Concurrency Across Different Languages

William Archbold   
*University of Colorado Denver*  
Denver, CO  
[william.archbold@ucdenver.edu](mailto:william.archbold@ucdenver.edu)

Kerry Gip  
*University of Colorado Denver*  
Denver, CO  
[kerry.gip@ucdenver.edu](mailto:kerry.gip@ucdenver.edu)

Samantha Melconian  
*University of Colorado Denver*  
Denver, CO  
[sam.melconian@ucdenver.edu](mailto:sam.melconian@ucdenver.edu)

Odyssey Villagomez  
*University of Colorado Denver*  
Denver, CO  
[odyssey.villagomez@ucdenver.edu](mailto:odyssey.villagomez@ucdenver.edu)

*Executive Summary*— This paper is the final report for the CU Denver Spring 2022 CSCI 5573 Rocketing Rockets’ class project. Each section (i.e. I, II) of this paper reflects a required section of the assignment guidance.

Keywords— multiprocessing, multithreading, concurrency, processors, Windows, Linux, Ubuntu, FreeBSD, C++, Java, Golang, Python

# Introduction

## Motivation

We want to study how an operating system handles concurrency and multiprocessing while varying the programming language creating the threads. We plan to create four separate implementations of a multiprocessing program in four different programming languages: C++, Python, Java, and Go. We will run each of those programs on Windows, FreeBSD, and Linux Ubuntu. We will compare the performance of each program within each operating system and measure the behavior of the operating system as it manages the processes created from each program written in the differing languages. We hypothesize that since Go can handle specific concurrency problems [1], it will perform the best out of the four languages. This project will help us to better understand the relationship between a programming language and an operating system. It will also give us the opportunity to explore the details of concurrency and the different ways that it can be implemented. We will gain a deep understanding of how an operating system handles multiple processes at once, including the operating system’s use of context switching algorithms.

## Background

In a concurrent process on a single processor, the processor can switch execution resources between processes and threads, resulting in concurrent execution. Concurrency indicates that more than one process is making progress, but the processes/threads are not actually running simultaneously.[1] While concurrency can lead to powerful increases in performance by maximizing CPU utilization, it also has the potential to create surprising or unexpected results. This happens because threads share access to the same resources—for example, memory—and the order in which each of the threads executes is nondeterministic. Languages provide various tools and libraries that allow for safe implementations of concurrency. These tools allow for performance improvements associated with concurrency without the danger of nondeterministic memory accesses. This paper will focus on concurrent performance across different languages in different operating systems.

Concurrency deals with managing the access to shared state from different threads and on the other side, parallelism deals with utilizing multiple CPUs or its cores to split a task across the cores. [2] Concurrency means that two or more calculations happen within the same time frame and parallelism means that two or more calculations happen at the same moment. Since parallelism requires multiple cores, it is a form of concurrency.[3]

Each thread in the process can run concurrently on a separate processor, resulting in parallel execution. When the number of threads in a process is less than or equal to the number of processors available, the operating system's language and thread support system ensures that each thread runs on a different processor. For example, in a matrix multiplication that is programmed with four threads and runs on a system that has two dual-core processors, each software thread can run simultaneously on the four processor cores to compute a row of the result at the same time. [4]

There are three levels of concurrency, low level, mid-level and high-level concurrency. Low-level concurrency involves atomic operations, mid-level does not use atomic operations but requires locks and high-level concurrency does not use atomic operations or locks. We will be focusing on high level concurrency. [2]

# Methods

## Software System Architecture and Main Modules Being Developeed

Each team member will write the same multi-process program in one of the four listed programming languages. The team members will coordinate the functioning of the programs to eliminate any bias from choice in code. For example, they will all need to call the same number of processes and each program’s newly created processes needs to perform the same functions. Each of us will have one million random numbers to sort through using merge sort. Each process will take a copy of an array of integers and merge sort it. We will be measuring each child process’ execution time, CPU time, CPU utilization, system memory usage, context switches, page faults, and peak memory usage. Each of the programs will be using the time command on Linux/FreeBSD and the deprecated, yet comparable command timeit.exe on Windows to measure the results.

