INTERACTIVITY PERFORMANCE BENCHMARK FOR WINDOWS AND MAC OS

By

Fan Wei Cong

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Kampar Campus)

JANUARY 2020

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: Interactivity Performance Benchmark For Windows and Mac OS

Academic Session: **January 2020**

I FAN WEI CONG

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

(Author's signature) (Supervisor's signature)

Address:
3, Persiaran Halaman Ampang 24,
Halaman Ampang Mewah, 31350,
Ipoh, Perak.

Wong Chee Siang
Supervisor's name

Date: 23 April 2020

Date: 23 April 2020

INTERACTIVITY PERFORMANCE BENCHMARK FOR WINDOWS AND MAC OS

By

Fan Wei Cong

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Kampar Campus)

JANUARY 2020

DECLARATION OF ORIGINALITY

I declare that this report entitled "**INTERACTIVITY PERFORMANCE BENCHMARK FOR WINDOWS AND MAC OS**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : 

Name : Fan Wei Cong

Date : 23 April 2020

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Mr. Wong Chee Siang for allowing me to involve in this research project titled **Interactivity Performance Benchmark for Windows and Mac OS** and trying his very best to assist me in making this project a success.

Besides that, I would also like to thank my dearest family and friends for the unconditional support and love throughout the process of completing this project.

ABSTRACT

Operating system (OS) is a piece of software that exists between computer hardware and programs. Communication between hardware and programs is impossible without the help of an operating system. Within a computer system, there will be a massive amount of tasks created by various types of installed programs which allows the user to perform their work, such as rendering an image and playing a video. However, the number of Central Processing Unit (CPU) available in a computer system will not be accessed by all of the tasks at the same time. In order to allow a fair CPU resource allocation to all the tasks, kernel scheduler is introduced as one of the fundamental component in operating systems.

In this report, the interactivity performance of macOS kernel scheduler will be measured and compared with other OS kernel schedulers using a benchmark program called Interbench. However, Interbench was only available in Linux, making interactivity performance benchmarking impossible without porting it to macOS. Changes involving various semaphores implementations and macOS-specific headers/libraries application are included in the process of porting the original Interbench to macOS.

The Interbench benchmark program which was ported to Windows in the past research will be reverified to ensure that the ported benchmark program is able to simulate the interactive tasks and background loads correctly when it is being executed in systems with different hardware configurations.

The final outcome of this research is to compare the interactivity performance of kernel schedulers in macOS, Linux and Windows with the help of original and ported versions of Interbench. The comparison shows that Linux kernel scheduler has the greatest advantage in terms of interactivity performance in various types of interactive tasks and background load conditions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Program Interface
CFS	Completely Fair Scheduler
CPU	Central Processing Unit
GCC	GNU Compiler Collection
GCD	Grand Central Dispatch
GUI	Graphical User Interface
OS	Operating System
RAM	Random Access Memory
SD	Staircase Deadline
SSD	Solid State Drive

CHAPTER 1 – INTRODUCTION

Operating system (OS) refers to a software that acts as an interface between programs and computer hardware (Silberschatz, Galvin & Gagne 2009). It is comprised of a fundamental component, known as kernel. The kernel will be loaded into the Random Access Memory (RAM) once the system is booted and remains in the RAM for the entire computer session to provide services such as process management and file management. At any moment within a computer session, there are more than one tasks in an OS requesting to execute. Task in OS terms is defined as a process or thread that carries out a set of operations in sequential order (Kamal 2011). Thus, the Central Processing Unit (CPU) scheduler, interrupt handler and process manager are designed to handle the requests from the tasks and to ensure fair execution of tasks. To achieve fairness in scheduling, the scheduler should ensure that the same task will not be utilizing the CPU all the time, causing other tasks to face starvation. Starvation is a condition where a task is ready to execute but it is not chosen to execute due to low priority. Besides, an efficient CPU scheduling algorithm should exist as it is necessary for making scheduling decision as soon as possible.

Most CPU scheduler benchmarking programs focused on determining the system performance based on the throughput of non-interactive tasks (Phoronix Test Suite 2020). These benchmarking programs are not appropriate for the measurement of an operating system's interactivity performance as the main goal of these benchmarking programs is to determine the number of tasks or instructions that can be executed over time. Instead, interactivity performance should be measured based on the response time which is defined as the time from the submission of a request until the time when the response is received (Stallings, n.d.). In other words, response time refers to the time required for a task to switch from Ready State to Running State. A task will be in Ready State when it makes requests for CPU resources to execute instructions. After a request is made by a task, the task will be enqueued to a data structure and will wait until it is chosen by the scheduler for execution. When it is chosen to execute on the CPU, the task will switch from Ready State to Running State. The transition of states for a task is shown in Figure 1.1 below. Interactivity performance plays a very important role in user-oriented systems such as desktops. These systems are used for various types of interactive tasks involving both user and computer which are equivalent to response time sensitive non-CPU bound tasks. The ideal condition for these tasks is when the

amount of response time is low because high response time will become noticeable by human when the response time exceeds the average response time for humans to a visual stimulus, which is 250ms (Backyard Brains, n.d.). For instance, in a video playback, bad interactivity performance will cause some frames in the video to be dropped, making it unnatural for the users to look at. When problems like this exist in a user-oriented system, the user experience of the system will be affected.

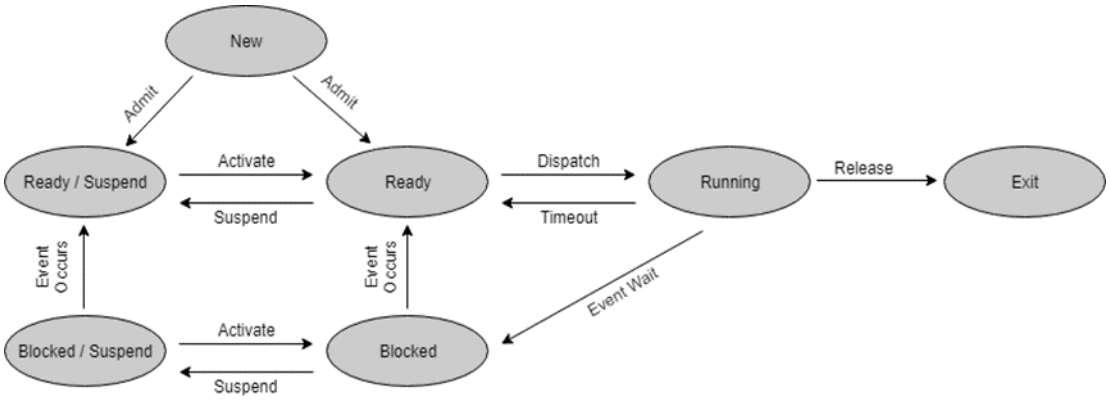


Figure 1.1 7-State Model

Interbench is one of the most well-known benchmark programs which can be used to determine interactivity performance for an operating system (Kolivas 2006). The main goal of this benchmark program is to measure the latencies and jitters that exist in Linux kernel schedulers under different simulated conditions, called interactivity. Interactivity can also be described as an interactive task's response time. However, Interbench is only available for Linux. Other operating systems designed for user-oriented applications such as macOS and Windows are not supported by the benchmark program. Therefore, there is no way to find out the interactivity performance for all operating systems without porting the benchmark program to the unsupported OS.

1.1 Problem Statement

Lack of support of interactivity benchmark program for different operating systems

A benchmark program should provide a standard set of tests to allow comparisons and evaluations of system with same set of hardware and different operating systems. However, Interbench only supports Linux. Because the benchmark program was originally intended to compare different schedulers in Linux, the performance of the same set of benchmark tests on other OS designed for user-oriented applications such as Windows and macOS is limited.

Increasing need for comparisons of benchmark performance in terms of interactivity between different OS kernel schedulers

If a benchmark program is not capable of running in different operating systems, OS developers are not able to obtain sufficient and accurate information regarding the interactivity performance of different OS kernel schedulers. Improvements of scheduling performance will become more challenging to achieve on newly rolled out operating systems or updates. Besides OS developers, normal users will also face a hard time finding the most efficient OS schedulers as there is no way to accurately determine their scheduling performance. The reason is because the users have no accurate information such as latency time, they can only make a guess from the jitters and lags based on their observations.

1.2 Background and Motivation

1.2.1 Project Background

Interbench was used in several past researches to measure the interactivity performance for OS kernel schedulers. One of the researches has figured out the interactivity performance and fairness of O(1) scheduler and Completely Fair Scheduler (CFS) which is included in Linux kernel version 2.6 and version 2.6.23 respectively (Wong et al. 2008). Both of the kernel schedulers were evaluated in terms of interactivity performance and fairness. At the end of the research, CFS is proven to be better in terms of efficiency and fairness of CPU bandwidth distribution without sacrificing its interactivity performance.

Besides that, Interbench was also applied in the study of latencies that exist in schedulers in Linux and FreeBSD kernels focusing on real-time system applications (Abaffy & Krajcovic 2009). In the research, 4 different kernel schedulers were involved, including O(1), CFS, 4BSD and ULE. Discovery of optimal kernel option had been carried out to minimize the amount of latencies. Interbench was used for interactivity benchmarks with different Linux kernel configurations applied. As Interbench does not support benchmarks for FreeBSD kernel, a new benchmark, namely PI-ping was developed for the comparison of all 4 kernel schedulers. Final results of the benchmarks showed that CFS has a greater advantage in servers which require high throughput and embedded systems which emphasize in producing low latencies.

Next, the application of Interbench can be also found in a past research which studied on the fairness, interactivity and multiprocessor performance of O(1) scheduler, Staircase Deadline (SD) scheduler and Completely Fair Scheduler (CFS) (Wang et al. 2009). Results for the interactivity test showed that interactive tasks in O(1) scheduler have the highest tendency to face starvation as the scheduler will raise the priority of one of the interactive tasks once it is finally considered as an interactive task, allowing that particular interactive task to stay in the active queue for a long time. In terms of fairness, good fairness in long term can be achieved by all three kernel schedulers in a uniprocessor environment. As for multiprocessor environment, CFS is the fairest compared to O(1) scheduler and SD scheduler.

A clear idea about the interactivity performance of Linux kernel schedulers can be provided by all three researches mentioned above. However, the coverage of these

researches only include Linux OS while the interactivity performance of kernel schedulers in other operating systems such as Windows and macOS are still unknown. Fortunately, a research has been conducted previously regarding the interactivity performance for Windows OS (Cheng 2015). In order to achieve this, the researcher ported Interbench to Windows 8 by implementing an alternative application program interface (API) for Linux API in Windows. Comparison by using the original Linux Interbench and ported Windows Interbench proved that Linux kernel scheduler has the ability to handle interactive tasks better compared to Windows 8 kernel scheduler. The great performance becomes more obvious when Linux kernel scheduler is operating in high background load conditions.

1.2.2 Motivation

The first motivation of this project is to achieve a fair and accurate comparison on the interactivity performance on different operating systems. In order to accomplish this, the original Interbench benchmark program developed specifically for Linux OS will require porting to macOS. Else, there is no way to figure out the differences in interactivity performance for all OS.

Next, the second motivation of this project is to ease the process of OS developments by providing a fair interactivity performance analysis solution to the OS developers. The developers are able to determine whether their design and implementation of schedulers in their operating systems' kernel are better than the others or the schedulers deployed previously. For normal users, they will be able to use the benchmark application to determine their systems' interactivity performance and figure out the system that produces the greatest interactivity performance with minimal response time. Past research showed that response time in a computing system can lead users to psychological consequences such as stress and emotion problems. Bad interactivity performance caused by great amount of response time can cause users to have a bad emotional state, especially when the users are under time pressure (Stupak 2009).

1.3 Project Scope

1.3.1 Porting Interbench to macOS

Due to the unavailability of Interbench in other operating systems other than Linux OS, porting of the benchmark program is a necessary step to achieve our objectives. In this project, Interbench will be ported into macOS to obtain the results in terms of interactivity performance for interactive tasks in macOS. In order to achieve this, syntax difference and parameter difference caused by different system libraries and headers will be figured out and appropriate modifications will be made.

1.3.2 Perform Reverification for Windows Interbench

Interbench was made available to Windows by Cheng (2015) in the previous research. Although a verification plan was completed by the researcher, however, Windows Interbench was only executed on Windows 8 with one set of system hardware which does not reflect the ported benchmark program's ability to work in different hardware configurations and different version of Windows OS. Besides, the past verification did not include information for Intel Turbo Boost Technology, which allows CPU to run at higher frequency than its base frequency for heavier workloads to achieve better performance. Intel Turbo Boost Technology can affect the simulation of interactive tasks and background loads because the behaviour of this technology differs from one OS to another. In order to allow fair comparisons of interactivity performance, Intel Turbo Boost Technology will be disabled for all three OS. Therefore, Windows Interbench will be reverified to ensure that the benchmark program is able to behave similarly to the original Interbench on different sets of system hardware and software configurations.

1.3.3 Run benchmark test and compare the performance in Linux, Windows and macOS

In order to compare the kernel schedulers' performance, Linux, Windows and macOS will be installed natively into the test computer so that fair comparisons can be done with same hardware resources, for instance, the CPU. The original Interbench and the modified Interbench will be executed in terminal of every OS. In the execution process, combination pair consisting one interactive task and one background load will be executed concurrently and the scheduling latencies will be measured.

For the actual benchmark test, 4 different interactive tasks will be simulated. Audio and Video tasks represent low CPU consumption task and medium CPU consumption task respectively. As for the high CPU consumption tasks, X and Gaming will be included. These 4 interactive tasks will be simulated under different loads. Detailed discussion of all possible combination pairs of interactive task and background loads is included in Chapter 5.2 of this report. Average scheduling latencies and maximum scheduling latencies that exist within each interactive task will be obtained by running Interbench repeatedly for up to 30 times. The obtained latencies results will be compared side-by-side in graphs to provide a clearer insight about the differences in interactivity performance for the schedulers of Linux, Windows and macOS.

1.4 Project Objectives

- To modify existing benchmark application to enable the ability to run interactivity benchmarking on macOS.
- To reverify Windows Interbench's ability to work in different system hardware configurations.
- To differentiate the interactivity performance of different OS kernel schedulers from different operating systems by using a benchmark program with the simulation of multiple computer related tasks under different types of load.

1.5 Proposed Approach

In the macOS Interbench, there are multiple critical sections consist of variables shared among threads such as emulation_thread and timekeeping_thread. The threads are able to access the shared variables and perform updates on the values of the variables during the execution of the program. If there is no proper synchronization done for threads' execution, the execution order of the program will be different every time, causing the values of the variables will be changed improperly, also known as a race condition. Semaphores are applied in macOS Interbench to ensure the correct thread execution sequence and make sure that the critical sections are accessed by only one thread at a time.

The thread synchronization for macOS Interbench is implemented with the help of Grand Central Dispatch's (GCD) semaphore (Apple Developer 2019). Semaphores which were originally implemented in the original Interbench benchmark program is not applicable in macOS as the functions located in semaphore.h header file are no longer supported by Apple. Table 1.1 below shows the mapping of the original functions from Linux semaphores to GCD semaphores.

Linux	macOS
sem_init(sem_t *sem, int pshared, unsigned int value)	dispatch_semaphore_create(int value)
sem_wait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_FOREVER)
sem_trywait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_NOW)
sem_post(sem_t *s)	dispatch_semaphore_signal(dispatch_semaphore_t *s)

Table 1.1 Mapping of Linux Semaphores to GCD Semaphores

1.6 Highlight of What Have Been Achieved

The first thing that have been achieved in this project is obtaining the total amount of physical memory which is available. This information is required for the simulation of multiple background loads involving Disk I/O operations such as Write and Read. The Write background load is simulated by writing to a file with the size of the physical memory while the Read background load is simulated by reading from a file with the size of the physical memory from the disk.

Followed by the implementation of Grand Central Dispatch's (GCD) semaphores for synchronization of threads involving accesses to shared resources.

After the achieving the things mentioned above, verification on the Interbench benchmark program on Linux, macOS and Windows was done to make sure that the simulation behaves correctly so that the final interactivity performance results will be reliable.

Lastly, Interbench was executed for 30 times repeatedly on each operating system to obtain the average scheduling latencies and maximum scheduling latencies. The final results were collected and presented in the form of bar charts for comparison purposes.

1.7 Report Organization

The details for this research are included in the following chapters. In Chapter 2, it covers literature review that contains work done in previous researches and related to the current project. Followed by Chapter 3 which contains discussions on the system design and explanation of Interbench benchmark program. Then, Chapter 4 discusses about the methodologies applied in this project. Next, the comparisons of interactivity performance on Linux, macOS and Windows are discussed in Chapter 5. Lastly, the project is concluded in Chapter 6.

CHAPTER 2 – LITERATURE REVIEW

2.1 Fairness and Interactivity Performance of O(1) and CFS Linux Kernel Schedulers

In this research done previously, the scheduling algorithms for two different Linux kernel schedulers were explained (Wong et al. 2008). The paper focused on the ways time sharing tasks work in the different kernel schedulers in terms of their interactivity performance and fairness of the schedulers.

In Linux kernel version 2.6, O(1) scheduler was introduced by Ingo Molnar. In this scheduler, each of the system's CPU will have their own run-queue where all the runnable tasks assigned to the specific CPU will handled. In order to achieve this, 2 different arrays will be used, an active array and an expired array. For a task to access the CPU, a runnable task which is stored within the active array and having the greater dynamic priority will be selected by the O(1) scheduler for execution with time quantum pre-specified. When several runnable tasks with the same priority values, a preemptive Round Robin scheduling style will be applied so that all the tasks can obtain a fair access to the CPU resources. After the time quantum is finished, the current task's next time quantum will be calculated and the task will be placed into the expired run-queue. Eventually the active array will become empty as all the tasks' time quantum have been finished. This is where the pointers of the 2 arrays will be switched. In other words, the previous active array will become the current expired array and the previous expired array will become the current active array.

In order to implement tasks prioritization in the system, 2-D array is used in the run-queue. The 2-D array contains 140 different priority levels consisting of real-time priority (ranging from 0 to 99) and static priority (ranging from 100 to 139). Each of the static priorities is assigned with a nice value, which determines a process's priority. The base amount of time quantum allocated to a task is based on the static priority. The equation below is followed:

$$\textit{Base time slice (in milliseconds)} = \begin{cases} \textit{if static priority} < 120, & (140 - \textit{static priority}) \times 20 \\ \textit{if static priority} \geq 120, & (140 - \textit{static priority}) \times 5 \end{cases}$$

Therefore, if a process has a higher static priority than others, a longer base CPU time quantum will be allocated to the other processes with lower static priorities. In other words, the task will be able to access the CPU for a longer period.

The O(1) kernel scheduler is being used in the next Linux kernel versions until Linux kernel version 2.6.23. It is being replaced by another kernel scheduler called Completely Fair Scheduler (CFS). To achieve a better efficiency, some components that were used in the O(1) scheduler are removed, including run queue arrays and allocation of time quantum based on priority values. The advantages of CFS include the ability to produce a good interactivity performance while pushing the CPU to its maximum utilization and allowing fair CPU resource access for all the tasks without dragging down the interactivity performance of the CPU. Fair amount of CPU time is assigned to each task by the application of proportional share algorithm to achieve fairness.

The desired situation the CFS try to achieve is to allow parallel execution of tasks in a uni-processor system. However, that is not possible as only one task is allowed to run in the CPU at a time, which causes other tasks to wait for the time quantum of the executing task to exhaust or request for I/O. Proportional share algorithm is used to break the time quantum to reduce the amount of lags to its minimum. The time quantum that will be allocated to a task is calculated based on the equation below:

$$slice = \frac{se \rightarrow load.weight}{cfs\_rq \rightarrow load.weight} \times period$$

se -> load.weight refers to the schedule-able entity's weight that can be obtained from prio_to_weight 2-D array using nice values, cfs_rq -> load_weight representing the sum of all the weights from all the entities within the CFS run-queue and period refers to the time quantum for the scheduler to run all the tasks.

At the end of the research, evaluations of fairness and interactivity performance in both O(1) and CFS are carried out. Final results showed that CFS is better than O(1) kernel scheduler in terms of efficiency and fairness of CPU bandwidth distribution without sacrificing its interactivity performance.

2.2 Latencies in Linux and FreeBSD kernels with different schedulers – O(1), CFS, 4BSD, ULE

Study was done by the researchers involved in this paper regarding the latencies present in schedulers included in different Linux 2.6 kernel versions with a focus on its application in real-time systems (Abaffy & Krajcovic 2009). Some soft real-time tuning options were used to discover the best kernel configuration with the least amount of latencies exists.

Comparisons had been carried out on O(1) scheduler in Linux kernel version 2.6.22 and Completely Fair Scheduler (CFS) in Linux kernel version 2.6.28. Linux kernel version 2.6.22 is the last kernel version that uses O(1) kernel scheduler before it was replaced by Completely Fair Scheduler (CFS) from Linux kernel version 2.6.23 and above. Making the comparisons more accurate, different kernel options were also compared as latency reduction is possible when other options are applied. A kernel which is designed for server usage, 4BSD scheduler and ULE scheduler available in FreeBSD 7.1 kernel were also included in the comparison.

The discovery of the optimal kernel configurations involves appropriate kernel options setting, kernel compilations and benchmark performance. Interbench was the main benchmark program used in the research for the testing various kernels. Each test was carried out for a period of 30 seconds with 1055301 CPU cycles used per second, enabling the collection of relevant data. Besides, the Linux kernels with default configuration applied were used as reference kernels. Then, some appropriate kernel options were added into the default kernel configurations and compared with the reference kernels to determine the presence of any improvement in terms of real-time performance.

Unfortunately, some problems were faced by the researchers in the middle of the study. One of the problems is that they were unable to compare Linux and FreeBSD kernels by only using Interbench, which is originally developed for Linux. As a solution, a new benchmark called PI-ping was developed to compare all 4 kernel schedulers.

As for the results, CFS was proved to have an advantage in both applications in servers and embedded systems, which requires high throughput or number of tasks completed within a specified time and low latencies respectively. The researchers also proved that CFS is able to perform better than O(1) for number of processes below than 500 due to

CFS's complexity of O(log n). Lastly, processes are managed by CFS in a red-black tree while others use run-queues.

2.3 Fairness and Interactivity of Three CPU Schedulers in Linux

Analysis has been conducted on three different schedulers in terms of their fairness, interactivity, and multi-processor performance by applying benchmarks (Wang et al. 2009). The three schedulers studied are O(1), Staircase Deadline (SD) schedulers and Completely Fair Scheduler (CFS). Comparison and evaluation in the schedulers were done using three different types of benchmarks, kernel codes remained unchanged except for the scheduler to introduce the goals of CPU scheduling and the schedulers.

The primary goal of the scheduler is fairness, which refers to sharing CPU time fairly with individual tasks and considering the priorities of the tasks at the same time. According to the researcher, the scheduling algorithm is considered as fairer when the amount of lag is smaller.

Followed by the next scheduling goals are regarding interactivity and impact on fairness. Minimal amount of latencies is necessary in order to achieve a good response time for interactive tasks. The tendency for an interactive task to sleep frequently has become an assumption made by the scheduler to identify it as an interactive task. For a situation which multiple interactive tasks existing the system, the scheduler needs to maintain the fairness between the tasks to prevent any of the tasks from facing starvation. Starvation refers to a situation where task is already in 'Ready' state but it is not chosen to execute. Besides that, interactive tasks should not let the non-interactive tasks to starve for a long time.

The next scheduling goal is load balance. In order to accelerate computation in a parallel way, many multi-processor architecture were introduced and used in embedded fields. Without the existence of load balance, CPUs in a system will face a significant amount of unfairness as there is no way to predict the load of each CPU in a specific time.

Lastly, application performance in the real world is also an important goal. The number of large real world applications that can be executed on both desktop environment and server environment increases. So, maximum throughput, fairness and interactivity are the important features that should be included in a well-designed scheduler.

To obtain the results for schedulers' fairness, n tasks with nice value of 0 were executed for uni-processor benchmark. Good long term fairness results were obtained for all O(1), SD and CFS. However, CFS has an advantage in producing the least average lag

compared to the other two schedulers. Results also showed that average lags increase linearly when the number of system tasks increases.

