Reducing Software Bloat in the Linux Kernel Through SLAB Allocation and Unfair Scheduling

Ramsey Alsheikh

Abstract

Software bloat is the inefficient or ineffective use of computer resources that affects many computer operating systems today; in particular the Linux kernel is known to suffer acutely from bloat. The purpose of this experiment was to provide evidence for whether or not the performance of the Linux kernel may be improved, and bloat reduced, through implementing SLAB memory allocation instead of SLUB, and the Multiple Queue Skiplist Scheduler (MuQSS) process scheduler instead of the Completely Fair Scheduler (CFS). Both SLUB and CFS sacrifice general system efficiency for failsafes in cases with extreme RAM and CPU time utilization; thus, it was hypothesized that they were detrimental to overall system performance, relatively inefficient, and sources of software bloat. The hypothesis was that if SLAB allocation was enabled and the MuQSS scheduler was set up in the Linux kernel, then the performance of the computer (measured by disk space, memory usage, and processing speed), would improve. An Arch Linux system was prepared with this configuration, and it outperformed the default configuration significantly on all three trials as measured by UnixBench, a standard benchmarking program for Unix-like operating systems. This experiment provides evidence for the effectiveness of this potential bloat-reducing and performance-enhancing Linux kernel patch. Due to the importance of Linux in several key technological areas, including smartphone operating systems, web server hosting, and supercomputer processing, any reduction of bloat in the Linux kernel is of particular interest to modern computer science and operating system design.

Introduction

Statement of Purpose

The purpose of this experiment would be to provide evidence for whether or not the performance of the Linux kernel may be improved, and bloat reduced, through implementing SLAB memory allocation and alternatives to the traditional Completely Fair Scheduler (CFS). A modified version of the Linux kernel would be configured to use SLAB allocation (instead of the default SLUB allocator) and patched to replace CFS with the MuQSS scheduler. The experimental group would be a build of Arch Linux that utilizes the modified kernel, and the control group would be a build of the same version of Arch Linux that utilizes an unmodified kernel. The dependent variable would be the disk space, memory usage, and processing speed benchmarked from the Arch Linux build, while the independent variable would be which kernel was installed on the build. If the modified kernel performs more efficiently than the default kernel, it would provide evidence for SLAB allocation and unfair scheduling as possible tools to reduce the encroaching bloat in the Linux kernel. Linux kernels are installed in every Linux system, which run most of the world's web servers, power all of the world's top five hundred supercomputers, and are the basis for the widely popular Android mobile operating system.

Background Research

Linux is an open-source family of operating systems that is used to host Internet web servers, run the world's five hundred most powerful supercomputers, and to operate all Android smartphones. (What is Linux?) All Linux operating systems use the Linux kernel, which is the centerpiece of the Linux family of operating systems; it is the code that connects the hardware to the software and is responsible for managing the system's resources and running programs (What

is the). Software bloat is the unnecessary complexity that arises in software over time as a result of increasing computer capability, high level of abstraction, and the increasing inclusion of niche features into software (Definition of Software). The Linux kernel suffers especially from software bloat because it is open source; there is less centralization in its development and update cycle, which helps proliferate bloating (Gralla, 2009). Software bloat is negative because the added complexity leads to less efficient programs that waste resources. Two areas in the Linux kernel that are bloated are identified for the purpose of this experiment: memory allocation and process scheduling (Bhartiya, 2020). Memory allocation is the process of sharing the machine's primary memory resources (usually in the form of physical or virtual Random Access Memory, or RAM) among the running programs on the computer. Process scheduling is the task of managing which program will have access to the CPU at a given point in time. Both subsystems have been subjected to recent updates that have added features that have ostensibly improved the kernel but may have done so at the expense of performance. With respect to memory allocation, the novel SLUB allocator, used in current versions of the Linux kernel, was designed to mitigate the occasional RAM spikes that the previous SLAB took up (What is Memory). However, SLUB uses more CPU time than SLAB did, and the RAM spikes that SLAB caused were infrequent and in only extreme cases. As such, there is a case to be made that SLUB may be detrimental to computer performance, and may be an instance of software bloat in the Linux kernel. With respect to process scheduling, the current versions of the Linux kernel use CFS as the process scheduler. CFS takes into consideration more factors than other schedulers when considering what program to run next, in order to distribute CPU access more evenly across priorities. While in certain high-intensity workloads this may offer benefits, it may slow

down the responsiveness of the system in more common scenarios. Because it adds niche features and support that may be detrimental to the performance of the machine overall, the Completely Fair Scheduler could be a potential source of software bloat in the Linux kernel. An alternative option to the Completely Fair Scheduler, the MuQSS scheduler may be a more reliable and streamlined option (in contrast to the “Completely Fair Scheduler”, it is considered an “unfair” scheduler). By removing these subsystem implementations from the kernel and replacing them with simpler alternatives, the performance of the Linux kernel may be able to improve for the vast majority of users. Software bloat may be able to be decreased in the kernel. Linux is the operating system installed on sixty seven percent of the world’s web servers and powers the Android mobile operating system, which accounts for eighty four percent of the smartphone market. (Finley, 2016) Most of the world's top supercomputers, many used for calculation-intensive science simulations and research, are powered by Linux. (Linux Took Over). By making the Linux kernel more efficient, much of our modern information infrastructure is improved and streamlined.

Hypothesis

If SLAB allocation is enabled and the MuQSS scheduler is installed in the Linux kernel, then the performance of the computer (measured by disk space, memory usage, and processing speed), will improve. The drawbacks of SLAB and MuQSS (high RAM spikes and uneven CPU time allocation) that SLUB and CFS purport to fix are only relevant under extreme and specific circumstances. The added complexity from SLUB and CFS may be detrimental overall to kernel performance, and may be causing software bloat in the kernel. However, if the specific situations

are common enough, then SLUB and CFS may counterbalance and offset their performance cost with a better handling of worst-case scenarios. This experiment will test this.

