Honors Program

Andrew Brand

Operating System Scheduling:
Lynux Preemptive Scheduling Algorithms

Computer Science
Mathematics
26 April 2013

Operating System Scheduling:

Linux Preemptive Scheduling Algorithms

Andrew Brand

Honors Program

March 31, 2013

Table of Contents

Abstract

In this study the effectiveness of three preemptive scheduling algorithms found in the Linux operating system was tested. The algorithms were tested using three types of software that college students most commonly use in their academic work and during their free time: a word processor, a web browser, and a media player. The data that are presented in this paper are based on an average of 100 trials for each different combination of the three applications, done for each of the three scheduling algorithms. These tests were run to determine which of the algorithms would work best for an operating system used primarily by a college student. The results indicate that the three applications perform the same regardless of what scheduling algorithm is being used.

Introduction

Operating system performance is an important topic of research. One of the main components of an operating system that performs well is the process scheduler. The scheduler is responsible for determining which processes should run on the processor and for how long. Scheduling is important because when a process is running on the processor, it is allocated resources by the operating system so it can complete its task effectively. These resources are important for all of the processes wanting to run, so it is important that the scheduler is coded so that the resources are allocated correctly, fairly, and efficiently. There are three main areas that the scheduler focuses on. The first is maximizing the number of processes that finish their tasks to completion, also known as throughput. The second is minimizing the amount of time it takes for said process to finish its task. The final important responsibility is making sure that each process is scheduled fairly. All three of these responsibilities are important to each other and to the operating system as a whole. Minimizing the time it takes for a process to finish allows more processes to finish their tasks, while the fairness of an algorithm makes sure that every process has time on the processor based on that process' importance and priority.

This may seem like an easy task, but maintaining the balance among fairness, throughput, and completion time is what makes developing scheduling algorithms difficult. Affecting one of the three areas, could lead to a decrease in efficiency in another, or lead to the entire scheduler to fail. Most of the time, a poorly designed algorithm leads to the errors of indefinite postponement or deadlock. These are two of the cardinal sins of operating system development. Deadlock occurs when a process waits for an event that will never occur while holding resources needed by other

processes. If the algorithm is not designed to handle this possibility, the process will never give up its resources on the processor, blocking other processes from using the processor. Indefinite postponement occurs when some processes keep getting processor time ahead of another process, forcing it to wait and not gain access to the processor. The poor prioritizing of processes usually cause this issue. If a process with a low priority is waiting to use the processor, processes with higher priority could keep being given processor time ahead of the low priority one. This effectively postpones that process from running its task to completion, sometimes indefinitely.

Many different scheduling algorithms have been developed not only to effectively combat deadlock and indefinite postponement, but also run efficiently on systems with different computing requirements. An operating system used for database work may need a different scheduler than an operating system in a college student's laptop. While there are many scheduling algorithms currently in use, the two most common categories of schedulers are preemptive or non-preemptive. Preemptive scheduling simply means that a process that is already running can be "kicked off" the processor to make way for another process. Non-preemptive is the opposite; a process remains on the processor until it finishes execution without interruptions (Deitel, Deitel, and Choffnes 333).

Review of Literature

Because an operating system scheduler an its performance are so important, numerous scholars have proposed various operating system features and tested them for other uses, as well. All of these variations are more complicated than this experiment, but they provide a solid basis for what has already been done and the variety of results that arise. It is not uncommon for these types of experiments to be inconclusive. They

may be inconclusive because affecting one portion of an OS could interfere another section of code. As stated before, developing a scheduling algorithm for an operating system is not a perfect art and no perfect solution has been developed for all scheduling problems. Some of these experiments had this problem, as well.

In *Preemptive online scheduling with reordering*, Dosa and Epstein created an experiment to test how preemptive scheduling algorithms worked on-line when the processes came in one-by-one. They hypothesized that a certain buffer size would help bring the processor time down to a certain threshold using preemptive scheduling. Their results showed that for a buffer of a certain size, the competition among processes was one, meaning that processes were not constantly getting preempted on the processor. This result showed that this algorithm successfully reduced the processing time by reducing the number of preemptions.

He, Zhang, and Zhou attempted to solve the problem of preemptive scheduling with two uniform machines in their article *Optimal preemptive online algorithms for scheduling with known largest size on two uniform machines*. For their experiment, He et al. assumed complete knowledge of the deadlines to finish each job, which is often helpful when scheduling. Using this knowledge, their goal was to maximize the throughput of both machines. One of the techniques they applied to their scheduling algorithm was randomization. Unfortunately, their experiment failed to produce the desired result, and they were unable to maximize throughput on both machines.

Dosa and He tested a problem similar to the one attempted by He, Zhang, and Zhou. The difference was that this experiment dealt with both preemptive and non-preemptive strategies, and introduced the idea of job rejection. Their experimental setup,

outlined in *Preemptive and non-preemptive on-line algorithms for scheduling with rejection on two uniform machines,* was also similar to that of the experiment conducted by He, Zhang, and Zhou. Their method was to start with an optimal scheduling solution and improve upon it if possible. They increased the lower bound of the non-preemptive strategy and came up with an overall better optimal solution for the preemptive scheduler.