## Measurements

* These are the metrics we will use to compare the performance of each programming languages and operating systems.
* Processes execution time
* CPU process time
* Percentage of CPU utilization
* Average system memory
* Number of context switches (voluntary and involuntary)
* Max resident set size
* Page faults (major and minor if applicable)
* Number of cores
* CPU type (will vary across team members)

## Intended Code

Below is the pseudocode for a program that can be implemented in multiple languages and run on multiple operating systems.

main()

randomNumbersArray = Read in file of a million random numbers

cores = Get number of cores on processor()

SpawnSortProcesses(*n* cores) //this will evenly distribute each process to each core

SpawnSortProcesses(2\*cores) //this will create double the number of processes than cores forcing the OS to start juggling processes according to its respective scheduling algorithm

SpawnSortProcesses(4\*cores) //further stress the OS. can keep calling the function with higher multiples to see how far we can stress the systemSpawnSortProcesses(x):

sort times = array //to store times each sub process takes to sort random numbers

from 0 to x:

timestart() //start timer to see how long subprocess takes

sorted array = exec(mergeSort, randomNumbersArray) //spawn 1 process per core that will sort randomNumbersArray

timestop() //stop timer

//place timestop - timestart into times array

return times

mergeSort():

//Basic mergeSort program code that can be found on GeeksForGeeks or one similarly written ourselves to accommodate our code.

This program is simple and effectively provides insight into how each operating system performs with each language. The basic idea of the program is to spawn many processes that will stress the operating system. We will follow the performance of the operating systems as the number of processes increases and the operating system becomes stressed. Initially, the program will create a process for every core in the central processing unit. Every process has at least one thread, so each process will be executed on a separate core. Each process will perform the same mergeSort() task on an array of the same random numbers read in from a file. Initially, the main program will call a function called SpawnSortProcesses(), which calls exec() for a number of processes equal to the number of cores and time how long each process takes to complete. This forces the operating system to schedule each process on a different core and run in parallel. The program will continue to call SpawnSortProcesses() but double the number of sub processes spawned each time so that t ratio of processes to cores keeps doubling than there are cores to handle them simultaneously. This will force each operating system to use its scheduling algorithm and perform context switches.

Though the pseudocode above shows how the program can measure the time each process takes to complete, additional measurements will be taken using Windows timeit.exe command and Linux/BSD time command For example, in Linux, there are numerous sources of information on processes and are shown in Figures 1-3 in section IX Figures.

As per Figures 1-3, Linux stores a copious amount of information for any given process in the /proc directory. Additionally, more information can be retrieved from the C library function system(). Unlike Windows10, Linux has a native time executable called time, which measures the performance of another process as shown in Figure 4 in section IX Figures. Time provides a lot more information than timeit in Windows10 does to include breaking down page faults into major and minor page faults and voluntary and involuntary context switches whereas Windows10 only provides one statistic for all page faults and all context switches respectively shown in Figure 5 in section IX Figures .

## Environment – Operating Systems, Machines, and Programming Languages

This project will use Windows, Linux and FreeBSD operating systems. Each member will standardize Oracle VM VirtualBox machine to host all three operating systems. Available memory and the number of cores will also be standardized. Each group member will write their respective program and then run all of them on their respective machines. A machine with ample memory and multicore performance will be needed because the project will measure whether performance changes as the amount of memory used by each process and number of cores change. The default settings on VirtualBox during installation are to have at least 1024 MB of RAM and 16 GB of Hard Disk Space. We will be using the default settings for each of the operating systems on VirtualBox as a means of standardizing the procedures and to control any outliers. A base description of each of the operating systems is below. XV6, a Unix based operating system was also considered but then discarded due to lack of compatibility with anything other than C/C++.

Windows is one of the most common household desktop/laptop operating systems with a 73% ownership as of June 2021 [5]. It was created by Microsoft in 1975 by Bill Gates and Paul Allen.[8] Unlike Linux and FreeBSD, Windows is a closed source system, and you must purchase a license to be able to run it on your hardware. Windows 10 will be used for this experiment. Windows 10 uses the Multi-level Feedback Queue job scheduling algorithm, which will be important when we analyze process execution time. Each of the authors has Windows installed natively on their main computers

FreeBSD is an open-sourced Unix-like operating system that was developed by the University of California, Berkeley [6].[12] FreeBSD compiles using its native compilers[10]. FreeBSD focuses on performance, network features and many others. It was publicly available in the 80s and has had many open-source contributors to make it the system that it is now. FreeBSD is now scalable; it contains management tools as well as security systems. It can host major websites thanks to the contributions of individual developers [6]. FreeBSD uses the ULE job scheduling algorithm. FreeBSD version 13.0 is the most recent one and is used for this experiment.