Method

Materials

- A computer capable of running Arch Linux 2021.07.01 with administrator access and at least one USB port
- An Arch Linux 2021.07.01 USB flash installation medium
- Charging Cable
- Internet Access

Procedure

1. Attach the charging cable to an outlet, and connect the computer to the charging cable.
2. Verify that the computer has an x86 processor capable of running Arch Linux 2021.07.01.
3. Verify that the computer has 'USB Boot' enabled in BIOS. Access to the BIOS is machine-specific and will depend on the computer make and model.
4. Shut down the computer.
5. Connect the USB flash installation medium to the USB port on the computer.
6. Boot the computer from the USB flash installation medium.
7. Install Arch Linux 2021.07.01. Using the pacman package manager and pacstrap command, ensure that the 'base-devel' package is installed in addition to the standard 'base' package during this process. Ensure a text editor is installed during this process. Ensure that Arch Linux is installed with the appropriate tools to access the internet, which are dependent on the computer make and model, as well. Set up a password for the root account.
8. Shut the computer down.
9. Remove the USB flash installation medium.
10. Boot the computer from the instance of Arch Linux that was just installed.
11. Log into Arch Linux using the root password you set during the installation process.
12. Using a text editor, open /etc/pacman.conf and insert the following lines of text, exactly as they appear below, excluding the quotes:

    "[repo-ck]
    Server = http://repo-ck.com/$arch
    Server = http://repo-ck.com/$arch
    Server = http://repo-ck.com/$arch
    Server = http://repo-ck.com/$arch"

    Ensure that the above lines of text are inserted directly above the following text, excluding the quotes, separated by one new line:

    "#[testing]"

13. Save the changes to /etc/pacman.conf in the text editor and exit the text editor
14. Run the following command: pacman -key -r 5EE46C4C
15. Run the following command: pacman -key -f 5EE46C4C
16. Run the following command: pacman -key --l-sign 5EE4
17. Run the following command: pacman -Syy
18. Run the following command: /lib/ld-linux-x86-64.so.2 --help | grep supported
19. If "x86-64-v3 (supported, searched)" was outputted, run the following command: pacman -Sg ck-generic-v3
20. If "x86-64-v3 (supported, searched)" was not outputted, but "x86-64-v2 (supported, searched)" was, run the following command: pacman -Sg ck-generic-v2
21. If neither of conditions for the previous two steps were met, run the following command: pacman -Sg ck-generic
22. Run the following command: pacman -S linux-ck-generic linux-ck-generic-headers
23. Run the following command: mkinitcpio -p linux-ck
24. Run the following command: grub-mkconfig -o /boot/grub/grub.cfg
25. Run the following command and wait for the computer to restart: shutdown -r now
26. Log in the root account. The MuQSS scheduler is now installed.
27. Run the following command: pacman -Sg git libncurses5-dev bison flex libssl-dev libelf-dev makepkg
28. If step 19 was executed, run the following command: cd /usr/src/linux-ck-generic-v3
29. If step 20 was executed, run the following command: cd /usr/src/linux-ck-generic-v2
30. If step 21 was executed, run the following command: cd /usr/src/linux-ck-generic
31. Run the following command: sudo make menuconfig
32. In the menu that appears, navigate to the "General Setup" subdirectory and enable SLAB allocation.
33. Save the configuration and exit the menu.
34. Run the following command and wait for the command to finish, which may take several hours: make
35. Run the following command and wait for the command to finish, which may take several hours: make modules_install install
36. Run the following command and wait for the computer to restart: shutdown -r now
37. Log in the root account. SLAB allocation is now enabled.
38. Run the following command: cd /root
39. Run the following command: git clone https://aur.archlinux.org/snapd.git
40. Run the following command: cd snapd
41. Run the following command: makepkg -si
42. Run the following command: systemctl enable --now snapd.socket
43. Run the following command: ln -s /var/lib/snapd/snap /snap
44. Run the following command and wait for the computer to restart: shutdown -r now

45. Log in the root account
46. Run the following command: cd /root
47. snap install unixbench
48. Run the following command: mkdir /root/results
49. Run the following command: cd /root/results
50. Run the following command and wait for the command to finish, which may take several hours: ubench > experimentaltrialone.txt
51. Run the following command and wait for the command to finish, which may take several hours: ubench > experimentaltrialtwo.txt
52. Run the following command and wait for the command to finish, which may take several hours: ubench > experimentaltrialthree.txt
53. Restart the computer, and in GRUB select “Advanced options for Arch Linux” and boot into the original kernel.
54. Log in the root account
55. Run the following command: cd /root/results
56. Run the following command and wait for the command to finish, which may take several hours: ubench > controltrialone.txt
57. Run the following command and wait for the command to finish, which may take several hours: ubench > controltrialtwo.txt
58. Run the following command and wait for the command to finish, which may take several hours: ubench > controltrialthree.txt

Results

Metric	Trial One Index Score	Trial Two Index Score	Trial Three Index Score
Dhrystone 2 using register variables	4423.1	4388.3	4441.0
Double-Precision Whetstone	1685.9	1685.8	1685.9
Execl Throughput	1880.4	1876.5	1875.7
File Copy 1024 bufsize 2000 maxblocks	3616.8	3625.3	3632.8
File Copy 256 bufsize 500 maxblocks	2559.2	2556.6	2538.2
File Copy 4096 bufsize 8000 maxblocks	3897.2	3863.2	3860.5
Pipe Throughput	2191.8	2164.7	2188.8
Pipe-based Context Switching	285.5	285.4	285.1
Process Creation	1395.1	1410.1	1384.4
System Benchmarks Index Score	2023.6	2019.4	2018.3

Table 1. The UnixBench Indices for the Experimental Group

Metric	Trial One Index Score	Trial Two Index Score	Trial Three Index Score
Dhrystone 2 using register variables	4396.5	4417.0	4512.2
Double-Precision Whetstone	1683.1	1683.4	1683.3
Execl Throughput	1844.0	1842.3	1835.7
File Copy 1024 bufsize 2000 maxblocks	3339.9	3350.3	3332.7
File Copy 256 bufsize 500 maxblocks	2325.8	2351.6	2325.4
File Copy 4096 bufsize 8000 maxblocks	3941.1	3760.0	3728.5
Pipe Throughput	2039.8	2164.7	2051.9
Pipe-based Context Switching	168.4	285.4	158.1
Process Creation	455.9	440.3	456.9
System Benchmarks Index Score	1578.6	1559.3	1564.2