Sevastyanov, Sitters, and Fishkin also looked at a preemptive scheduling problem in *Preemptive scheduling of independent jobs on identical parallel machines subject to migration delay.* Their goal was to minimize the makespan using preemptive scheduling on multiple parallel processors that allows migration of processes. (Makespan is a common term in operating system development. It simply means the time it takes for a process to finish completely from the time it initially started.) To carry out the experiment, they put a threshold on the number of process migrations for each trial. In doing this, they have developed an optimal solution for, at most, m-1 migrations where m is the number of parallel machines. Their results also yielded a linear algorithm for when there are two migrations.

Instead of looking at a preemptive scheduler, Paletta and Pietramala were interested in finding an optimal solution for using a non-preemptive scheduling algorithm for n processes on m different processors. In *A new approximation algorithm for the nonpreemptive scheduling of independent jobs on identical processors,* they analyzed the average-, worst-, and best-case scenarios and attempted to find a new algorithm to describe this behavior. They managed to find a new algorithm that runs in O(n log(n)+mn) time. In computer science, Big O notation is used to describe an upper bound on the worst-case performance of an algorithm. Generally, O(nlog(n)) translates to a

relatively good algorithm, in terms of performance. So, O(n log(n)+mn) is slightly worse than O(nlog(n). Paletta and Pietramala were surprised that they found an algorithm that performed that well.

The research group of Chekuri, Motwani, Natarajan, and Stein look at the problem of non-preemptive scheduling allowing release dates, parallel machines, and precedence restraints, all while trying to minimize completion time. The release date of a process is the time at which it is sent to a processor. They consulted other experiments to help them come up with some algorithms to test their schedulers. Their results showed that they had an optimal on-line algorithm, and their algorithm is useful for parallel machines. These results were published in *Approximation techniques for average completion time scheduling.*

Ebenlendr and Sgall presented a unified optimal algorithm for preemptive scheduling on uniformly related machines in their work, *Semi-online preemptive scheduling: one algorithm for all variants*. To back up the claim of an optimal algorithm, Ebenlendr and Sgall compared it to other algorithms that attacked the same problem. The results showed that their algorithm worked for all semi-online restrictions, adapting to create an optimal algorithm for any restrictions applied to it.

The problem studied in *Algorithms for the single machine total weighted completion time scheduling problem with release times and sequence-dependent setups* has not been researched as thoroughly as other scheduling problems. Chou, Wang, and Chang looked at the problem of scheduling n jobs on a single machine to minimize the completion time in the presence of sequence-dependent setup times and release times. This simply means that the order of the processes matters. They created two new

algorithms to evaluate the performance. The two algorithms performed in $O(n^4)$ and $O(n^3)$ time, which gave the researchers ideas as to what to improve upon in the next study. While O(nlog(n)) and O(log(n)) are good performance times for algorithms, any algorithm with performance times that exceed $O(n^2)$ needs to be looked at again and, if at all possible, rewritten to improve its performance. Sometimes this is not possible, but Chou, Wang, and Chang are hoping that they can achieve this in the future.

Reisi and Moslehi took a different approach to scheduling. In *Minimizing the number of tardy jobs and maximum earliness in the single machine scheduling using an artificial immune system,* the group attempted to find a scheduling algorithm that maximizes earliness in the processes and reduce the number of tardy ones. The uniqueness of their study stems from their use of data from an artificial immune system that was used in anatomy and biology research. The results show that this immune system algorithm is more favorable than other methods that have been used to solve this problem.

A problem found in scheduling is creating an algorithm that correctly determines if a task has performed correctly, or if it should be allowed to use the processor again to fix mistakes that it made. In *Approximation algorithms for multiprocessor scheduling under uncertainty*, Lin and Rajaraman looked at this very problem. They looked at certain cases to help minimize the run time and makespan. Their work resulted in an algorithm that ran in O(log(n)) time, but only for some of their specific cases. After all of the cases were tested, the algorithm performed, on average, in polynomial runtime.

In *Preemptive scheduling of two unrelated processors*, Gonzalez, Lawler, and Sahni tried to make an algorithm for preemptively scheduling n jobs on m unrelated

parallel processors. The results of this experiment are quite good. Gonzalez, Lawler, and Sahni present an algorithm that, on average, has only two preemptions for each trial. The lower the number of preemptions, the less time spent on changing processes. The algorithm also runs in linear time, which is a very good performance time.

In *Single machine scheduling to minimize total weighted late work*, Hariri, Potts, and Wassenhove attempted to use preemptive scheduling to schedule a single processor machine where each process has a "due date" and they must minimize the number of late processes. Their "due date" is like a deadline that the Deadline algorithm (discussed later) implements. Their research consisted of developing both non-preemptive and preemptive scheduling algorithms and comparing them. Their preemptive algorithm ran in O(nlog(n)) time. While their non-preemptive algorithm worked for some trials, an algorithm that performed well in all cases escaped the researchers.

Hoogeveen, Skutella, and Woeginger approached their research problem from a reverse order. They conjectured what sorts of problems they would encounter when the processor was allowed to reject jobs, and then developed their algorithm based on these issues. One of the issues discussed in *Preemptive scheduling with rejection* is the increase in processing time when the processor rejected the wrong jobs. The retrieval of the rejected jobs adds to the processing time. Taking this and other problems into account, the group developed an algorithm that ran in polynomial time.

There are many more articles written on the subject of operating system performance, and many more algorithms that have been developed in hopes of making operating systems more efficient. Like with the algorithms described in the articles, every algorithm has to undergo some sort of analysis to determine the its strengths and

weaknesses. Experiments are set up not only to create algorithms, but to test their performance within specific computing environments.