As for the interactivity results, interactive tasks in O(1) faced starvation the most as the scheduler will raise the priority of a task considered as an interactive task and allowing the particular task to stay in the active queue for a long time. As for SD and CFS, the strict fairness scheduler design allowing both of the schedulers to achieve good fairness.

In conclusion, all schedulers are able to achieve a decent fairness in long term on uni-processor. As for fairness on multi-processors, CFS is the fairest among the 3 schedulers.

2.4 Interactivity Performance Benchmark on Windows OS

Studies had been conducted on the interactivity performance of the OS schedulers in Linux OS and Windows OS (Cheng 2015). Interbench was used by the researcher to perform the benchmark. But unfortunately, Interbench is only made available for Linux by the original developer. Porting the benchmark program to Windows was done by the researcher with the application of C programming language, producing the final product named Windows Interbench. Tweaks have been made in the porting process due to different system call and application program interface (API) that exists in Linux and Windows.

In order to port the program to Windows, issues and challenges had been faced by the researcher in the porting process. A lot of time and effort was required to find an alternative API for Linux API to be implemented in Windows. Besides that, accuracy in Windows Interbench were not as good as the original Interbench benchmark program for Linux. Both operating systems come with a structure called timeval in Windows and timespec in Linux, primarily for time interval definition. The main reason of the difference in accuracy is because the precision of microsecond is supported in Windows OS's timeval structure while the precision of nanosecond is supported in Linux OS's timespec structure.

After the completion of the program's porting process, interactive tasks such as Audio, Video, X and Gaming were simulated with different loads. First, number of meaningless loops executed by the system within one millisecond was determined. This process is required for the reproduction of a constant CPU usage in every test. After that, the number of meaningless loops executed within one millisecond is saved to a file, allowing the emulation CPU usage to remain unchanged for the following tests. For every benchmark test performed, amount of latency that exists between the starting time of an interactive task and the time the task finally acquires the CPU was recorded. Each benchmark test was repeated for 10 times to obtain the average scheduling latency and maximum scheduling latency.

Results for interactivity performance for the four different interactive tasks were obtained as the descriptions below. For interactive tasks with low CPU consumption, such as Audio task that consumed only 5% of CPU resources and Video task that consumed 40% of CPU resources, both operating systems showed a very similar results

under low CPU load. However, when both of the interactive tasks were tested in Windows under high amount of load in Burn load, the average latency in Windows was significantly higher than in Ubuntu, a Linux distribution. As for interactive tasks with high CPU consumption, including X task consuming 0% to 100% of CPU resources and Gaming task which tries to consume as much CPU resources as possible, slight performance advantages were shown in Windows compared to Ubuntu. Under low load, latencies in both operating systems were not obvious. However, under high load, the amount of latencies were high until they were noticeable to human's senses.

In conclusion, in terms of overall interactivity performance, Linux scheduler is able to handle interactive tasks better than the scheduler included in Windows, especially in conditions with high amount of background load.

2.5 Synchronization Primitives

There are two mechanisms made available by the kernel for kernel programming in macOS, which includes semaphores and locks (Apple Developer 2013). These mechanisms can be applied when multiple threads in a system need to access shared resources or critical section to accomplish their tasks, so that the shared resources can be protected.

For lock, a lock request can be made by multiple threads, however the lock will be allocated to one and only one thread at a time. As for the remaining threads requested for the lock, they have to wait until the thread which is holding the lock to release it.

As for the next synchronization mechanism semaphore, it is quite similar to a lock. The difference between a semaphore and a lock is that a shared resource can be accessed by more than one threads at the same time. Instead of protecting one shared resource by lock, multiple indistinct shared resources can be protected with the use of semaphores.

In macOS, counting semaphores are used instead of binary semaphores which behaves like a lock. There is some potentials for a thread to face starvation because Mesa semantics are applied in Mach semaphores. This is because before Thread A is allowed run and Thread B has a faster execution speed, Thread B will run first, causing Thread A to wait indefinitely.

CHAPTER 3 – SYSTEM DESIGN

3.1 General Workflow of macOS Interbench

In the main( )function, it is used to display the table outputs for each interactive task and to loop through the interactive tasks and background loads defined in the threadlist[ ] array. Checking is done in the function to determine whether the current value in the array is an interactive task or a background load.

In order to reproduce a fixed percentage of CPU utilization in every benchmark run, the system will be benchmarked by Interbench in the first execution by invoking a function called calibrate_loop( ) to determine the number of meaningless loops that can be executed by the system within one millisecond. Then the value will be written into a file for the subsequent benchmark runs.

The bench( ) function will be invoked to perform synchronization between interactive task's child task and background load's child process. The synchronization is required to ensure that the correct execution sequence can be achieved.

For the simulation of interactive tasks and background loads, run_loadchild( ) and run_benchchild( ) are responsible to control the thread such as starting and stopping a thread.

Followed by emulation_thread( ) function which is responsible to point to the interactive task and background load which corresponds to the i and j integer values indicating the current values of the interactive task and background load to be executed. These values are being tracked by the previous main function using a nested for-loop. For instance, when the value for i is equals to 4 and value for j is equals to 2 in the same iteration, the emulation_thread( ) function will call the emulate_game( ) function which simulates Gaming interactive task and emulate_video( ) function which simulates Video background load.

Lastly, timekeeping_thread( ) is responsible to keep track of the time taken for an interactive task to be scheduled under each background load.

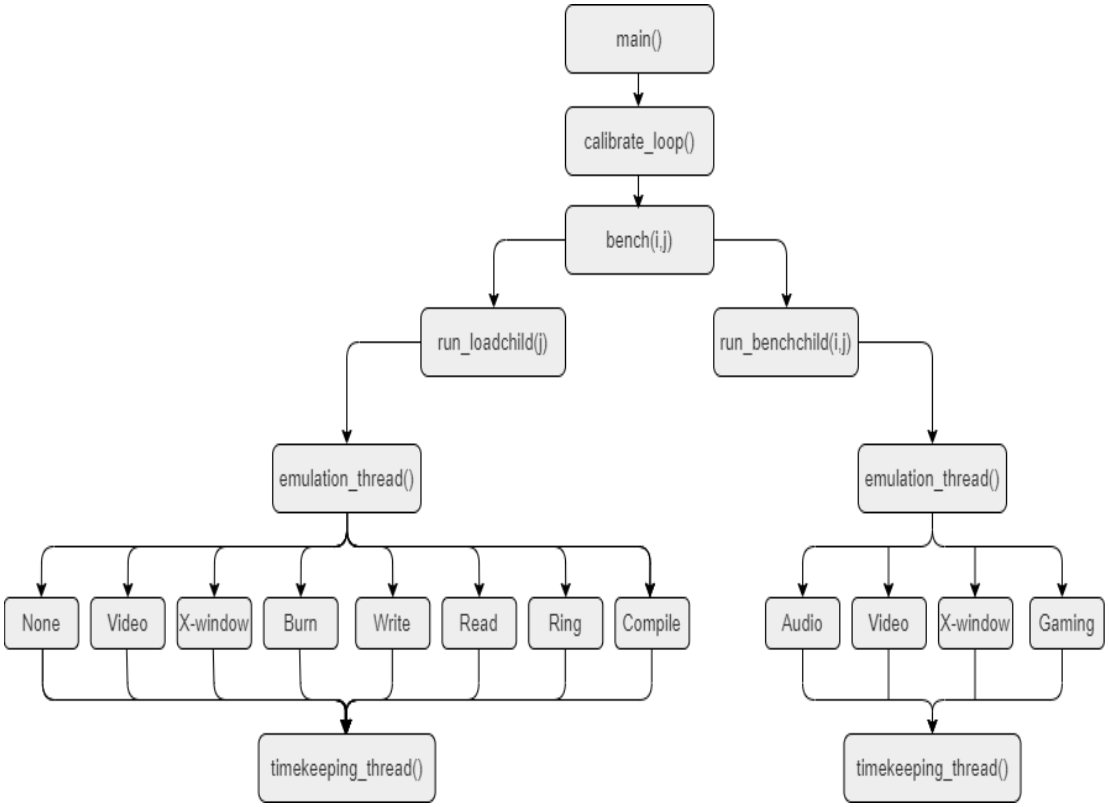
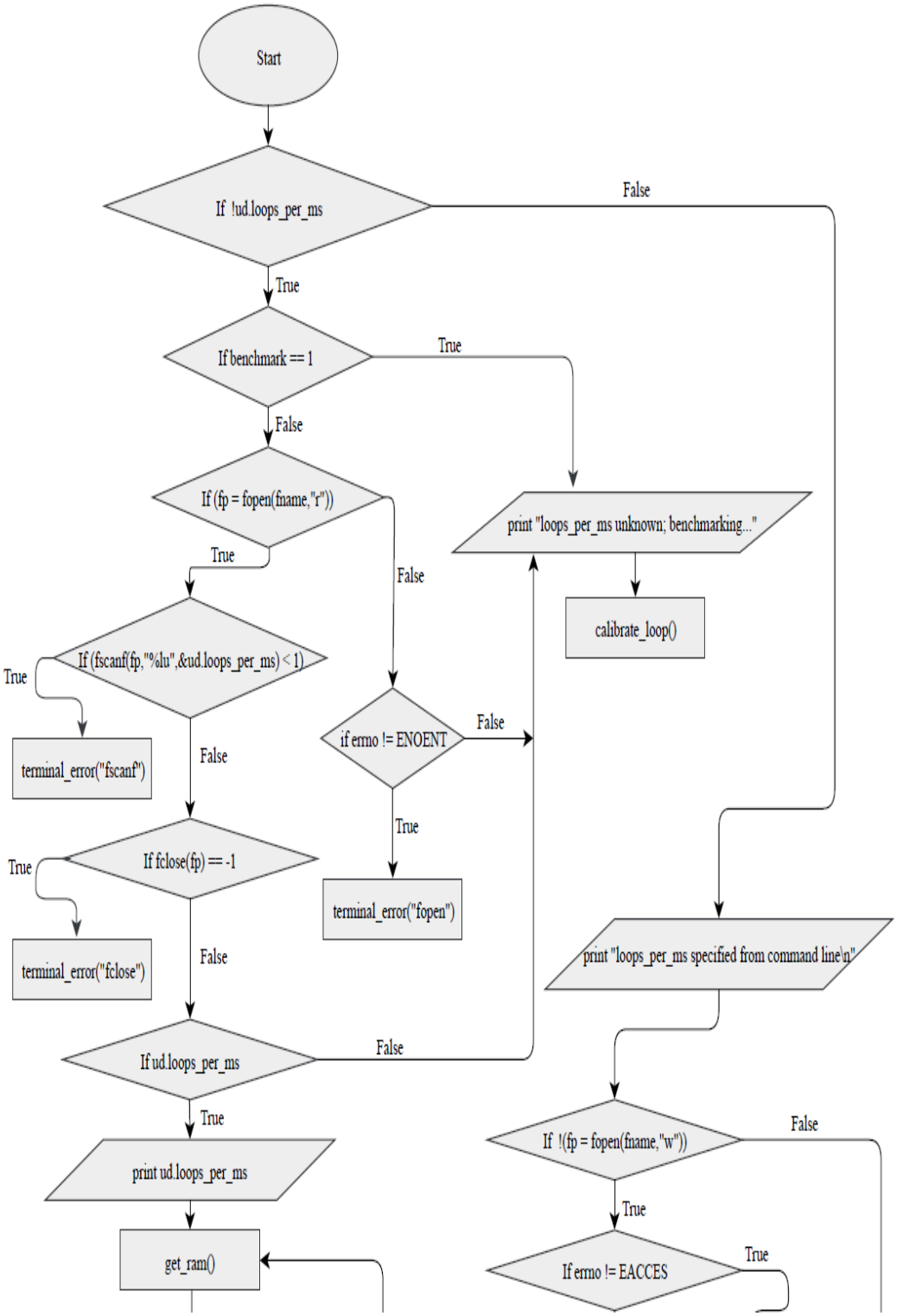
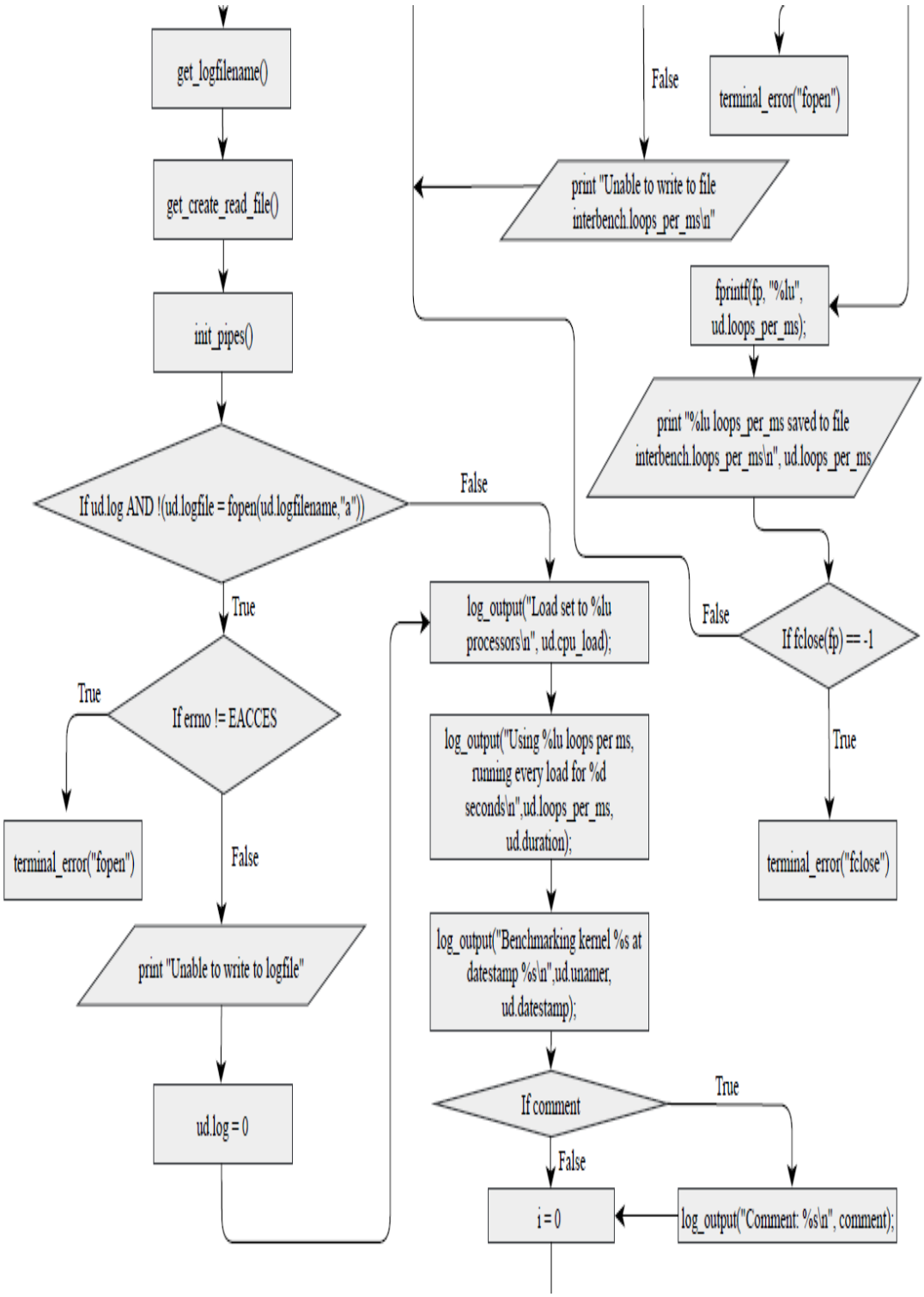


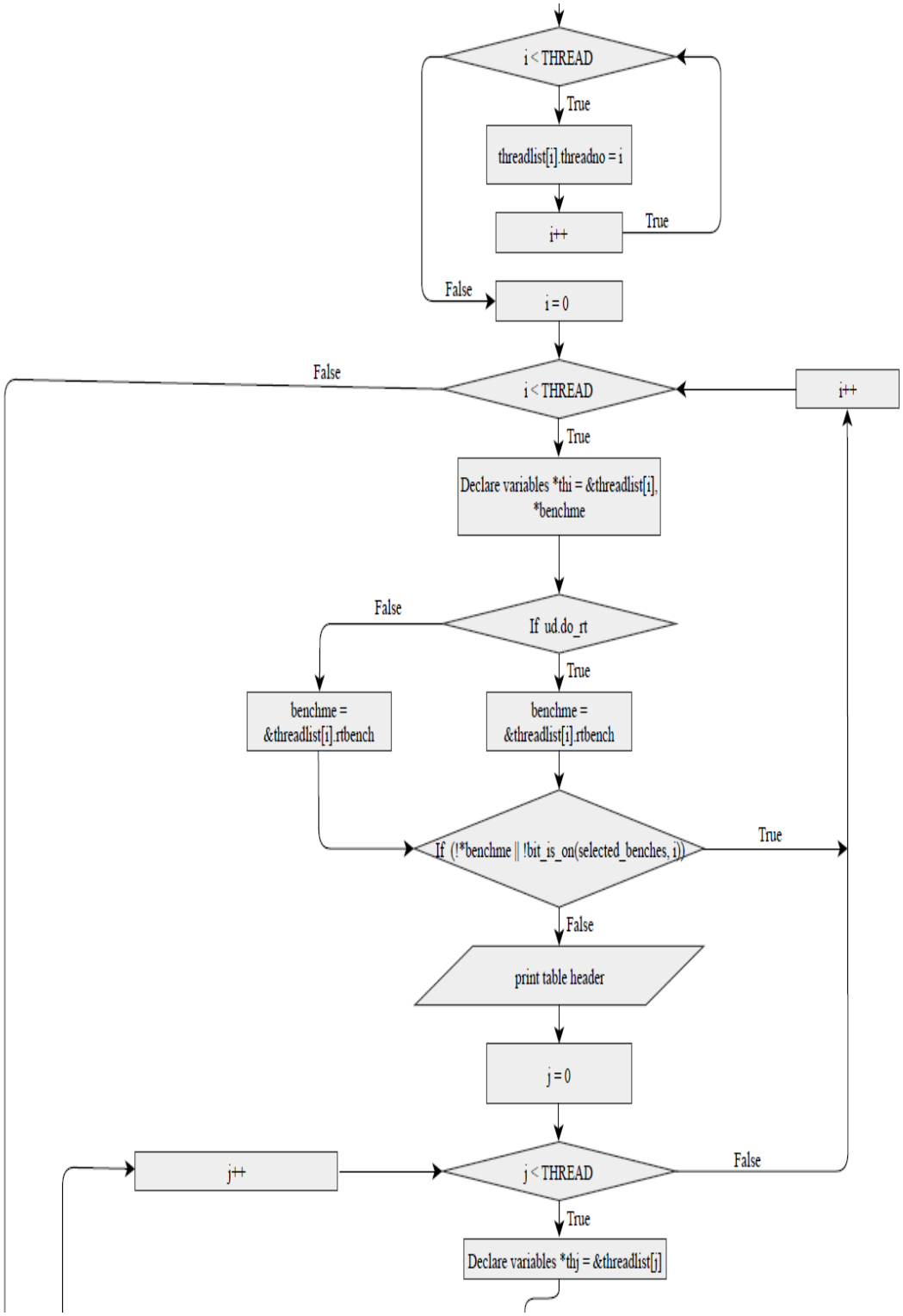
Figure 3.1 General Workflow of macOS Interbench

3.2 Detailed Function Flowcharts

3.2.1 main( ) Function Flowchart




get_logfilename()
get_create_read_file()
init_pipes()
If ud.log AND !(ud.logfile = fopen(ud.logfilename,"a"))
True
False
If errno != EACCES
True
terminal_error("fopen")
False
print "Unable to write to logfile"
ud.log = 0
log_output("Load set to %lu processors\n", ud.cpu_load);
log_output("Using %lu loops per ms, running every load for %d seconds\n",ud.loops_per_ms, ud.duration);
log_output("Benchmarking kernel %s at datestamp %s\n",ud.unamer, ud.datestamp);
If comment
True
False
log_output("Comment: %s\n", comment);
i = 0
False
terminal_error("fopen")
print "Unable to write to file interbench.loops_per_ms\n"
fprintf(fp, "%lu", ud.loops_per_ms);
print "%lu loops_per_ms saved to file interbench.loops_per_ms\n", ud.loops_per_ms
False
If fclose(fp) == -1
True
terminal_error("fclose")


i < THREAD
True
threadlist[i].threadno = i
i++
True
False
i = 0
False
i < THREAD
i++
True
Declare variables *thi = &threadlist[i], *benchme
False
If ud.do_rt
True
benchme = &threadlist[i].rtbench
benchme = &threadlist[i].rtbench
If (!*benchme || !bit_is_on(selected_benches, i))
True
False
print table header
j = 0
j++
j < THREAD
False
True
Declare variables *thj = &threadlist[j]

