

# Programming Assignment 3: Investigating the Linux Scheduler

Stephen Bennett  
CSCI 3753 - Operating Systems  
University of Colorado at Boulder  
Spring 2013

*Due Date: Sunday, March 31, 2013 11:55pm*

## **Abstract**

# 1 Introduction

## 2 Method

### 2.1 Benchmarks

Three benchmarks were written, each testing one of the three possible program types: CPU bound, I/O bound, and a mix of the two. In creating the CPU bound benchmark `pi.c`, some computationally intensive algorithm had to be chosen such that no time would be spent waiting on I/O operations or blocked on any other resource. As a result of these constraints, the statistical algorithm for calculating the value of pi based on the Monte Carlo method was chosen[1]. This algorithm is relatively slow and very CPU intensive. It creates an imaginary quarter circle of radius `RAND_MAX`, generates a random pair (x,y) of coordinates  $\{x, y \in \mathbb{R} \mid 0 \leq x, y \leq \text{RAND\_MAX}\}$ , then calculates whether the (x,y) coordinate is within the quarter circle. Additionally, there are two counters: one that keeps track of the total number of iterations and another that keeps track of the number of times the random (x,y) coordinate is found to be within the quarter circle. Finally, once all iterations are complete, all that must be done is to calculate the probability of being in the quarter circle by dividing the two counters (inside circle divided by number of iterations) and multiplying the result by 4 to create a full circle rather than a quarter circle.

The I/O bound benchmark `rw.c` was written in such a way as to “minimize the effects of filesystem buffering and maximize I/O delays”[2]. In order to do this, the low level-level `read()` and `write()` system calls were used in conjunction with files opened in `O_SYNC` mode. `O_SYNC` causes `write()` operations to block until the data is physically written to the disk rather than until the data is simply copied to a kernel buffer[3]. An input file and an output file are given to the program which then reads blocks of data from the input file and writes said data to the output file. This process occurs multiple times with minimal CPU involvement, thus creating the I/O bound benchmark. There is only a small amount of CPU use involved, primarily in setting up the input/output files and verifying the passed parameters.

The third and final benchmark program was the mixed benchmark `mix.c`. I wrote this benchmark as a combination of the ideas from the previous two benchmarks. The program performs the same computation as the CPU bound process statistically calculating the value of pi, but every so many iterations it writes the current values of the counters and the estimated value of pi to a file. This `write()` operation once again uses `O_SYNC` mode to maximize I/O delays.

For each benchmark, I wrote a separate program (e.g. `rw-sched.c` for I/O benchmark, etc.) that took care of setting the correct scheduling policy and `fork()`ing the desired number of child processes. Each time a process needs to read from or write to a file, it needs its own input or output file in order to prevent additional waiting due to mutual exclusions and not actual I/O as is desired; the housekeeping programs ensure that each process has its own unique input/output file.

## 2.2 Testing and Data

I wrote a Bash script to take care of automation for running the 27 different test cases. Each test case was run three times and the results were averaged in order to get more accurate data. While compiling the source code with the Makefile, a single 1 MB file called `rwinput` is created. It is created by reading 1024 blocks of size 1024 kB of random data from `/dev/urandom` and writing the data to `rwinput`. To ensure that each instance of the I/O bound benchmark program had its own input file, the Bash script then copies `rwinput` as many times as necessary and gives each copy a unique name. The `rw-sched.c` program then passes the proper unique input file name to each child process `fork()`ed. Through the use of the `time` command and nested `for` loops iterating over both the number of child processes to `fork()` and the scheduling algorithms, I gathered results for each of the test cases. From the `time` command I gathered the following aggregate information for each set of processes:

- Wall time (turnaround time) - the real time the process took to complete from when it entered the system to when it completed
- User time - the amount of CPU-seconds the process spent executing in user mode
- System time - the amount of CPU-seconds the process spent executing in kernel mode
- The percentage of the CPU that the process got
- The number of times the process was context switched
- The number of times the process was blocked on I/O