The purpose of this experiment is to statistically analyze the performance and efficiency of scheduling algorithms while using a word processor, a media player, and a web browser. These are the three most common applications found on a computer used for learning/academic use, especially in a college setting. Analyzing these algorithms could be important for to decide which one will run a college students laptop more efficiently.

Errors

The original proposal for this thesis project was to compare a preemptive scheduler against a non-preemptive scheduler running the three applications previously stated. The preemptive portion of the experiment was straightforward; Linux, the operating system used in this experiment, is already preemptive (it uses the NOOP scheduling algorithm described below). The challenging part of the experiment was writing and testing a non-preemptive scheduler. However, after a couple of failed attempts and a dwindling time frame, this portion of the experiment was scrapped in favor of analyzing the three preemptive algorithms. The reason the three preemptive algorithms worked better is because Linux's scheduler can easily be switched via a terminal command while the operating system is running. This approach allowed me to gather more data and evaluate the effectiveness of three preemptive algorithms: Deadline, Completely Fair Queuing (CFQ), and No Operation (NOOP).

The problems and obstacles that I ran into while attempting to code a non-preemptive scheduler are instructive. These problems add nothing to the overall

experiment at this point, but show how difficult operating system development is. In my first attempt, I opened and edited the source code files for the Linux scheduler, sched.c and sched.h. The initial plan to make the scheduler non-preemptive was to get rid of any time slices and deadlines that are imposed on the processes and only have the process leave the processor if it remained idle for a certain amount of time, since a non-preemptive scheduler cannot kick a process off the processor until the process itself yields its resources. After coding, I accidentally exited out of the source code, only to find that the algorithm had not worked at all and now Linux was not working properly. I could not get back into the source code to undo my mistakes, so I had to reinstall Linux on the computer.

Before launching into the next attempt, I decided to study the source code more thoroughly than I had previously. This lead me to discover that the code for the scheduling algorithms was not just in the two files I had looked at, but spread out through even more files in multiple directories. Changing code in the main scheduler file caused errors in other scheduler files located through Linux's directory system. This was the first and biggest obstacle that hindered me from completing the non-preemptive scheduler. Linux's directory structure and the interconnectedness of the files is very complicated. If I wrote code in a couple of files, it would have adverse affects elsewhere that I, even through patient research, could not find. This was very frustrating, because I could never be sure whether the errors were caused specifically by the algorithm, or because I had to change something in a file somewhere else before it could work.

From then on, every subsequent attempt to modify the scheduler was met with failure, mainly due to the obstacle just described, as well as issues with indefinite

postponement and deadlock. In the end, I ended up failing to make a non-preemptive scheduler, partially due to the errors and obstacles previously described, mostly due to my lack of knowledge and naiveté when it came operating systems. Perhaps after a few more years of operating system studies and learning C, I may be able to pursue my initial project.

Definitions

The replacement experiment developed was to analyze the performance of three preemptive algorithms. The Linux operating system has three preemptive scheduling algorithms already coded into the operating system: Deadline, CFQ, and NOOP. Having the schedulers already written by skilled programmers is advantageous to the analysis of their performance. Coding these algorithms myself might have affected the final analysis of the algorithms. There might have been errors in my program that would have led to issues during the tests, giving false data that the scheduler itself was performing poorly, rather than the problem being with my code. This way, the analysis of the three algorithms can be done without fear of extraneous coding errors.

The NOOP scheduler is probably the simplest of the algorithms to understand. Any incoming requests by processes to use the processor are simply placed in a first-in-first-out (FIFO) queue. A FIFO queue is one in which processes leave the queue in the same order they entered the queue. The requests are processed in the order that they arrived and are removed when they have completed or their time slice has expired. If a process runs out of time, it is simply placed back in the queue to wait its turn.

The Deadline scheduling algorithm, as its name implies, adds a deadline to requests. This deadline is basically a set amount of time during which the process has to

complete its task. It implements two separate deadline queues, as well as read and write queues. The first step for the algorithm is to determine which queue to pick a process from. This is decided based on the priority assigned to the queues. Next, the algorithm checks to see if the process in the queue has run past its deadline. If it hasn't, the process is then run. The Deadline algorithm is best used if a database is required (Collinson 1).

The CFQ scheduling algorithm is similar to the Deadline algorithm. In CFQ, requests are given a time slice and placed in a queue that corresponds to the process making the request. A time slice is similar to a deadline, but when the slice expires, the process is just removed from the processor, and can run again at a later time if it has not completed. The length of the time slice and the number of requests that can be processed from a specific queue are based on the priority placed on said queue. A higher priority means a longer time slice and more requests from that queue can be processed together. An interesting feature of CFQ is that it lets each process idle a little bit after it appears it has completed. A process that is running idle is usually a bad thing in operating systems, because that process is still allocated resources that could be used by other processes. The reason CFQ allows a small amount of idle time is because requests may appear idle when they are in fact still processing data. Stopping a request in the middle of processing data is detrimental to the performance of the operating system, so CFQ tries to fight this false idle time ("Chapter 13. Tuning I/O Performance.").