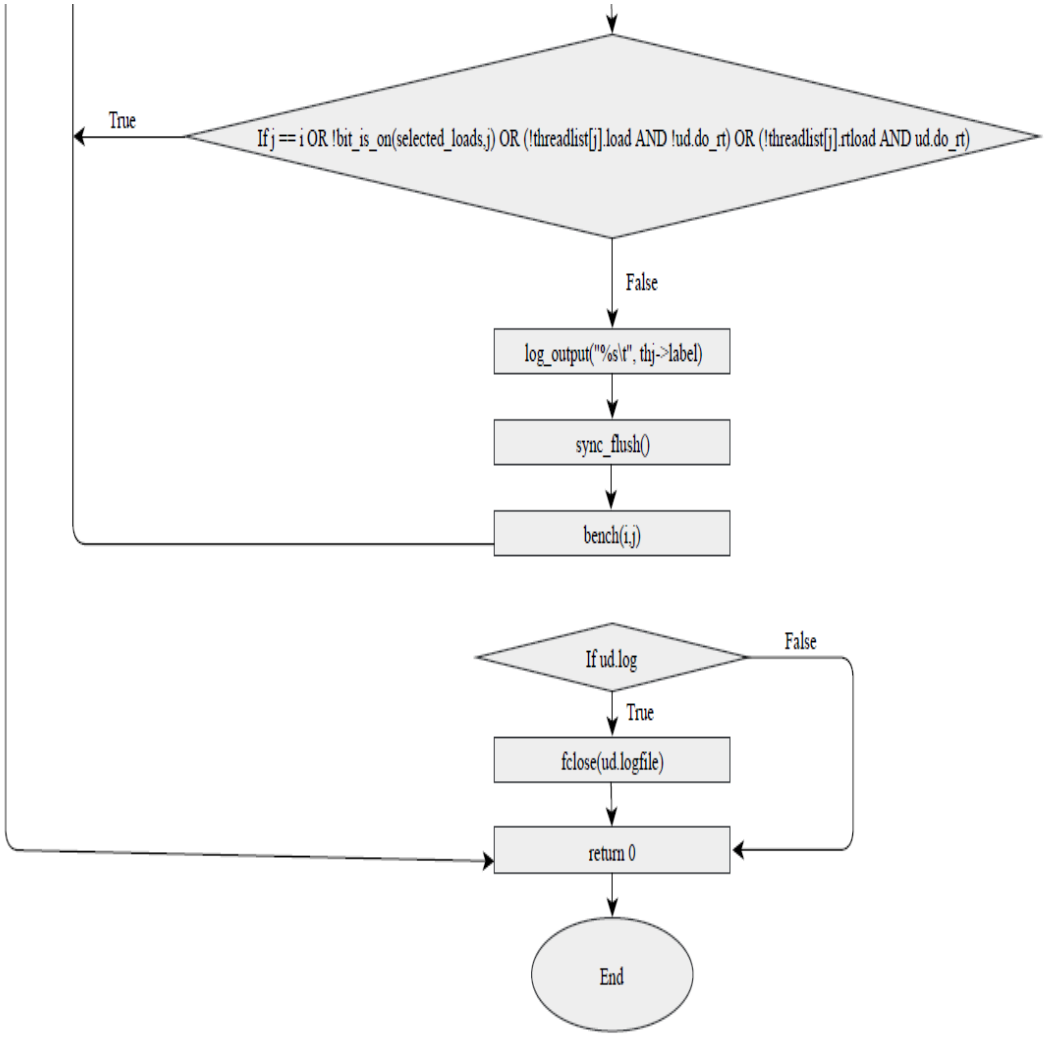
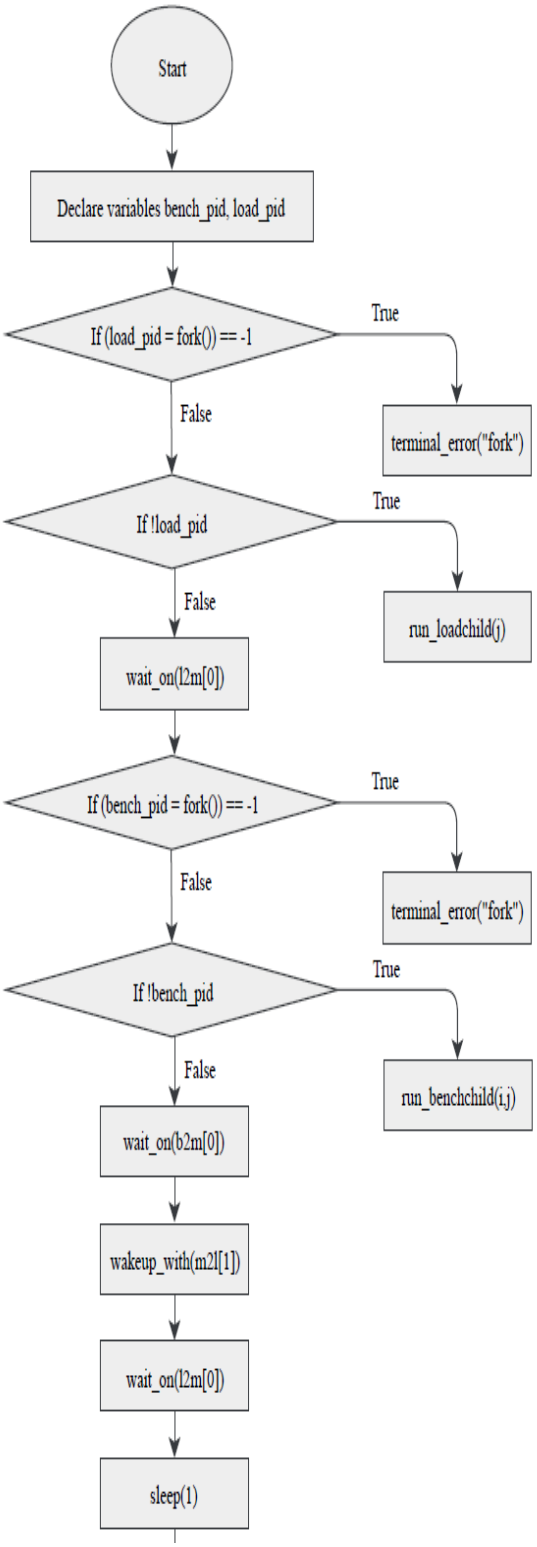


Figure 3.2 Flowchart for main( )

3.2.2 bench( ) Function Flowchart



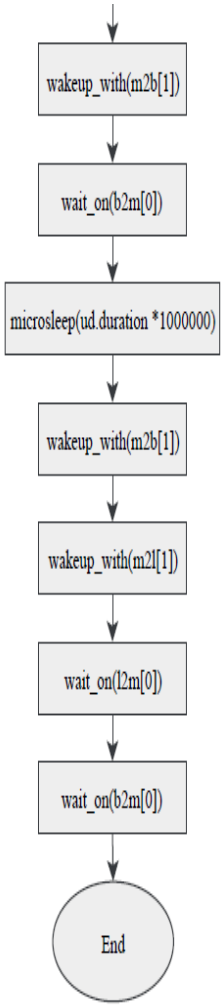
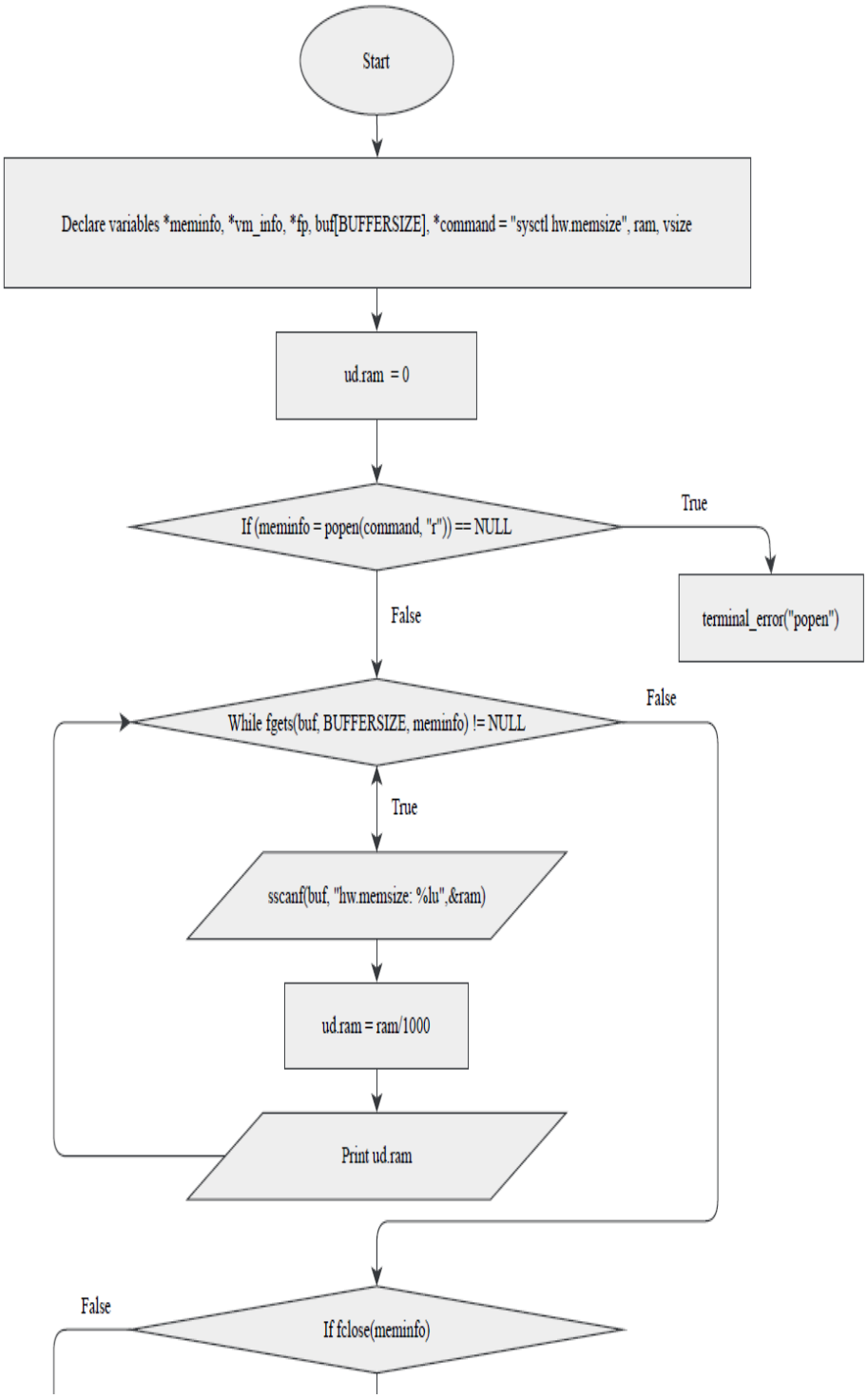


Figure 3.3 Flowchart for bench( )

3.2.3 get_ram( ) Function Flowchart



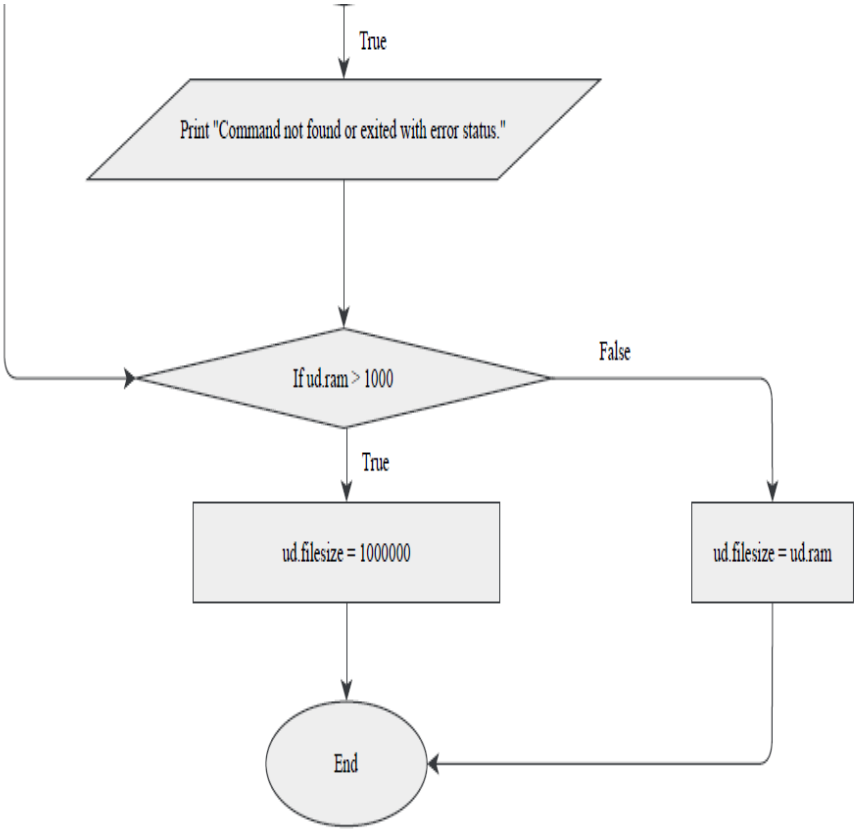


Figure 3.4 Flowchart for get_ram( )

3.2.4 run_loadchild() Function Flowchart

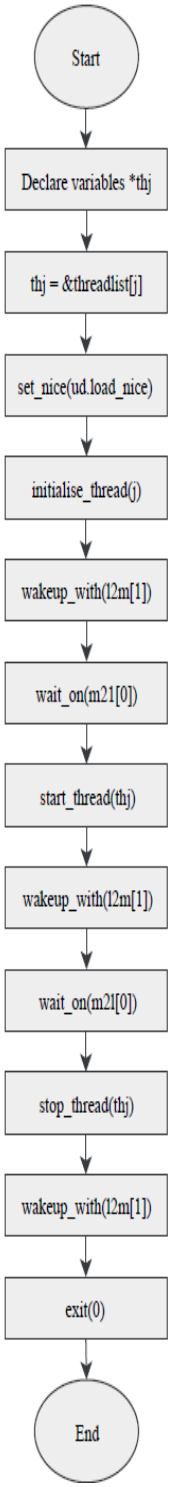
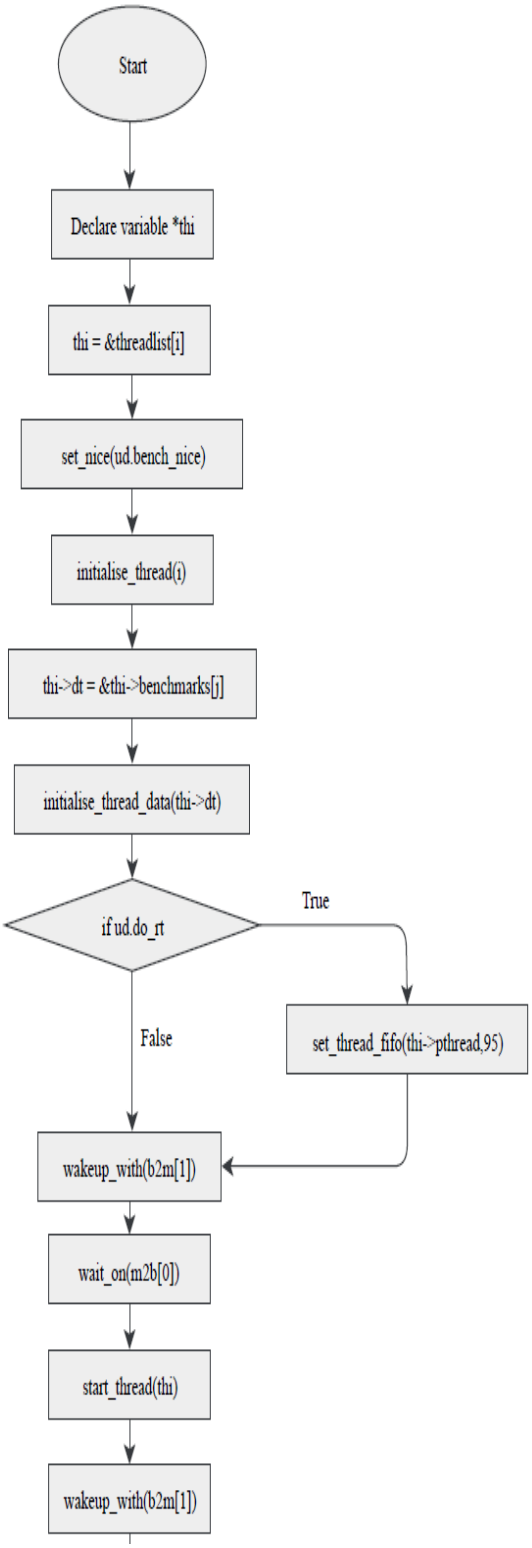


Figure 3.5 Flowchart for run_loadchild( )

3.2.5 run_benchchild( ) Function Flowchart



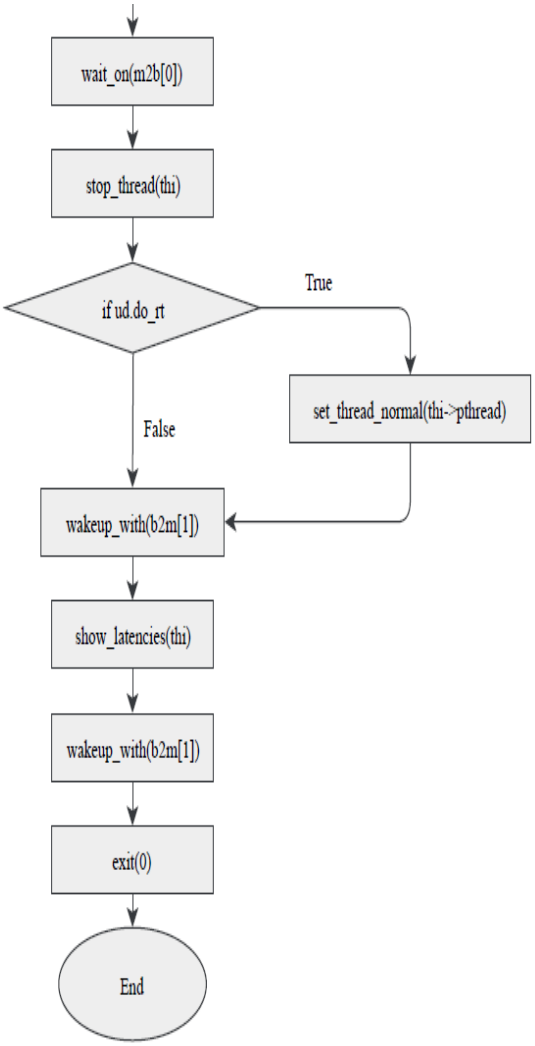
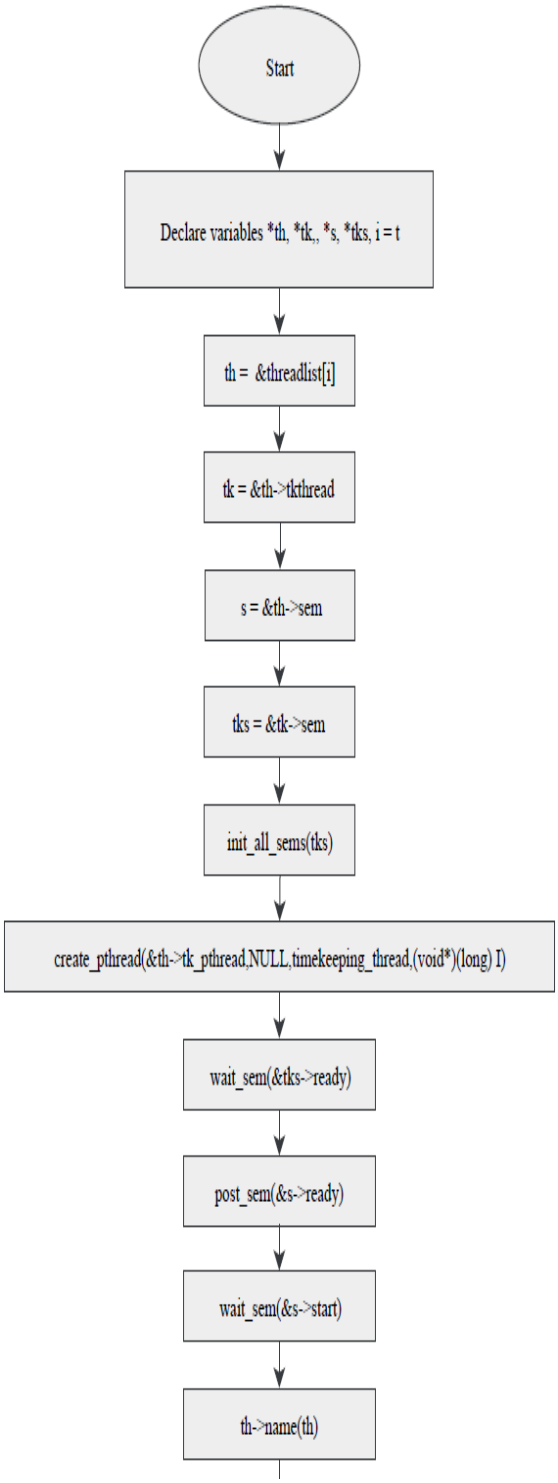


Figure 3.6 Flowchart for run_benchchild( )

3.2.6 emulation_thread( ) Function Flowchart



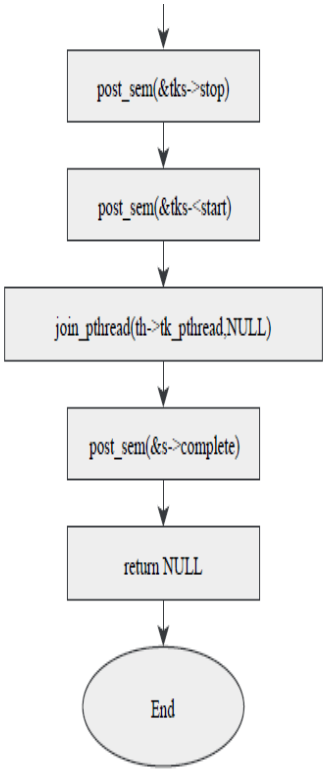
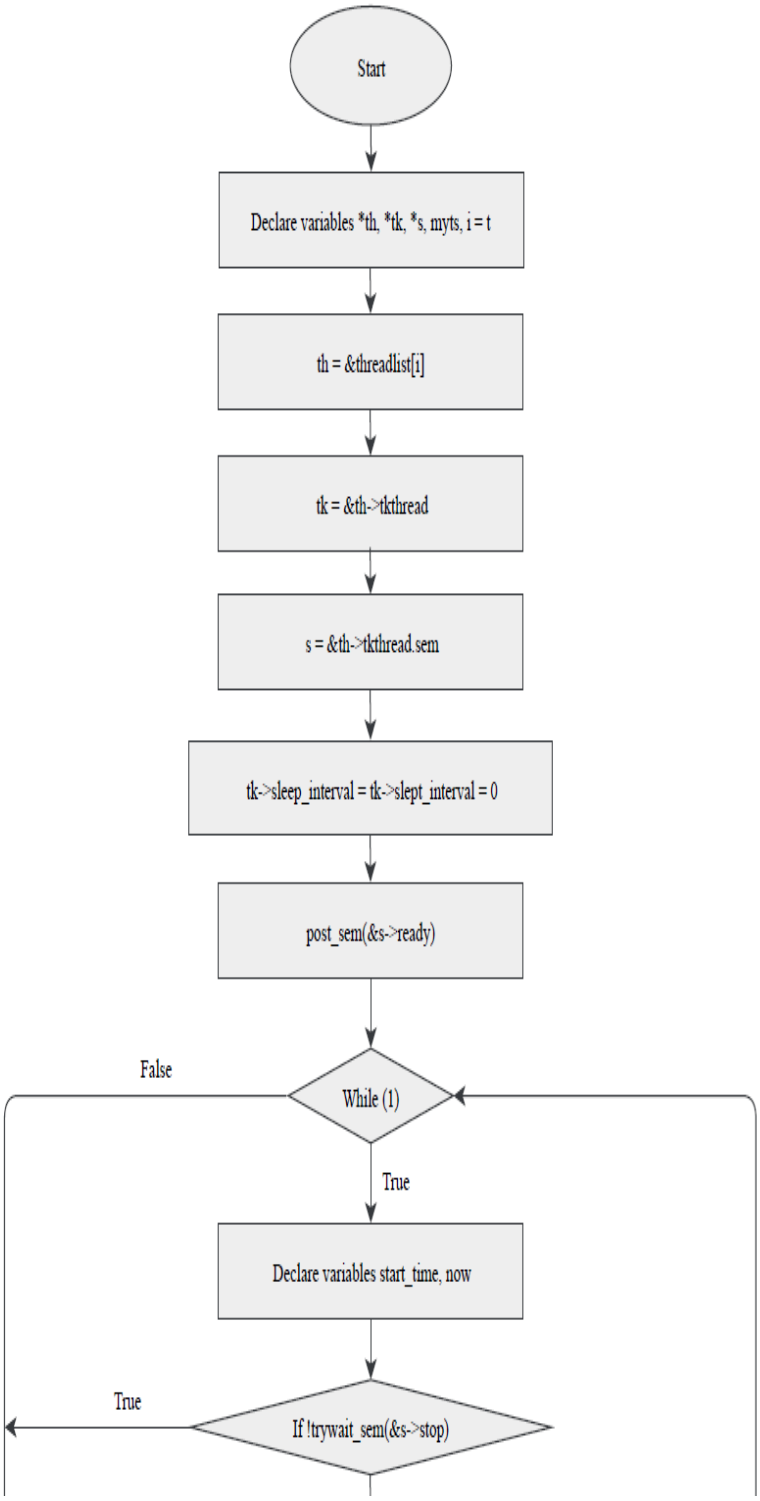
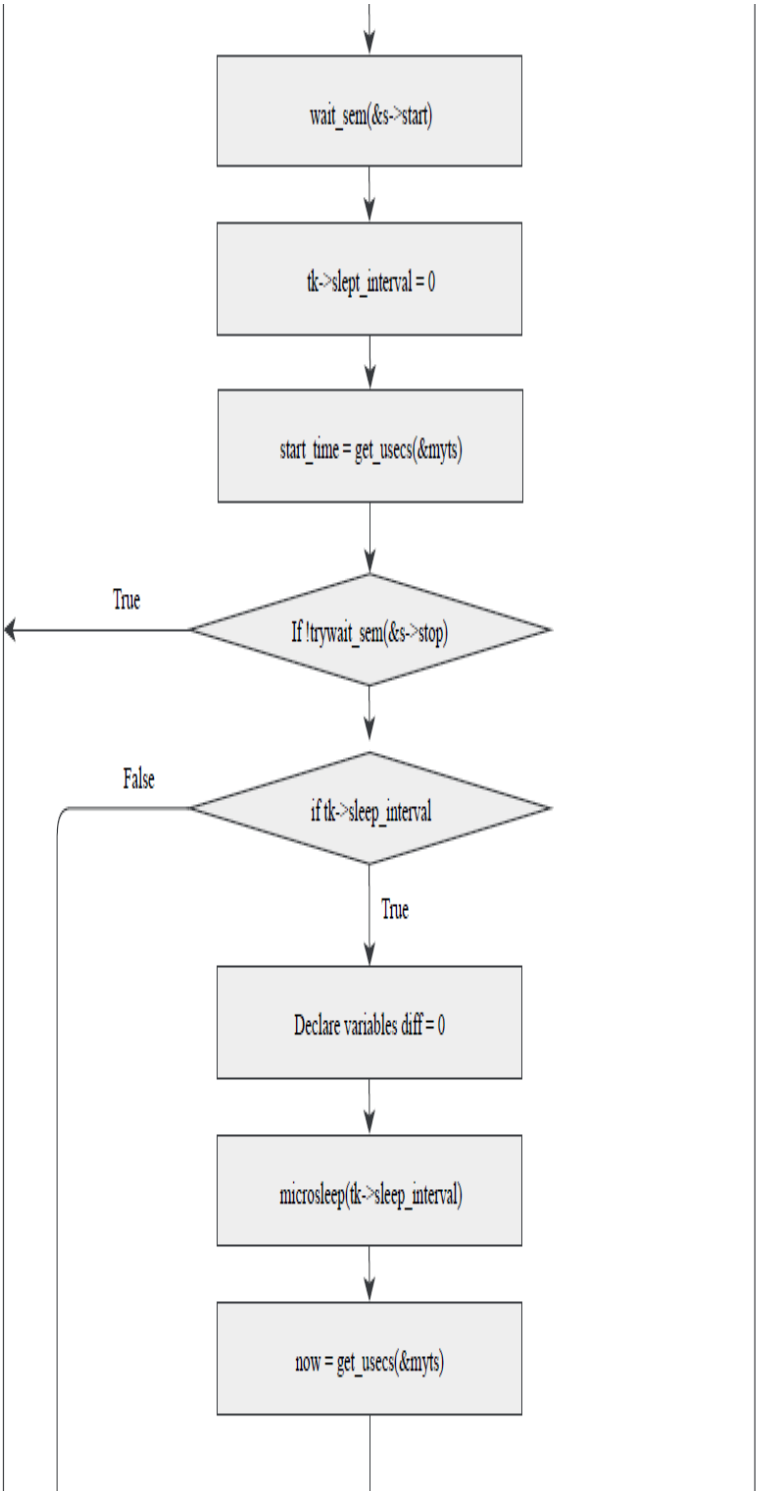