Table 2. The UnixBench Indices for the Control Group

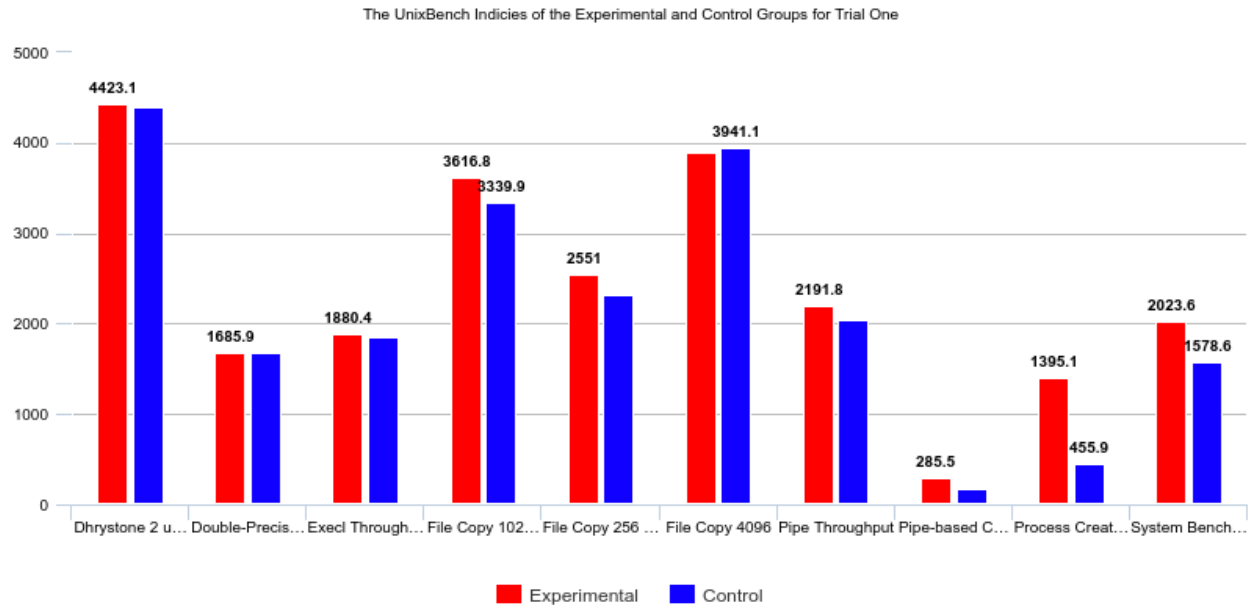

Figure 1. UnixBench Indices of the Experimental and Control Groups for Trial One Bar Chart

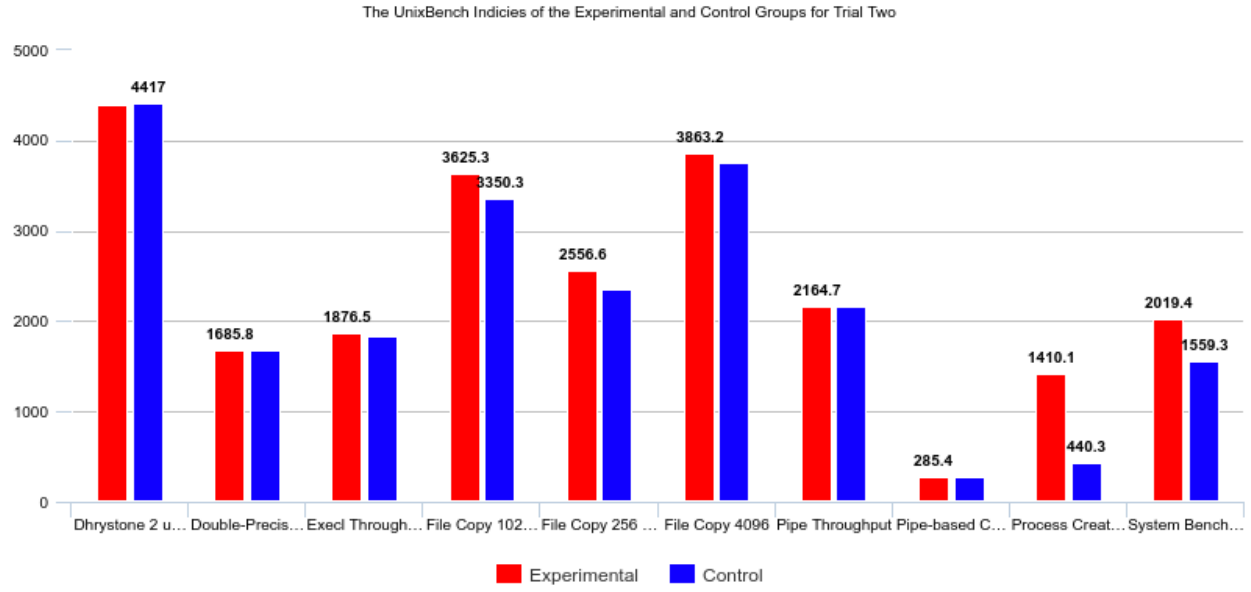

Figure 2. UnixBench Indices of the Experimental and Control Groups for Trial Two Bar Chart

