**Programming Assignment 4: Investigating the Linux Scheduler**

CSCI 3753 - Operating Systems

University of Colorado at Boulder

Fall 2015 Due Date: Sunday, November 8, 2015 11:55 PM

**Author: Juan Carlos Herrera**

**CUID: 102963035**

Table of Contents

[Abstract 3](#_Toc434785546)

[Introduction 3](#_Toc434785547)

[Method 4](#_Toc434785548)

[Results 5](#_Toc434785549)

[Analysis 10](#_Toc434785550)

[Conclusion 10](#_Toc434785551)

[Appendix A: Raw Data 11](#_Toc434785552)

[Appendix B: Code 13](#_Toc434785553)

# Abstract

Computers today have multiple-cores which means that Operating Systems are stuck with the task of scheduling processes accordingly. The scheduling process can be very complex so OS’s have been adapting to new more efficient scheduling methods. In this report I will examine different scheduler policies in a variety of different conditions. This will allow me to find the Linux scheduler that is the most efficient under each condition.

## Introduction

Schedulers

For this assignment the scheduler policies I will be testing are:

1. Non-preemptive (First-in, First-out) – SCHED\_FIFO
2. Round Robin – SCHED\_RR
3. Completely-fair-scheduler – SCHED\_OTHER

These policies will be tested for performance as well as their ability to schedule jobs under certain conditions such as different level of I/O and CPU processes.

Tests

There are three different programs that are used to test this. The first program is used to test a CPU-bound task so this program will be calculating Pi (π) a certain **n** amount of times. The second program will perform I/O bounds tasks by reading from an input file and writing the same contents into another file. Finally, the third program will create a combination of both of these techniques and test.

In order to make this test realistic I am testing each program with a number **n** arguments that sets up the independent variables for the test. There will also be a fixed number of child processes that are spawned via the fork call and there will be different workloads ranging from 1-128 (this will emulate different difficulties in the system). I will also emulate a 1 core system vs a 4 core system so we can see if the schedulers have a different effect.

The results showed in this paper are from a 2011 Mac Pro. Technical specifications below.

# Method

In order to make this test as realistic as possible I will be measuring a several dependent variables with some tests (explained below).

* CPU
  + The percentage of the CPU that the job used.

Calculated as:

* System
  + The total CPU time spent processing this program in kernel mode bit.
* User
  + The total CPU time spend processing this program in user mode bit.
* Wall
  + The total time for every program to execute.
* Wait-Time
  + The total CPU time that was not scheduled.

Calculated as:

* V-Switched
  + The total times the program context switched (voluntarily) – I/O bound
* I-Switched
  + The total times the program context switched (involuntarily) – Swapped out

In order to run theses tests I used a bash script which iterates through all specified conditions and displays results.

# Results

These test results are the output from the testscript that was executed using the bash script.

The computer used in these tests is a 2011 Mac Pro running under Virtual Box with the following specifications:

* VM: CU-CS-VM from 2015
* CPU: Intel Xeon CPU W3530 @ 2.80 Ghz (The VM has 4 cores allocated)
* Samsung SSD

**1-Core System:**

🡪 For this 1 Core system we can see that the best scheduler is the First-come, First-serve (SCHED\_FIFO) as can be expected. We expect this because FIFO runs until a process yields or completes and all of the processes are finite.

**4-Core System:**

# Analysis

**1-Core System:**

🡪 For this 1 Core system we can see that the best scheduler is the First-come, First-serve (SCHED\_FIFO) as can be expected. We expect this because FIFO runs until a process yields or completes and all of the processes are finite. Something to note is that the Round-Robin scheduler (SCHED\_RR) switches out a process after its quanta has elapsed so there is more overhead. Another thing to note is the SCHED\_OTHER which has a many involuntary switches.

**4-Core System:**

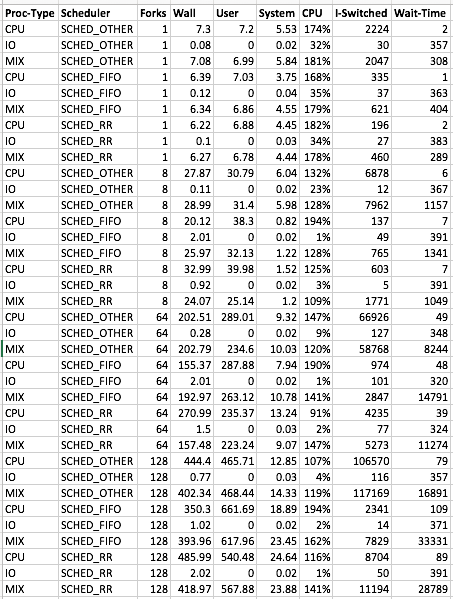
🡪 The 4 core system is very similar to the 1 core system, though much faster. We would obviously expect the system to be faster but we have to account for possible errors which seem somewhat unapparent on this finite system.