Figure 3.7 Flowchart for emulation_thread( )

3.2.7 timekeeping_thread( ) Function Flowchart




wait_sem(&cs->start)
tk->slept_interval = 0
start_time = get_usecs(&myts)
True
If !trywait_sem(&cs->stop)
False
if tk->sleep_interval
True
Declare variables diff = 0
microsleep(tk->sleep_interval)
now = get_usecs(&myts)

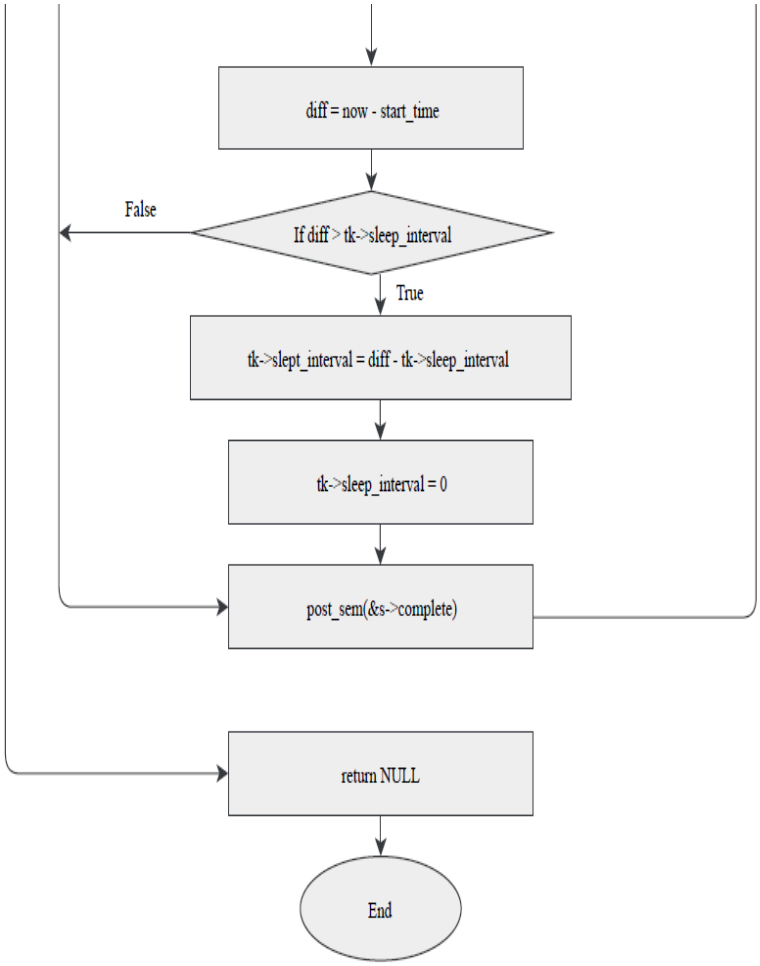


Figure 3.8 Flowchart for timekeeping_thread( )

3.2.8 init_sem( ) Function Flowchart

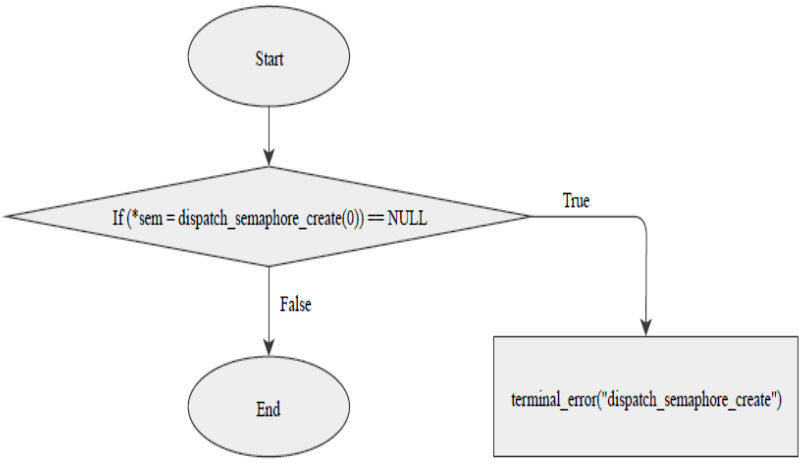


Figure 3.9 Flowchart for init_sem( )

3.2.9 wait_sem( ) Function Flowchart

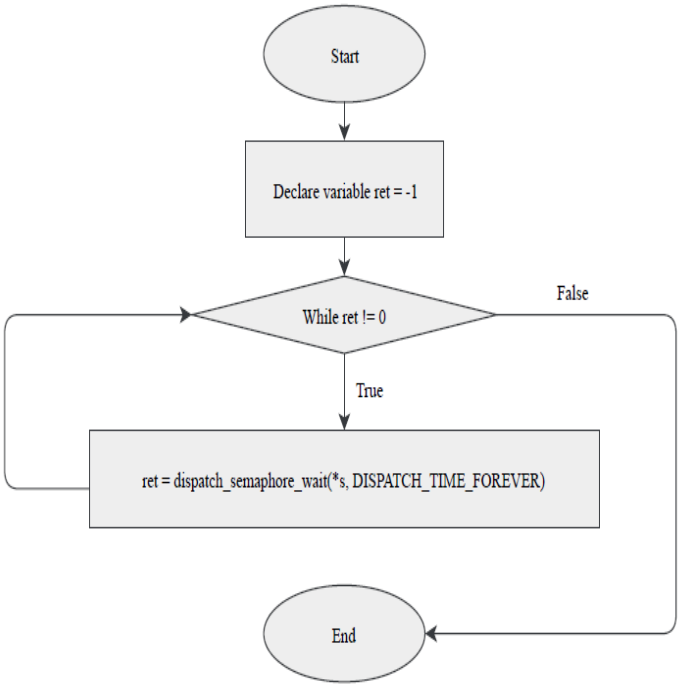


Figure 3.10 Flowchart for wait_sem( )

3.2.10 trywait_sem( ) Function Flowchart

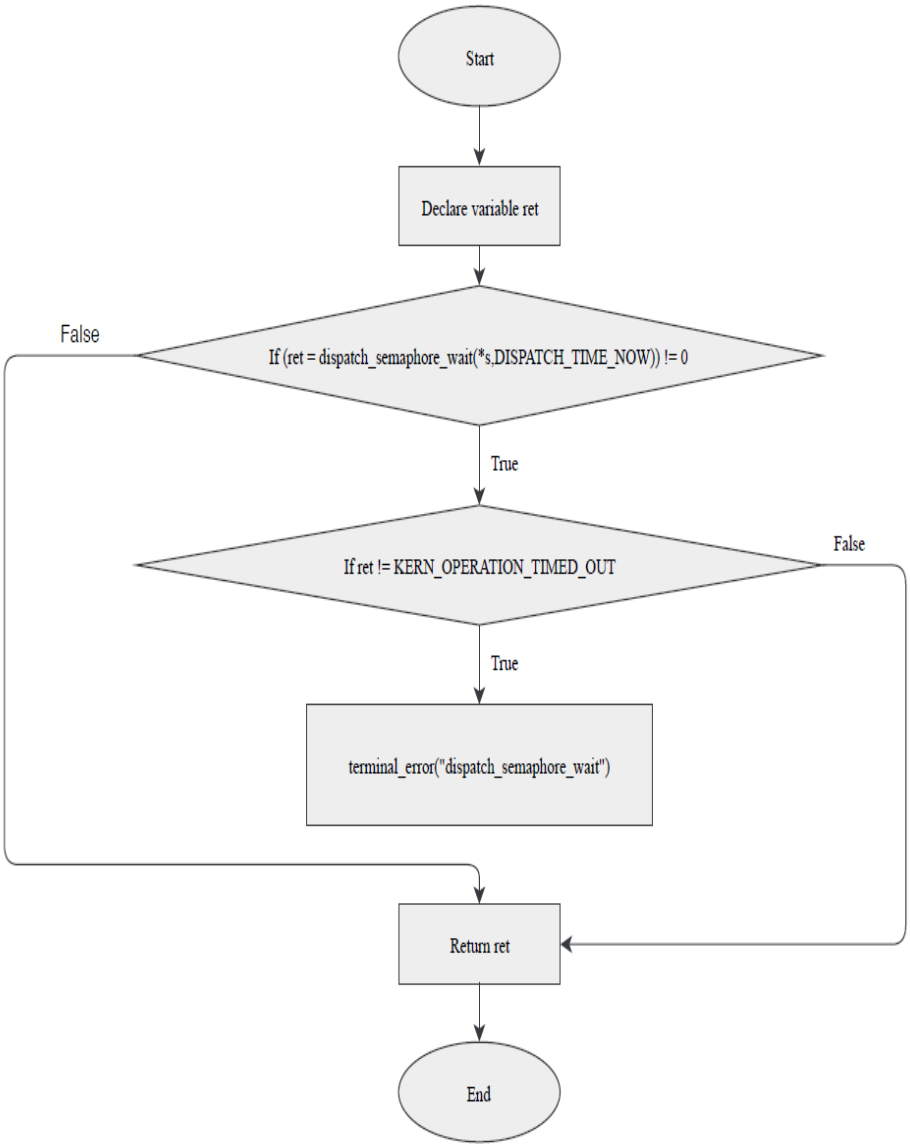


Figure 3.11 Flowchart for trywait_sem( )

3.2.11 post_sem( ) Function Flowchart

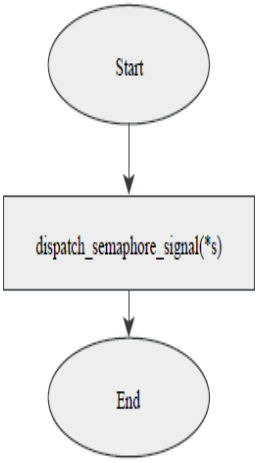


Figure 3.12 Flowchart for post_sem( )

3.2.12 emulate_none( ) Function Flowchart

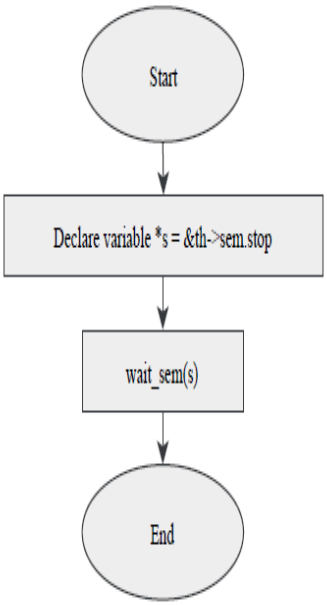


Figure 3.13 Flowchart for emulate_none( )

3.2.13 emulate_audio( ) Function Flowchart

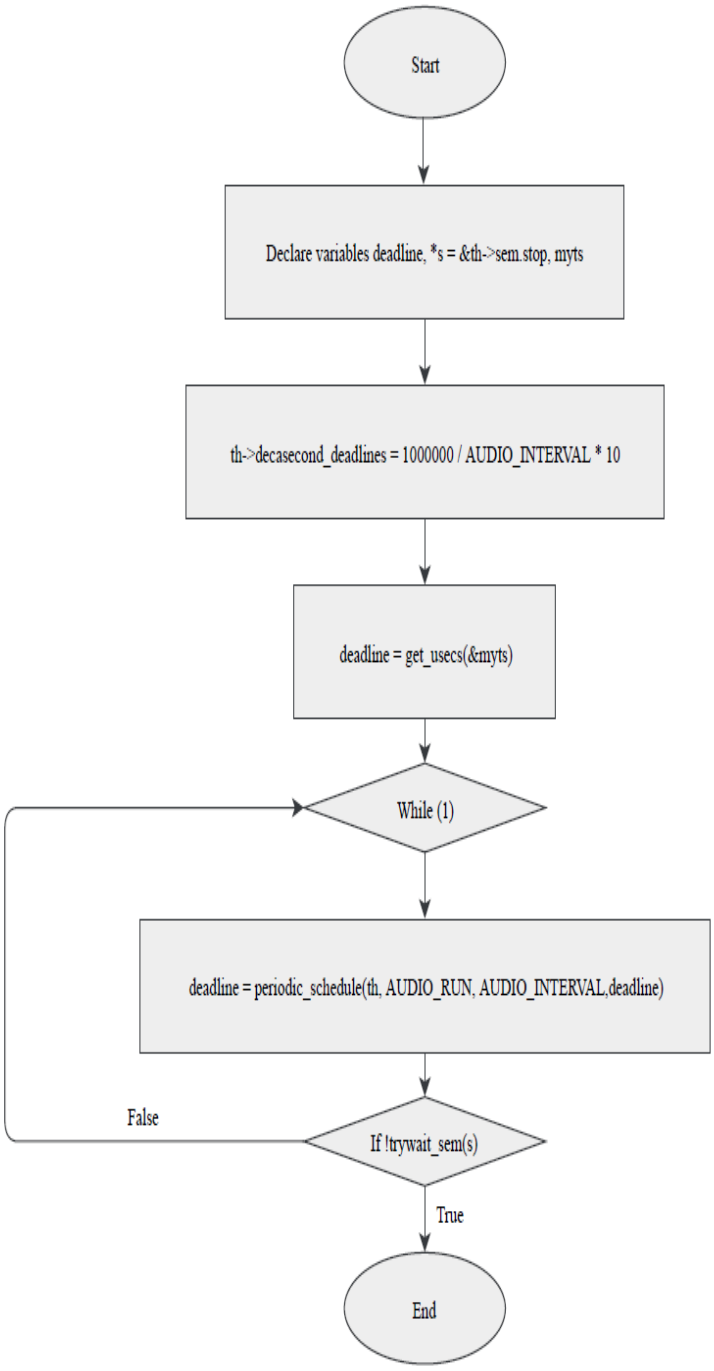


Figure 3.14 Flowchart for emulate_audio( )

3.2.14 emulate_video( ) Function Flowchart

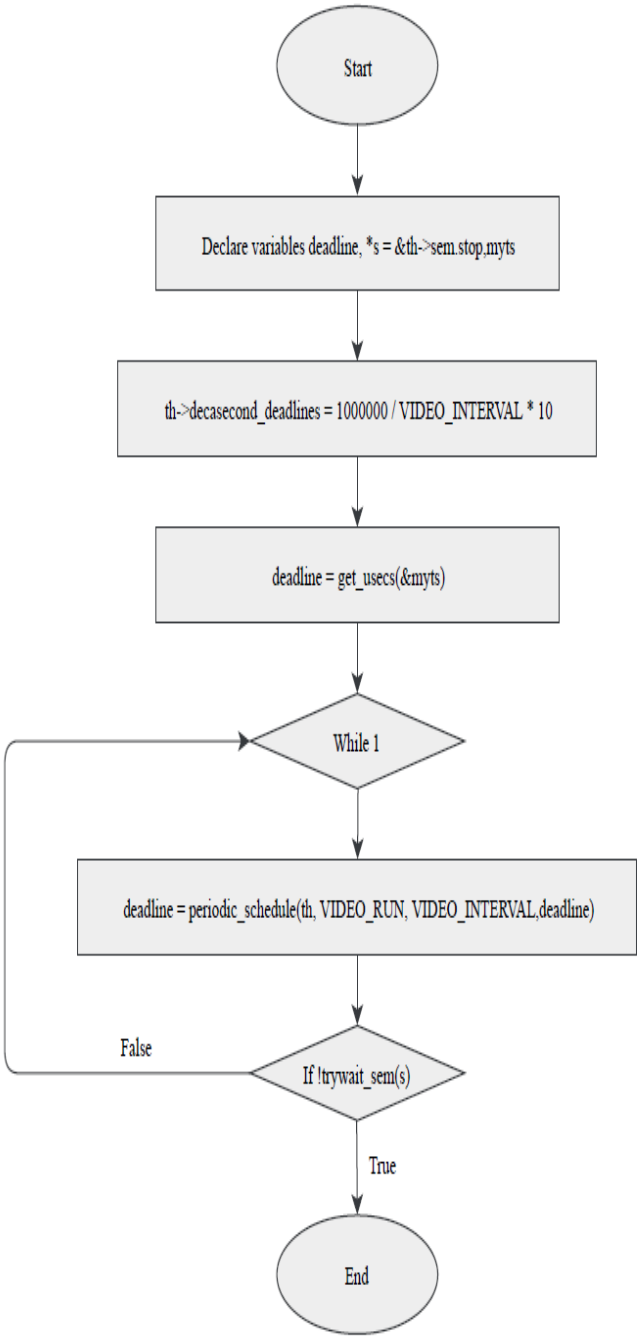
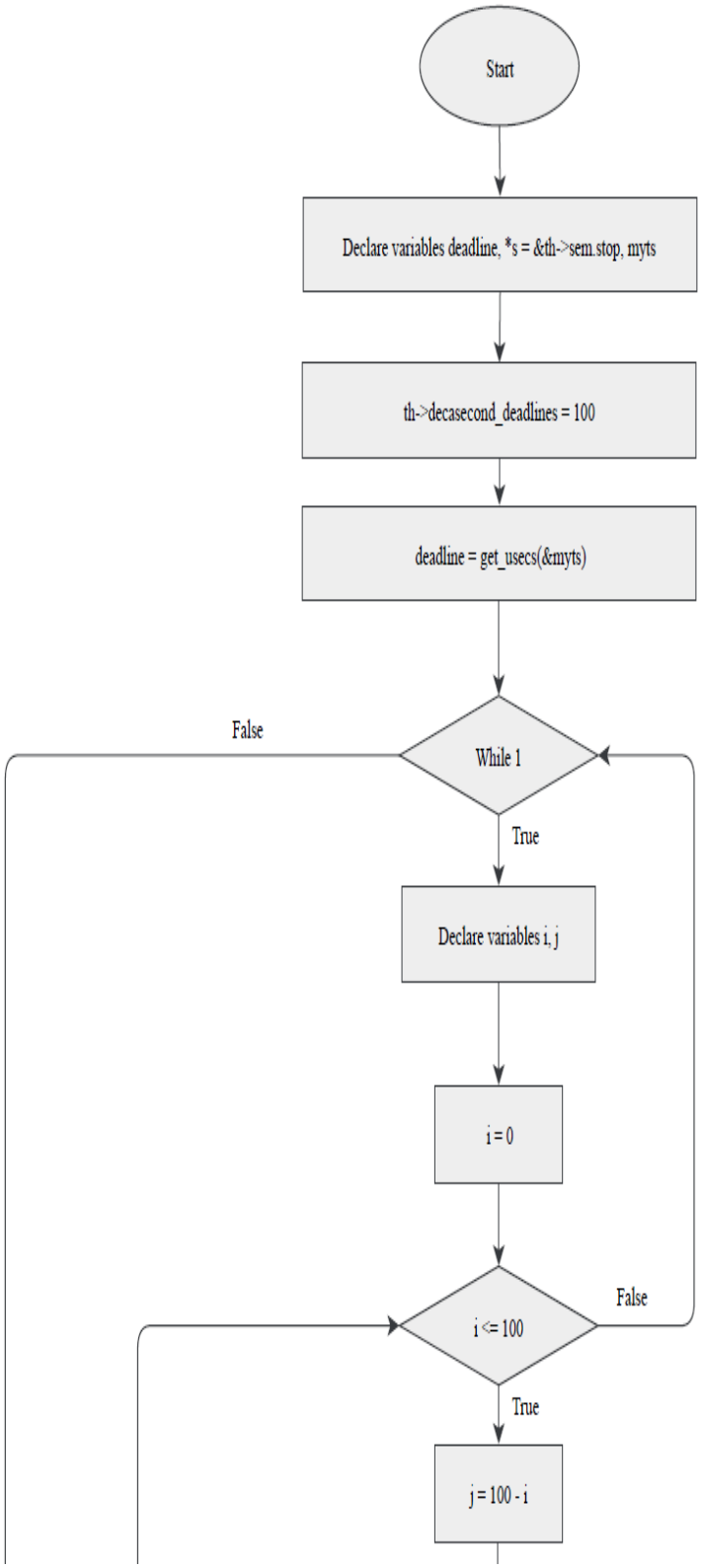


Figure 3.15 Flowchart for emulate_video( )