Linux is popular and the largest open-sourced operating system project in the world [7]. It compiles using its native compiler [8]. Linux is derived from Unix and has evolved to run on several diverse types of hardware. Many developers have contributed their ideas to the Linux kernel and have given it back to the community. Because of that Linux is extremely diverse and has multiple different distributions of Linux and is extremely customizable. Different distributors of Linux include Red Hat, Ubuntu, and Mint, to name a few. The creator of Linux is Linus Torvalds, and he owns the trademark. Linux uses the Completely Fair Scheduler to schedule its jobs. Linux Ubuntu version 20.04 LTS was used for this experiment.

# Group Member Roles

Each team member will download the proper ISOs for both operating systems and install them on Oracle’s VM VirtualBox. They will write the same program in their respective language and collaborate with their fellow team members to ensure the program is standardized using the same functions and process production. An analysis of how each operating system handles the program will be done. Each team member will also record the individual processor they are using. This will provide another opportunity for comparing future results.

The team members will need to have a thorough understanding of how all the operating systems function to make the tests sensitive to changes in performance.

# implementation

The below sections will describe how each language was implemented.

## C++

An AMD Ryzen 3900 12-Core Processor 3.1 GHz processor was used to implement the C++ programs.

C++ is a compiled language that doesn’t require a virtual machine as Java or Python does to intermediate between the code and the operating system. Therefore, implementing our pseudocode in C++ across three different operating systems sometimes required small changes to the code in order to make the code compatible with the specific, hosting operating system. The rest of this section will describe how gathering C++ results had to change across the OS’s.

Windows10 was the hardest operating system to implement a C++ solution on. Unlike with FreeBSD and Ubuntu Linux, there is no equivalent /proc folder storing process information that a user can investigate for gathering performance metrics. Instead, the user has to rely on a program called Process Monitor. Alternatively, the user can gather metrics about performance about the C++ sort program by using a deprecated executable called timeit. timeit isn’t native to Windows10 and required some difficult web searching to find an installation package, but it ultimately provided valuable information as shown in Figure 5 shown in section IX Figures.

Windows10 also required some dramatic changes to the C++ code implementation. With Linux and FreeBSD, we were able to use the unistd.h library to gather system information such as the number of cores or the parent process ID, but that library isn’t available in Windows10. Getting the parent process ID required the process.h, windows.h, tlhelp32.h, sysinfoapi.h library headers. Whereas with Linux based OS’s, you simply call getppid(), we had to implement the below code in Figure 6 to get the parent process ID shown in section IX Figures.

Lastly, in order to spawn threads to call the merge sort executable numerous times, we could not use fork() calls, but rather had to use thread instantiations as shown in Figure 7 shown in section IX Figures.

Implementing the C++ solution in Linux for Ubuntu allowed us to gather additional operating system performance information. Unlike Windows10, Linux has a native time executable called time, which measures the performance of another process as shown in Figure XX. Time provides a lot more information than timeit in Windows10 does to include breaking down page faults into major and minor page faults and voluntary and involuntary context switches whereas Windows10 only provides one statistic for all page faults and all context switches respectively.

We were also able to gather command line snapshots of operating system performance by calling htop and ps intermittently while the numerous merge sort executables ran. This could only be done on Windows10 by looking at the Process Monitor GUI, which isn’t conducive to saving and analyzing data. Figure 8 shown in section IX Figures shows the results of htop, with one of the machines used for testing has all 12 processors running at 100% utilization executing numerous merge sort programs in parallel. Figure 9 shown in section IX Figures further confirms that each processor is running not only an individual mergeSort() but having to perform context switches between different merge sorts because the “PSR” column shows the same core being utilized more than once for multiple merge sorts.