A feature that all three of these algorithms implement is request merging. A request merge occurs when a request is made that is located on the same part of the disk as the request/process that is already running. Both the running request and new request are run. For example, using the NOOP algorithm, there is a request in the FIFO

queue waiting to use the processor and that this request will need to take place at a section of the disk called Block 1. When it starts running, another request comes in at the back of the queue. However, since this request must also be run on Block 1 of the disk, it is then run at the same time as the current request. Request merging can help reduce the distance the disk head has to move by processing all requests that are located in the same place all at once, thereby improving overall system performance (Love).

Given the similarity of these algorithms, it was unclear which algorithm would perform the best. Since this experiment only ran three common applications during the trials, the algorithms should been able to schedule them effectively. My initial conjecture was that Deadline would do slightly worse, based on the fact that it is better suited for database work, but the expectation was that the results would be statistically equivalent for all three algorithms, with only slight variation that was not statistically significant.

Equipment

The experiment would consist of analysis of data collected from the algorithms and testing whether there is any statistical difference among them. In order for this experiment to be as bereft of as many outside influencing factors as possible, I bought a brand new HP Pavilion g6 laptop. The computer came with 320 gigabytes of hard drive, 4096 megabytes of DDR3 SDRAM, and Windows 7. The reason I bought a new laptop is because other hardware I had access to had extra applications and software that could affect the results of the experiment. I partitioned the hard disk drive of the laptop so that I could install the Linux operating system alongside the Windows operating system. The version of Linux I used was Ubuntu version 12.04 LTS.

The three applications I used during the testing were a web browser, a word processor, and a media player. For the web browser, I used Firefox version 19.0.2, because this was the default browser on Linux. The default word processor on Linux is LibreOffice Writer version 3.5.7.2. Finally, the media player I chose was Rhythmbox version 2.96. Unfortunately, iTunes requires few extra installations for it to work on Linux, so I chose Rythymbox instead because it was already downloaded. I uploaded 809 songs to Rythymbox, which is approximately 4.11 gigabytes of data.

Procedure

Another slight modification to the original proposal was how I was going to collect the data. The original proposal was to run the three applications in different combinations and time how long it took them to complete their tasks. Upon reflection, this method would have been sloppy and produced inaccurate results. It would have been difficult to determine if one of the applications was complete and defining what "complete" meant would have led to some inaccuracies. Instead, I used a performance-checking tool called vmstat, a program that reports virtual memory statistics. This program prints out certain statistics about how the system was performing during the experiment.

The first step of the data collecting procedure was choosing the scheduler. The three scheduling algorithms that I was testing could easily be switched to by this command: *echo noop > /sys/block/had/queue/scheduler.* The next step after the scheduler was set was to decide which order I should run the applications in. There are three applications, so there are a total of six different orders in which the applications can be started up. I wasn't sure if the order would have any affect on the results, so I tested

all of the orders just to be thorough. For example, I would start up a web browser, then begin a new word document, and lastly, start playing some music.

This is where vmstat came into play. Vmstat can, if told to, print out a line of statistics approximately every second. Running vmstat and the applications for a 100 second interval at a time yields 100 lines of statistics, which I would then average. I did this for 100 trials for each ordering of the applications. 100 lines of statistics for 100 trials give 10,000 lines of statistics. There are six different ways the applications can be ordered, so this leads to 60,000 lines of statistics that we can use to analyze each scheduling algorithm. Statistically speaking, each trial had 100 observations that made up the trial, so for each scheduling algorithm, there was a total of 600 trials. After all of these statistics were gathered, I found the average for each of the six application orderings, and then averaged those to get the results for the scheduling algorithm. 60,000 lines of statistics may seem like a lot to average, but it was straightforward to import the statistics into a spreadsheet. From there, I averaged each of the columns and record those averages in another document. This can all be done via this command: *vmstat 1 100 | tee Desktop/data.ods.* This command states that vmstat should print out 100 lines of statistics (about 1 per second) before exporting that data to a numbers file, here entitled "data.ods."

Before I state the results, I should describe the types of statistics I looked at during the trials. Here is a sample line of what vmstat would print out:

r	**b**	swpd	free	buff	cache	si	so	bi	bo	**in**	**cs**	**us**	**sy**	**id**	**wa**
1	**0**	181024	26276	35316	502960	0	0	3	2	**6**	**1**	**5**	**1**	**94**	**0**

Even though I averaged all of these statistics, I ignored some of them in the final analysis because they had to do with memory. Since the laptop was new and had a lot of memory, I didn't think these statistics would matter in the final analysis. Just to make sure that the memory statistics would not be statistically relevant to the results, I used the GNOME System Monitor software, another performance tester, to see how much memory was allocated to each of the three applications. When Firefox was running, it was allocated about 40-50 megabytes of memory. Rhythmbox was allocated about 40 megabytes of memory, as well. LibreOffice Writer barely uses any memory, as is expected, because word documents don't require much memory. This paper itself is only about 150 kilobytes of data. Because these three applications don't require a lot of memory, the memory usage statistics are not useful for determining the algorithms' performance.

The bolded data are the information I was interested in. "r" and "b" have to do with the number of processes waiting and the number in uninterruptible sleep, respectively. A process sometimes enters the sleep state if it does not have any work to complete at the moment. If the number of processes sleeping is high, this also increases the processing time, because the processor has to wait until the process wakes up for it to check if it needs the processor. A sleeping process cannot use a processor, even if one is available, which can cause an algorithm with a lot of sleeping processes to be inefficient (Ware).