3.2.15 emulate_x( ) Function Flowchart



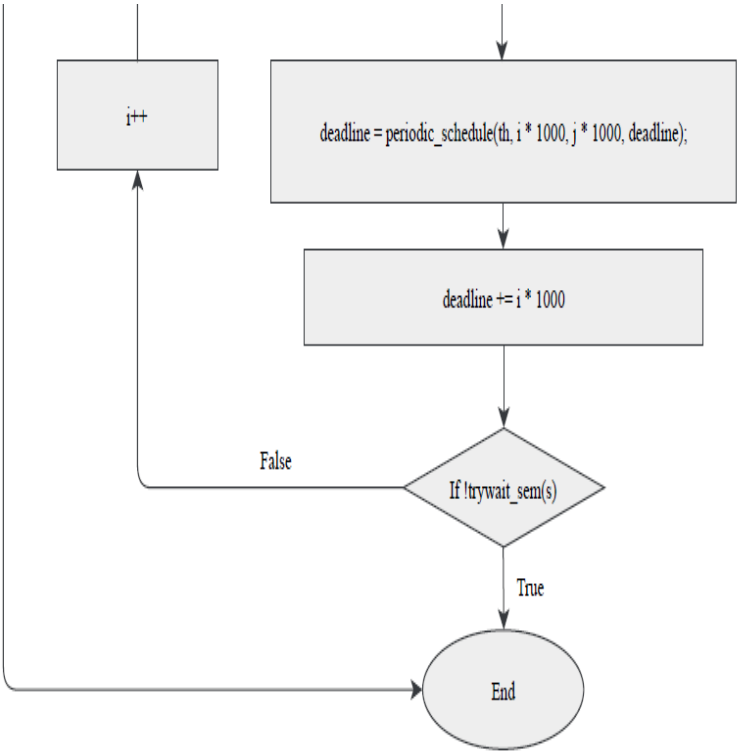
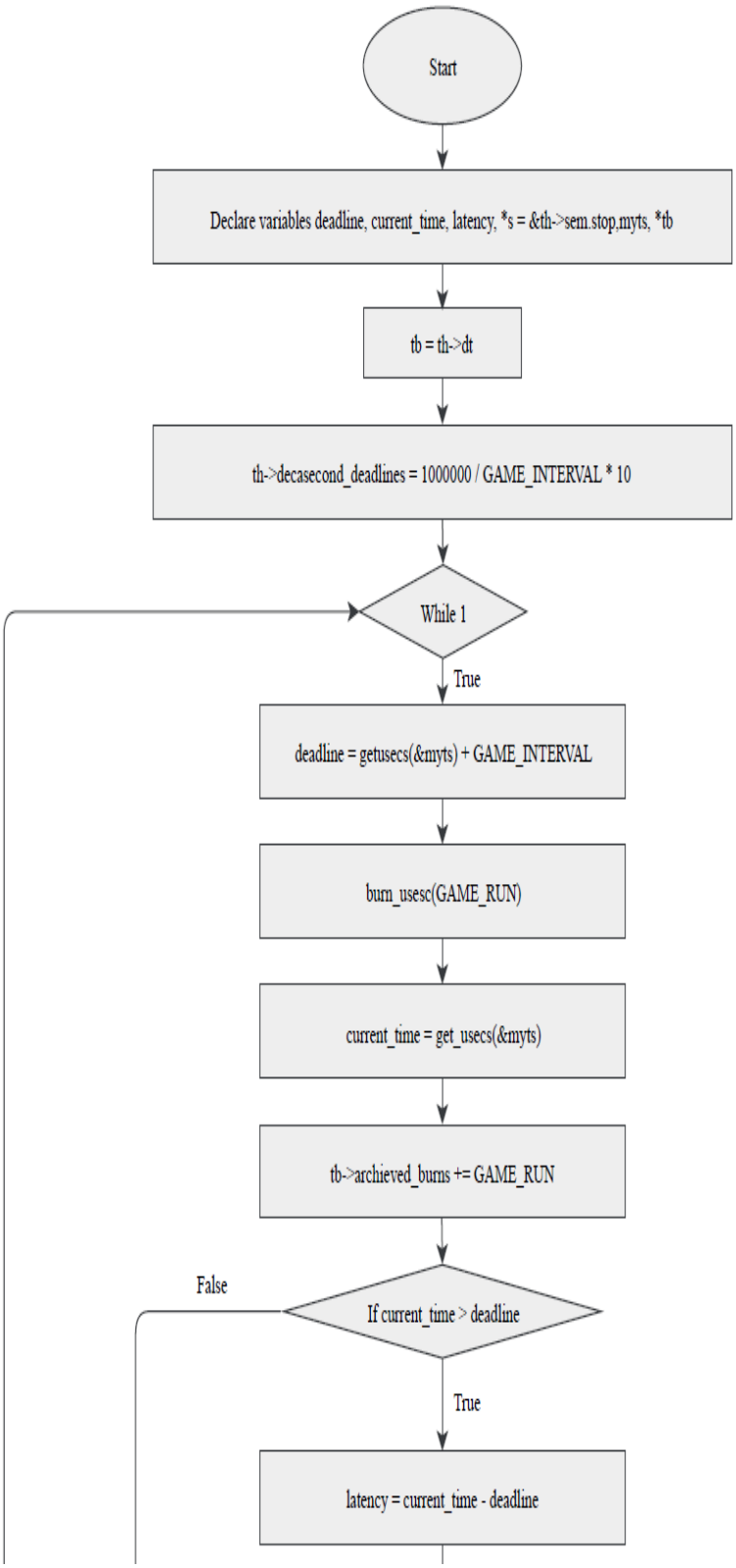


Figure 3.16 Flowchart for emulate_x( )

3.2.16 emulate_game( ) Function Flowchart



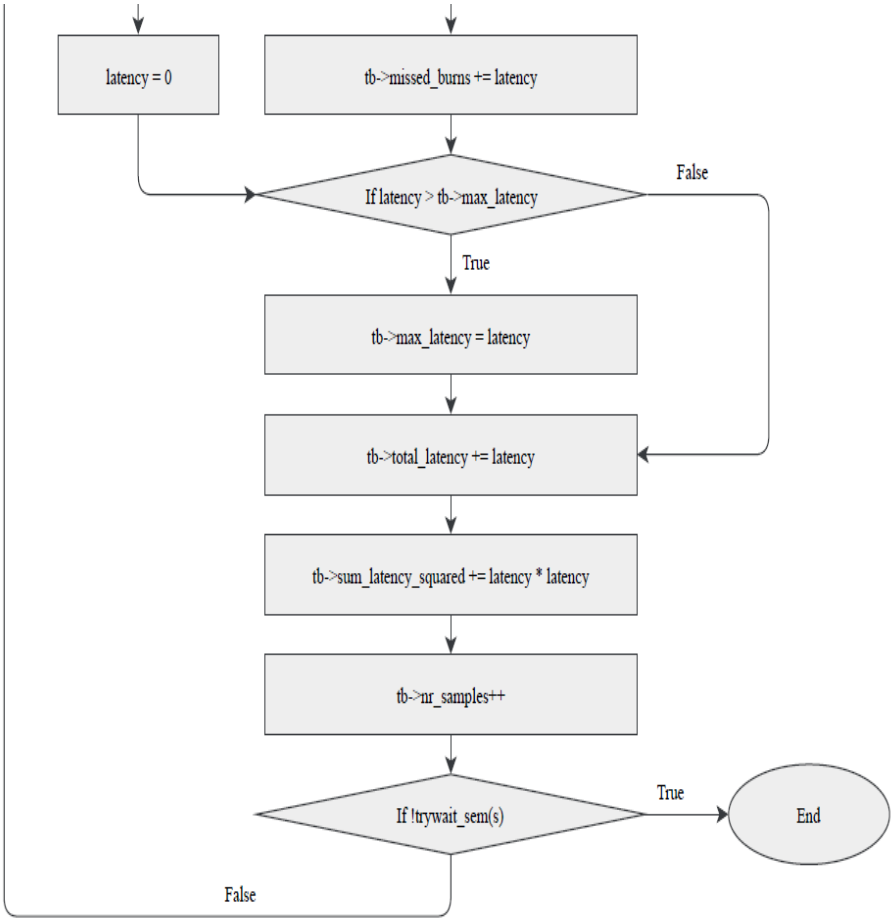
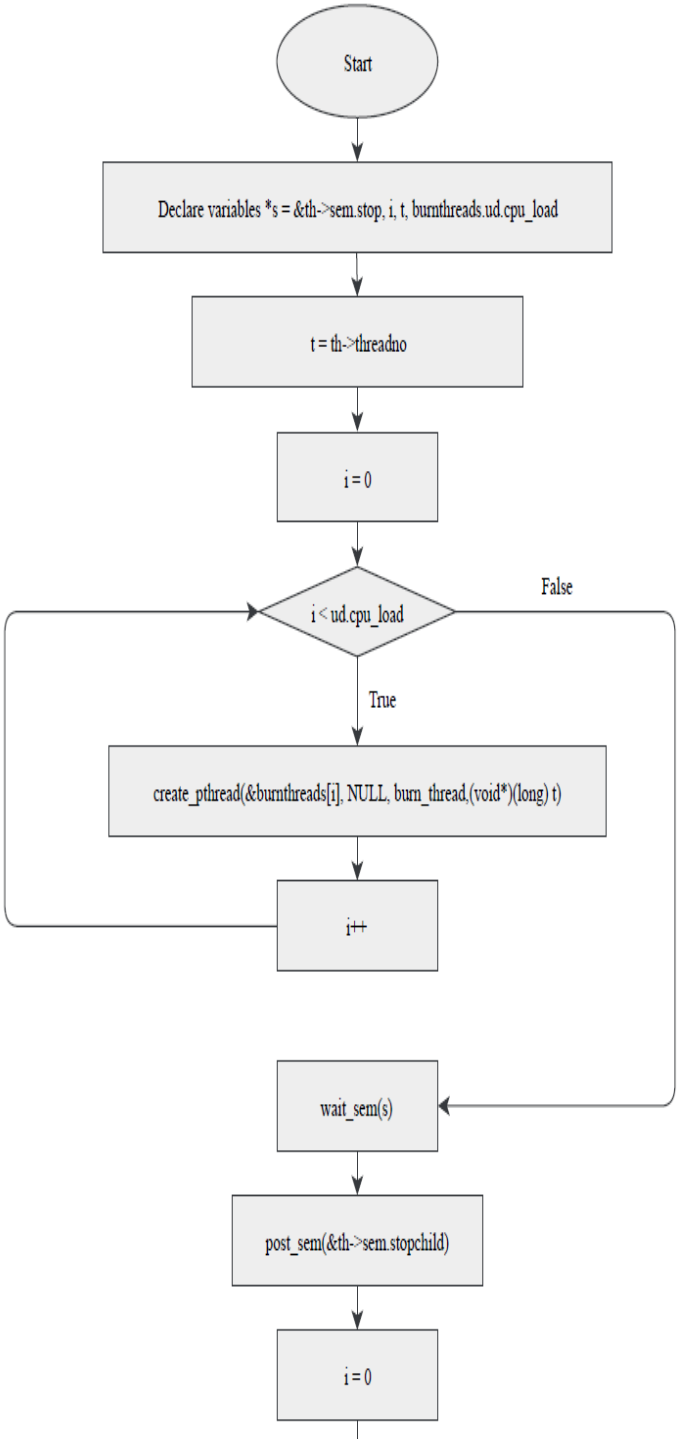


Figure 3.17 Flowchart for emulate_game( )

3.2.17 emulate_burn( ) Function Flowchart



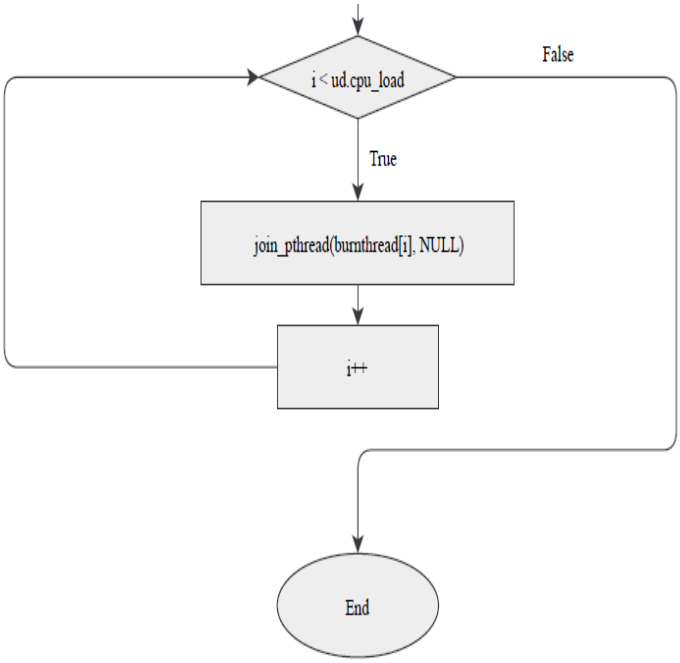
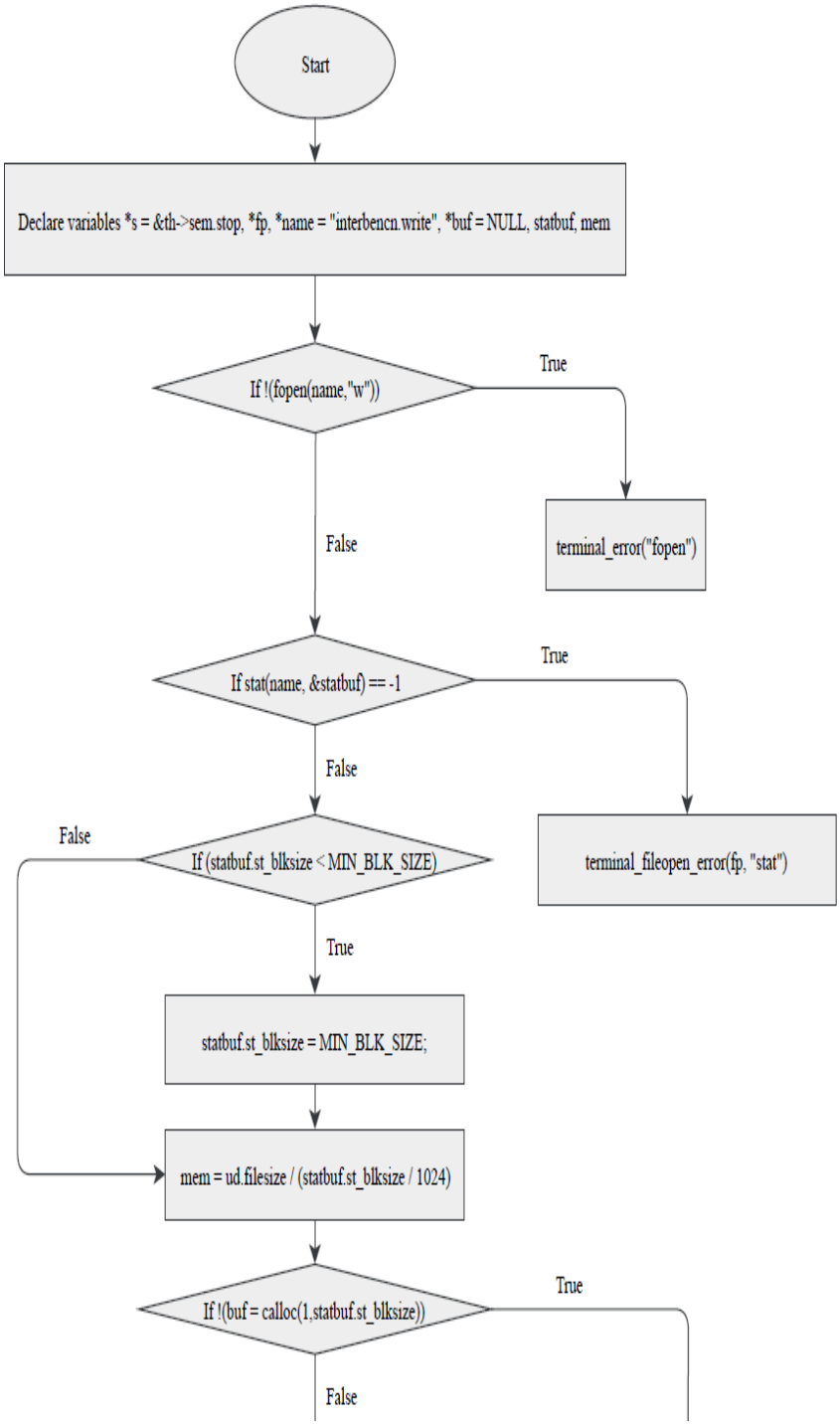
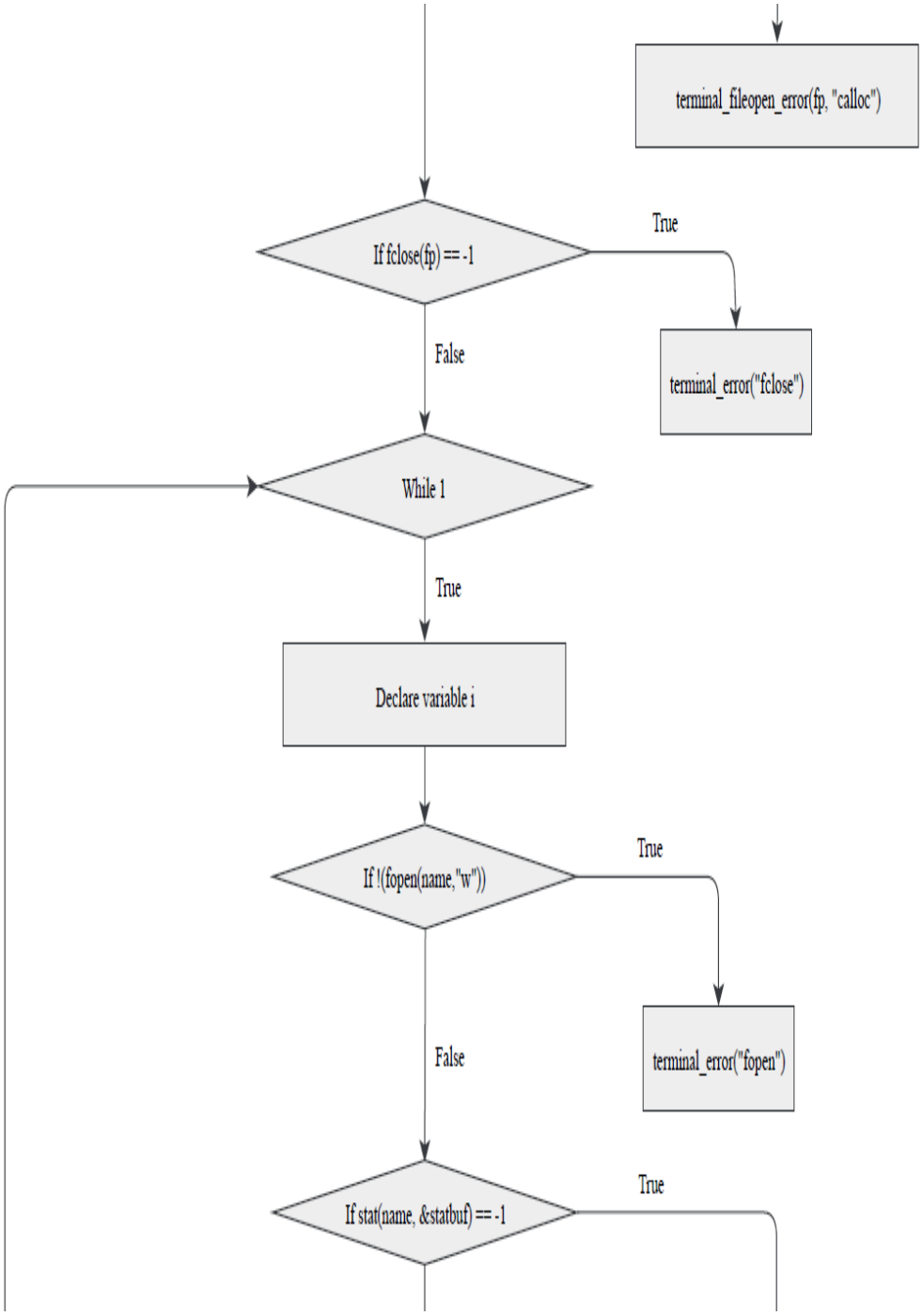
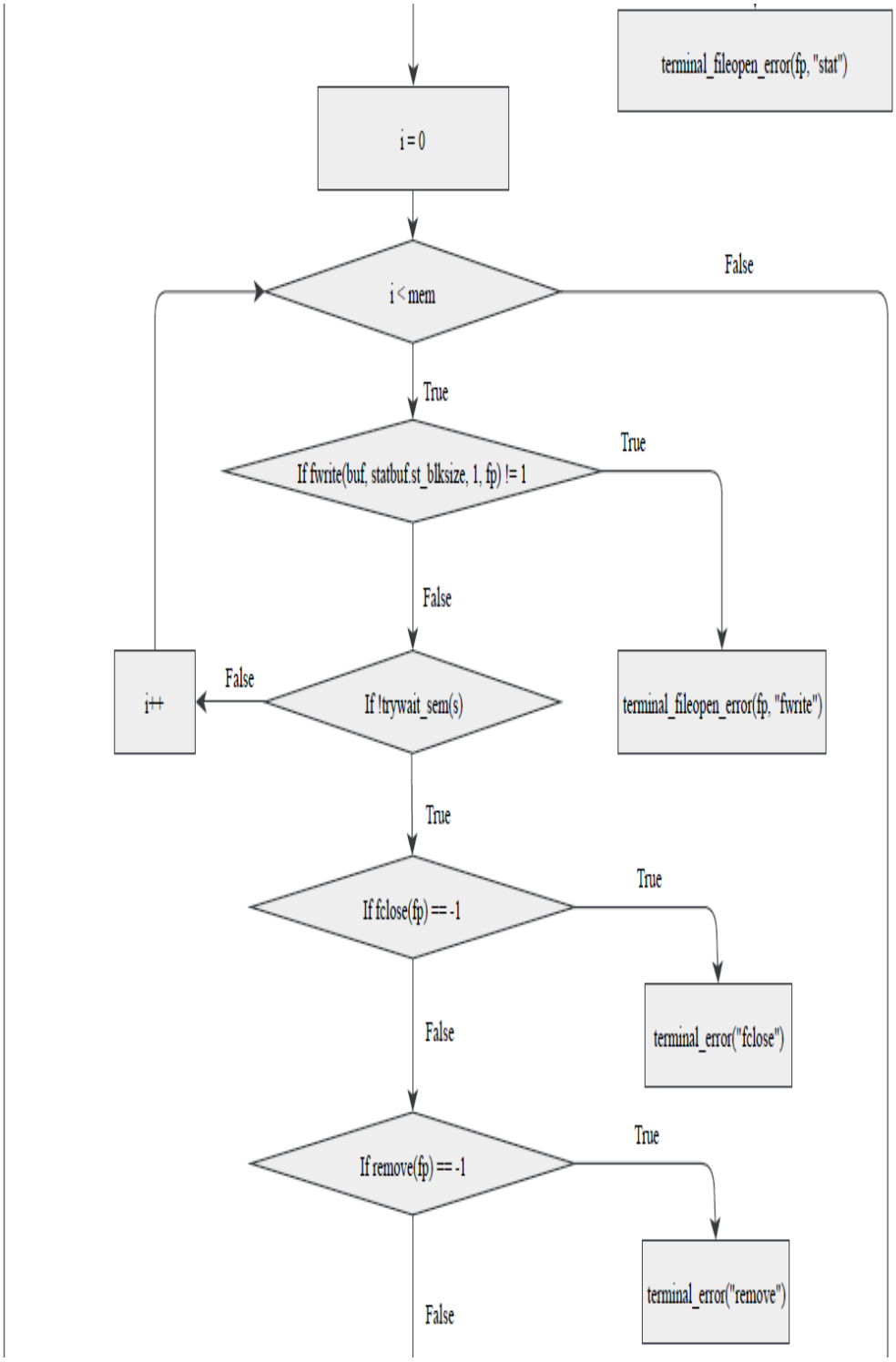


Figure 3.18 Flowchart for emulate_burn( )

3.2.18 emulate_write( ) Function Flowchart




terminal_fileopen_error(fp, "calloc")
If fclose(fp) == -1
True
terminal_error("fclose")
False
While 1
True
Declare variable i
If !(fopen(name,"w"))
True
terminal_error("fopen")
False
If stat(name, &statbuf) == -1
True


terminal_fileopen_error(fp, "stat")
i = 0
i < mem
False
True
If fwrite(buf, statbuf.st_blksize, 1, fp) != 1
True
False
i++
False
If !trywait_sem(s)
terminal_fileopen_error(fp, "fwrite")
True
If fclose(fp) == -1
True
terminal_error("fclose")
False
If remove(fp) == -1
True
terminal_error("remove")
False