## 2.3 Test System

All of the tests were run on a desktop running 64-bit Linux Mint 14 Nadia with version `3.5.0-26-generic` of the Linux kernel. This all runs on a quad-core Intel Core i5-2500K CPU running at 3.30GHz with 8GB of RAM. Each core is capable of executing a single hardware thread; thus four processes may be running concurrently in this system. The primary disk for the system is a 1TB 7200 RPM Western Digital with 32MB of cache which uses the SATA II interface. All of the programs were compiled using `GNU C Compiler version 4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1)`. The time slice for the Round Robin scheduling algorithm for this setup is 0.1000006 seconds.

## 3 Results

Using the methodology described above, I gathered the results found in Figure 1.

### Wall Time CPU Bound Benchmark Results

Total wall time divided by number of child processes vs.  
number of child processes and scheduling algorithm

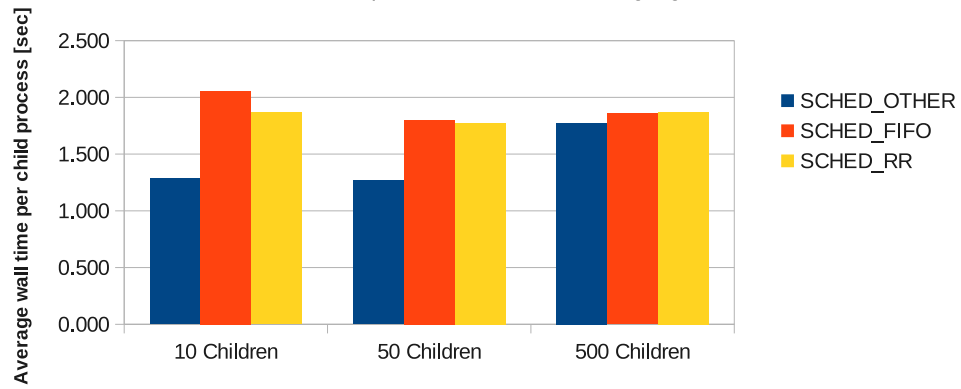


Figure 1:

### Wall Time I/O Bound Benchmark Results

Total wall time divided by number of child processes vs.  
number of child processes and scheduling algorithm

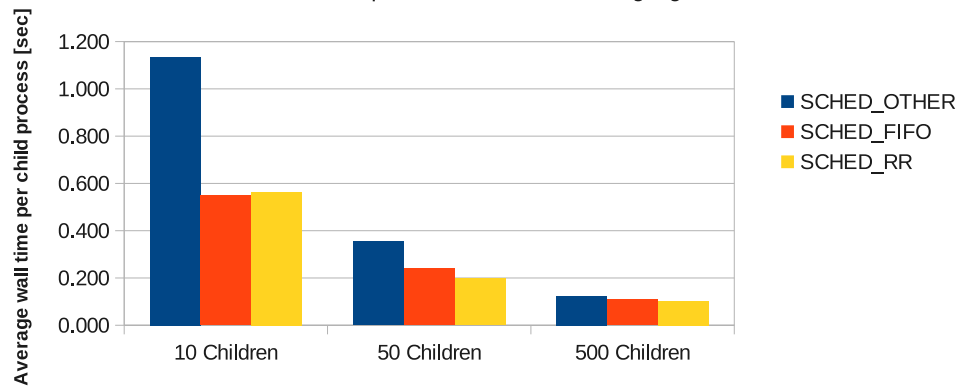


Figure 2:

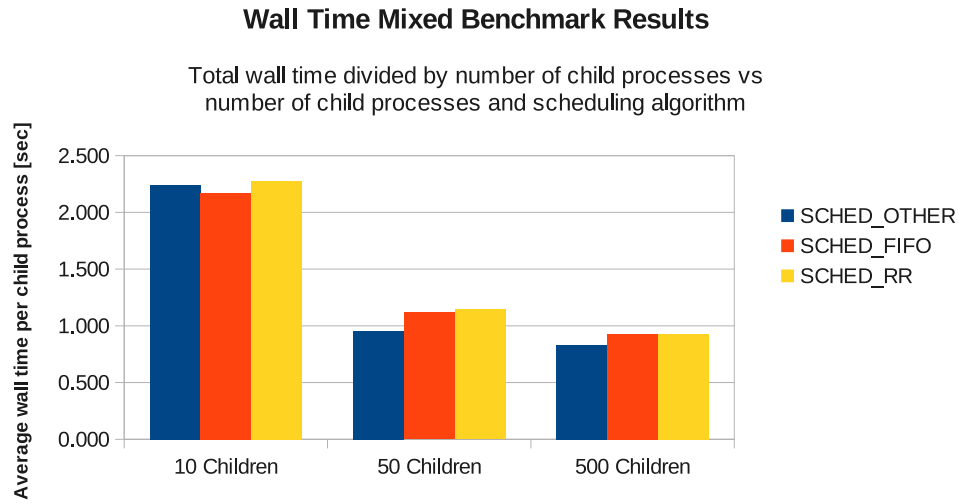


Figure 3:

This comes after the image. Where is it?

## 4 Analysis

## 5 Conclusion

## References

- [1] Sayler, Andy. *pi.c*. 03/09/2012. Forked 03/17/2013. <https://github.com/asayler/CU-CS3753-PA3/blob/master/pi.c>.
- [2] Sayler, Andy. *Programming Assignment 3: Investigating the Linux Scheduler*. 03/06/2012 <https://github.com/cgartrel/CU-CS3753-PA3/blob/master/handout/pa3.pdf>.
- [3] Linux man-pages project. *open(2)*. 2013/02/18. Access 03/23/2013. <http://man7.org/linux/man-pages/man2/open.2.html>.
- [4] Jones, M. Tim. *Inside the Linux 2.6 Completely Fair Scheduler*. IBM developerWorks: 2009. Accessed 06/01/12. <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>.
- [5] Kernighan, Brian and Dennis, Ritchie. *The C Programming Language*. Second Edition: 2009. Prentice Hall: New Jersey.
- [6] Kolivas, Con. *BFS - The Brain Fuck Scheduler*. 2009. Accessed 06/01/12. <http://ck.kolivas.org/patches/bfs/sched-BFS.txt>.
- [7] Kolivas, Con. *FAQS about BFS*. v0.330: 2009. Accessed 06/01/12. <http://ck.kolivas.org/patches/bfs/bfs-faq.txt>.
- [8] Kumar, Avinesh. *Multiprocessing with the Completely Fair Scheduler*. IBM developerWorks: 2008. Accessed 06/01/12. <http://www.ibm.com/developerworks/linux/library/l-cfs/>.
- [9] Le, Thang Minh. *A Study on Linux Kernel Scheduler: version 2.6.32*. 2009. Accessed 06/01/12. <http://www.scribd.com/thangmle/d/24111564-Project-Linux-Scheduler-2-6-32>.
- [10] Molnar, Ingo. *This is the CFS scheduler*. 2007. Accessed 06/01/12. <http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt>.

## A Appendix - Raw Data

Figure 4:

Process Type	Scheduler	Children	wall [sec]	user [CPU-sec]	system [CPU-sec]	CPU [%]	preempted [count]	blocked [count]
CPU Bound	SCHED_OTHER	10	12.84	50.40	0.03	392	6871	22
		50	63.39	252.20	0.10	398	65376	104
		500	888.47	3532.61	1.15	397	911979	1014
	SCHED_FIFO	10	20.55	63.56	0.02	309	50	15
		50	89.99	326.81	0.08	363	326	55
		500	928.95	3523.70	0.98	379	3685	505
	SCHED_RR	10	18.68	64.82	0.01	346	215	18
		50	88.57	329.68	0.07	371	1240	76
		500	934.63	3552.97	0.76	380	13501	791
I/O Bound	SCHED_OTHER	10	11.36	0.02	0.03	0	19	2687
		50	17.72	0.02	0.06	0	114	16957
		500	61.60	0.04	0.07	0	544	163474
	SCHED_FIFO	10	5.50	0.00	0.00	0	1	2012
		50	12.01	0.02	0.02	0	1	12494
		500	54.24	0.02	0.06	0	1	142977
	SCHED_RR	10	5.64	0.00	0.00	0	1	2012
		50	9.89	0.00	0.00	0	1	10052
		500	51.40	0.02	0.04	0	1	133495
Mixed	SCHED_OTHER	10	22.42	25.30	0.06	114	3657	4969
		50	47.76	126.03	0.22	264	33244	25253
		500	416.60	1588.08	2.95	381	461419	236701
	SCHED_FIFO	10	21.73	29.24	0.05	134	1	4262
		50	55.80	152.80	0.24	274	3	21425
		500	464.28	1747.49	2.67	376	1732	186482
	SCHED_RR	10	22.79	30.00	0.05	131	123	4343
		50	57.12	150.48	0.19	263	1258	21701
		500	464.80	1753.86	2.27	377	18655	172993



Figure 5:

Process Type	Scheduler	wall/child [sec]	user/child [CPU-sec]	system/child [CPU-sec]	preempted/child [count]	blocked/child [count]
CPU Bound	SCHED_OTHER	1.284	5.040	0.00333	687	2
		1.268	5.044	0.00200	1308	2
		1.777	7.065	0.00229	1824	2
	SCHED_FIFO	2.055	6.356	0.00167	5	2
		1.800	6.536	0.00167	7	1
		1.858	7.047	0.00195	7	1
	SCHED_RR	1.868	6.482	0.00133	21	2
		1.771	6.594	0.00140	25	2
		1.869	7.106	0.00153	27	2
I/O Bound	SCHED_OTHER	1.136	0.002	0.00267	2	269
		0.354	0.000	0.00113	2	339
		0.123	0.000	0.00014	1	327
	SCHED_FIFO	0.550	0.000	0.00000	0	201
		0.240	0.000	0.00047	0	250
		0.108	0.000	0.00013	0	286
	SCHED_RR	0.564	0.000	0.00000	0	201
		0.198	0.000	0.00000	0	201
		0.103	0.000	0.00009	0	267
Mixed	SCHED_OTHER	2.242	2.530	0.00567	366	497
		0.955	2.521	0.00433	665	505
		0.833	3.176	0.00591	923	473
	SCHED_FIFO	2.173	2.924	0.00533	0	426
		1.116	3.056	0.00487	0	428
		0.929	3.495	0.00534	3	373
	SCHED_RR	2.279	3.000	0.00500	12	434
		1.142	3.010	0.00373	25	434
		0.930	3.508	0.00453	37	346

## B Appendix - All Code

Listing 1: CPU Bound Benchmark pi.c

```
/*
 * File:          pi.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.
 */

/* 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)

/* 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 method 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;
}

```

Listing 2: CPU Bound Benchmark Scheduler pi-sched.c

```

/*
 * File:          pi-sched.c
 * Author:        Andy Sayler
 * Modified by:   Stephen Bennett
 * Project:       CSCI 3753 Programming Assignment 3
 * Create Date:   2012/03/07
 * Modify Date:   2013/03/21
 * Description:
 *   A program for setting the desired scheduling policy and creating
 *   the desired number of child processes of pi.c
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h> // Used for fork
#include <sys/wait.h>   // Used for fork
#include <unistd.h>     // Used for fork