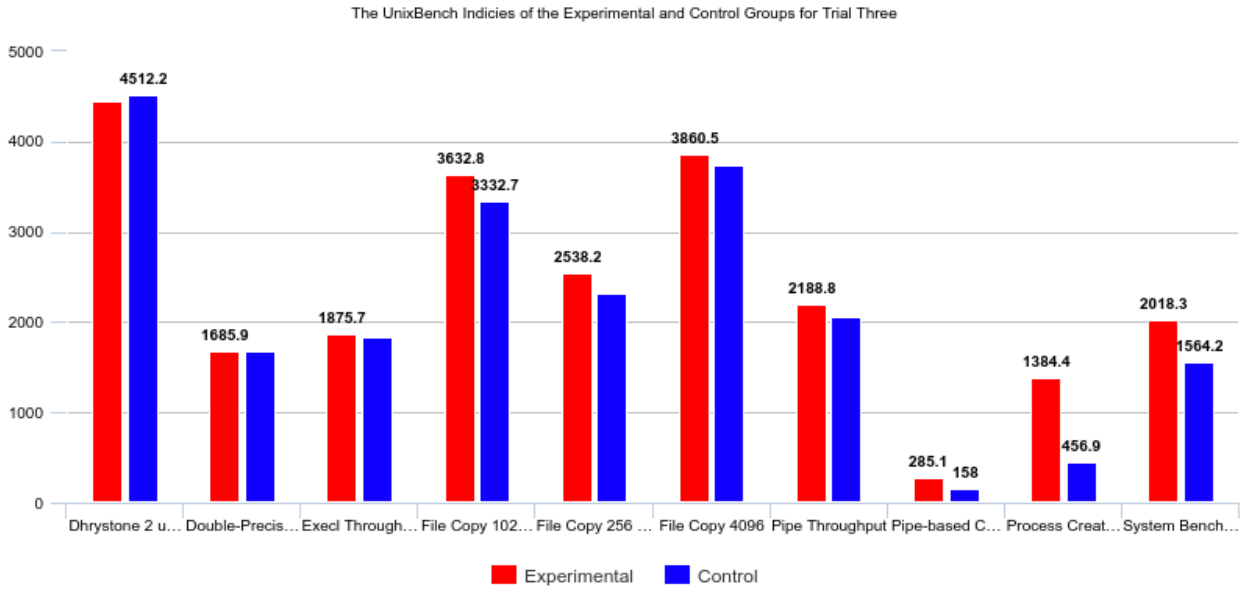

Figure 3. UnixBench Indices of the Experimental and Control Groups for Trial Three Bar Chart

```
Benchmark Run: Mon Aug 23 2021 15:30:34 - 15:58:36
16 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables       51617617.7 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                     9272.7 MWIPS (9.9 s, 7 samples)
Execl Throughput                               8085.9 lps   (29.7 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks       1432266.1 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks          423553.9 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks       2260349.7 KBps  (30.0 s, 2 samples)
Pipe Throughput                             2726638.2 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                 114206.3 lps   (10.0 s, 7 samples)
Process Creation                              17578.6 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                   5367.0 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)                   2851.2 lpm   (60.0 s, 2 samples)
System Call Overhead                        2663073.1 lps   (10.0 s, 7 samples)

System Benchmarks Index Values               BASELINE       RESULT    INDEX
Dhrystone 2 using register variables         116700.0   51617617.7   4423.1
Double-Precision Whetstone                       55.0       9272.7   1685.9
Execl Throughput                                 43.0       8085.9   1880.4
File Copy 1024 bufsize 2000 maxblocks          3960.0    1432266.1   3616.8
File Copy 256 bufsize 500 maxblocks            1655.0     423553.9   2559.2
File Copy 4096 bufsize 8000 maxblocks          5800.0    2260349.7   3897.2
Pipe Throughput                               12440.0    2726638.2   2191.8
Pipe-based Context Switching                   4000.0     114206.3    285.5
Process Creation                                126.0      17578.6   1395.1
Shell Scripts (1 concurrent)                     42.4       5367.0   1265.8
Shell Scripts (8 concurrent)                      6.0       2851.2   4752.1
System Call Overhead                          15000.0    2663073.1   1775.4
                                                                     ========
System Benchmarks Index Score                                         2023.6
```

Figure 4. Sample UnixBench Output for the Experimental Group

```
Benchmark Run: Wed Aug 25 2021 13:08:17 - 13:36:23
16 CPUs in system; running 1 parallel copy of tests

Dhrystone 2 using register variables       51307562.4 lps   (10.0 s, 7 samples)
Double-Precision Whetstone                     9257.0 MWIPS (9.9 s, 7 samples)
Execl Throughput                               7929.0 lps   (29.6 s, 2 samples)
File Copy 1024 bufsize 2000 maxblocks       1322599.9 KBps  (30.0 s, 2 samples)
File Copy 256 bufsize 500 maxblocks          384916.7 KBps  (30.0 s, 2 samples)
File Copy 4096 bufsize 8000 maxblocks       2285820.4 KBps  (30.0 s, 2 samples)
Pipe Throughput                             2537501.7 lps   (10.0 s, 7 samples)
Pipe-based Context Switching                  67361.1 lps   (10.0 s, 7 samples)
Process Creation                               5744.4 lps   (30.0 s, 2 samples)
Shell Scripts (1 concurrent)                   2543.0 lpm   (60.0 s, 2 samples)
Shell Scripts (8 concurrent)                   2120.6 lpm   (60.0 s, 2 samples)
System Call Overhead                        2591261.8 lps   (10.0 s, 7 samples)

System Benchmarks Index Values               BASELINE       RESULT    INDEX
Dhrystone 2 using register variables         116700.0   51307562.4   4396.5
Double-Precision Whetstone                       55.0       9257.0   1683.1
Execl Throughput                                 43.0       7929.0   1844.0
File Copy 1024 bufsize 2000 maxblocks          3960.0    1322599.9   3339.9
File Copy 256 bufsize 500 maxblocks            1655.0     384916.7   2325.8
File Copy 4096 bufsize 8000 maxblocks          5800.0    2285820.4   3941.1
Pipe Throughput                               12440.0    2537501.7   2039.8
Pipe-based Context Switching                   4000.0      67361.1    168.4
Process Creation                                126.0       5744.4    455.9
Shell Scripts (1 concurrent)                     42.4       2543.0    599.8
Shell Scripts (8 concurrent)                      6.0       2120.6   3534.4
System Call Overhead                          15000.0    2591261.8   1727.5
                                                                     ========
System Benchmarks Index Score                                         1578.6
```