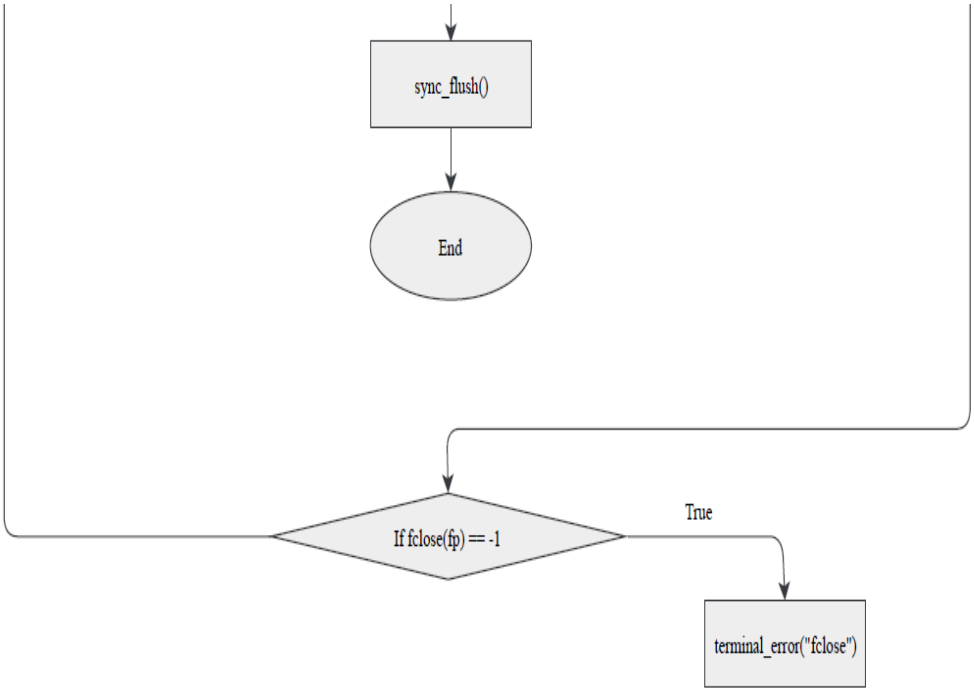
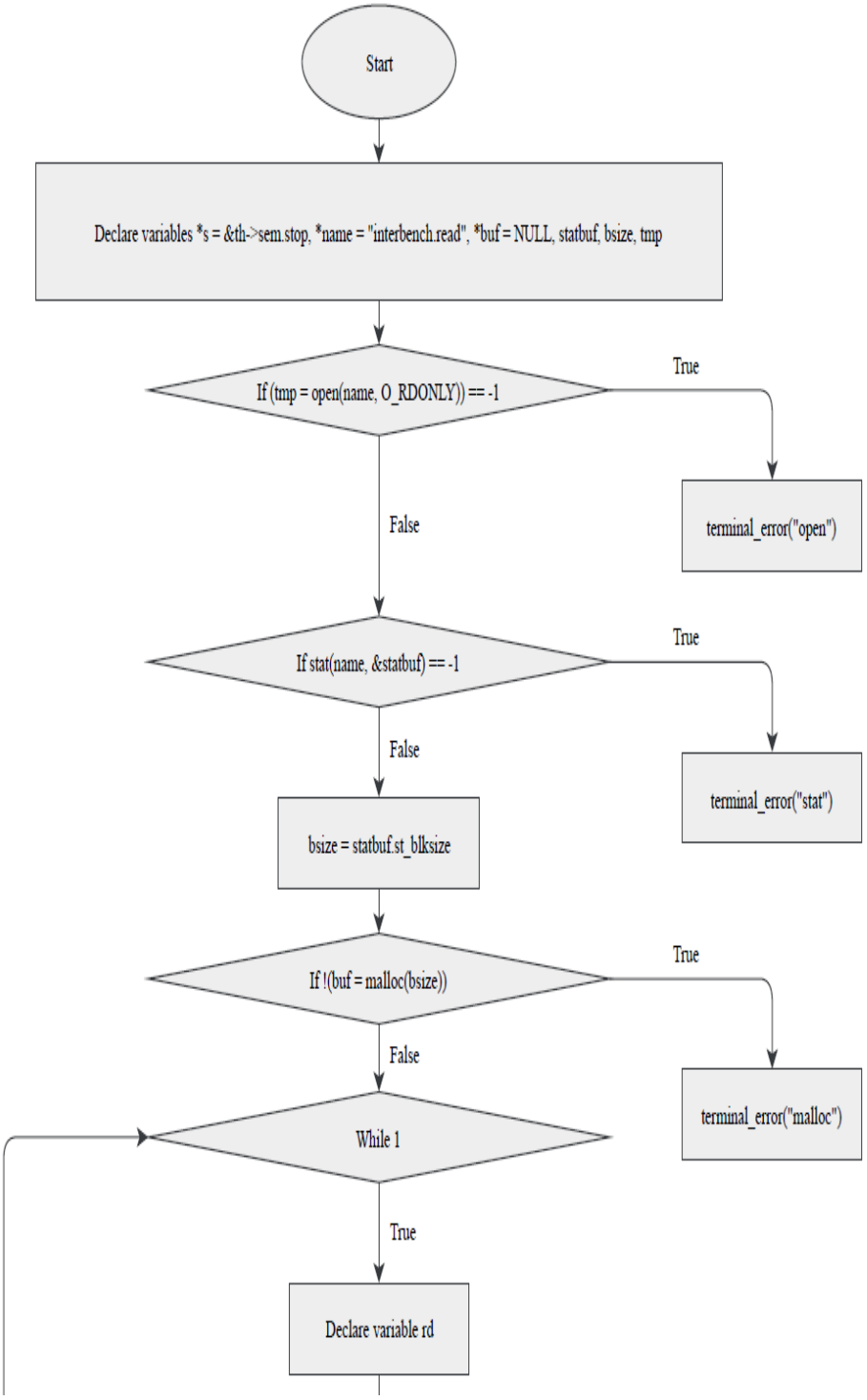


Figure 3.19 Flowchart for emulate_write( )

3.2.19 emulate_read( ) Function Flowchart



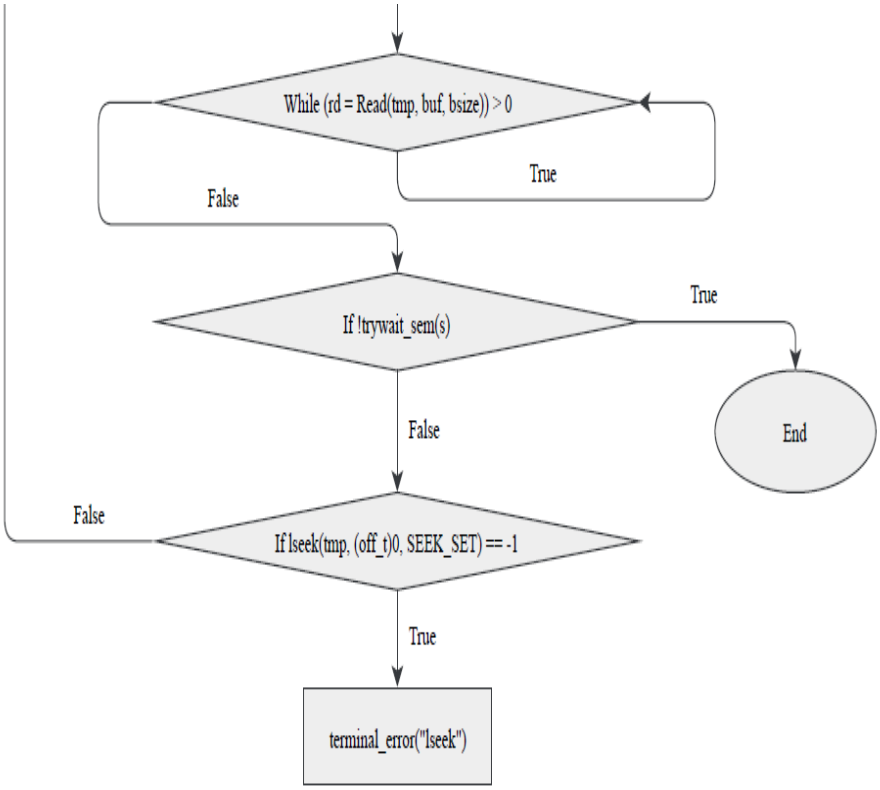
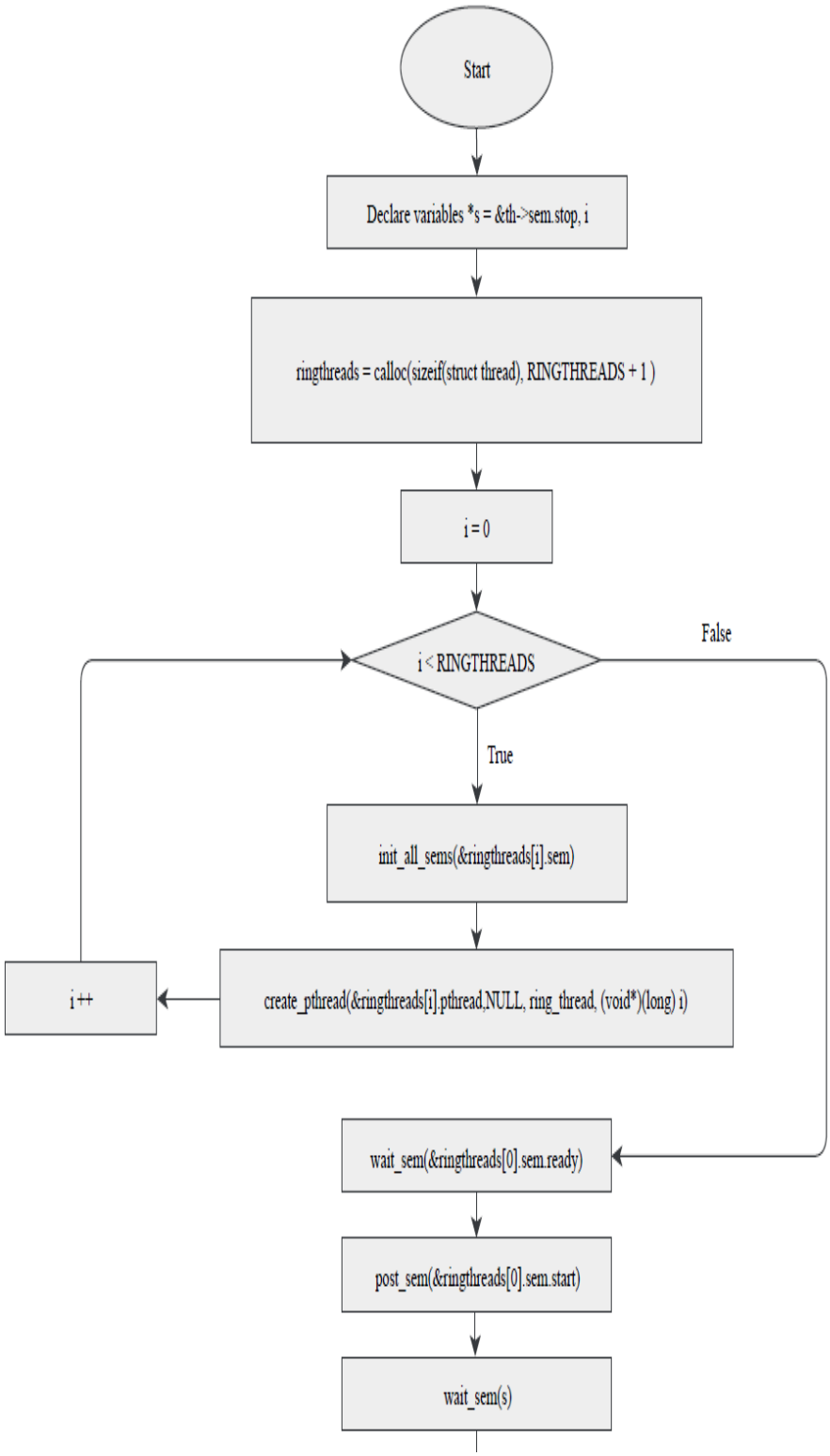


Figure 3.20 Flowchart for emulate_read( )

3.2.20 emulate_ring( ) Function Flowchart



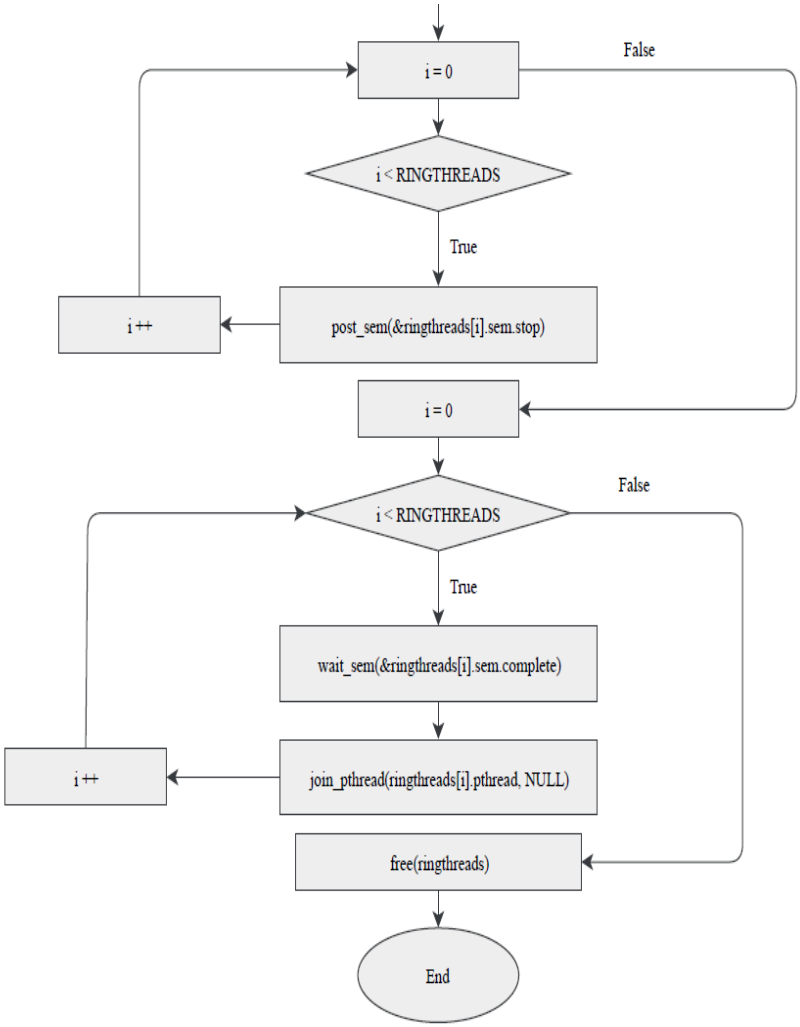
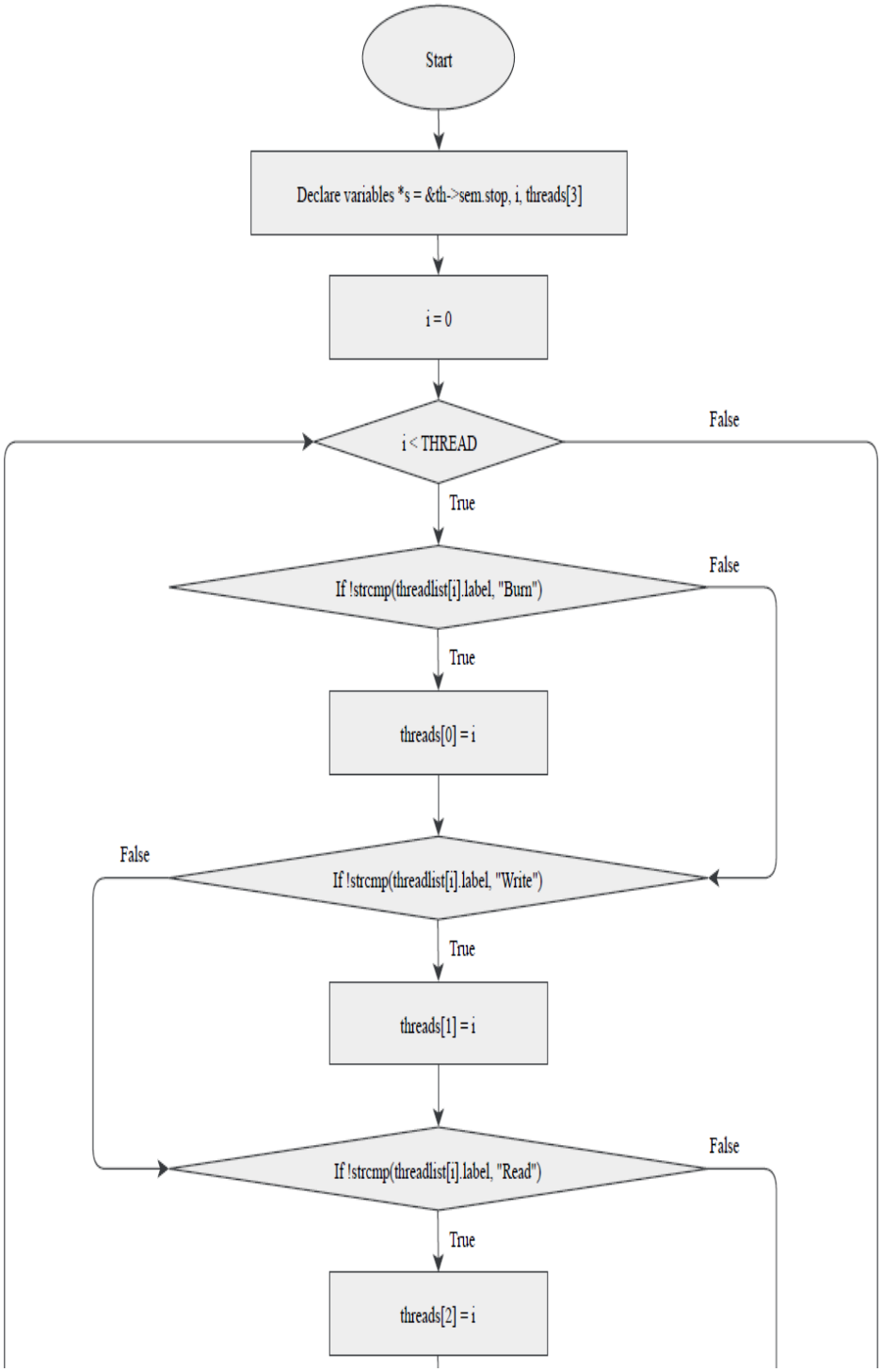
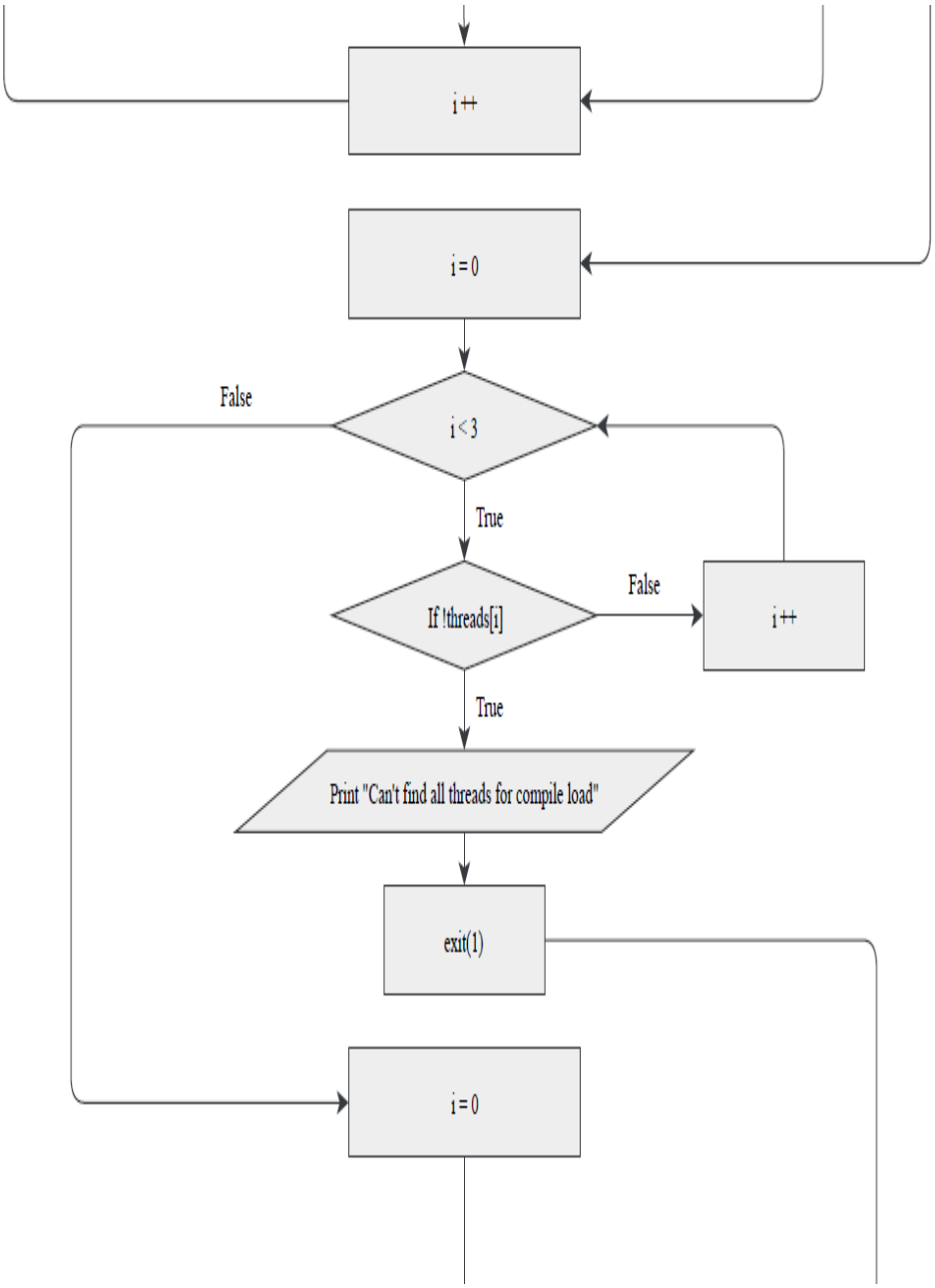


Figure 3.21 Flowchart for emulate_ring( )

3.2.21 emulate_compile( ) Function Flowchart




i ++
i = 0
False
i < 3
True
If !threads[i]
False
i ++
True
Print "Can't find all threads for compile load"
exit(1)
i = 0

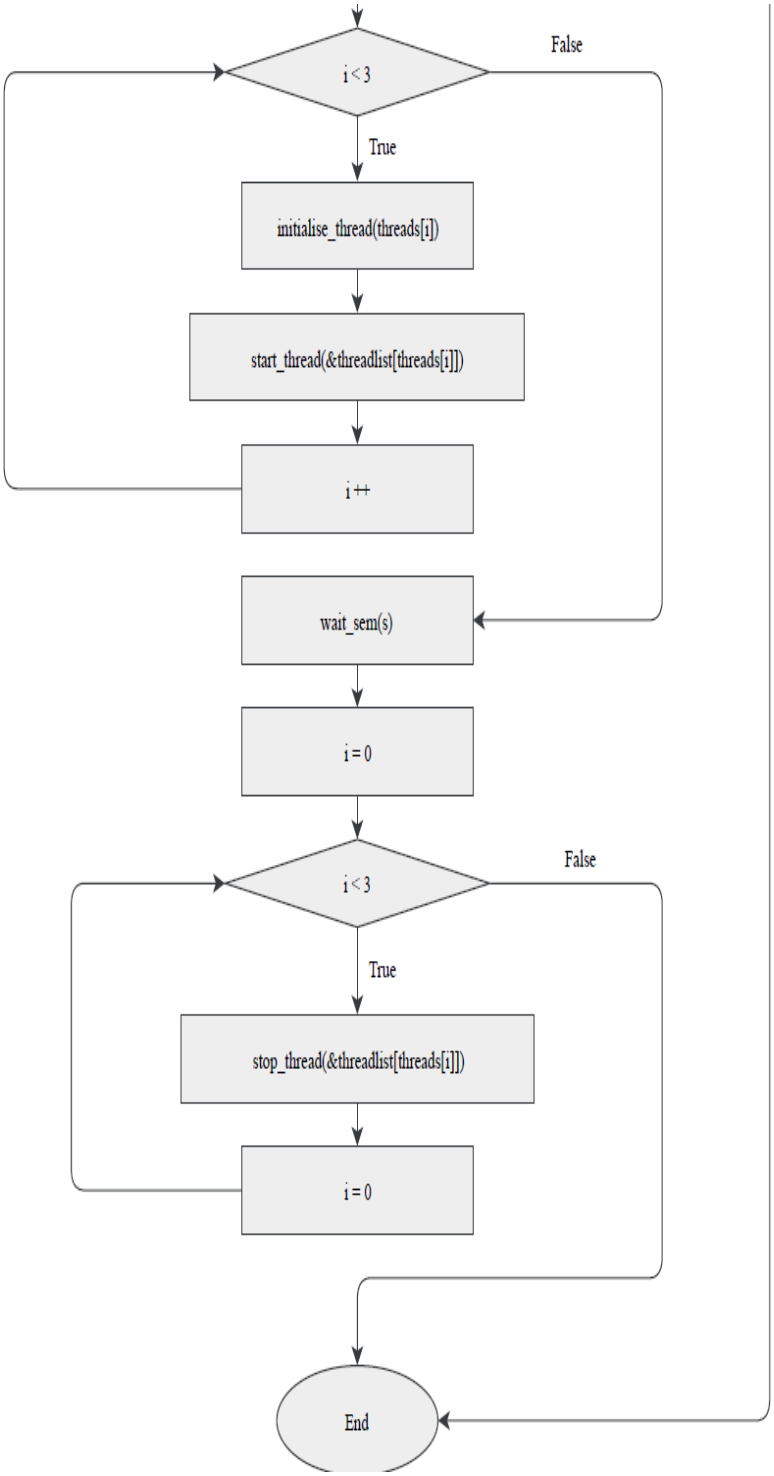


Figure 3.22 Flowchart for emulate_compile( )

3.3 System Design Explanation

3.3.1 Obtaining Total Physical Memory

```
void get_ram(void)
{
    FILE *meminfo,*fp;
    char aux[256] = {}, buf[BUFFERSIZE];
    char *command = "sysctl hw.memsize";
    long ram;

    /* Get RAM using bash script method [1.3] [Start] */
    ud.ram = 0;
    if ((meminfo = popen(command,"r")) == NULL)
    {
        printf("\nError opening pipe for get_ram \n");
    }
    while (fgets(buf,BUFFERSIZE,meminfo) != NULL)
    {
        sscanf(buf,"hw.memsize: %lu",&ram);
        ud.ram = ram/1000; // byte to kB conversion
        printf("RAM in kB : %lu kB \n",ud.ram);
    }
    if (fclose(meminfo))
    {
        printf("Command not found or exited with error status. \n");
    }
    /* Get RAM using bash script method [1.3] [End] */

    /* Limit filesize to 1GB */
    if (ud.ram > 1000)
        ud.filesize = 1000000;
    else
        ud.filesize = ud.ram;
}
```

Figure 3.23 Changes Made in get_ram( )

When the ported Interbench is executed, the main( ) function of the benchmark program will call get_ram( ) function to determine the total physical memory in the system. This system information is required for emulate_read( ) to simulate reading of a file with the size of the physical memory from the disk and emulate_write( ) to simulate a write operation to a file located in a disk with the size of physical memory.