```

```

/* Local Defines */
#define DEFAULT_ITERATIONS 1000000
#define DEFAULT_CHILDREN 10

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

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

    /* 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 default number of child processes to spawn */
    if (argc < 4) {
        children = DEFAULT_CHILDREN;
    }

    /* Set iterations if supplied */
    if (argc > 1) {
        iterations = atol(argv[1]);
        if (iterations == 0) {
            /* Set default iterations */
            iterations = DEFAULT_ITERATIONS;
        }
        else if (iterations < 0) {
            fprintf(stderr, "Bad iterations value [%li]\n", iterations);
            fprintf(stderr, " 0: Default iterations [%d]\n", DEFAULT_ITERATIONS);
            fprintf(stderr, "<0: Bad iterations value\n");
            fprintf(stderr, ">0: Good 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, "Unhanded scheduling policy [%s]\n", argv[2]);
        fprintf(stderr, "Available scheduling policies:\n");
        fprintf(stderr, "    SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
        exit(EXIT_FAILURE);
    }
}

/* Set children if supplied */
if (argc > 3) {
    children = atol(argv[3]);
    if (children == 0) {
        /* Set default iterations */
        children = DEFAULT_CHILDREN;
    }
    else if (children < 0) {
        fprintf(stderr, "Bad children value [%d]\n", children);
        fprintf(stderr, "    0: Default children [%d]\n", DEFAULT_CHILDREN);
        fprintf(stderr, "    <0: Bad children value\n");
        fprintf(stderr, "    >0: Good children value\n");
        exit(EXIT_FAILURE);
    }
}

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

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

/* Fork children child processes */
for (i = 0; i < children; ++i) {
    if ((pid = fork()) == -1) exit(EXIT_FAILURE);    /* Fork Failed */

    if (pid == 0) { /* Child process */
        // execl(exe, argv[0], argv[1], argv[2], ..., NULL)
        execl("pi", "pi", argv[1], NULL);
        exit(EXIT_SUCCESS);
    } else { /* Parent process */
        printf("Forked %d pid = %d\n", i, pid);
    }
}

/* Wait for children child processes to finish */
for (i = 0; i < children; ++i) {
    pid = wait(NULL);
}

```

```

        printf("Waited %d pid = %d\n", i+1, pid);
    }

    return 0;
}

```

Listing 3: I/O Bound Benchmark rw.c

```

/*
 * File:          rw.c
 * Author:        Andy Sayler
 * Project:       CSCI 3753 Programming Assignment 3
 * Create Date:   2012/03/19
 * Modify Date:   2013/03/21
 * 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>

/* 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];

    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 == 0) {
        transfersize = DEFAULT_TRANSFERSIZE;
    }
    else if (transfersize < 0) {
        fprintf(stderr, "Bad transfersize value\n");
        fprintf(stderr, "    Default value = %d\n", DEFAULT_TRANSFERSIZE);
        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 == 0) {
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else if (blocksize < 0) {
        fprintf(stderr, "Bad blocksize value\n");
        fprintf(stderr, "    Default value = %d\n", DEFAULT_BLOCKSIZE);
        exit(EXIT_FAILURE);
    }
}

/* Set supplied input filename or default if not supplied */
if (argc < 4) {
    if (strlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
    {
        fprintf(stderr, "Default input filename too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(inputFilename, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
}
else {
    if (!strcmp(argv[3], "default")) {

```

```

        strncpy(inputFilename, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else {
        if (strlen(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 (strlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >=
        MAXFILENAMELENGTH) {
        fprintf(stderr, "Default output filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
}
else {
    if (!strcmp(argv[4], "default")) {
        strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
            MAXFILENAMELENGTH);
    }
    else {
        if (strlen(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 filename 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 helpful info to make it seem like we were doing stuff
       */
    fprintf(stdout, "Read:      %zd bytes in %d reads\n",
            totalBytesRead, totalReads);
    fprintf(stdout, "Written: %zd bytes in %d writes\n",
            totalBytesWritten, totalWrites);
    fprintf(stdout, "Read input file in %d pass%s\n",
            (inputFileResets + 1), (inputFileResets ? "es" : ""));
    fprintf(stdout, "Processed %zd bytes in blocks of %zd bytes\n",
            transfersize, blocksize);

    /* Free Buffer */
    free(transferBuffer);

    /* Close Output File Descriptor */
    if (close(outputFD)) {
        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;
}