Figure 5. Sample UnixBench Output for the Control Group

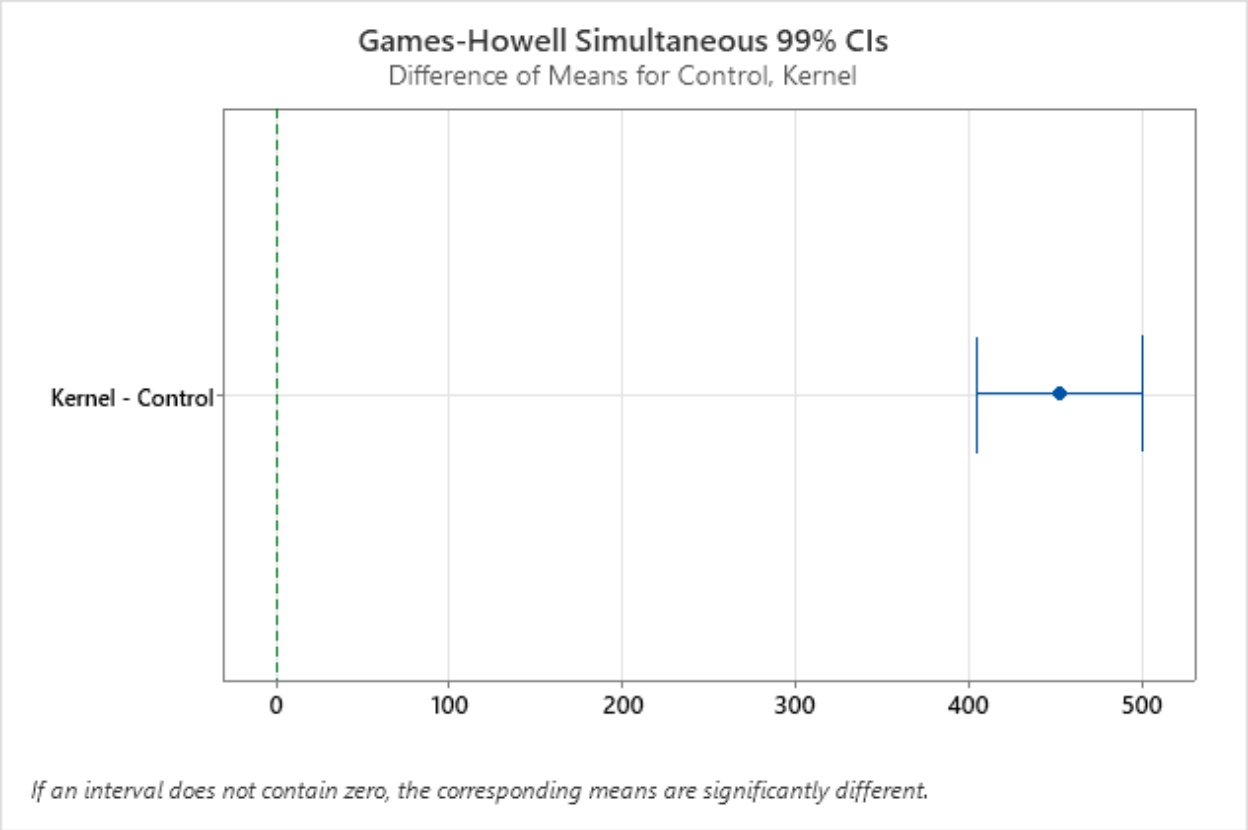

Figure 6. Graphical Representation of the Games-Howell 99% Confidence Interval for the Difference Between Kernel and Control