Since /proc/meminfo file is missing in macOS, bash scripting method is used to access the required information instead of reading from a file directly. Unlike Linux, “sysctl hw.memsize” which allows the retrieval of total physical memory returns output in bytes instead of Kilobytes (KB). Therefore, the returned output is divided by 1000 to

convert bytes into kilobytes before assigning it to ud.ram. For verification purpose, the value stored in ud.ram is included in the output of the ported Interbench as shown in Figure 3.24 below.

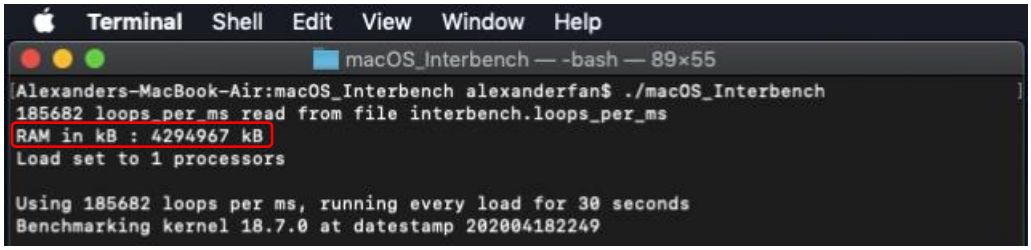


Figure 3.24 Value for ud.ram in Program Output

3.3.2 Semaphore Operations

In the benchmark program, emulation threads and timekeeping threads will gain access to shared resources by using semaphores. Semaphores for both Linux and macOS are based on the traditional counting semaphores. However, for macOS, Grand Central Dispatch's (GCD) dispatch semaphores are used in the ported Interbench because the semaphore.h in macOS which is similar to Linux's semaphore.h header contains deprecated functions. Table 3.1 below shows the mapping of the original functions from Linux semaphores to GCD semaphores.

Linux	macOS
sem_init(sem_t *sem, int pshared, unsigned int value)	dispatch_semaphore_create(int value)
sem_wait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_FOREVER)
sem_trywait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_NOW)
sem_post(sem_t *s)	dispatch_semaphore_signal(dispatch_semaphore_t *s)

Table 3.1 Mapping Semaphores in Linux to macOS GCD semaphores

dispatch_semaphore_create is used to create a semaphore with argument "value" as the initial value of the semaphore.

If the second argument of dispatch_semaphore_wait is "DISPATCH_TIME_FOREVER", the value of the semaphore will be decremented directly. If the semaphore value falls below 0, the function will wait until a signal is received.

If the second argument of dispatch_semaphore_wait is "DISPATCH_TIME_NOW", the function will return a value immediately instead of blocking.

dispatch_semaphore_signal is used to increment the value of the semaphore, if the semaphore value falls below 0, a waiting thread will be woke before the function returns.

In order to solve the “Function not implemented” warning shown when POSIX semaphore’s sem_init( ) is used to create semaphores for emulation threads and timekeeping thread, Grand Central Dispatch’s dispatch_semaphore_create( ) function is used to create a new semaphore with value 0 as the semaphore’s initial value. If the operation is successful, the function will return the newly created semaphore. Else, NULL will be returned and an error message will be shown in the terminal.

```
void init_sem(dispatch_semaphore_t *sem)
{

    if ((*sem = dispatch_semaphore_create(0)) == NULL)
    {
        terminal_error("dispatch_semaphore_create");
    }
}
```

Figure 3.25 Changes Made in init_sem( )

The dispatch_semaphore_wait( ) function will perform a decrement on the semaphore’s value similar to the sem_wait( ) function for POSIX semaphores. If the value of semaphore s falls below 0, the function will wait indefinitely until a signal is received. A value of 0 will be returned to “ret” if the operation is successful.

```
void wait_sem(dispatch_semaphore_t *s)
{
    int ret = -1;
    while (ret != 0)
    {
        ret = dispatch_semaphore_wait(*s,DISPATCH_TIME_FOREVER);
    }
}
```

Figure 3.26 Changes Made in wait_sem( )

POSIX semaphore’s sem_trywait( ) function is implemented by using the same dispatch_semaphore_wait( ) as in the one applied in wait_sem( ). However, the second argument is replaced with the DISPATCH_TIME_NOW constant which allows the function to return immediately instead of waiting indefinitely. The function will return 0 if the operation is completed successfully. If the operation has timed out, a value of 49 will be returned by the function, indicating KERN_OPERATION_TIME_OUT.

```c
int trywait_sem(dispatch_semaphore_t *s)
{
    int ret;

    if ((ret = dispatch_semaphore_wait(*s,DISPATCH_TIME_NOW)) != 0)
    {
        if (ret != KERN_OPERATION_TIMED_OUT)
        {
            terminal_error("dispatch_semaphore_wait");
        }
    }
    return ret;
}
```

Figure 3.27 Changes Made in trywait_sem( )

To release a shared resource, the sem_post( ) function in the original Interbench is replaced by dispatch_semaphore_signal( ). When this function is called, the value of semaphore s will be incremented by one and checking will be done on the value. If the value is negative, the function will wake a thread in the waiting queue before returning.

```c
void post_sem(dispatch_semaphore_t *s)
{
    dispatch_semaphore_signal(*s);
}
```

Figure 3.28 Changes Made in post_sem( )

With the functions of original POSIX semaphores reimplemented successfully using GCD semaphores, the ported Interbench can finally proceed with the simulation of interactive tasks and background loads. Figure 3.29 below shows the output containing the interactivity performance benchmark results for Audio interactive task.

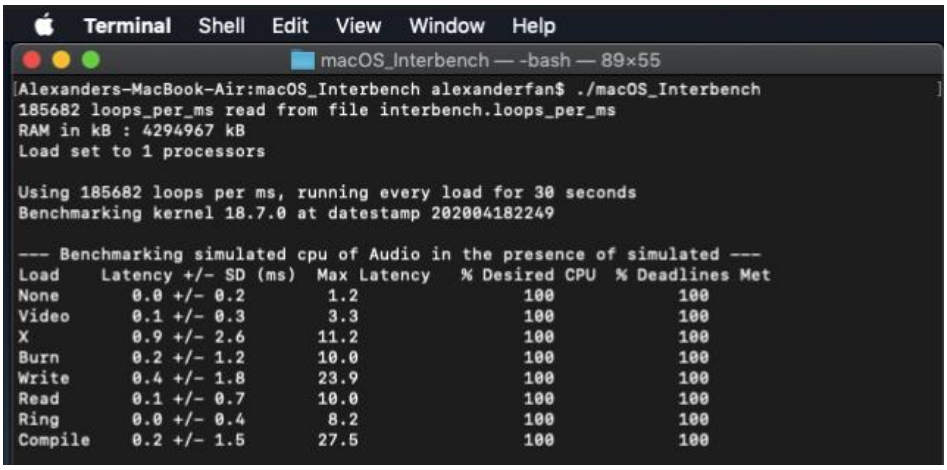


Figure 3.29 Output Containing Benchmark Results for Audio

3.4 Implementation Issues and Challenges

In the process of porting the original Interbench benchmark program from Linux to macOS, there are a few challenges faced that require a great amount of research and time. The first challenge is regarding the implementation of semaphores in macOS. In the development environment running macOS 10.14.6 (Mojave), some of the functions related to semaphore have been made unavailable in macOS's "semaphore.h" system header. This causes warnings to pop up during the compilation of the ported Interbench for functions that worked fine in Linux such as sem_init( ) for the creation of a semaphore although both of the operating systems are Unix-liked systems.

Besides, some files which is available in Linux is not accessible by macOS to obtain some important system information. For instance, a function in Interbench namely get_ram( ) is used by the main( ) function to obtain details about the system's physical memory for the use of one of the simulation load called Write. In Linux, those details can be obtained by accessing "/proc/meminfo" file. However in macOS, the physical memory info can only be obtained using the bash scripting method.

CHAPTER 4 – METHODOLOGIES

4.1 Methodology and General Work Procedures

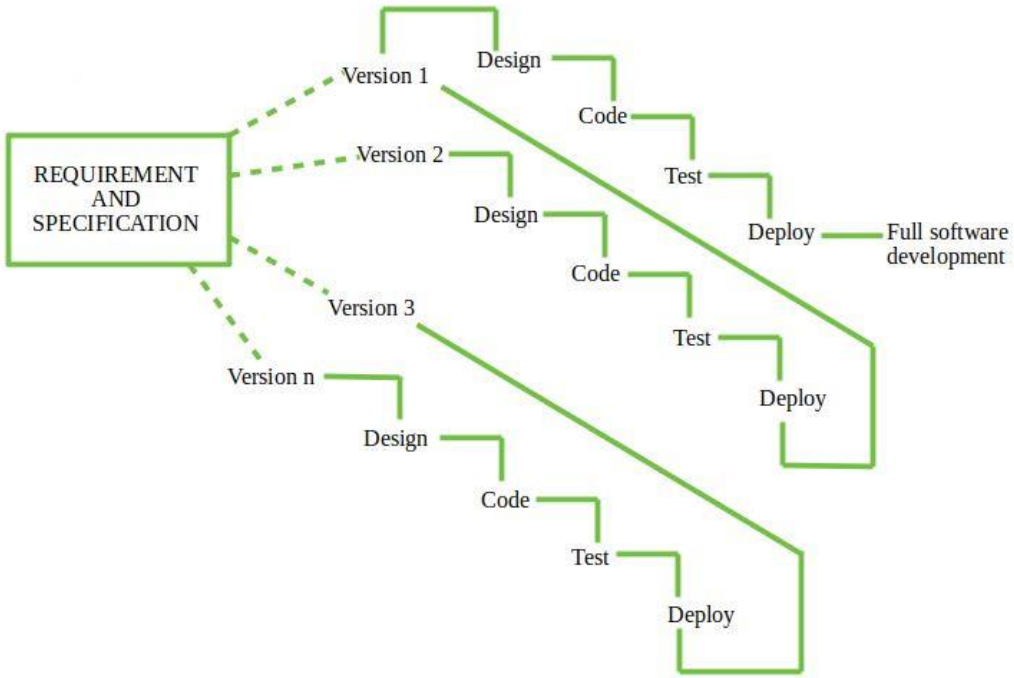


Figure 4.1 Incremental Model

Considering the time and effort required to put on the project, which requires going through more than a thousand lines of pre-developed codes in Interbench to make sure that the simulation tasks and loads can be implemented similar the software's behavior in Linux OS, the whole research and development process is based on the incremental model (GeeksforGeeks, n.d.). Porting the entire Interbench benchmark program by modifying all the simulation tasks and loads from one OS to another at once can be a difficult task. If the development is not divided properly, confusion will occur, making the researcher unsure about the suitable fundamental component to work on first to make the next component work.

If benchmark program is unable to fulfil the software requirements in the middle of the development process, it is still possible to make modifications on the original set of requirements thanks to the flexibility in incremental model. Besides that, the delivery time required for a working benchmark program can be reduced because it is easier to port working module at a time rather than porting a large and complex program at once.

Therefore, by following the incremental model, the whole benchmark program can be broken down into smaller and less complex modules. Researcher will work on the modules until the specific module is able to function as in the original Interbench benchmark program in Linux OS. After the completion of each module, the researcher will proceed to work on the next module until the ported Interbench benchmark program is fully completed. Below are the stages that will be implemented in each increment.

Design :

In this phase, the module's architectural design will be identified. The researcher will figure out the ways to design each simulation tasks and loads so that each module works as the same as in the original Interbench benchmark program. The design phase is highly demanded as an input for the coding phase.

Code :

After the completion of the module's architectural design, researcher should be able to gain more insight on the ways to implement the porting process of the simulation tasks and loads. The original design will be converted from a concept to actual C language source code. This is the stage where coding skills and debugging skills play a vital role. At the end of this stage, the researcher should be able to run the module as an executable program.

Testing :

To ensure that the module has the ability to solve the known problems obtained within the requirement analysis phase, the module's code will be tested against the defined requirements after the completing of coding phase in each increment. Various kinds of testing such as integration testing and unit testing will be performed to measure the quality of the ported benchmark program before deploying it.

Deploy :

Lastly, after the researcher performed various types of tests on the developed module, simulation tasks and loads are now ready to be used. Comparisons of interactivity can be done using same set of simulation tasks, simulation loads and system hardware between different operating systems.

4.2 Tools To Use

4.2.1 GNU Compiler Collection (GCC)

GNU Compiler Collection, also known as GNU C Compiler previously is developed by the founder of the GNU Project, Richard Stallman. It is a component which is very important for the development of applications and operating systems. For this project, several tools in the GNU Toolchain will be used. The Interbench program written in C language will be compiled using the command "gcc" in the terminal. Besides the "make" command is used to automate the compilation and build of the application. Lastly, GNU Debugger (GDB) is triggered by the command "gdb" in the terminal. GDB allows multiple breakpoints to be set to ease the troubleshooting process.

4.4.2 Interbench

Interbench simulates different tasks that use multiple system generated loads to simulate different conditions to see how well the scheduler performs. In this project, comparison will be made on the interactivity in different OS schedulers for the same set of hardware components with the usage if this benchmark program.

The simulation tasks are:

X: A thread that utilizes amount of CPU that change from time to time ranging from 0% to 100%. An idle Graphical User Interface (GUI) with a window being grabbed and dragged across the screen is simulated

Audio: A thread which to execute at an interval of 50ms and requires 5% of CPU utilization is used to simulate the Audio task.

Video: A thread which requires to utilize the 40% of CPU and attempts to receive from the CPU 60 times per second is used to simulate the Video task.

Gaming: Emulates a CPU-bound game by using as much CPU as possible.

Load simulated for the tasks:

None: Idle system.

Video: Background load generated by using video simulation thread.

X: Load generated by using X simulation thread.

Burn: Number of threads fully CPU bound that can be configured.

Write: Write a file of the size of Random Access Memory (RAM) as a stream.

Read: Reading a file of the size of RAM from the disk repeatedly.

Compile: Concurrent execution of Burn, Write and Read to simulate a heavy 'make –j4' compilation.

4.2.3 Vim Editor

As both Linux and macOS are based on Unix, Vim Editor is used for efficient addition and modification of source codes in the project. This editor provides supports for many programming languages including C.

4.2.4 Turbo Boost Switcher

In most computer systems, there will be some applications that require more processing power for users to accomplish certain tasks. To cope with these heavier workloads, a solution called Intel Turbo Boost Technology is introduced which dynamically adjusts the frequency of the CPU cores to the maximum turbo frequency to create a balance between system performance and workloads (Intel Corporation, n.d.). This technology will only be activated when the power, temperature and current specification limits of the CPU are not exceeded when the CPU is in operation. The technology's frequency and the amount of time for the CPU to remain in the turbo state differ because they are relying on factors including the hardware, software, workload and configuration of the system.

Turbo Boost Switch is a program which allows users to disable or enable Intel Turbo Boost Technology in systems running Apple's macOS as BIOS in an Apple computer is not accessible (rugarciap 2019). Intel Turbo Boost Technology is disabled in all three operating systems when Interbench benchmarks were carried out. The reason why it needs to be disable is because different operating systems can interpret the same workload in a very different manner and one of the factors affecting the frequency of Intel Turbo Boost Technology is the system's software including operating systems. So, in order to achieve a fair comparison between the three operating systems, maximum turbo frequency is not allowed for tasks involving high workloads.

4.2.5 Gnuplot

Gnuplot is a free and command-line based graph plotting tool available for macOS, Linux and Windows which allows users to obtain a visualized form of the data (Gnuplot.info 2020). The tool is convenient to use as it can be started from the terminal. In this project, it was used to generate CPU utilization graphs for macOS by providing an input data file containing CPU utilization values with an interval of one second. The generated graphs were compared with the ones from Linux and Windows for verification purpose before proceeding to the final comparison of interactivity performance.

4.3 System's Specification

Component	Specification
Name	MacBook Air A1466 (13-inch, Mid 2013)
Processor	Intel Core i5-4250U Processor (2 Cores, 4 Threads)
Memory	4GB DDR4
Disk Storage	128GB Solid State Drive (SSD)
Graphics Processor	Intel HD Graphics 5000
Display	1440x990 Backlit Display
Operating System	macOS Mojave version 10.14.6 64-bit OS Ubuntu 18.04.4 LTS 64-bit OS Windows 10 Home 64-bit OS

Table 4.1 System's Specification

4.4 Verification Plan

Before the actual comparison is conducted, the macOS Interbench which is ported in this project is verified by comparing the CPU utilization when the interactive tasks and background loads are executing. The comparison involves the ported macOS Interbench, ported Windows Interbench done by the previous researcher and the original Linux Interbench. This verification process is crucial because it allows us to ensure that the simulation can be done correctly and the results of the final comparison on interactivity performance is reliable. Although the verification was already conducted by the previous researcher for Windows Interbench and Linux Interbench, the same verification process is repeated in this project to confirm that Windows Interbench is able to behave as expected by using a different set of system hardware and Windows OS version compared to the previous research.

The CPU utilization values on macOS were obtained through the “top” and “grep” command in the terminal. The values were collected and stored into a .dat data file which is used for graph plotting using Gnuplot. As for Linux and Windows, the CPU utilization graphs were monitored and collected from System Monitor and Task Manager.

The section below contains screenshots of the Gnuplot on macOS, System Monitor on Linux and Task Manager on Windows for every interactive tasks. According to the screenshots, the pattern of CPU utilization for the three operating systems involved are identical to each other.

Interactive Task 1: Audio (5% CPU Utilization)

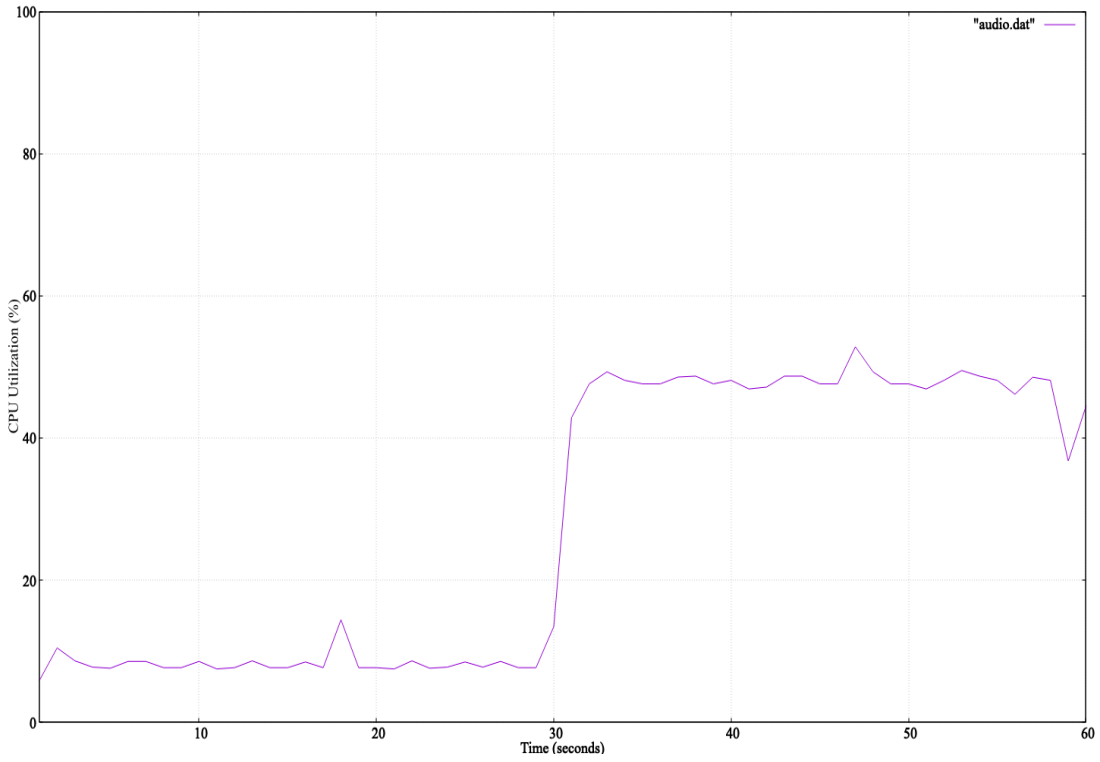


Figure 4.2 CPU Utilization Graph for Audio on macOS

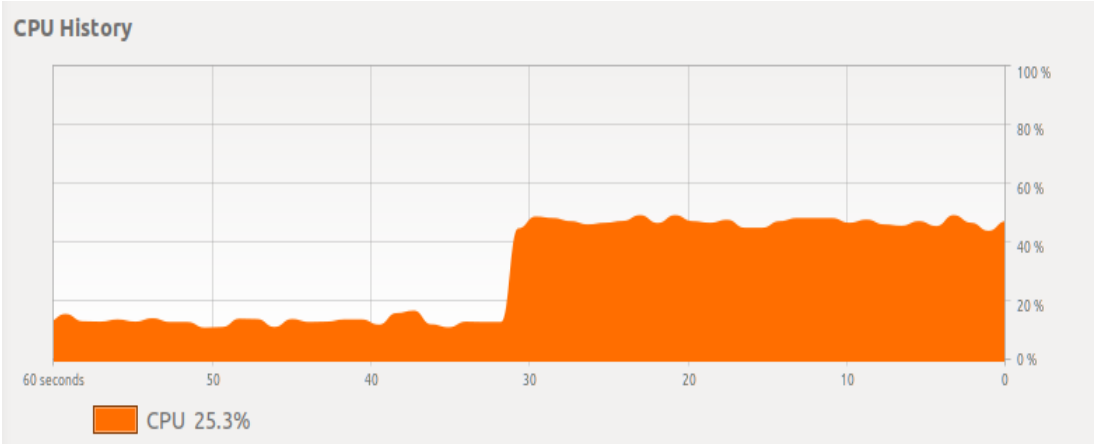


Figure 4.3 CPU Utilization Graph for Audio on Linux

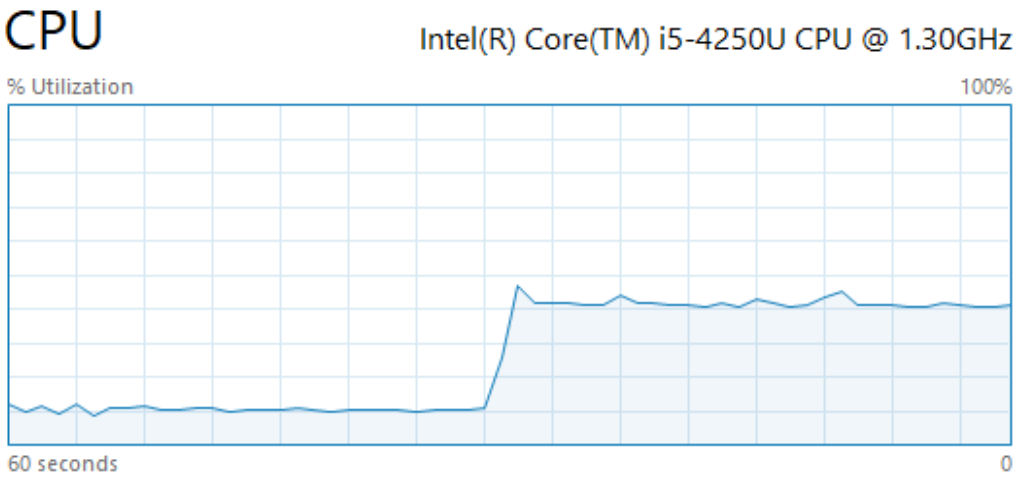


Figure 4.4 CPU Utilization Graph for Audio on Windows

Interactive Task 2: Video (40% CPU Utilization)