```

Listing 4: I/O Bound Benchmark Scheduler `rw-sched.c`

```

/*
 * File:      rw-sched.c
 * Author:    Stephen Bennett
 * Project:   CSCI 3753 Programming Assignment 3
 * Create Date: 2013/03/21
 * Modify Date: 2013/03/21
 * Description:
 *   A program for setting the desired scheduling policy and creating
 *   the desired number of child processes of rw.c.
 */

/* Local Includes */

```

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h> // Used for fork
#include <sys/wait.h> // Used for fork
#include <unistd.h> // Used for fork

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEFAULT_CHILDREN 10

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

    ssize_t transfersize = 0;
    ssize_t blocksize = 0;
    char    inputFilenameBase[MAXFILENAMELENGTH];
    char    inputFilename[MAXFILENAMELENGTH];
    char    outputFilenameBase[MAXFILENAMELENGTH];
    int     policy;
    int     children;

    struct sched_param param;
    int     i;
    int     rv;
    pid_t   pid;

    /* 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 == 0) {
            transfersize = DEFAULT_TRANSFERSIZE;
        }
        else if (transfersize < 0) {
            fprintf(stderr, "Bad transfersize value\n");
            fprintf(stderr, " Default value = %d\n", DEFAULT_TRANSFERSIZE);
            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 == 0) {
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else if (blocksize < 0) {
        fprintf(stderr, "Bad blocksize value\n");
        fprintf(stderr, "    Default value = %d\n", DEFAULT_BLOCKSIZE);
        exit(EXIT_FAILURE);
    }
}

/* Set supplied input filename or default if not supplied */
if (argc < 4) {
    if (strlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
    {
        fprintf(stderr, "Default input filename too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(inputFilenameBase, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
}
else {
    if (!strcmp(argv[3], "default")) {
        strncpy(inputFilenameBase, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else {
        if (strlen(argv[3], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
            fprintf(stderr, "Input filename too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(inputFilenameBase, argv[3], MAXFILENAMELENGTH);
    }
}

/* Set supplied output filename base or default if not supplied */
if (argc < 5) {
    if (strlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >=
        MAXFILENAMELENGTH) {
        fprintf(stderr, "Default output filename base too long\n");
        exit(EXIT_FAILURE);
    }
    strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
}
else {
    if (!strcmp(argv[4], "default")) {
        strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE,
            MAXFILENAMELENGTH);
    }
    else {
        if (strlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH) {
            fprintf(stderr, "Output filename base is too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
    }
}

```

```

}

/* Set policy or default if not supplied */
if (argc < 6) {
    policy = SCHED_OTHER;
}
else {
    if (!strcmp(argv[5], "SCHED_OTHER")) {
        policy = SCHED_OTHER;
    }
    else if (!strcmp(argv[5], "SCHED_FIFO")) {
        policy = SCHED_FIFO;
    }
    else if (!strcmp(argv[5], "SCHED_RR")) {
        policy = SCHED_RR;
    }
    else {
        fprintf(stderr, "Unhanded scheduling policy [%s]\n", argv[5]);
        fprintf(stderr, "Available scheduling policies:\n");
        fprintf(stderr, "    SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
        exit(EXIT_FAILURE);
    }
}

/* Set number of child processes to spawn or default if not supplied */
if (argc < 7) {
    children = DEFAULT_CHILDREN;
}
else {
    children = atol(argv[6]);
    if (children == 0) {
        /* Set default iterations */
        children = DEFAULT_CHILDREN;
    }
    else if (children < 0) {
        fprintf(stderr, "Bad children value [%d]\n", children);
        fprintf(stderr, "    0: Default children [%d]\n", DEFAULT_CHILDREN);
        fprintf(stderr, "    <0: Bad children value\n");
        fprintf(stderr, "    >0: Good children value\n");
        exit(EXIT_FAILURE);
    }
}

