|  |  |
| --- | --- |
|  | LINUX scheduler Analysis |
|  |  |
| 11/8/2015 | Comparison of CFS, FIFO, and RR by Daniel Maxson |
|  | The CPU scheduler is an important part of a computer. How do we determine the best one? Which is best for a given situation? What offers the best performance? What are we optimizing for? This paper will attempt to shed some light on these Linux scheduling algorithms. |

LINUX scheduler Analysis

Comparison of CFS, FIFO, and RR by Daniel Maxson

# Introduction.

Operating systems include many parts and pieces that are not always know or even used. I had no idea that Linux included more than one scheduling algorithm. This report will bring to light the differences and intricacies of three of the available Linux Scheduling algorithms.

The Default scheduler is Completely Fair Scheduler (CFS) and is uses two classes, real-time and default scheduling, to determine priority for scheduling processes.

First in first out (FIFO) acts like a que and the first process that is available to process gets to run to completion.

Round Robin (RR) time slices all active processes and allocates equal time segments to each.

# Method.

I was tasked with analysis of these algorithms during specific CPU loads using a CPU bound program, an I/O bound program, and a mixed combination of the two.

The necessary CPU loads were achieved using a Linux program called stress. It allows you to choose a number of workers to send to the CPU that can use CPU time as CPU bound and spinning on square root calculations, and/or spend time requesting I/O. This allowed me to emulate the required CPU loading and simulated different environments.

For the CPU bound program I used a pi calculator that was modified to fork which allowed for a large aggregate of data to be average using the Linux time command.

For the I/O bound program a read write program was used. The program took two arguments and allowed the size of the write to be used specified as the first argument and then the second was the block size. I used a multiple of 4096b block size, 8192000kb or about 7.8mb file. The test system is running RAID 0 ssd’s so this large file size was necessary to have read write times in the second range that could be comparable to normal computer read write.

Included in the documentation is a truncated CPU-Z report of my test system specifications, but I will summarize them here.

The analysis was performed on an AMD FX-8350 8-core CPU running at 4.4GHz with 16GB of ram. The I/O was performed on two Intel SSD’s in a RAID 0 configuration.

The results of the programs time on CPU were recorded using the Linux time command with the –v flag.

# Results & Analysis.

The results from the Linux time command provide valuable accurate CPU timing metrics. This data has been used to create the following graphs and they will aid in our analysis and subsequent conclusions about the separate schedulers.

The first metric is the total active time for each process.

This shows a maximum difference of 0.32 seconds between all schedulers. This was also not during full CPU load as I had expected. I hypothesized that because there would be full process queues and the CPU would not be idle the programs would be active the longest during full load.

I believe that this data is showing a single processor scheduling under different loads as the processor affinity may have held throughout the 0% to 100% loading. I am under the impression that forking a program and all the inheritance that come from the parent including processor affinity is passed to the child. This does not guarantee we are testing SMP scheduling in a multi-processor environment.

This data suggests that CFS is in fact an optimal scheduler for single processor or multi-processor environments. This difference between RR and CFS at 0% load is only 0.1s. One tenth of a second could be an error attributed to the context switching from idle processes. This context switch error seems to be eliminated as the processors load increases, at 50% load the difference between RR and CFS is 0.02s. A small difference which favors CFS at higher loads. Then at 100% load there is an improvement in CFS over FIFO and RR, with CFS beating RR by 0.16s, and FIFO by 0.25s.

But this graph is showing total time and we have data for user time and kernel time to refine our conclusions about schedulers.

Our user time Graph shows a similar 0.1s difference between CFS and RR at 0% load but doubles the gap of 0.2s to 0.4s at 50% load. An improvement of 0.5s at 100% load hints that the CFS scheduler is allowing our process more process time and less total time spent active.

The Kernel time data provides an interesting look at how the schedulers operate. CFS here at 100% load has spent an impressive amount of time on the CPU which doesn’t seem optimal. CFS has our processes running 3.5X longer than RR. But this makes sense when we think about how CFS operates. Preemption and higher priority processes explains our extended CPU time. I’ll explain my reasoning in a later graph detailing voluntary and involuntary context switching. Our kernel time graph hints at RR being optimal for providing processor time. But the number also suggest that the time increments are too small to be accurate.

The unusual linear distribution of the 50% load shows that even at hundredths of a second our data is blocky and we see a 0.01s difference as a large amount. In my opinion more granularity is needed and data accuracy should be in the thousandths to millionths or even billionths of a second. Our processors are running in the GHz range and the data cannot accurately model small differences, less than one hundredth of one second, which could be poised on rounding error divides or actually be a hundredth of a second difference. In CPU time those are large differences in time.