Implementing our C++ solution for FreeBSD followed a more similar path to our Linux implementation than to our Windows10 implementation. Though we could not find any good system monitor for taking snapshots of operating system stress as we could with Process Montior or htop, we still could use a time executable (see Figure 10 shown in section IX Figures) to measure merge sort individual process performance. Unfortunately, unlike the Linux version, there was no verbose option for detailed output.

## Python

An AMD Ryzen 5 3600 6-Core Processor 3.59 GHz processor was used to implement the Python program.

Python has three modes for concurrency, multiprocessing, threading and asyncio. The main library that was used is the multiprocessing library. The package allows for local and remote concurrency [9] . Multiprocessing allows the user to fully leverage multiple processors on any machine or OS. Multiprocessing manager and multiprocessing pool are the two classes used. Pooling is a convenient way to execute a function across multiple processes. Map asynchronous is used in conjunction with pool to ensure that the function applies to every iterable item. It allows the results that are completed to be processed immediately in the next step, allowing for concurrency. Multiprocessing manager is a method that allows processes to share resources and run concurrently, bypassing the Global Interpreter Lock found in Python threads, which forces one native thread to run at a time. Due to python being an interpreted language, many of the Linux commands were not compatible with the program and thus the library psutil was used to capture a lot of the internal data, such as page faults, CPU time, Resident Set Size, Context Switches, etc. However, psutil was limited to the data that it can capture across different OS’s. Windows allowed the most data while Linux and FreeBSD were limited. Windows timeit.exe and Linux/BSD version of time were used to capture similar results of the whole program and psutil was used to track the results inside of the program.

The Python multiprocessing style guide recommends placing the multiprocessing code inside the \_\_name\_\_ == '\_\_main\_\_' instead of a regular main() function. This is due to the way the processes are created on Windows. The guard is to prevent the endless loop of process generations and to protect the script [3]. A command line argument function was used to allow the user to dictate how many processes should be spawned. Next the file of 1 million random numbers is read. A normal unsplit merge sort was run to guarantee that the merge sort and multiprocessor merge sort correctly and that both arrays were equivalent. The program then uses the number that the user entered to create that many number of processes. The merge sort array is then split into that number of processes to ensure concurrency. Each process takes a section of the merge sort and then sorts that section before merging it back together into the original array. A process pool function was used to create a process block so that all processes may run. Psutils was used to observe the output of the process time and completion, but the timeit and time functions on Windows and Linux/BSD respectively were used to actually measure the entire process.

## Java

An Intel® Core™ i7-6500U CPU 2.5GHz 2.59 GHz processor was used to implement the Java program

The main Java program takes in 1 parameter, the number of processes the user would like to generate.

The Java Process Builder class was used to generate N new processes from within the main Java program. This class provides an interface to the developer to create and manage child processes from within a Java application. [10] The command() function was called to specify the separate Sort.java file that each newly created process would run. Then each process was started using the start() method. The Sort code would then call MergeSort on an array of 1,000,000 random integers. [11] In the Sort code that each process runs, the Java Management Factory class was called to generate statistics unique to the child process. The Management Factory class manages the instance of the process and provides the amount of physical memory used, process execution time, CPU use as a percentage, and total memory available to this process. Once the process completes the sort function, it would end, and control would return to the parent process. After all processes are finished, the main process would print statistics on the parent process execution time, memory use, and CPU use as a percentage. Java does offer built in synchronization techniques for concurrency, like locks, but locks were not covered in the scope of this project, as the performance of each language was left to the operating system and underlying language functionality. In addition to the built in Java process information, the Operating System specific timeit and time commands were used to provide system wide information on the main process, such as the total context switches, and page faults that happened during the execution.

## Go

Go was created by Google. It was designed to be quick to write and easy to read (like Python) but also efficient to run (like C++). Go is a statically typed, compiled language like C++; however, unlike C++, its powerful compiler allows for it to be easily compiled across different platforms. For the purposes of this project, the same process-spawning code was run on all three platforms without any changes necessary. The code was run on a machine with Intel® Core™ i5-8350U CPU @ 1.70 GHz 1896 Mhz processors.

Go offers a powerful concurrency framework based off of “goroutines” that avoids spawning processes unless absolutely necessary. In order to force Go to spawn new processes (instead of lighter goroutines), the program made exec() calls to the operating system.