# Conclusion

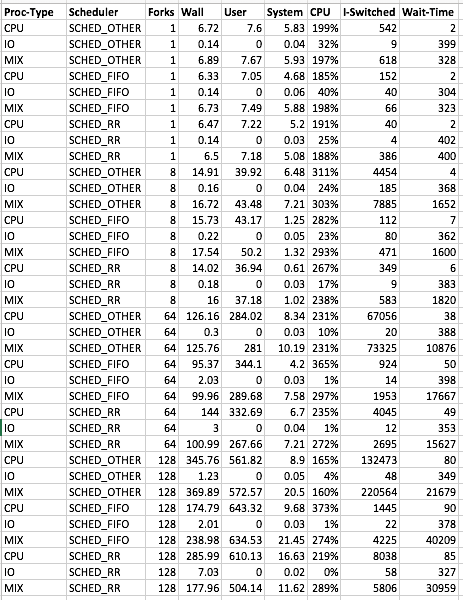
After running multiple tests on two different core systems we can conclude that there are many conditions and properties that can arise and different schedulers will suit different needs. For this case the optimal scheduler is the First-come, First-serve (FIFO) since it is optimizing for finite CPU-bound processes. When dealing with longer processes this may not be the optimal scheduler since it may lead to starvation or other possible errors. We would not expect the Round-Robin scheduler to be the best choice though for this case it proved to be very efficient with time performance. Finally the SCHED\_OTHER is the best option since it provides a balance between speed and fairness. This is not the fastest scheduler but it is better than FIFO since it also provides voluntary switches which will allow resources within processes.

# Appendix A: Raw Data

**1-Core System Raw Data:**



**4-Core System Raw Data:**



# Appendix B: Code

#!/bin/bash

# File: testscript

# Author: Juan Carlos Herrera

# Project: CSCI 3753 Programming Assignment 4

# Create Date: 5/11/15

# Description:

# A simple bash script to run a signle copy of each test case

# and gather the relevent data.

ITERATIONS=100000000

BYTESTOCOPY=102400

BLOCKSIZE=1024

# APPENDIX

# e - elapsed real time in seconds

# u - Total number of CPU-seconds that the process spent in user mode.

# S - seconds spent in kernel mode,

# P - Percentage of the CPU that this job got, computed as (%U + %S) /E,

# c - number of times the program context sitched voluntarily,

# w - Number of waits: times that the program was context-switched voluntarily.

#WaitTime: Realtime - (System time + user time) -> e - (u+s)

#TIMEFORMAT="wall=%e,user=%U,system=%S,CPU=%P,i-switched=%c,v-switched=%w"

TIMEFORMAT="%e,%U,%S,%P,%c,%w"

MAKE="make -s"

echo Building code...

$MAKE clean

$MAKE

COUNT=0

SCHED=SCHED\_RR

ARR=(cpu io mix)

#Loop through Fork COUNT

for i in `seq 1 4`;

do

case $i in

1) #Single

COUNT=1

;;

2) #Low

COUNT=8

;;

3) #Medium

COUNT=64

;;

\*) #High

COUNT=128

;;

esac

#Loop through SCHEDuler type

for j in `seq 1 3`;

do

case $j in

1) #OTHER

SCHED=SCHED\_OTHER

;;

2) #FIFO

SCHED=SCHED\_FIFO

;;

\*) #RR

SCHED=SCHED\_RR

;;

esac

#Loop through process type

#CPU bound

echo CPU,$SCHED,$COUNT

/usr/bin/time -f "$TIMEFORMAT" ./pi\_fork $ITERATIONS $SCHED $COUNT > /dev/null

#I/O bound

echo IO,$SCHED,$COUNT

/usr/bin/time -f "$TIMEFORMAT" ./rw\_fork $BYTESTOCOPY $BLOCKSIZE rwinput rwoutput $SCHED $COUNT > /dev/null

#BOTH

echo MIX,$SCHED,$COUNT

/usr/bin/time -f "$TIMEFORMAT" ./mix\_fork $BYTESTOCOPY $BLOCKSIZE rwinput rwoutput $ITERATIONS $SCHED $COUNT > /dev/null

done

done

rm -rf rwoutput-\*

rm -rf rwinput\*