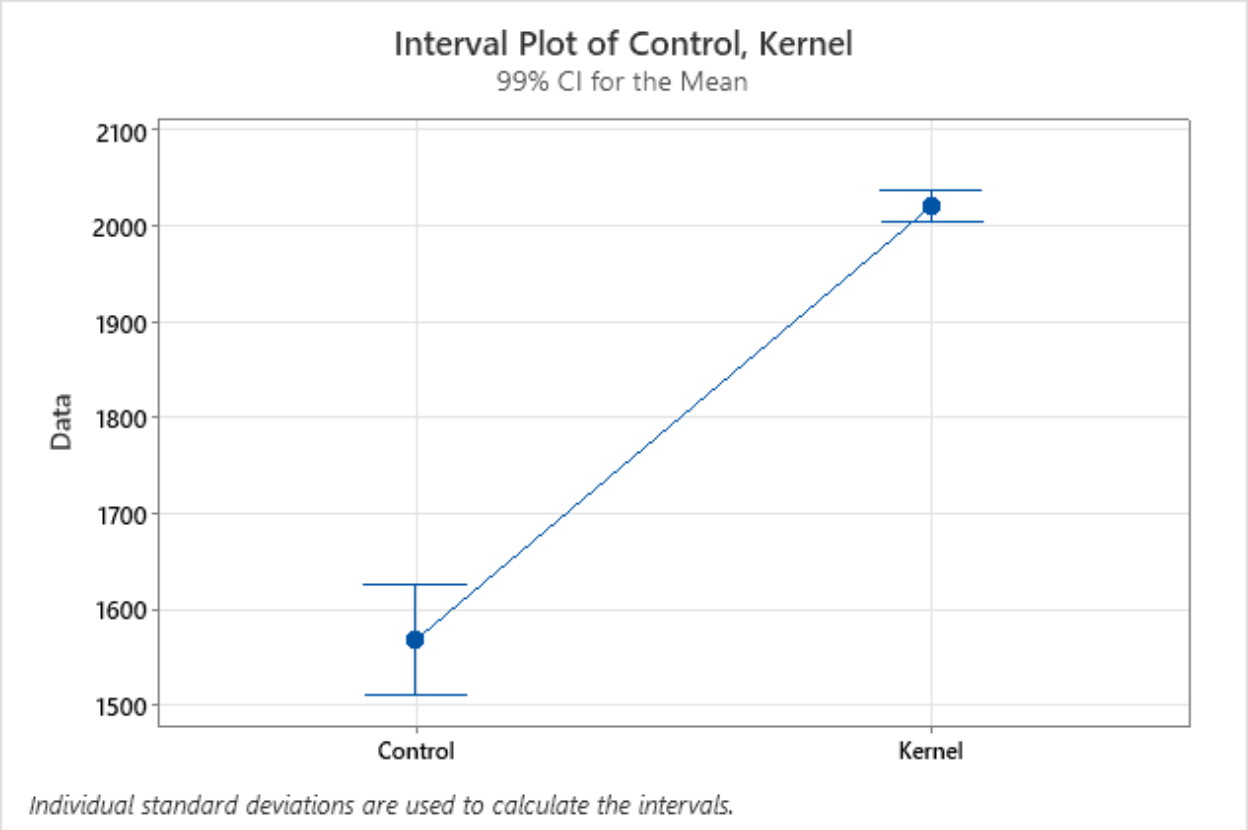

Figure 7. Interval Plot of the 99% Confidence Intervals of the Two Groups

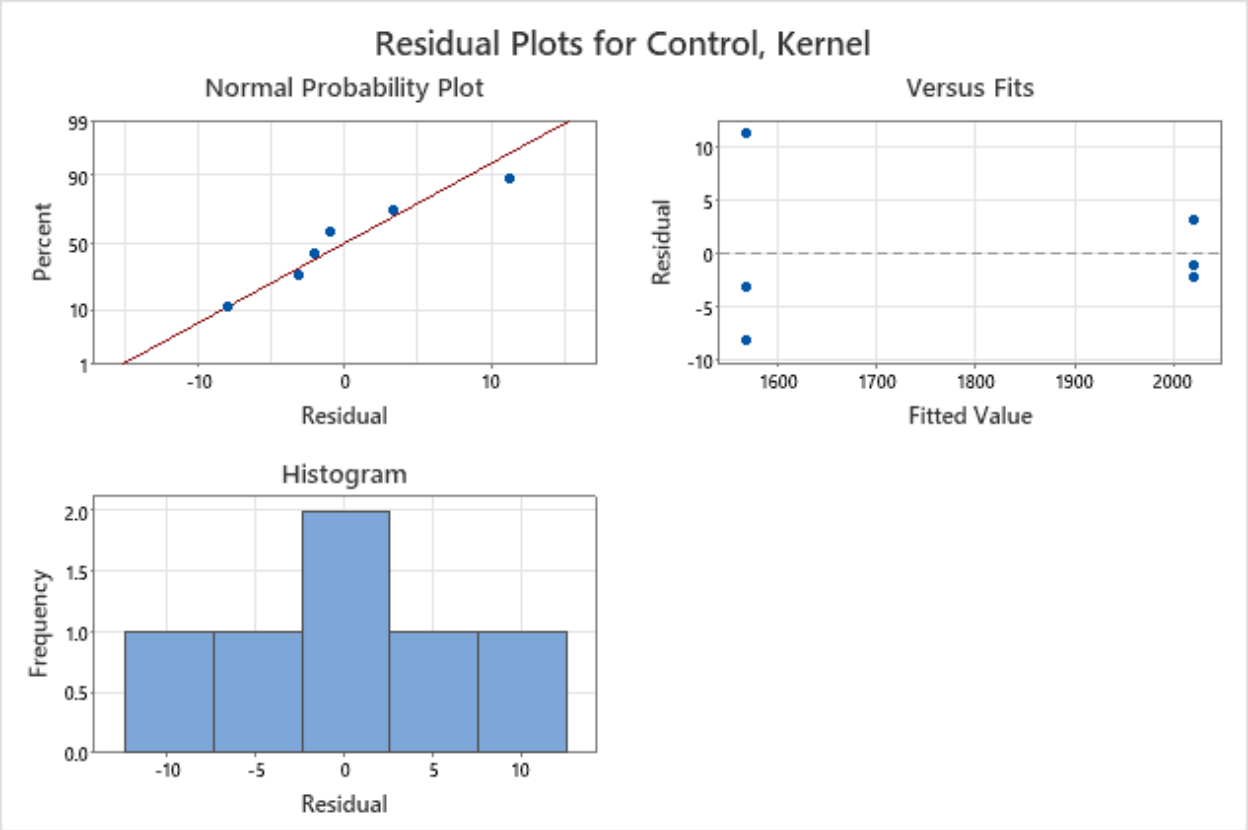

Figure 8. Residual Plots of the Control and Experimental Groups

Figure 9. Computer Used to Run Both the Control and Experimental Groups

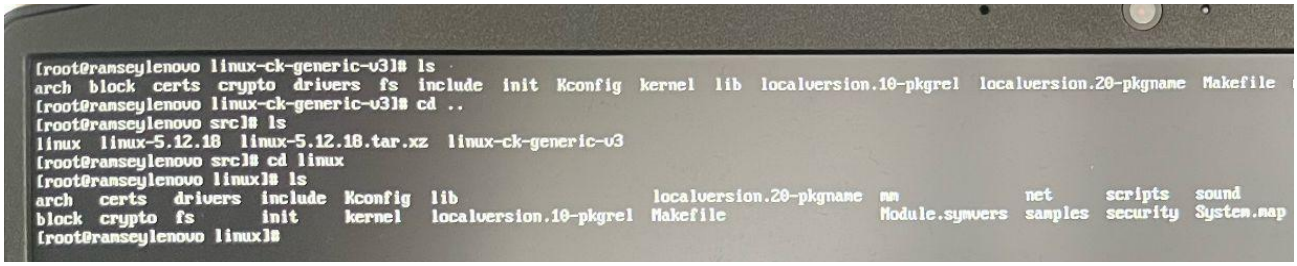

Figure 10. The Contents of the Control Source Directory

Figure 11. The Contents of the Experimental Source Directory

Figure 12. The Output of the “make” Command for Both Groups

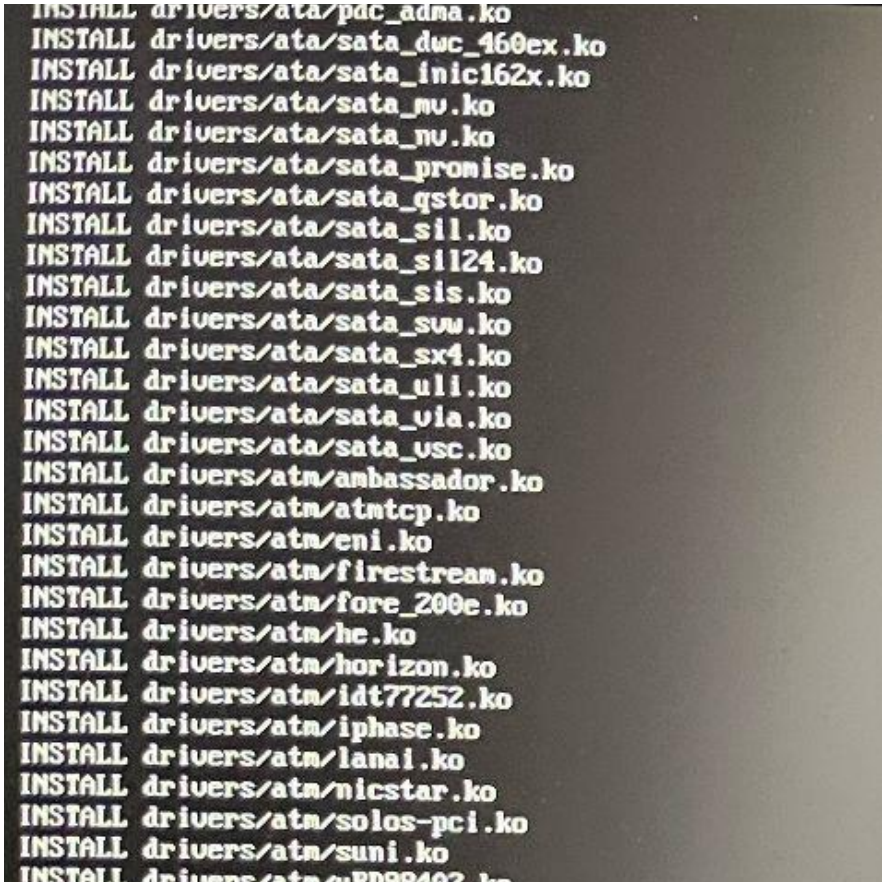

Figure 13. The Output for the “make modules_install” Command for Both Groups

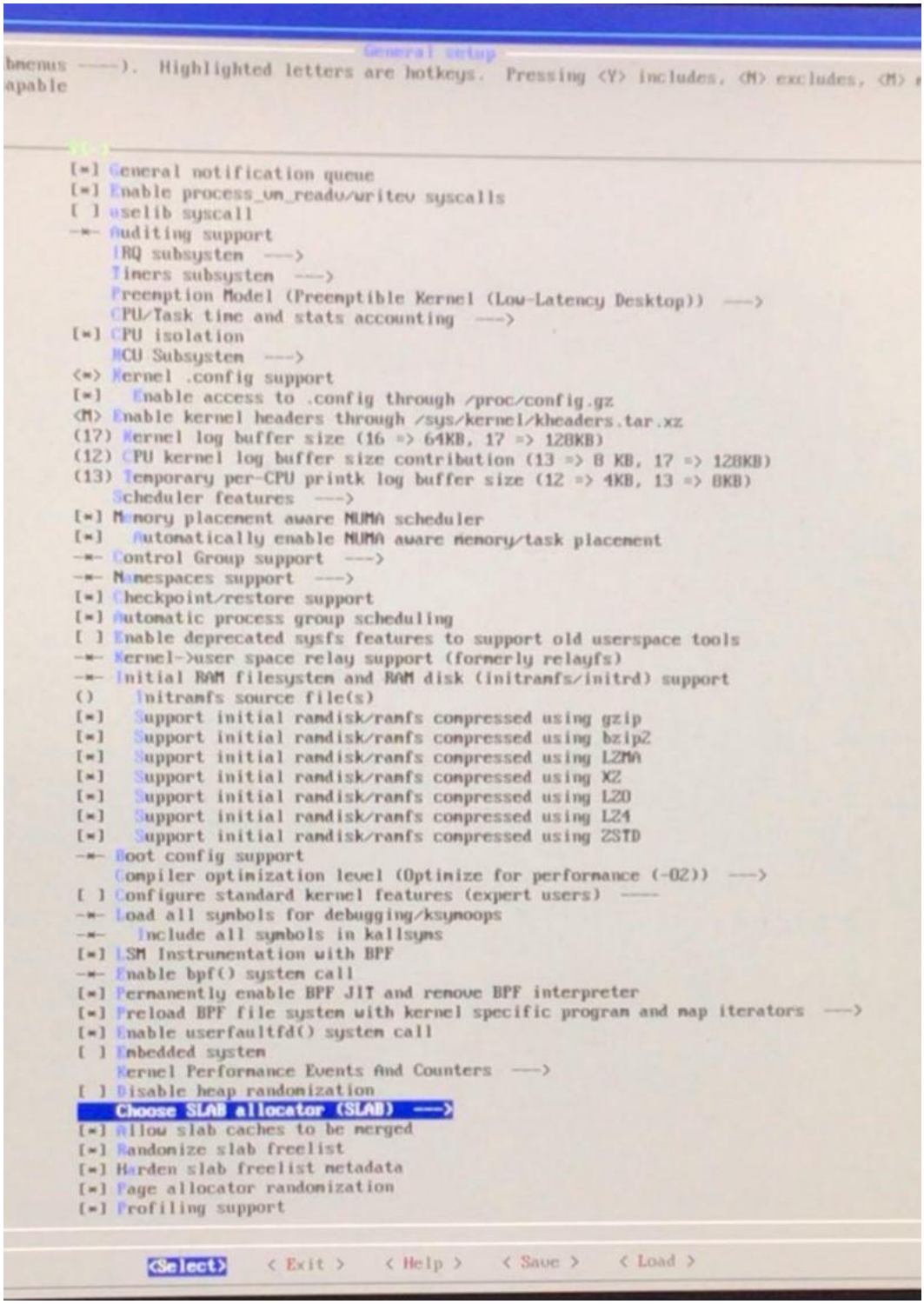

Figure 14. The SLAB Allocation Setting in the Kernel Configuration Makefile for Both Groups

Calculations

$$\text{Index} = \frac{10 * Result}{Baseline}$$

$$\text{Mean} = \bar{x} = \frac{Sum\ of\ data\ points}{Number\ of\ data\ points}$$

$$\text{Standard Deviation (Std.Dev)} = \sqrt{\frac{\sum_{i}^{n}(x_i - \bar{x})^2}{n-1}}$$

The tabulated data is taken directly from the output of the Unix Bench tests; there were no intermediate calculations performed by the researcher. The bash command used to generate the output was "ubench > *[output filename]*". The purpose of the angle bracket was to redirect the output to a text file. The command was run three times for three trials for both the experimental and control groups, and the output was recorded from the text file "*[output filename]"*. The indices for ten different benchmarking tests were recorded for each trial. The index results were calculated according to the above equation by the Unix Bench program during the execution of the bash command. "Result" is the observed value that a specific benchmarking test returned over a given test instance, while "Baseline" is a pre-recorded, fixed test result observed from a test computer owned by the Unix Bench development team. The Baseline parameter differs from benchmark test to benchmark test, but does not change over different instances of Unix Bench. Since benchmarking tests are only useful in comparing the relative performance of two machines, the choice of computer used to generate the Baseline parameters is completely arbitrary and has no effect on the results. The results of the machine used to generate the Baseline parameters were arbitrarily set as 10 index points as well. Several trials were performed to minimize the effect of random chance on the readings of Unix Bench. In addition, a one-way analysis of variance (ANOVA) test was conducted on the System