This context switch graph show the voluntary and involuntary context switches each scheduler allowed during the different program execution. The voluntary context switching for the I/O bound program at the top shows us that going out to the hard drive takes time and all the schedulers fairly and morally give up the CPU when they do not need it. RR has a bit of an issue because it has to wait for a time slice to inform the CPU it can be taken off the processor and put into a wait queue.

# Conclusion.

Taking all of our results into consideration we can make a few conclusions about CPU bound, I/O, and mixed scheduling optimizations.

CPU bound – RR seems to be the optimal solution if your focus is solely on heavy computation. This scheduler might be most useful for scientific calculation.

I/O bound – FIFO notably seemed to spend less time on the CPU. I would surmise that this is because it has the time to initiate its process and tell the CPU its waiting on I/O before being switched off. RR and FIFO are close winners. CFS notably lost because of the priority and I/O being low priority.

Mixed – CFS and RR could both be optimal, but again the data granularity prevents a clearly optimal scheduler from showing its marks. FIFO falls behind at higher CPU loads.

During my experimentation and data collection phase I had an inkling that this programming assignment would give an interesting insight to how and why schedulers were important, but I also noticed from the data points being very close that if I were to continue this analysis in a scholarly setting my data would need to gain significant granularity and extended time testing. Ten to twenty minutes gave good data for the behavior of the schedulers but the inaccurate timing provided by default by the Linux Time program quickly became a large source of error and frustration.

Final thoughts - This programming assignment and the Operating Systems class in general have provided interesting and useful knowledge for my future career hopefully in industry.

Appendix A:

Raw data

Please see the included excel sheet for an understanding of the included graphs.

Command being timed: "./rw 8192000 4096"

User time (seconds): 0.01

System time (seconds): 2.92

Percent of CPU this job got: 12%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:23.78

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1424

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 71

Voluntary context switches: 9224

Involuntary context switches: 1097

Swaps: 0

File system inputs: 0

File system outputs: 148160

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./rw 8192000 4096"

User time (seconds): 0.00

System time (seconds): 1.69

Percent of CPU this job got: 23%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:07.24

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1424

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 72

Voluntary context switches: 8487

Involuntary context switches: 449

Swaps: 0

File system inputs: 0

File system outputs: 144800

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./rw 8192000 4096"

User time (seconds): 0.00

System time (seconds): 2.71

Percent of CPU this job got: 53%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:05.10

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1420

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 71

Voluntary context switches: 8015

Involuntary context switches: 98

Swaps: 0

File system inputs: 0

File system outputs: 319904

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 11.74

System time (seconds): 0.01

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.79

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1480

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 650

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./rw 16384000 4096"

User time (seconds): 0.00

System time (seconds): 3.99

Percent of CPU this job got: 48%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:08.30

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1532

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 72

Voluntary context switches: 15998

Involuntary context switches: 378

Swaps: 0

File system inputs: 0

File system outputs: 662880

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 12.09

System time (seconds): 0.02

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:12.14

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1480

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 652

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./rw 16384000 4096"

User time (seconds): 0.00

System time (seconds): 4.57

Percent of CPU this job got: 10%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:42.53

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1528

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 73

Voluntary context switches: 17805

Involuntary context switches: 1884

Swaps: 0

File system inputs: 0

File system outputs: 292416

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 12.00

System time (seconds): 0.01

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:12.05

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1560

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 682

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./rw 16384000 4096"

User time (seconds): 0.00

System time (seconds): 4.59

Percent of CPU this job got: 14%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:31.41

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1528

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 72

Voluntary context switches: 17939

Involuntary context switches: 1851

Swaps: 0

File system inputs: 0

File system outputs: 294048

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_OTHER 15"

User time (seconds): 11.29

System time (seconds): 0.06

Percent of CPU this job got: 97%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.66

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1560

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 683

Voluntary context switches: 31

Involuntary context switches: 295

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_FIFO 15"

User time (seconds): 11.58

System time (seconds): 0.03

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.65

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1480

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 666

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 11.50

System time (seconds): 0.02

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.57

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1484

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 650

Voluntary context switches: 31

Involuntary context switches: 6

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_OTHER 15"

User time (seconds): 11.26

System time (seconds): 0.03

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.38

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1556

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 668

Voluntary context switches: 31

Involuntary context switches: 103

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_FIFO 15"