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

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

```

```

/* Fork children child processes */
for (i = 0; i < children; ++i) {
    rv = snprintf(inputFilename, MAXFILENAMELENGTH, "%s-%d",
                  inputFilenameBase, i+1);
    if (rv > MAXFILENAMELENGTH) {
        fprintf(stderr, "Output filename 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 ((pid = fork()) == -1) exit(EXIT_FAILURE);    /* Fork Failed */

    if (pid == 0) { /* Child process */
        // execl(exe, argv[0], argv[1], argv[2], ..., NULL)
        execl("rw", "rw", argv[1], argv[2], inputFilename, argv[4], NULL);
        exit(EXIT_SUCCESS);
    } else {      /* Parent process */
        printf("Forked %d pid = %d\n", i, pid);
    }
}

/* Wait for children child processes to finish */
for (i = 0; i < children; ++i) {
    pid = wait(NULL);
    printf("Waited %d pid = %d\n", i+1, pid);
}

return 0;
}

```

Listing 5: Mixed Benchmark mix.c

```

/*
 * File:          mix.c
 * Author:        Stephen Bennett
 * Project:       CSCI 3753 Programming Assignment 3
 * Create Date:   2013/03/21
 * Modify Date:   2013/03/21
 * Description:
 *   This file contains a small program for statistically calculating pi
 *   as well as writing the intermediate results of this calculation
 *   to a file.
 */

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

```

```

#include <unistd.h>          // getpid()
#include <errno.h>
#include <fcntl.h>           // open()
#include <string.h>
#include <math.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_OUTPUTFILENAMEBASE "mixoutput"
#define DEFAULT_ITERATIONS 10000
#define DEFAULT_INTERMEDIATERESULTS 100
#define RADIUS (RAND_MAX / 2)

/* 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[]) {

    /* Parameters */
    long intermediateResults;
    long iterations;
    char outputFilename[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];

    /* I/O Bound Variables */
    int rv;
    int outputFD;
    char writeBuffer[78];

    /* CPU Bound Variables */
    long i, j;
    double x, y;
    double inCircle = 0.0;
    double inSquare = 0.0;
    double pCircle = 0.0;
    double piCalc = 0.0;

    /* Process program arguments to run-time parameters */
    /* Set intermediate results or default if not supplied */
    if (argc < 2) {
        intermediateResults = DEFAULT_INTERMEDIATERESULTS;
    }
    else {
        intermediateResults = atol(argv[1]);
        if (intermediateResults== 0) {
            intermediateResults = DEFAULT_INTERMEDIATERESULTS;
        }
        else if (intermediateResults < 0) {

```

```

        fprintf(stderr, "Bad intermediate results value [%li]\n",
            intermediateResults);
        fprintf(stderr, "  0: Default intermediate results [%d]\n",
            DEFAULT_INTERMEDIATERESULTS);
        fprintf(stderr, " <0: Bad intermediate results value\n");
        fprintf(stderr, " >0: Good intermediate results value\n");
        exit(EXIT_FAILURE);
    }
}

/* Set iterations or default if not supplied */
if (argc < 3) {
    iterations = DEFAULT_ITERATIONS;
}
else {
    iterations = atol(argv[2]);
    if (iterations == 0) {
        iterations = DEFAULT_ITERATIONS;
    }
    else if (iterations < 0) {
        fprintf(stderr, "Bad iterations value [%li]\n", iterations);
        fprintf(stderr, "  0: Default iterations [%d]\n", DEFAULT_ITERATIONS);
        fprintf(stderr, " <0: Bad iterations value\n");
        fprintf(stderr, " >0: Good iterations value\n");
        exit(EXIT_FAILURE);
    }
}