Benchmark Index Scores to assess the statistical significance of the observed results. In the ANOVA test, the experimental group is labelled the Kernel group.

Experimental System Benchmark Index Score Mean = $\frac{2023.6+2019.4+2018.3}{3}$= 2020.43

Control System Benchmark Index Score Mean = $\frac{1578.6+1559.3+1564.2}{3}$=1567.37

Experimental System Benchmark St.Dev = $\sqrt{\frac{(2023.6-2020.4)^2(2019.4-2020.4)^2(2018.3-2020.4)^2}{2}}$

= 2.80

Control System Benchmark St.Dev = $\sqrt{\frac{(1578.6-1567.4)^2(1559.3-1567.4)^2(1564.2-1567.4)^2}{2}}$

= 10.03

Factor	N	Mean	St.Dev	99% Confidence Interval
Control	3	1567.37	10.03	(1509.88, 1624.85)
Kernel	3	2020.43	2.80	(2004.41, 2036.46)

Table 1. One-way ANOVA Means

Difference of Levels	Difference of Means	SE of Difference	99% Confidence Interval	Adjusted P-Value
Kernel - Control	453.07	6.01	(405.20, 500.93)	0.00

Table 2. One-way ANOVA Games-Howell Simultaneous Tests for Differences of Means

Sample	Confidence Interval
Control	(4.09851, 200.517)
Kernel	(1.14269, 55.905)

Table 3. One-way ANOVA 99% Bonferroni Confidence Intervals for Standard Deviations

Source	DF	Seq SS	Contribution	Adj SS	Adj MS	F-Value	P-Value
Factor	1	307904	99.93%	307904	307904	5677.40	0.00
Error	4	217	0.07%	217	54		
Total	5	308121	100.0%				

Table 4. One-way ANOVA Analysis of Variance

Discussion

Conclusion

In this experiment, both the 5.13.12-arch1-1 kernel and an experimental kernel (with both unfair scheduling and SLAB allocation installed) were tested. The UnixBench program was used to benchmark the two groups and compare their performance. From the various tests in this program, the System Benchmarks Index Score was calculated, which represents how many orders of magnitude faster the test groups were when compared to a baseline metric (Kelly, 2018). The experimental kernel averaged a 2020.3 on the System Benchmarks Index Score, while the control kernel averaged a 1567.4. As verified by the one-way Anova, the 99% confidence intervals for these means do not overlap, which is evidence of a statistically significant increase in the performance of the experimental group over the control group.