These statistics are important for scheduling, because they can tell the programmer if there is a problem with the scheduling algorithm. For example, if the time slices or deadlines set by the CFQ and Deadline algorithms are too long, the number of

processes waiting and sleeping will be higher because each process is given a large amount of time on the processor. Forcing the algorithms to give out shorter time slices and deadlines may lead to more processes being run, which may lead to a more efficient algorithm. However, this might also increase the number of interrupts and context switches.

The number of interrupts and context switches happening per second are indicated by the "in" and "cs" columns, respectively. Interrupts are a signal to the operating system that something needs immediate attention. An interrupt could, in fact, be signaling for a context switch, which occurs when one process is replaced by another. We would like these numbers to be as low as possible, because interrupts and context switches take time on the CPU, preventing it from doing useful work for the applications (Kaedrin).

The next four pieces of data ("us", "sy", "id", and "wa") are all connected because their numbers represent a percentage of the CPU time. "us" represents the percentage of time spent running code in user mode, while "sy" tells us the percentage of time running kernel code. Code that is executable in kernel mode has access to the functions and methods that control not only very low-level parts of the computer, but also the hardware itself. Kernel mode code is highly restricted; an error in the kernel code will have major consequences for the entire machine. Code executed in user mode is basically run on top of the kernel code. User mode only has access to some of the functionality that is performed by the kernel. Errors in the user mode are recoverable. Based on the three applications being used in this experiment, we would expect the percentage of time in user mode to be higher than the time in kernel mode, because the

applications shouldn't need much access to the lower-level functions. The time spent in kernel mode should have some correlation to the number of context switches, because every context switch happens in kernel mode. If one algorithm has more context switches than another, we should expect the time spent in kernel mode to be slightly higher (Atwood).

"id" represents the percentage of time spent idling, and "wa" is the percentage of time spent waiting for I/O requests to complete. Idling simply means the processor is not being used by anything; there are no processes waiting to use the processor. For example, we would expect the idle time for the CFQ algorithm to be higher than the other two, because part of its algorithm dictates that there be a small amount of idle time after a process appears to be finished, even though it may just be waiting for some other event to happen before it starts running again. A high amount of idling could be a sign that the processor is being poorly optimized, or it could mean that the algorithm is working well enough that processes are finishing relatively fast. It was hard to discern whether the amount of time waiting for I/O requests was high or low. The three applications rely heavily on I/O requests, so we would expect "wa" to be high, but if the algorithm is efficient enough, the time spent waiting could be minimal. (Ware).

Results

A sample of the 60,000 lines of data collected is included in the appendix. The appendix also includes the final averages for each of the three algorithms. They are summarized and discussed next.

After all of the data was collected, here were the final averages of the three algorithms:

	r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
NOOP	2.485	0.038	0	2422061.87	71860.73	55724.95	0	0	45.536	175.656	2120.69	4581.597	25.611	8.02	63.39	2.978
CFQ	2.484	0.079	0	2423323.67	70134.29	56046.57	0	0	215.46	243.373	2110.9	4482.333	25.509	8.056	63.53	2.907
DEADLINE	2.48	0.037	0	2422292.87	71907.22	59836.12	0	0	45.378	174.953	2125.05	4493.902	25.638	8.105	63.31	2.948

One way to analyze this data is to compare each of the means as they appear above. However, just looking at these means and coming to a conclusion would be naïve, because even though these numbers are all different from each other, they may or may not be statistically different. The sample size plays a role in this calculation. The larger the sample size, the less variability there may be between means. Because the results are each created from a sample size of 600, it may very well be possible that these means are statistically the same.

To determine if the means of the data (μ_{NOOP}, μ_{CFQ}, μ_{Dead}) are statistically the same, we must apply a hypothesis test to this data. A hypothesis test consists of two statements, the null hypothesis (H_0) and the alternate hypothesis (H_a). For each set of means we can state our hypotheses in this fashion:

$$H_0: \mu_{NOOP}=\mu_{CFQ}=\mu_{Dead}$$

$$H_a: \mu_{NOOP}\neq\mu_{CFQ}$$
$$\mu_{CFQ}\neq\mu_{Dead}$$
$$\mu_{NOOP}\neq\mu_{Dead}$$

So, we would apply this hypothesis test for each type of statistic, starting with the "r" statistic, and work our way down to "wa."