User time (seconds): 11.62

System time (seconds): 0.02

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.67

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1484

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 668

Voluntary context switches: 31

Involuntary context switches: 3

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 11.33

System time (seconds): 0.01

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.38

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1560

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 683

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_OTHER 15"

User time (seconds): 11.00

System time (seconds): 0.01

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.06

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1484

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 650

Voluntary context switches: 31

Involuntary context switches: 48

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_FIFO 15"

User time (seconds): 11.02

System time (seconds): 0.02

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.08

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1480

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 666

Voluntary context switches: 31

Involuntary context switches: 7

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Command being timed: "./pi-sched\_fork 10000000 SCHED\_RR 15"

User time (seconds): 10.90

System time (seconds): 0.02

Percent of CPU this job got: 99%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:10.97

Average shared text size (kbytes): 0

Average unshared data size (kbytes): 0

Average stack size (kbytes): 0

Average total size (kbytes): 0

Maximum resident set size (kbytes): 1484

Average resident set size (kbytes): 0

Major (requiring I/O) page faults: 0

Minor (reclaiming a frame) page faults: 665

Voluntary context switches: 31

Involuntary context switches: 6

Swaps: 0

File system inputs: 0

File system outputs: 0

Socket messages sent: 0

Socket messages received: 0

Signals delivered: 0

Page size (bytes): 4096

Exit status: 0

Appendix B:

Code

/\*

\* File: pi-sched.c

\* Author: Andy Sayler

\* Project: CSCI 3753 Programming Assignment 3

\* Create Date: 2012/03/07

\* Modify Date: 2012/03/09

\* Description:

\* This file contains a simple program for statistically

\* calculating pi using a specific scheduling policy.

\*/

/\* Local Includes \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <math.h>

#include <errno.h>

#include <sched.h>

#include <sys/wait.h>

#include <unistd.h>

#define DEFAULT\_ITERATIONS 1000000

#define RADIUS (RAND\_MAX / 2)

int policy;

struct sched\_param param;

long iterations;

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 piFunction(){

long i;

double x, y;

double inCircle = 0.0;

double inSquare = 0.0;

double pCircle = 0.0;

double piCalc = 0.0;

//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;

}

// Exec multiple of piFunction for testing

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

if (argc != 4){

fprintf(stderr, "Incorrect number of args given");

return 1;

}

/\* 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);

}

long processes = atol(argv[3]);

int i = 0;

pid\_t pid, w\_pid;

int status;

for ( i= 0 ; i < processes ; i++ ) {

if((pid = fork()) == 0){

piFunction();

exit(0);

}

while((w\_pid = waitpid(-1, &status, 0) > 0)){

if(!WIFEXITED(status)){

printf("Child terminated abnormally");

}

}

}

return 0;

}

/\*

\* File: rw.c

\* Author: Andy Sayler

\* Project: CSCI 3753 Programming Assignment 3

\* Create Date: 2012/03/19

\* Modify Date: 2012/03/20

\* Description: A small i/o bound program to copy N bytes from an input

\* file to an output file. May read the input file multiple

\* times if N is larger than the size of the input file.

\*/

/\* Include Flags \*/

#define \_GNU\_SOURCE

/\* System Includes \*/

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sched.h>

#include <sys/wait.h>

#include <math.h>

/\* Local Defines \*/

#define MAXFILENAMELENGTH 80

#define DEFAULT\_OUTPUTFILENAMEBASE "rwoutput"

#define DEFAULT\_BLOCKSIZE 1024

#define DEFAULT\_TRANSFERSIZE 1024\*100

#define DEFAULT\_ITERATIONS 1000000

#define RADIUS (RAND\_MAX / 2)

int policy;

long iterations;

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 piFunction(){

long i;

double x, y;

double inCircle = 0.0;

double inSquare = 0.0;

double pCircle = 0.0;

double piCalc = 0.0;

//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;

}

int rwFunction(char inFN[MAXFILENAMELENGTH], char outFN[MAXFILENAMELENGTH]){

int rv;

int inputFD;

int outputFD;

char inputFilename[MAXFILENAMELENGTH];

char outputFilename[MAXFILENAMELENGTH];

char outputFilenameBase[MAXFILENAMELENGTH];

ssize\_t transfersize = 1024\*100;

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 \*/

transfersize = DEFAULT\_TRANSFERSIZE;

/\* Set supplied block size or default if not supplied \*/

blocksize = DEFAULT\_BLOCKSIZE;

/\* Set supplied input filename or default if not supplied \*/

strncpy(inputFilename, inFN, MAXFILENAMELENGTH);