To obtain runtime metrics (such as CPU time and memory usage), a GitHub repository called “gopsutil” was used. Gopsutil was created by a user named Shirou and is a port of Python’s psutil to Go. Gopsutil was compatible with all three operating systems, so it was used on all those three to collect metrics.

# Results & Analysis

## General results

Results were gathered from different computers with varying numbers of cores. In order to standardize results, results were computed using spawned processes as a percentage of number of cores. For example, a user with four cores would first run their analysis using 4 spawned processes which equaled 100% of the number of cores. That user would then double the number of spawned processes to eight to get 200% use of the number of cores. A different user, say with 8 cores, would first spawn 8 processes to get the same percentage as the 4-core user and then 16 processes to get the same 200%. Tables 1 through 3 are shown enlarged in Section X Tables. They show the raw data results for each operating system broken down by coding language and processes as percentages of cores. Section XI Graphs show the comparison of the different languages across different metrics.

The metrics that were used to measure concurrent processes were program execution completion time, CPU process time, percentage of CPU utilization, average system memory, number of context switches (voluntary and involuntary), max resident set size, page faults (major and minor if applicable), number of cores and CPU type. Execution speed was a necessary metric to determine how well each language performed on each operating system. CPU process time is also a necessary metric to see how much time the CPU was used for processing instructions. The average system memory measures the total memory obtained from the OS for running the processes. Context switches is also an important measure of concurrency. Voluntary context switches means that a process willingly gives up its space to another process and involuntary means that the system scheduler actively suspends or saves the state of a process before the process has completed. Max resident set size (RSS) is the total amount of physical memory assigned to a process that is held in main memory (RAM) [12]. Page faults occur when a program attempts to access data or code in the address space, but it is not currently located in RAM. Major faults occur when the required page is not in RAM and minor faults occur when the page is present but not properly mapped. The number of cores was an important necessity to stress the system and see if performance is hindered by over allocating processes to cores. CPU type is also another metric to be considered since each CPU will have a different architecture and specifications but was not explored within the scope of this project. Each of the individual performance of the language will be explained in further detail below.

## C++ Results

C++ code performed in a predictable manner across Linux and FreeBSD and also in line with the majority of the languages when applied to those operating systems. Context switches increased and total wall time (process start time to process finish time) halved as the number of processes doubled, which was expected. An interesting statistic was the number of major page faults. There were more page faults at the first run of the program than later runs of the program probably because the data with the correct page wasn’t loaded yet for the first run, but by the second run it was already available, which creates some bias favoring later runs. However, C++ execution time was significantly higher in Windows10 from both an absolute perspective and relative perspective when compared to the other languages. The most significant factor that probably contributed to this was the compiler rather than the operating system used. Both FreeBSD and Ubuntu Linux come with the g++ compiler pre-installed. Windows10 doesn’t come with a pre-installed compiler. Microsoft’s Visual Studio (which was used in all three operating systems to write the code) Code’s webpage recommends installing the clang compiler for C++ compilation on Windows so that was what we used. C++ is available for windows, but it needs to be downloaded from a third-party website, [www.MSYS2.org](http://www.MSYS2.org), but downloading from third parties introduces unverified sources and security risks.

## Python Results

Python performed surprisingly well. It was expected that python would be one of the worst performers due to the fact that it is an interpreted language. The few things that stood out were that Python consistently utilized 100% of the CPU across all three operating systems. That is due to the multiprocessor library that was used in the code. The multiprocessor library spawns multiple interpreters for each of the number of processors that the user inputted. This monopolizes the CPU. In Ubuntu Linux, Python had the highest minor page faults and no major page faults. This is probably since python is running concurrently, it is looking for similar memory in spatial locality. In FreeBSD, it had similar execution time and CPU time, which showed that the code was the most efficient on FreeBSD. It also had one of the highest involuntary context switches as the system was stressed up to 400% as well as the highest number of page faults. Page faults can increase as the number of concurrent processes exist and the number of context switches occur. A context switch is when a CPU process switches from one task to another. One process saves state and then becomes suspended by the CPU and another process executes. That process is running a similar program but needs a different type of memory bit access, this causes an increase in page faults. As the number of processes increases, the number of page faults and context switches should also increase. In Windows, Python had a considerably high execution time, but its CPU time was on par with the other languages. This might be due to the high overhead when the user running the program has a lot of background processes. It also had the highest average system memory and almost the lowest number of page faults

## Java Results

Java's execution time was the highest across all languages on both the Linux and FreeBSD operating systems. On Windows, Java had the 2nd lowest execution time.