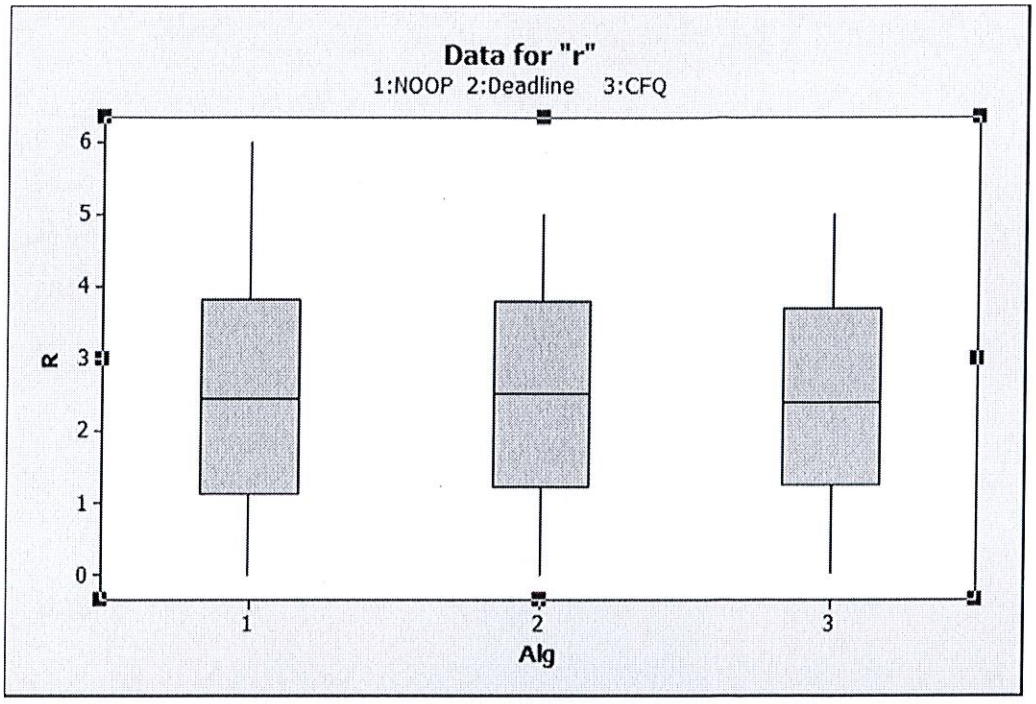
There are few more components to a hypothesis testing. The next component is the level of significance (α). The level of significance is our cut-off for the p-value. The p-value, along with the level of significance, is used to determine whether we should reject or accept the null hypothesis for the data. The level of significance is predetermined; it is independent of the data and is usually either 0.05 or 0.01. The p-value is constructed from the data being tested. If the p-value were less than the level of significance, we would reject the null hypothesis in favor of the alternate hypothesis. If it

were greater than the level of significance, we would accept the null hypothesis. For these tests, I found the p-value for both α of 0.05 and 0.01.

With the hypothesis test set, the type of test now needs to be chosen. Since we are testing whether the means are statistically different, we want to look at the variability between the means. The proper test for this situation is a 1-Way Analysis of Variance (ANOVA) test. The ANOVA test is used to determine the levels of variance between the three means and determine if they are statistically different. The ANOVA test is specially suited to compare three or more means. Fortunately, Minitab, a statistics software program, has the capabilities of performing an ANOVA test and showing us the p-value for each set of means. Here are the p-values for all eight statistical means:

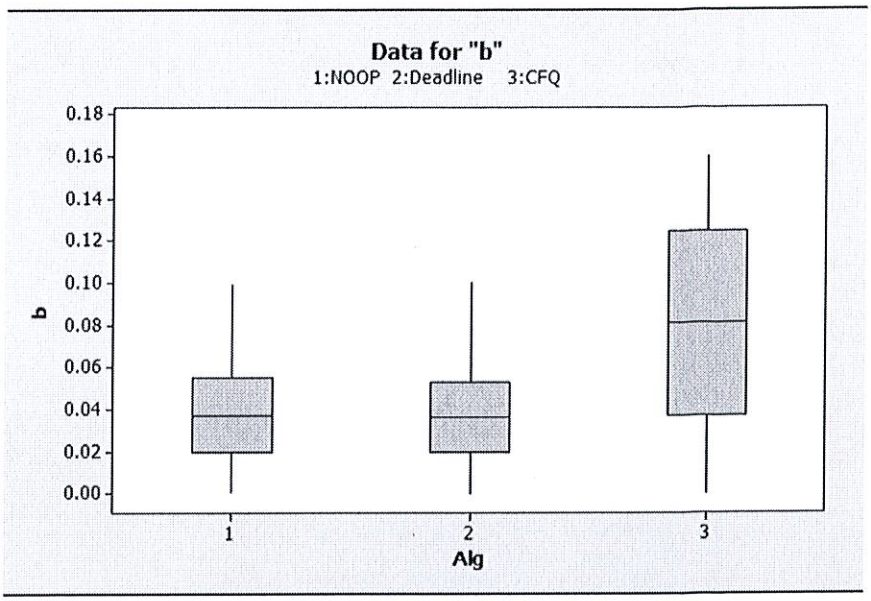
	r	b	in	cs	us	sy	id	wa
p-values	0.811	0	0.122	0.106	0.421	0.994	0.319	0.755

As we can see, most of these p-values are quite high. Recall, to reject the null hypothesis, our p-values must be less than the level of significance. In both cases, (α of 0.05 and 0.01) all except for one of the data is over the levels of significance. This means that for "r", "in", "cs", "us", "sy", "id", and "wa", the variability among the three algorithms' means was so small, that they should be considered statistically the same mean. A boxplot of the data visually confirms that the means are almost identical.



As we can see, it is nearly indiscernible if the means are different. While this is only the boxplot for "r", the rest of the data, except for "b", look almost identical to this graph, and so it is not be particularly helpful to include them. This also clears up the issue of the correlation between the number of context switches and time spent in kernel mode. Because the mean time spent in kernel mode is statistically the same for all three algorithms, we would expect that the number of context switches would be the same, as well. The data supports this.