/\* Set supplied output filename base or default if not supplied \*/

strncpy(outputFilenameBase, outFN, 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",

outputFilenameBase);

if(rv > MAXFILENAMELENGTH){

fprintf(stderr, "Output filenmae length exceeds limit of %d characters.\n",

MAXFILENAMELENGTH);

exit(EXIT\_FAILURE);

}

else if(rv < 0){

perror("Failed to generate output filename");

exit(EXIT\_FAILURE);

}

if((outputFD =

open(outputFilename,

O\_WRONLY | O\_CREAT | O\_TRUNC | O\_SYNC,

S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH)) < 0){

perror("Failed to open output file");

exit(EXIT\_FAILURE);

}

/\* Print Status \*/

fprintf(stdout, "Reading from %s and writing to %s\n",

inputFilename, outputFilename);

/\* Read from input file and write to output file\*/

do{

/\* Read transfersize bytes from input file\*/

bytesRead = read(inputFD, transferBuffer, buffersize);

if(bytesRead < 0){

perror("Error reading input file");

exit(EXIT\_FAILURE);

}

else{

totalBytesRead += bytesRead;

totalReads++;

}

/\* If all bytes were read, write to output file\*/

if(bytesRead == blocksize){

bytesWritten = write(outputFD, transferBuffer, bytesRead);

if(bytesWritten < 0){

perror("Error writing output file");

exit(EXIT\_FAILURE);

}

else{

totalBytesWritten += bytesWritten;

totalWrites++;

}

}

/\* Otherwise assume we have reached the end of the input file and reset \*/

else{

if(lseek(inputFD, 0, SEEK\_SET)){

perror("Error resetting to beginning of file");

exit(EXIT\_FAILURE);

}

inputFileResets++;

}

}while(totalBytesWritten < transfersize);

/\* Output some possibly helpfull info to make it seem like we were doing stuff \*/

fprintf(stdout, "Read: %zd bytes in %d reads\n",

totalBytesRead, totalReads);

fprintf(stdout, "Written: %zd bytes in %d writes\n",

totalBytesWritten, totalWrites);

fprintf(stdout, "Read input file in %d pass%s\n",

(inputFileResets + 1), (inputFileResets ? "es" : ""));

fprintf(stdout, "Processed %zd bytes in blocks of %zd bytes\n",

transfersize, blocksize);

/\* Free Buffer \*/

free(transferBuffer);

/\* Close Output File Descriptor \*/

if(close(outputFD)){

perror("Failed to close output file");

exit(EXIT\_FAILURE);

}

/\* Close Input File Descriptor \*/

if(close(inputFD)){

perror("Failed to close input file");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

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

if(argc != 4){

fprintf(stderr, "Incorrect input arguments");

}

int policy;

struct sched\_param param;

long processes;

/\* Set policy if supplied \*/

if(!strcmp(argv[1], "SCHED\_OTHER")){

policy = SCHED\_OTHER;

}

else if(!strcmp(argv[1], "SCHED\_FIFO")){

policy = SCHED\_FIFO;

}

else if(!strcmp(argv[1], "SCHED\_RR")){

policy = SCHED\_RR;

}

else{

fprintf(stderr, "Unhandeled scheduling policy\n");

exit(EXIT\_FAILURE);

}

processes = atol(argv[2]);

/\* 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);

}

iterations = atol(argv[3]);

/\* Run with different files \*/

char outputFNArray[processes][MAXFILENAMELENGTH];

char inputFNArray[processes][MAXFILENAMELENGTH];

int i, status;

pid\_t pid, w\_pid;

for (i = 0 ; i < processes ; i++){

sprintf(inputFNArray[i],"input/InputFile%d.txt",i);

}

for (i = 0 ; i < processes ; i++){

sprintf(outputFNArray[i],"output/OutputFile%d.txt",i);

}

/\* Run processes \*/

for (i = 0 ; i < processes ; i++){

// Fork it

if ((pid = fork()) == 0){

piFunction();

rwFunction(inputFNArray[i], outputFNArray[i]);

exit(0);

}

while((w\_pid = waitpid(-1, &status, 0)) > 0){

if(!WIFEXITED(status)){

printf("child failed to terminate");

}

}

}

return 0;

}

/\*

\* File: rw.c

\* Author: Andy Sayler

\* Project: CSCI 3753 Programming Assignment 3

\* Create Date: 2012/03/19

\* Modify Date: 2012/03/20

\* Description: A small i/o bound program to copy N bytes from an input

\* file to an output file. May read the input file multiple

\* times if N is larger than the size of the input file.