Furthermore, the Games-Howell Simultaneous Tests returned a 99% confidence interval confirming the statistical significance of the increased performance of the experimental group over the control group, with a p-value of 0.00. The Bonferroni confidence intervals of the two different variances of the observed System Benchmark Index Scores do overlap, which is evidence that the two groups have similar variances. When taken along with the 0.00 p-value in the analysis of variances, the demonstrated normally distributed nature of the variances as per Figure 8, as well as the aforementioned conclusions from the analysis of means, there is much evidence to support that the increase in performance of the experimental group is statistically significant. The experimental group performed approximately 1.29 times as efficiently compared to the control group. This experiment provided evidence that the experimental kernel is more efficient than the default kernel, and that SLAB allocation and unfair scheduling are viable options to reduce bloat in the Linux kernel. Thus, the hypothesis was supported.

Applications

This experiment is of use to scientists through it's demonstration of the potential for SLAB allocation and Unfair scheduling to reduce software bloat in the Linux kernel. This experiment suggests that Linux kernel developers should consider implementing main-line versions of these subsystems into the Linux kernel in the near future. As demonstrated, doing so could have tangible performance benefits to most of the end-user base. Linux is deeply ingrained in modern internet infrastructure, making up a majority of the world's supercomputers and web servers (Linux Took Over). By improving the performance of Linux, the efficiency of modern internet communications is directly improved as well. Furthermore, the Linux kernel is embedded in all Linux-derived operating systems, like the Android smartphone operating

system, which increases the applicability of this experiment (Finley, 2016). Additionally, other operating systems beside Linux, including Windows, which also suffers heavily from bloat, have cause to implement both of these subsystems into their own operating system kernels as well as a potential way to reduce software bloat. This experiment has general and wide-spread applications that touch nearly all aspects of computing and our world's modern communication infrastructure.

Limitations

Due to the length of time required to run the UnixBench performance tests, only three trials per group could be collected. The experiment could be improved with more trials to increase the sample size in the tests for statistical significance. Furthermore, due to the costs associated with buying a new computer, only one computer architecture was able to be tested. This limits the general applicability of the results, as the experiment could theoretically produce different results across different CPU architectures (Weixiang, 2015). As the author of UnixBench notes, small changes in environmental software could affect the UnixBench program in disproportionate ways (Kelly, 2018). Thus, a system with similar hardware but slightly different software could perform differently. Overall, due to practical constraints there were many artificial limitations that had to be imposed on the experiment. Yet, seeing as the x86_64 architecture and standard ArchLinux environmental programs are standard across most Linux distributions, the experiment is still generally applicable and still offers new, valid insights into operating system theory and Linux kernel design.

Error Analysis

While the computer was kept in the same place for all benchmarking, experiencing the same workload, it is theoretically possible that the results were influenced by their environment. For example, the programs open in the background could have theoretically influenced the benchmarking performance results, though this is unlikely. The computer also underwent normal variations in temperature and noise. Additionally, there is the possibility of human error. The UnixBench program yielded vast amounts of data for every trial, and in the process of recording them the researcher may have misread a number or made a typo.

Future Analysis

It would be beneficial to this experiment's results to have the code peer reviewed by experienced programmers, to verify that the procedure used was logical and contained no errors. It would be worth investigating the performance of the described kernel subsystems on different operating systems, as the operating system is intimately involved with managing bloat. Additionally, since process scheduling and memory allocation are not the only sources of software bloat in the linux kernel, there may exist other subsystems that could be replaced with alternative implementations to mitigate bloat. Testing these subsystems would also be of use to the scientific community. Finally, running the benchmark tests on different kinds of hardware would be worthwhile as well.

References

Bhartiya, S. (2020, January 7). *Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd*. Linux.com. Retrieved June 8, 2021, from https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/

Definition of software bloat. (n.d.). PC Mag. Retrieved June 6, 2021, from https://www.pcmag.com/encyclopedia/term/software-bloat

Finley, K. (2016, August 25). *Linux Took Over the Web. Now, It's Taking Over the World*. Wired. Retrieved June 7, 2021, from https://www.wired.com/2016/08/linux-took-web-now-taking-world/

Gralla, P. (2009, September 22). *Linux creator: Linux is "bloated...huge and scary."* Computer World. Retrieved June 7, 2021, from https://www.computerworld.com/article/2467715/linux-creator--linux-is--bloated---huge-and-scary-.htmlhttps://www.computerworld.com/article/2467715/linux-creator--linux-is--bloated---huge-and-scary-.html

International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2021–2022. (2021). In *International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2021–2022*. Retrieved October 22, 2021, from https://sspcdn.blob.core.windows.net/files/Documents/SEP/ISEF/2022/Rules/Book.pdf

Lucas, K. (2018, January 29). *USAGE*. GitHub. Retrieved October 24, 2021, from https://github.com/kdlucas/byte-unixbench/blob/master/UnixBench/USAGE

Meyran, M. (2010, January). *Linux performance: is Linux becoming just too slow and bloated?* freesoftwaremagazine.com. Retrieved July 16, 2021, from http://freesoftwaremagazine.com/articles/linux_performance_linux_slow_bloated/

Palm Beach Regional Science & Engineering Fair 2021 -22 Handbook for Parents, Students, and Teachers. (2021, June). In *Palm Beach Regional Science & Engineering Fair 2020 -21 Handbook for Parents, Students, and Teachers*. Retrieved October 22, 2021, from https://docs.google.com/document/d/1xuVzBzjuXDSHCnCZbRaTxb3FxzrW0A-ohVoHqN7RAgo/edit#heading=h.t7hq47div9em

Process Schedulers in Operating System. (n.d.). geeksforgeeks.org. Retrieved July 16, 2021, from https://www.geeksforgeeks.org/process-schedulers-in-operating-system/

Society for Science and the Public (2016-17). International Science and Engineering Fair 2016-17: International Rules & Guidelines. Washington, DC: Society for Science and the Public.

Weixiang, C. (2015, December 14). *Linux: Linux Supported Architectures*. Retrieved June 9, 2021, from http://chenweixiang.github.io/2015/12/14/linux-series-03-linux-supported-architectures.html

What is Linux? (n.d.). Linux.com. Retrieved June 6, 2021, from https://www.linux.com/what-is-linux/

What is Memory Allocation? (n.d.). www.technopedia.com. Retrieved July 16, 2021, from https://www.techopedia.com/definition/27492/memory-allocation

What is the Linux kernel? (n.d.). Red Hat. Retrieved June 6, 2021, from https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel

2021-2022 RULES SUPPLEMENT To the International Science and Engineering Fair Rules. (2021, October). In *2021-2022 RULES SUPPLEMENT To the International Science and Engineering Fair Rules*. Retrieved October 22, 2021, from https://drive.google.com/file/d/1z1k5HELlCGTCjpUZAaBKPkmoDwWPmGji/view