The statistic of interest is "b". It has a p-value of 0, so for both levels of confidence, we should reject the null hypothesis and accept the alternate hypothesis, which was that the three means are statistically different. This means that even though the sample size was large, there was still enough variability among the means to make them statistically different. Here's the boxplot for "b":



Here there is visual evidence that the mean is statistically different for the CFQ algorithm. It appears that the means for NOOP and Deadline are statistically the same. Recall, that we would like the number of processes in the sleep state to be kept as low as possible, because this may increase processing time.

Conclusion

Looking at the results, I conclude that my findings were that this selection of scheduling algorithms has no impact on the performance of the three applications. The NOOP and Deadline algorithms are statistically the same for all parts of the data, so there are no data to support one of them performing better.

Based on the "b" data, we could possibly say that the CFQ algorithm performed slightly worse than the NOOP and Deadline algorithms. However, I hesitate to make that conclusion, as well. The only piece of data that CFQ differed on was the number of processes sleeping; all other facets of the data were statistically the same as the other two algorithms. A process enters the sleep state only if it has nothing to execute at the

moment. One could conjecture that the algorithm handled the processes that were sleeping efficiently; the algorithm ran so that any process that was sleeping would not be requested until it woke up, saving processor time. I say this because CFQ was statistically the same for all of the other pieces of data. It is very possible that CFQ had more sleeping processes, but still worked as efficiently as the other two algorithms.

My final conclusion is that the data collected shows that the three applications, statistically, performed the same. These results are a bit anti-climactic, but the data shows this much. So, for a laptop that uses the three most common applications used by college students, the results suggest that any of the three preemptive scheduling algorithms provided by the Linux operating system will perform as efficiently as the other two.

Works Cited

Atwood, Jeff. "Understanding User and Kernel Mode."*Coding Horror*. N.p., 03 Jan 2008. Web. Web. 1 Apr. 2013.

"Chapter 13. Tuning I/O Performance." (2012): n. page. Web. 24 Mar. 2013. <http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.io.html>.

Chekuri, C., R. Motwani, B. Natarajan, and C. Stein. "Approximation techniques for average completion time scheduling.." *SIAM Journal on Discrete Mathematics* 31.1 (2001): 146. *A*. Web. 30 Jun 2012.

Chou, Fuh-Der, Hui-Mei Wang, and Tzu-Yun Chang. "Algorithms for the single machine total weighted completion time scheduling problem with release times and sequence-dependent setups." *International Journal of Advanced Manufacturing Technology* 43.7 (2009): 810-21. *Academic Search Premier*. Web. 30 Jun 2012.

Collinson, Ronnie. "Linux Io Scheduler." *Waikato Linux Users Group*. (2012): 1. Web. 24 Mar. 2013. <http://www.wlug.org.nz/LinuxIoScheduler>.

Deitel, H.M., P.J. Deitel, and D.R. Choffnes. *Operating Systems*. 3. 1. Upper Saddle River: Pearson Prentice Hall, 2004. 291-359. Print.

Dosa, G., and Y. He. "Preemptive and non-preemptive on-line algorithms for scheduling with rejection on two uniform machines." *Computing* 76.1 (2006): 149-64. *Academic Search Premier*. Web. 10 Jun 2012.

Dosa, Gyorgi, and Leah Epstein. "Preemptive online scheduling with reordering." *SIAM Journal on Discrete Mathematics* 25.1 (2011): 21-49. *Academic Search Premier*. Web. 10 Jun 2012.

Ebenlendr, Tomáš , and Jiří Sgall. "Semi-online preemptive scheduling: one algorithm for all variants." *Theory of Computing Systems* 48.3 (2011): 577-613. *Academic Search Premier*. Web. 26 Jun 2012.

Gonzalez, Teofilo, Eugene L. Lawler, and Sartaj Sahni. "Optimal preemptive scheduling of two unrelated processors." *ORSA Journal on Computing* 2.3 (1990): 219. *Academic Search Premier*. Web. 26 Jun 2012.

Hariri, A., C. Potts, and L. Van Wassenhove. "Single machine scheduling to minimize total weighted late work." *ORSA Journal on Computing* 7.2 (1995): 232. *Academic Search Premier*. Web. 1 Jul 2012.

He, Young, Yi Wei Jiang, and Hao Zhou. " Optimal preemptive online algorithms for scheduling with known largest size on two uniform machines." *Acta Mathematica Sinica* 23.1 (2007): 165-74. *Academic Search Premier*. Web. 10 Jun 2012.

Hoogeveen, Han, Martin Skutella, and Gerard Woeginger. "Preemptive scheduling with rejection." *Mathematical Programming* 94.2 (2003): 361. *Academic Search Premier*. Web. 1 Jul 2012.

"Interrupts and Context Switching." *Kaedrin*. (2009): n. page. Web. 24 Mar. 2013. <http://kaedrin.com/weblog/archive/001657.html>.

Lin, Guolong, and Rajmohan Rajaraman. "Approximation algorithms for multiprocessor scheduling under uncertainty." *Theory of Computing Systems* 47.4 (2010): 856-77. *Academic Search Premier*. Web. 27 Jun 2012.

Love, Robert. "Kernel Korner - I/O Schedulers." *Linux Journal*. (2004): n. page. Web. 24 Mar. 2013. <http://www.linuxjournal.com/article/6931>.