\*/

/\* Include Flags \*/

#define \_GNU\_SOURCE

/\* System Includes \*/

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <fcntl.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sched.h>

#include <sys/wait.h>

/\* Local Defines \*/

#define MAXFILENAMELENGTH 80

#define DEFAULT\_INPUTFILENAME "rwinput"

#define DEFAULT\_OUTPUTFILENAMEBASE "rwoutput"

#define DEFAULT\_BLOCKSIZE 1024

#define DEFAULT\_TRANSFERSIZE 1024\*100

int rwFunction(){

int rv;

int inputFD;

int outputFD;

char inputFilename[MAXFILENAMELENGTH];

char outputFilename[MAXFILENAMELENGTH];

char outputFilenameBase[MAXFILENAMELENGTH];

ssize\_t transfersize = 1024\*100;

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 \*/

transfersize = DEFAULT\_TRANSFERSIZE;

/\* Set supplied block size or default if not supplied \*/

blocksize = DEFAULT\_BLOCKSIZE;

/\* 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",

outputFilenameBase);

if(rv > MAXFILENAMELENGTH){

fprintf(stderr, "Output filenmae length exceeds limit of %d characters.\n",

MAXFILENAMELENGTH);

exit(EXIT\_FAILURE);

}

else if(rv < 0){

perror("Failed to generate output filename");

exit(EXIT\_FAILURE);

}

if((outputFD =

open(outputFilename,

O\_WRONLY | O\_CREAT | O\_TRUNC | O\_SYNC,

S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH)) < 0){

perror("Failed to open output file");

exit(EXIT\_FAILURE);

}

/\* Print Status \*/

fprintf(stdout, "Reading from %s and writing to %s\n",

inputFilename, outputFilename);

/\* Read from input file and write to output file\*/

do{

/\* Read transfersize bytes from input file\*/

bytesRead = read(inputFD, transferBuffer, buffersize);

if(bytesRead < 0){

perror("Error reading input file");

exit(EXIT\_FAILURE);

}

else{

totalBytesRead += bytesRead;

totalReads++;

}

/\* If all bytes were read, write to output file\*/

if(bytesRead == blocksize){

bytesWritten = write(outputFD, transferBuffer, bytesRead);

if(bytesWritten < 0){

perror("Error writing output file");

exit(EXIT\_FAILURE);

}

else{

totalBytesWritten += bytesWritten;

totalWrites++;

}

}

/\* Otherwise assume we have reached the end of the input file and reset \*/

else{

if(lseek(inputFD, 0, SEEK\_SET)){

perror("Error resetting to beginning of file");

exit(EXIT\_FAILURE);

}

inputFileResets++;

}

}while(totalBytesWritten < transfersize);

/\* Output some possibly helpfull info to make it seem like we were doing stuff \*/

fprintf(stdout, "Read: %zd bytes in %d reads\n",

totalBytesRead, totalReads);

fprintf(stdout, "Written: %zd bytes in %d writes\n",

totalBytesWritten, totalWrites);

fprintf(stdout, "Read input file in %d pass%s\n",

(inputFileResets + 1), (inputFileResets ? "es" : ""));

fprintf(stdout, "Processed %zd bytes in blocks of %zd bytes\n",

transfersize, blocksize);

/\* Free Buffer \*/

free(transferBuffer);

/\* Close Output File Descriptor \*/

if(close(outputFD)){

perror("Failed to close output file");

exit(EXIT\_FAILURE);

}

/\* Close Input File Descriptor \*/

if(close(inputFD)){

perror("Failed to close input file");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

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

if(argc != 3){

fprintf(stderr, "Incorrect input arguments");

}

int processes = 5;

/\* Run with different files \*/

char outputFNArray[processes][MAXFILENAMELENGTH];

char inputFNArray[processes][MAXFILENAMELENGTH];

int i, status;

pid\_t pid, w\_pid;

for (i = 0 ; i < processes ; i++){

sprintf(inputFNArray[i],"input/InputFile%d.txt",i);

}

for (i = 0 ; i < processes ; i++){

sprintf(outputFNArray[i],"output/OutputFile%d.txt",i);

}

/\* Run processes \*/

for (i = 0 ; i < processes ; i++){

// Fork it

if ((pid = fork()) == 0){

rwFunction(inputFNArray[i], outputFNArray[i]);

exit(0);

}

while((w\_pid = waitpid(-1, &status, 0)) > 0){

if(!WIFEXITED(status)){

printf("child failed to terminate");

}

}

}

return 0;

}