,1000,FIFO,22.88,89.9,0.15,393%,2159,1237
CPU,1000,OTHER,22.55,89.8,0.14,398%,24869,2108
CPU,1000,RR,23.11,89.82,0.14,389%,2213,1202
CPU,10000,FIFO,229.17,901.53,1.27,393%,21456,12163
CPU,10000,OTHER,224.86,896.72,1.34,399%,258793,11785
CPU,10000,RR,228.82,900.64,1.27,394%,22499,11992
IO,100,FIFO,1.45,0.01,0.56,39%,253,30137
IO,100,OTHER,1.4,0,0.43,31%,177,30354
IO,100,RR,1.44,0.02,0.61,44%,254,30167
IO,1000,FIFO,0.11,0,0.34,314%,75,1054
IO,1000,OTHER,5.99,0.19,4.09,71%,5329,331011
IO,1000,RR,4.82,0.15,3.87,83%,3429,336151
IO,10000,FIFO,94.24,2.39,64.83,71%,48871,4185427
IO,10000,OTHER,94.22,2.26,65.19,71%,84235,4882731
IO,10000,RR,100.56,2.28,65.27,67%,39934,4178608
Mix,100,FIFO,2.61,4.36,0.22,175%,176,15270
Mix,100,OTHER,2.47,4.36,0.25,187%,172,15399
Mix,100,RR,2.38,4.44,0.22,195%,204,15198
Mix,1000,FIFO,14.96,45.21,2.25,317%,1937,180345
Mix,1000,OTHER,11.72,44.47,1.74,394%,34307,175439
Mix,1000,RR,14.68,45.15,2.07,321%,1984,156759
Mix,10000,FIFO,157.22,449.44,30.71,305%,20624,2026760
Mix,10000,OTHER,116.7,446.26,16.21,396%,369629,1609316
Mix,10000,RR,163.22,449.91,29.18,293%,20739,1975673
```

Appendix b: All Code

testscript

```
#/!/bin/bash

#File: testscript
#Author: Andy Sayler/Thomas Lillis
#Project: CSCI 3753 Programming Assignment 3
#Create Date: 2012/03/09
#Modify Date: 2012/03/21
#Description:
#    A simple bash script to run a signle copy of each test case
#    and gather the relevent data.

ITERATIONS=1000000
BYTESTOCOPY=102400
BLOCKSIZE=1024
FORKS1=100
FORKS2=1000
FORKS3=10000
PROGRAM1=test.c
PROGRAM1=rw-sched.c
PROGRAM1=mixed
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...

### testing CPU bound ###
echo
echo TESTING CPU BOUND...
#scalability small
echo testing small scalability...
echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER
with $FORKS1 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" ./test ./pi-sched $FORKS1 $ITERATIONS
SCHED_OTHER  2> pi-sched-$FORKS1-SCHED_OTHER > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with
$FORKS1 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS1
$ITERATIONS SCHED_FIFO 2> pi-sched-$FORKS1-SCHED_FIFO > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with
$FORKS1 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS1
$ITERATIONS SCHED_RR 2> pi-sched-$FORKS1-SCHED_RR > /dev/null

#scalability medium
```

```
echo testing medium scalability...
echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER
with $FORKS2 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" ./test ./pi-sched $FORKS2 $ITERATIONS
SCHED_OTHER 2> pi-sched-$FORKS2-SCHED_OTHER > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with
$FORKS2 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS2
$ITERATIONS SCHED_FIFO 2> pi-sched-$FORKS2-SCHED_FIFO > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with
$FORKS2 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS2
$ITERATIONS SCHED_RR 2> pi-sched-$FORKS2-SCHED_RR > /dev/null

#scalability large
echo testing large scalability...
echo Calculating pi over $ITERATIONS iterations using SCHED_OTHER
with $FORKS3 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" ./test ./pi-sched $FORKS3 $ITERATIONS
SCHED_OTHER 2> pi-sched-$FORKS3-SCHED_OTHER > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_FIFO with
$FORKS3 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS3
$ITERATIONS SCHED_FIFO 2> pi-sched-$FORKS3-SCHED_FIFO > /dev/null

echo Calculating pi over $ITERATIONS iterations using SCHED_RR with
$FORKS3 simultaneous process...
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./pi-sched $FORKS3
$ITERATIONS SCHED_RR 2> pi-sched-$FORKS3-SCHED_RR > /dev/null


### testing IO bound ###
echo
echo TESTING IO BOUND...
#scalability small
echo testing small scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS1 simultaneous processes and
SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS1
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> rw-sched-$FORKS1-SCHED_OTHER
> /dev/null

echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS1 simultaneous processes and
SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS1
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> rw-sched-$FORKS1-SCHED_FIFO
> /dev/null

echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS1 simultaneous processes and
SCHED_RR
```

```
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS1
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> rw-sched-$FORKS1-SCHED_RR >
/dev/null


#scalability medium
echo testing medium scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS2 simultaneous processes and
SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" ./test ./rw-sched $FORKS2 $BYTESTOCOPY
$BLOCKSIZE SCHED_OTHER 2> rw-sched-$FORKS2-SCHED_OTHER > /dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS2 simultaneous processes and
SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test /.rw-sched $FORKS2
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> rw-sched-$FORKS2-SCHED_FIFO
> /dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS2 simultaneous processes and
SCHED_RR
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS2
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> rw-sched-$FORKS2-SCHED_RR >
/dev/null