Paletta, Giuseppe, and Paolamaria Pietramala. "A new approximation algorithm for the nonpreemptive scheduling of independent jobs on identical processors." *SIAM Journal on Discrete Mathematics* 21.2 (2007): 313-28. *Academic Search Premier*. Web. 27 Jun 2012.

Reisi, Mohammad, and Ghasem Moslehi. "Minimizing the number of tardy jobs and maximum earliness in the single machine scheduling using an artificial immune system." *International Journal of Advanced Manufacturing Technology* 54.5 (2011): 749-56. *Academic Search Premier*. Web. 27 Jun 2012.

Sevastyanov, S.V., R.A. Sitters, and A.V. Fiskin. " Preemptive scheduling of independent jobs on identical parallel machines subject to migration delays." *Automation & Remote Control* 71.10 (2010): 2093-101. *Academic Search Premier*. Web. 10 Jun 2012.

Ware, Henry. "vmstat." *Linux Administrator's Manual*. (1994): n. page. Web. 24 Mar. 2013. <http://linuxcommand.org/man_pages/vmstat8.html>.

Sample Trial Data

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
1	0	0	2442176	71412	546024	0	0	386	47	459	1013	12	4	79	5
1	0	0	2441540	71420	546028	0	0	0	44	1149	2848	14	4	81	2
2	0	0	2440616	71420	546028	0	0	0	0	1984	6430	51	10	39	0
0	0	0	2440384	71420	546176	0	0	128	0	2491	4995	17	7	77	0
1	0	0	2439044	71420	546156	0	0	0	0	2295	3850	35	9	56	0
2	0	0	2438920	71420	546156	0	0	0	44	2706	4569	28	7	65	0
0	0	0	2438920	71428	546156	0	0	0	28	2453	4150	25	6	66	2
0	0	0	2438672	71428	546156	0	0	0	0	2413	4106	29	5	66	0
1	0	0	2435680	71436	546196	0	0	0	32	2726	4592	37	7	54	3
1	0	0	2432672	71436	546216	0	0	0	0	1854	2858	19	7	74	0
0	0	0	2432300	71444	546208	0	0	0	72	1441	3900	14	5	78	3
0	0	0	2431904	71464	546200	0	0	0	412	1211	2269	19	3	74	5

Final Averages

NOOP

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
35	0.035	0	2422557.8	71830.61	59811.21	0	0	45.568	174.627	2164.58	4446.54	25.499	8.15	63.24	3.1
36	0.04	0	2422034.48	71824.83	54916.55	0	0	45.356	176.802	2141.02	4817.18	25.398	7.8	63.99	2.8
75	0.04	0	2420486.54	71909	54891.81	0	0	45.706	176.23	2063.85	4378.29	25.97	8.15	62.83	3.0
54	0.04	0	2422476	71878.33	54905.71	0	0	45.478	176.833	2083.43	4569.55	25.572	8.12	63.28	3.0
58	0.03	0	2421206.23	71879.61	54924.98	0	0	45.681	175.047	2154.21	4571.23	25.812	8	63.03	3.1
33	0.04	0	2423610.18	71842	54899.44	0	0	45.427	174.398	2117.06	4706.79	25.412	7.9	63.96	2.7
485	0.038	0	2422061.87	71860.73	55724.95	0	0	45.536	175.656	2120.69	4581.597	25.611	8.02	63.39	2.9

CFQ

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
493	0.08	0	2422946	70272.56	56289.48	0	0	221.48	249.974	2110.5	4386	25.439	7.864	63.75	2.9
384	0.078	0	2424293	70198.49	55921.16	0	0	210.33	244.672	2131.37	4299	25.403	8.212	63.49	2.8
577	0.08	0	2423266	70166.22	55808.41	0	0	214.34	247.54	2030.53	4553	25.653	8.212	63.2	2.9
338	0.075	0	2423801	70104.54	56215.63	0	0	216.51	237.012	2169.15	4690	25.44	8.229	63.32	3.0
311	0.083	0	2421727	70002.81	56078	0	0	215.92	236.267	2115.33	4682	26.079	7.843	63.43	2.6
301	0.077	0	2423909	70061.09	55966.73	0	0	214.16	244.774	2108.51	4284	25.042	7.976	64	3.0
484	0.079	0	2423323.67	70134.29	56046.57	0	0	215.46	243.373	2110.9	4482.333	25.509	8.056	63.53	2.9

Deadline

r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
449	0.034	0	2422554.47	71929.28	59833.99	0	0	45.08	175.54	2092.58	4590.98	26.08	8.359	62.58	2.9
449	0.034	0	2422096.35	71821.09	59838.38	0	0	45.18	175.91	2124.37	4706.63	25.37	8.079	63.52	3.0
416	0.039	0	2422869.05	71917.7	59844.87	0	0	45.39	175.29	2185.78	4285.72	25.49	7.933	63.94	2.6
516	0.038	0	2422343.44	71921.52	59846.14	0	0	45.75	175.39	2081.97	4501.14	25.45	8.215	63.24	3.1
405	0.04	0	2421426.63	71946.59	59836.56	0	0	45.58	173.66	2157.89	4447.43	25.67	8.101	63.27	2.9
643	0.036	0	2422467.29	71907.15	59816.81	0	0	45.29	173.93	2107.71	4431.51	25.77	7.941	63.33	2.9
48	0.037	0	2422292.87	71907.22	59836.12	0	0	45.378	174.953	2125.05	4493.902	25.638	8.105	63.31	2.9