It is expected that the interpreted implementation of Java through a Java Virtual machine and additionally the Oracle Virtual Machine used to run Linux and FreeBSD, would have a larger execution time when compared to the compiled C++ and Go languages. When compiled, the Go and C++ languages optimize the code to consume fewer resources like Memory, and the resulting machine code can be directly run on the CPU. [13] When interpreted, Java has the additional translation step, which adds extra overhead and worsens the execution time. [14]. Java’s worse execution time compared to Python, which is also translated, can also be accounted for by looking at the amount of memory utilized by each language. Java consistently used more memory resources than Python, except for Windows, where Java performed better than Python in regard to execution time. Because Java accessed memory more often than Python, it’s execution time and memory use were worse. The Java program also used more CPU time, showing that in regard to CPU and memory, Java used more resources.

It seems that the Windows operating system was also able to more effectively handle the increasing processes spawned for the Java program, seen by how well Java performed on the Windows system compared to FreeBSD and Linux using execution time and memory use, regardless of the number of processes spawned. The resources (memory, and CPU) were more balanced in the Windows operating system.

In regard to context switching, Java had the least context switches on the FreeBSD platform, but the FreeBSD operating system used almost 6 times as much Memory, for example 340 MB compared to Python, Go, and C++ 21.59 MB, 43.86 MB, 61.89 MB use respectively. On Windows, context switches were positively correlated with the number of processes created, and on Linux, context switches were high across the board regardless of number of processes spawned.

It is interesting that Java performed the best on the Windows operating system, which I believe can be accounted for by considering that the Java Windows code was run natively on a Windows machine while FreeBSD and Linux ran Java through a Virtual Box. The Virtual Box doesn’t have access to as many resources as the native Windows operating system, like memory.

## Go Results

Across all three operating systems, Go had a lightning-fast execution time. While we expected that Go would have the fastest execution time, it was surprising just how much faster it was compared to the other programming languages. One possible explanation for this is that slices were used to efficiently index subsets of the array during MergeSort(). Another explanation for the fast execution time could be that Go is statically typed.

On Windows10, there was a positive correlation between the number of processes spawned and most of the metrics measured: average execution time, turnaround time, average memory usage, context switches, and page faults all increased linearly with the number of processes. This shows that Windows was able to effectively manage the increased process load without noticeable thrashing.

On Linux, there seemed to be a sharper increase in execution time and turnaround time as the process load increased. CPU utilization at a low process load was at 5%, however at a high process load utilization dropped to 1.18%. We also saw the number of major page faults and involuntary context switches increase with the increase in process load. This demonstrates an observable difference between Windows’s multi-level feedback queue and Linux’s fair scheduler: Windows seems to scale a little more elegantly when managing multiple processes.

On FreeBSD, the program had the longest execution time and the lowest CPU utilization. The number of page faults and context switches on FreeBSD was a little higher than that on Linux, but it was much lower than that on Windows. FreeBSD also uses a multi-level feedback queue, but it seems that Windows’s scheduler is more efficient than FreeBSD’s (despite the 170-fold increase in the number of context switches executed on Windows).

# Conclusion

## Go was consistently the fastest language by both CPU time and Execution time. Go has a CSP (communication sequential process) paradigm that is designed for efficient concurrency. This means that Go processes do not require the management of synchronization primitives, as the processes communicate through channels rather than shared memory. The CPU time was constant. Execution time grows as we increase processes. CPU % utilization was proportional to the ratio of threads to cores. Python’s built in multiprocessing module spawns multiple interpreters so it monopolizes the CPU and forces 100% CPU utilization. Voluntary context switches had stagnant growth while involuntary switches increased with more processes. Different operating systems use different scheduling algorithms which likely contributed to some of the difference in performance for the same language on different machines.

## Some of the physical limitations when running each of the individual languages involve the number of cores that each CPU has and the different type of CPU that was in each computer. AMD CPU’s have been known for better multithreading whereas Intel CPU is more geared towards throughput and power.