#scalability large
echo testing large scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS3 simultaneous processes and
SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" ./test ./rw-sched $FORKS3 $BYTESTOCOPY
$BLOCKSIZE SCHED_OTHER 2> rw-sched-$FORKS3-SCHED_OTHER > /dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS3 simultaneous processes and
SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS3
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> rw-sched-$FORKS3-SCHED_FIFO
> /dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE from rw-
schedinput to rw-schedoutput with $FORKS3 simultaneous processes and
SCHED_RR
/usr/bin/time -f "$TIMEFORMAT" sudo ./test ./rw-sched $FORKS3
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> rw-sched-$FORKS3-SCHED_RR >
/dev/null
```

```
### testing mix ###
echo
echo TESTING MIX...
#scalability small
echo testing small scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS1 simultaneous processes and SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS1 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> mixed-$FORKS1-SCHED_OTHER >
/dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS1 simultaneous processes and SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS1 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> mixed-$FORKS1-SCHED_FIFO >
/dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS1 simultaneous processes and SCHED_RR
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS1 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> mixed-$FORKS1-SCHED_RR >
/dev/null


#scalability medium
echo testing medium scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS2 simultaneous processes and SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" ./test mixed $FORKS2 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_OTHER 2> mixed-$FORKS2-SCHED_OTHER >
/dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS2 simultaneous processes and SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS2 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> mixed-$FORKS2-SCHED_FIFO >
/dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS2 simultaneous processes and SCHED_RR
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS2 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> mixed-$FORKS2-SCHED_RR >
/dev/null


#scalability large
echo testing large scalability...
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS3 simultaneous processes and SCHED_OTHER
/usr/bin/time -f "$TIMEFORMAT" ./test mixed $FORKS3 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_OTHER 2> mixed-$FORKS3-SCHED_OTHER >
/dev/null
```

```
echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS3 simultaneous processes and SCHED_FIFO
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS3 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_FIFO 2> mixed-$FORKS3-SCHED_FIFO >
/dev/null


echo Copying $BYTESTOCOPY bytes in blocks of $BLOCKSIZE and calc pi
with $ITERATIONS with $FORKS3 simultaneous processes and SCHED_RR
/usr/bin/time -f "$TIMEFORMAT" sudo ./test mixed $FORKS3 $ITERATIONS
$BYTESTOCOPY $BLOCKSIZE SCHED_RR 2> mixed-$FORKS3-SCHED_RR >
/dev/null

make clean

echo
echo "FINISHED! :)"
```

pi-sched.c

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

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

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

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

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;
    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;
    }
    /* Set iterations if supplied */
    if(argc > 1){
        iterations = atol(argv[1]);
        if(iterations < 1){
            fprintf(stderr, "Bad iterations value\n");
            exit(EXIT_FAILURE);
        }
    }
    /* Set policy if supplied */
    if(argc > 2){
        if(!strcmp(argv[2], "SCHED_OTHER")){
            policy = SCHED_OTHER;
        }
        else if(!strcmp(argv[2], "SCHED_FIFO")){
            policy = SCHED_FIFO;
        }
        else if(!strcmp(argv[2], "SCHED_RR")){
            policy = SCHED_RR;
        }
        else{
            fprintf(stderr, "Unhandeled scheduling policy\n");
            exit(EXIT_FAILURE);
        }
    }

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

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

    /* 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);

    return 0;
}
```

rw-sched.c

```
/*
 * File: rw.c
 * Author: Andy Sayler
 * Project: CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/19
 * Modify Date: 2012/03/20
 * Description: A small i/o bound program to copy N bytes from an
input
 *              file to an output file. May read the input file
multiple
 *              times if N is larger than the size of the input file.
 */

/* Include Flags */
#define _GNU_SOURCE

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

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

int main(int argc, char* argv[]){
    int rv;
    int inputFD;
    int outputFD;
    char inputFilename[MAXFILENAMELENGTH];
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];

    struct sched_param param;
    int policy;

    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 < 1){
        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 < 2){
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else{
        blocksize = atol(argv[2]);
        if(blocksize < 1){
            fprintf(stderr, "Bad blocksize value\n");
            exit(EXIT_FAILURE);
        }
    }

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

```
*/
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
MAXFILENAMELENGTH);
    /* Set policy if supplied */
    if(argc > 3){
        if(!strcmp(argv[3], "SCHED_OTHER")){
            policy = SCHED_OTHER;
        }
        else if(!strcmp(argv[3], "SCHED_FIFO")){
            policy = SCHED_FIFO;
        }
        else if(!strcmp(argv[3], "SCHED_RR")){
            policy = SCHED_RR;
        }
        else{
            fprintf(stderr, "Unhandeled scheduling policy\n");
            exit(EXIT_FAILURE);
        }
    }

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

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

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

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

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

    /* Open Output File Descriptor in Write Only mode with standard
permissions*/
    rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d",
            outputFilenameBase, getpid());
    if(rv > MAXFILENAMELENGTH){
        fprintf(stderr, "Output filenmae length exceeds limit of %d
characters.\n",
                MAXFILENAMELENGTH);
        exit(EXIT_FAILURE);
    }
    else if(rv < 0){
        perror("Failed to generate output filename");
        exit(EXIT_FAILURE);
    }
    if((outputFD =
                open(outputFilename,
                    O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH))
< 0){
        perror("Failed to open output file");
        exit(EXIT_FAILURE);
    }

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

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

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

    }while(totalBytesWritten < transfersize);

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

    /* Free Buffer */
    free(transferBuffer);

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

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

    return EXIT_SUCCESS;
}
```