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

/* 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 filename 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);
}

for (j = 0; j < intermediateResults; ++j) {
    /* Calculate pi using statistical method 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;

    snprintf(writeBuffer, 78, "In Circle Count = %-58f\n", inCircle);
    write(outputFD, writeBuffer, 77);
    snprintf(writeBuffer, 78, "In Square Count = %-58f\n", inSquare);
    write(outputFD, writeBuffer, 77);
    snprintf(writeBuffer, 78, "Calculated pi = %-58f\n\n", piCalc);
    write(outputFD, writeBuffer, 77);
}

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

return EXIT_SUCCESS;
}

```

Listing 6: Mixed Benchmark Scheduler mix-sched.c

```

/*
 * File:          mix-sched.c

```

```

* Author:      Stephen Bennett
* Project:     CSCI 3753 Programming Assignment 3
* Create Date: 2013/03/21
* Modify Date: 2013/03/21
* Description:
*   A program for setting the desired scheduling policy and creating
*   the desired number of child processes of mix.c.
*/

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <sys/types.h> // Used for fork
#include <sys/wait.h>  // Used for fork
#include <unistd.h>     // Used for fork

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100
#define DEFAULT_CHILDREN 10

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

    /* Parameters */
    int policy;
    int children;

    /* Scheduler/Fork Variables */
    struct sched_param param;
    int i;
    pid_t pid;

    /* Process program arguments to select run-time parameters */
    /* Set policy or default if not supplied */
    if (argc < 5) {
        policy = SCHED_OTHER;
    }
    else {
        if (!strcmp(argv[4], "SCHED_OTHER")) {
            policy = SCHED_OTHER;
        }
        else if (!strcmp(argv[4], "SCHED_FIFO")) {
            policy = SCHED_FIFO;
        }
        else if (!strcmp(argv[4], "SCHED_RR")) {
            policy = SCHED_RR;
        }
        else {

```

```

        fprintf(stderr, "Unhanded scheduling policy [%s]\n", argv[4]);
        fprintf(stderr, "Available scheduling policies:\n");
        fprintf(stderr, "    SCHED_OTHER, SCHED_FIFO, SCHED_RR\n");
        exit(EXIT_FAILURE);
    }
}

/* Set number of child processes to spawn or default if not supplied */
if (argc < 6) {
    children = DEFAULT_CHILDREN;
}
else {
    children = atol(argv[5]);
    if (children == 0) {
        /* Set default iterations */
        children = DEFAULT_CHILDREN;
    }
    else if (children < 0) {
        fprintf(stderr, "Bad children value [%d]\n", children);
        fprintf(stderr, "    0: Default children [%d]\n", DEFAULT_CHILDREN);
        fprintf(stderr, "    <0: Bad children value\n");
        fprintf(stderr, "    >0: Good children value\n");
        exit(EXIT_FAILURE);
    }
}

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

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

/* Fork children child processes */
for (i = 0; i < children; ++i) {
    if ((pid = fork()) == -1) exit(EXIT_FAILURE);    /* Fork Failed */

    if (pid == 0) { /* Child process */
        execv("mix", argv);
        exit(EXIT_SUCCESS);
    } else {
        /* Parent process */
        printf("Forked %d pid = %d\n", i, pid);
    }
}

/* Wait for children child processes to finish */
for (i = 0; i < children; ++i) {
    pid = wait(NULL);
    printf("Waited %d pid = %d\n", i, pid);
}

```

```

    }

    return 0;
}

```

Listing 7: 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)

EXECUTABLES = pi pi-sched rw rw-sched mix mix-sched rr_quantum
OBJECTS = $(EXECUTABLES:%=%.o)
#OBJECTS = $(foreach exe,$(EXECUTABLES),$(exe).o)

.PHONY: all clean

all: $(EXECUTABLES)

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

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

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

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

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

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

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

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

%.o: %.c

```

```
$(CC) $(CFLAGS) $<

clean: testclean
    rm -f $(EXECUTABLES)
    rm -f *.o
    rm -f *~
    rm -f handout/*~
    rm -f handout/*.log
    rm -f handout/*.aux

testclean:
    rm -f mixoutput*
    rm -f rwoutput*
    rm -f rwinput*
```