A likely significant contributor to the differences in performance is the use of different scheduling algorithms across the operating systems. Ubuntu uses Completely Fair Scheduler (CFS), but also allows for customization of scheduling. FreeBSD uses ULE, which is associated with processor affinity and sets priority based on interactivity. Windows10 uses multilevel feedback queue (MLFQ), which prioritizes assigning processes based on CPU burst characteristics.

# Future Work

Future work that has been considered is standardizing the CPU’s used. Every individual has a different CPU installed on their computers and that has a potential to affect performance of each of the languages concurrent processes on the operating systems. A consideration would be to determine the performance between same type of CPU’s with different GHz. As we have a range from 1.59 to 3.59GHz with both AMD and Intel which allows the CPU to interpret instructions. Additional work could include manipulating Ubuntu’s scheduling algorithms. The chrt utility specifically can change Ubuntu’s scheduling policy.

# Bibliography

|  |  |
| --- | --- |
| [1] | "Multithreading Concepts," Oracle, [Online]. Available: https://docs.oracle.com/cd/E19253-01/816-5137/mtintro-25092/index.html. [Accessed 22 February 2022]. |
| [2] | "Concurrency vs Parallelism," TutorialsPoint, 2022. [Online]. Available: https://www.tutorialspoint.com/concurrency\_in\_python/concurrency\_in\_python\_concurrency\_vs\_parallelism.htm. [Accessed 2 May 2022]. |
| [3] | "Python Multiprocessing," ZetCode, 29 Nov 2021. [Online]. Available: https://zetcode.com/python/multiprocessing/. [Accessed 2 May 2022]. |
| [4] | "Multithreading Concepts," Oracle, 2010. [Online]. Available: https://docs.oracle.com/cd/E19253-01/816-5137/mtintro-25092/index.html. [Accessed 2 May 2022]. |
| [5] | "Desktop PC operating system market share worldwide, from January 2013 to June 2021," Statista, Jan 2022. [Online]. Available: https://www.statista.com/statistics/218089/global-market-share-of-windows-7/. [Accessed 2 May 2022]. |
| [6] | "FreeBSD The Power To Serve," The FreeBSD Project, 1995. [Online]. Available: https://www.freebsd.org/. [Accessed 31 03 2022]. |
| [7] | "What is linux?," Red Hat - We make open source technologies for the enterprise, 27 02 2019. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-linux. [Accessed 31 03 2022]. |
| [8] | R. Cox, F. Kaashoek and R. Morris, "Xv6, a simple Unix-like teaching operating system," Massachusets of Institute of Technology, 22 09 2021. [Online]. Available: https://pdos.csail.mit.edu/6.828/2012/xv6.html. [Accessed 31 March 2022]. |
| [9] | "Python," Python Software Foundation, 26 April 2022. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes. [Accessed 30 April 2022]. |
| [10] | "Class ProcessBuilder," Java Platform, May 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html. [Accessed May 2022]. |
| [11] | "Merge sort using ArrayList in Java with example," 27 July 2017. [Online]. Available: https://www.withexample.com/merge-sort-using-arraylist-java-example/. [Accessed May 2022]. |
| [12] | "What is the maximum resident set size," Quick Adviser, 1 Feb 2020. [Online]. Available: https://quick-adviser.com/what-is-the-maximum-resident-set-size/. [Accessed 8 May 2022]. |
| [13] | "Code Optimization in Compiler Design," 03 Jul 2020. [Online]. Available: https://www.geeksforgeeks.org/code-optimization-in-compiler-design/. [Accessed May 2022]. |
| [14] | Jakub, "Compiled vs Interpreted Programming Languages," 27 May 2020. [Online]. Available: https://finematics.com/compiled-vs-interpreted-programming-languages/. [Accessed May 2022]. |
| [15] | A. Thomasian, "Concurrency Control: Methods, Performance, and Analysis," *ACM Computing Surveys,* vol. 30, no. 1, pp. 70-199, 1998. |
| [16] | R. Agrawal, M. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems,* vol. 12, no. 4, pp. 609-654, 1987. |
| [17] | "Microsoft.com," 2016. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/05/chapter3.pdf. [Accessed 1 April 2022]. |
| [18] | "Two Phase Locking Protocol," GeeksforGeeks.org, 2 June 2021. [Online]. Available: https://www.geeksforgeeks.org/two-phase-locking-protocol/. [Accessed 1 April 2022]. |
| [19] | D. Poudel, "Our Tech Room," 8 Nov 2021. [Online]. Available: https://ourtechroom.com/facts/history-microsoft-windows-os/. [Accessed 30 Apr 2022]. |
| [20] |  |

# Figures

A picture containing graphical user interface

Description automatically generated

Figure 1. Linux output of top command



Figure 2. Linux directory of process information

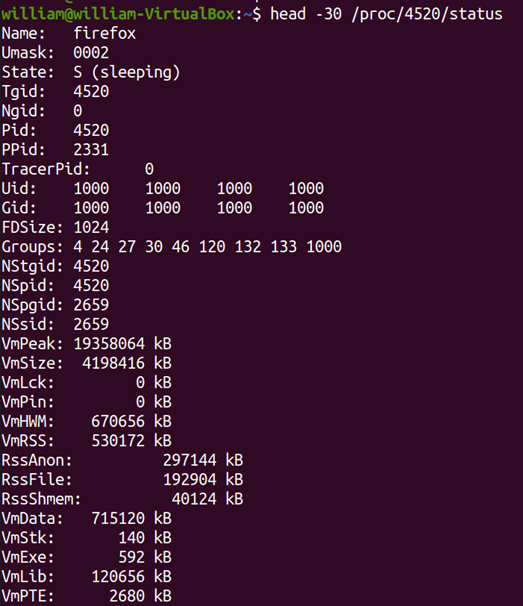


Figure 3. Linux process status information

Text

Description automatically generated

Figure 4. Time executable in Linux provides abundant information

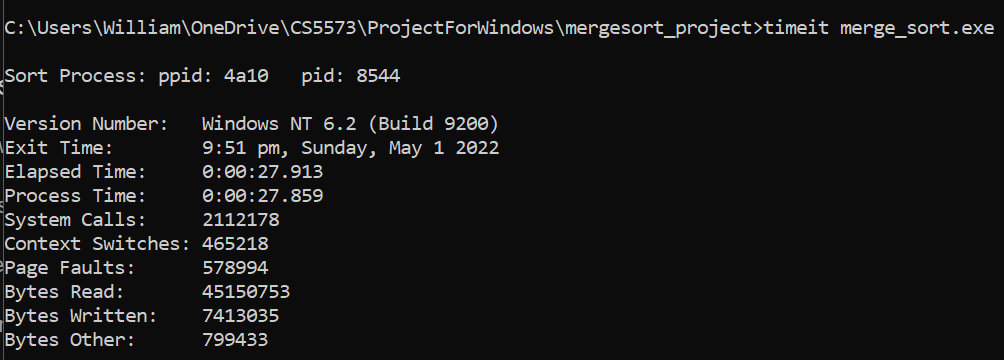


Figure 5. Deprecated executable timeit used to gather metrics.

Text

Description automatically generated Figure 6. Windows10 specific implementation to gather PPIDs.

Text

Description automatically generatedText

Description automatically generated

Figure 7. Spawning threads in Windows (left) vs Linux (right).

A screen shot of a computer

Description automatically generated with low confidence

Figure 8. One of many snapshots showing current system utilization.

Calendar

Description automatically generated

Figure 9. One of many ps cmd snapshots showing all cores in use.

Table

Description automatically generated with low confidence

Figure 10. FreeBSD time executable output.

# Tables

Table 1. Linux Results

Table

Description automatically generated

Table 2. Windows Results

Table

Description automatically generated

Table 3. FreeBSD Results

Table

Description automatically generated

# Graphs

## Execution Time

Graphical user interface, application

Description automatically generated

## CPU Usage

Graphical user interface, chart, bar chart

Description automatically generated

## CPU Utilization as percentage of System

Bar chart

Description automatically generated with low confidence

## Maximum Memory Used

Graphical user interface, application

Description automatically generated

## Total Context Switches

Graphical user interface, application

Description automatically generated

## Total Page Faults

Graphical user interface

Description automatically generated