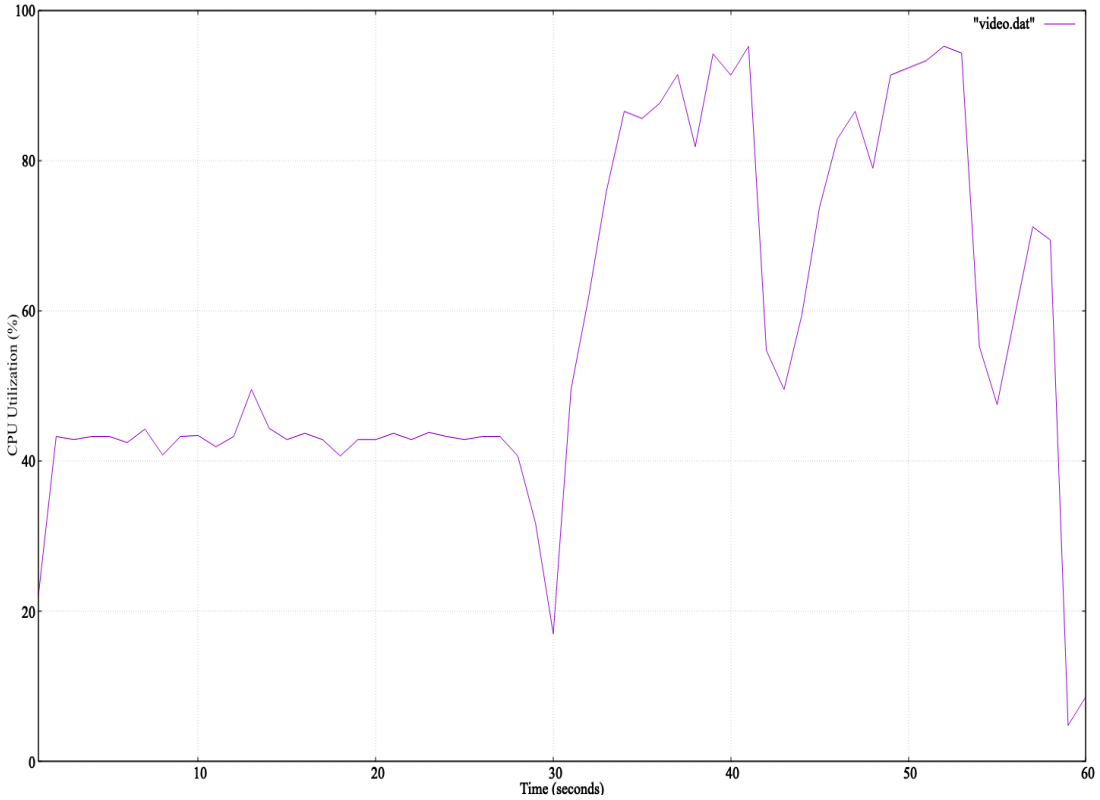


Figure 4.5 CPU Utilization Graph for Video on macOS

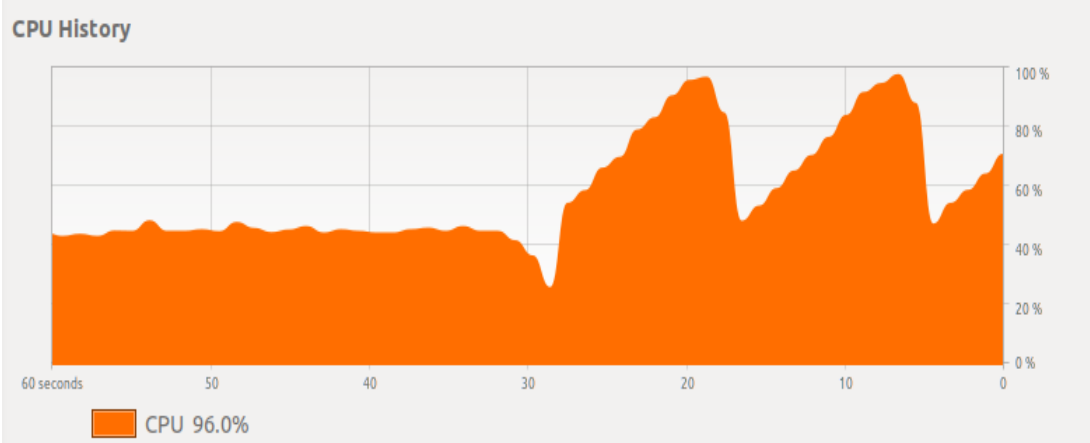


Figure 4.6 CPU Utilization Graph for Video on Linux

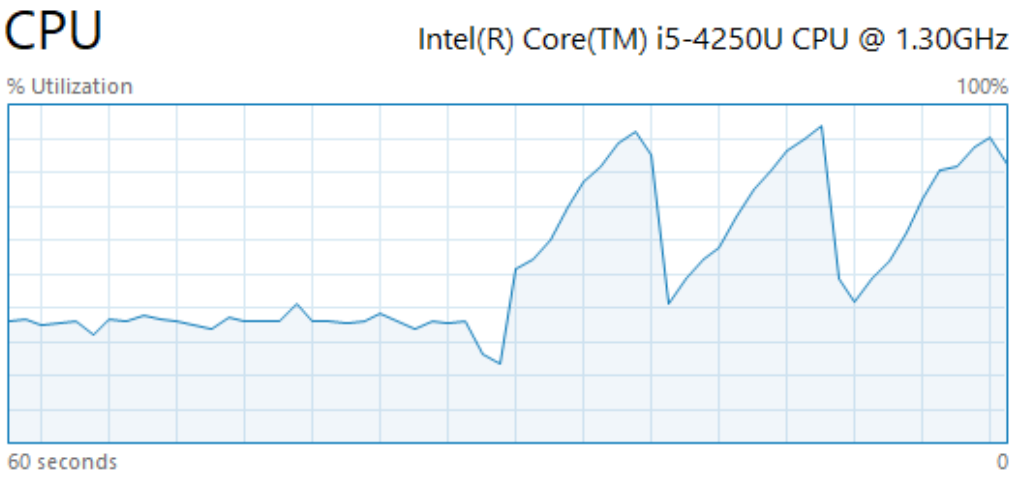


Figure 4.7 CPU Utilization Graph for Video on Windows

Interactive Task 3: X-window (0 - 100% CPU Utilization)

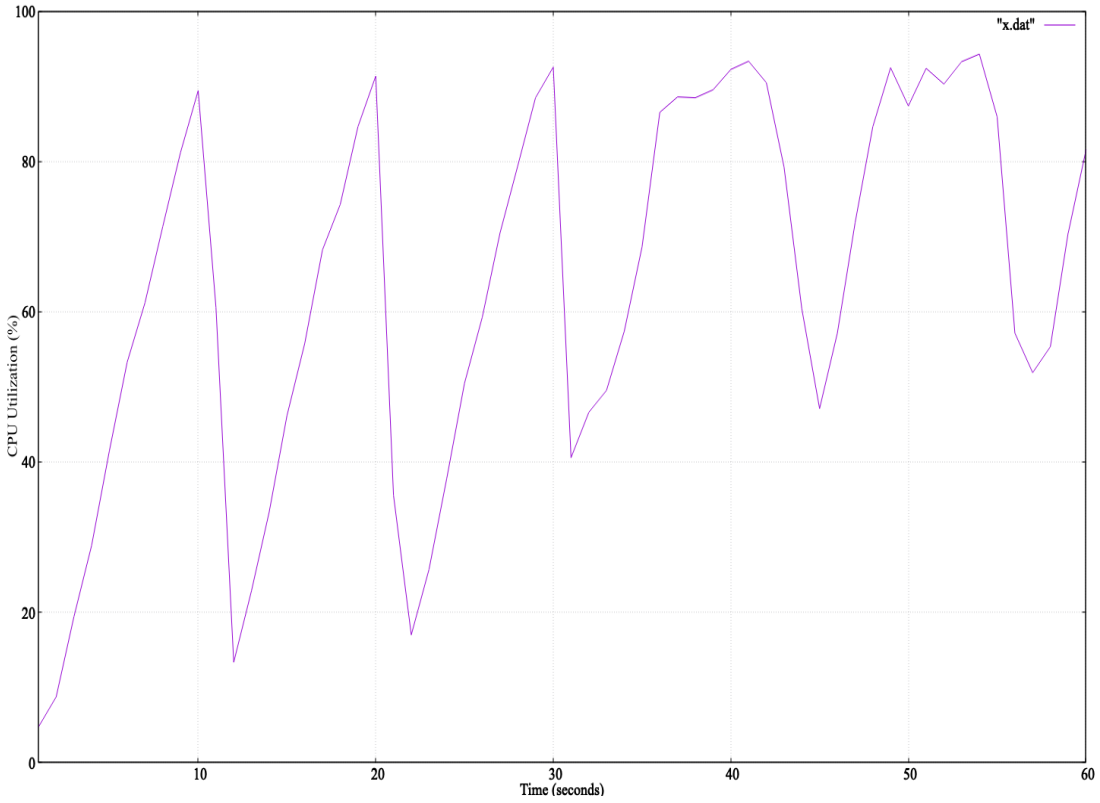


Figure 4.8 CPU Utilization Graph for X-window on macOS

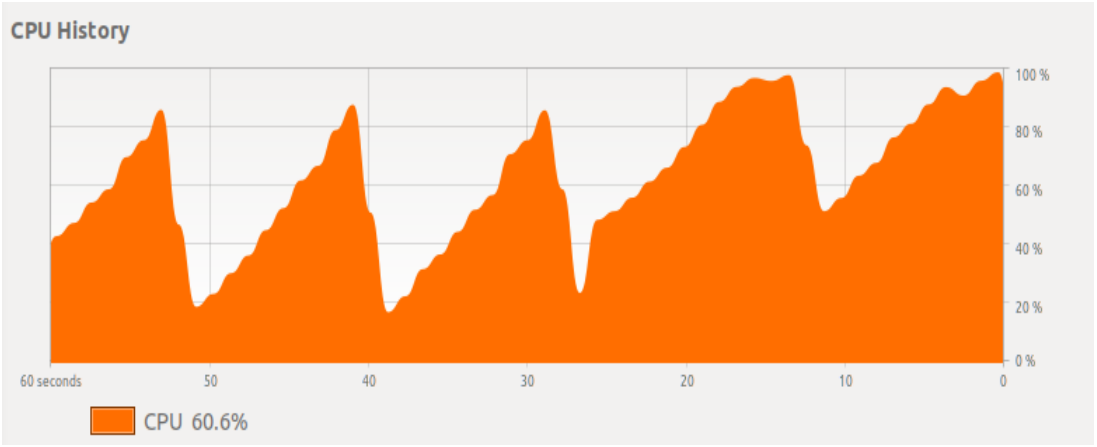


Figure 4.9 CPU Utilization Graph for X-window on Linux

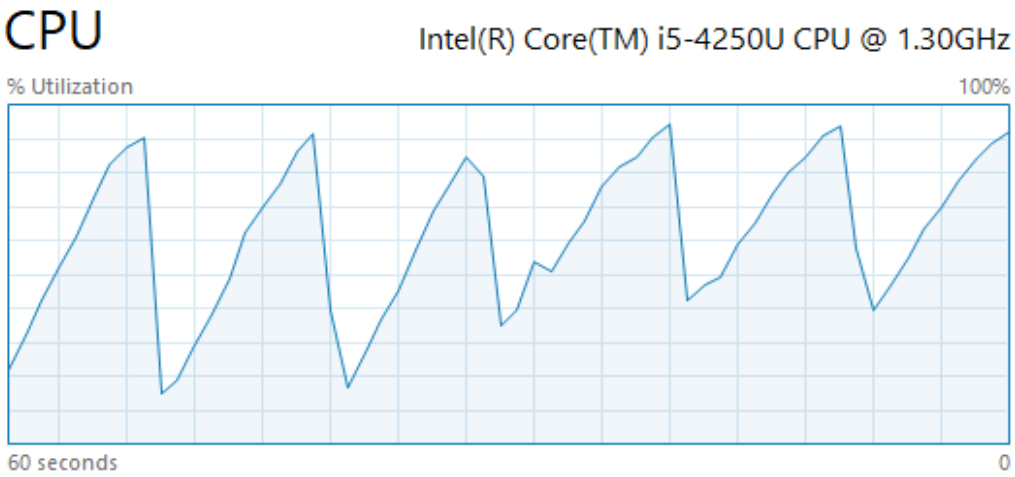


Figure 4.10 CPU Utilization Graph for X-window on Windows

Interactive Task 4: Gaming (Maximum CPU Utilization)

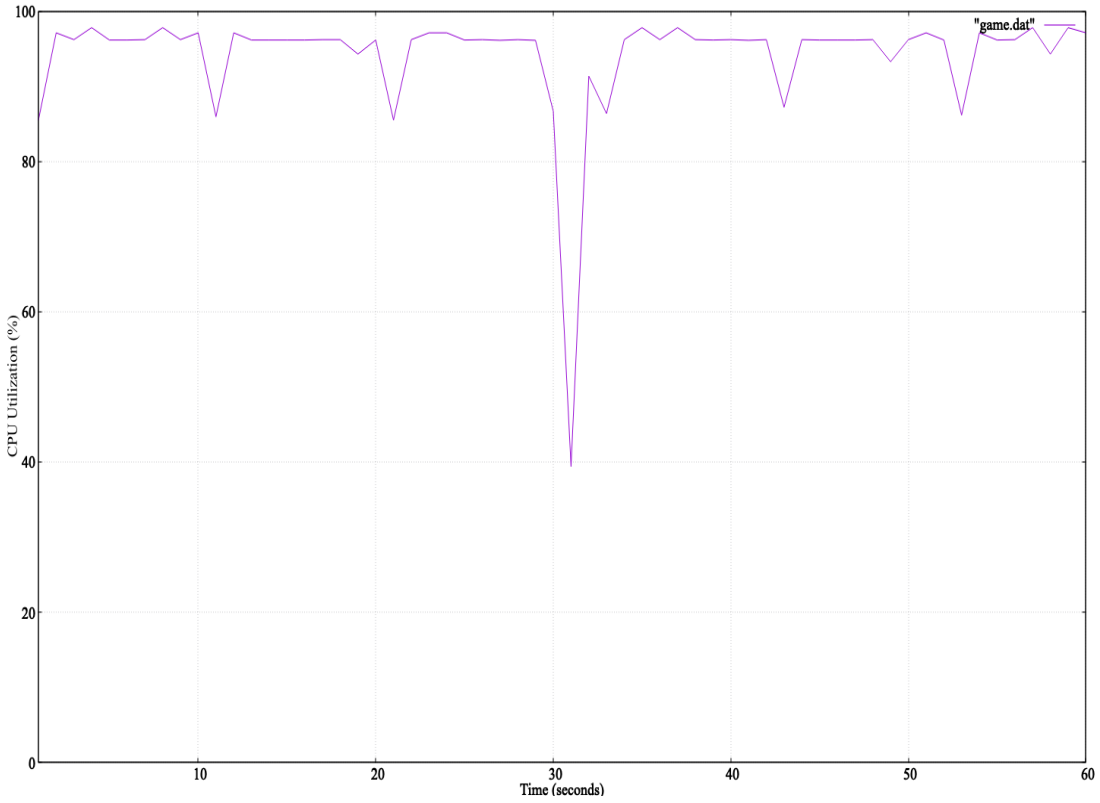


Figure 4.11 CPU Utilization Graph for Gaming on macOS

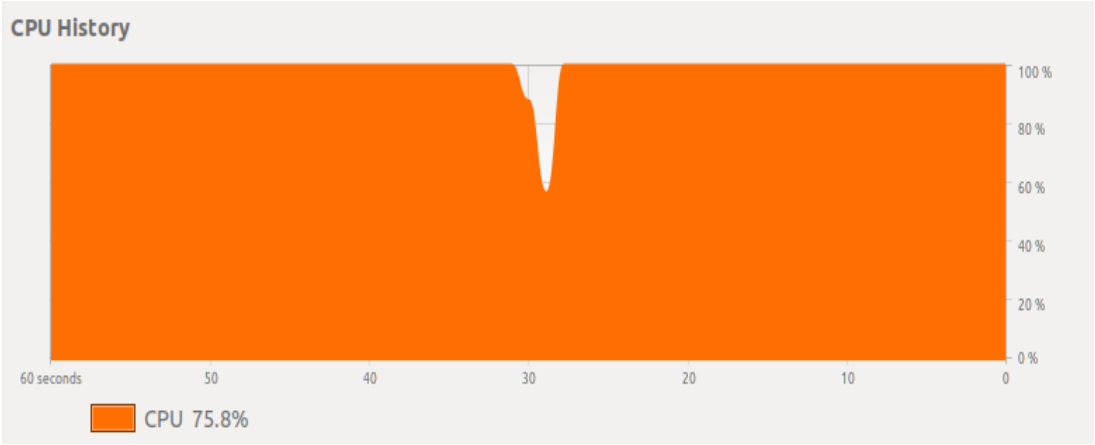


Figure 4.12 CPU Utilization Graph for Gaming on Linux

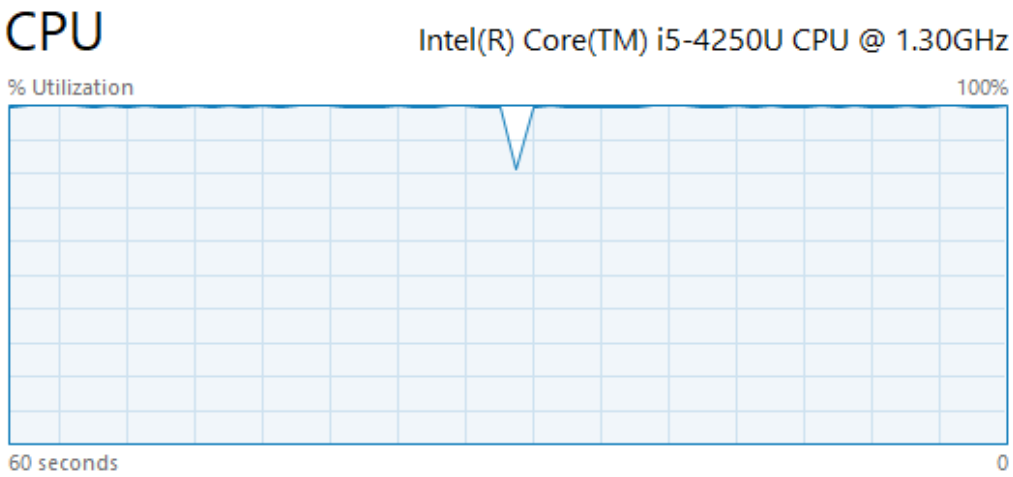


Figure 4.13 CPU Utilization Graph for Gaming on Windows

CHAPTER 5 – INTERACTIVITY PERFORMANCE BENCHMARK

5.1 Interactivity Test

With the help of Interbench, different interactive tasks can be simulated concurrently with various type of background loads. Each interactive task and background load is represented by different amount of CPU utilization. The intended purpose of this benchmark program is to measure the latency that exists within the scheduling process which is represented by the time difference between the time when an interactive task makes a CPU resources request and the time when the specific task actually acquires the requested resources in order to start its execution. The involved interactive tasks cover all workload conditions including low, medium and high workloads.

Two different comparisons for interactivity performance were conducted. The first comparison covers all three operating systems which are Linux, macOS and Windows. For this comparison, Four different interactive tasks are involved which are Audio, Video, X-window and Gaming. These interactive tasks were executed concurrently with various background loads which include None, Video, X-window and Burn.

As for the second comparison, the interactive tasks in the first comparison were used. However, the background loads involved are Write, Read, Ring and Compile. This comparison only involves Linux and macOS because the four background loads specified are not available in Windows Interbench.

5.2 Simulation Environment

In order to compare the schedulers' performance in these three different operating systems using the same set of hardware, Linux, macOS and Windows will be installed natively into the test computer so that fair comparisons can be done with same hardware resources, for instance, the CPU. The original Interbench and the modified Interbench will be executed in terminal of every OS. Before the benchmark starts, the number of active CPU cores is adjusted to only one in order to determine the operating system kernel scheduler's interactivity performance in uniprocessor environment.

Besides that, Intel Turbo Boost Technology which allows the CPU to operate in a higher frequency than usual is disabled to ensure that the interactive tasks and background loads can be simulated with the correct amount of CPU usage for all the involved operating systems.

When Interbench benchmark program in all three different operating systems is being executed for the first time, the number of meaningless loops that can be executed by the system within one millisecond is determined and recorded in a text file for subsequent benchmarks runs. This is done to ensure that the same CPU usage can be emulated in the next benchmark runs to improve consistency and accuracy of the benchmark results.

When a combination pair of interactive task and background load is executed, a thread namely "timekeeping_thread" is responsible to keep track of the time taken for an interactive task to be scheduled. After the execution of the combination pair is completed, the average scheduling latency and maximum scheduling latency are displayed as output.

Each combination pair consisting of one of the interactive tasks and one of the background loads are conducted repeatedly for 30 times. The average and maximum scheduling latencies are recorded and calculated at the end of the benchmark.

Table 5.1, Table 5.2, Table 5.3 and Table 5.4 below show the interactive tasks and background loads carried out for the benchmark involving Linux, macOS and Windows while Table 5.5, Table 5.6, Table 5.7 and Table 5.8 show the interactive tasks and background loads carried out for the benchmark involving Linux and macOS only.

Interactive Task	Background Load
Audio	None
	Video
	X-Window
	Burn

Table 5.1 Simulation of Audio Interactive Task for Linux, macOS and Windows

Interactive Task	Background Load
Video	None
	X-Window
	Burn
	-

Table 5.2 Simulation of Video Task for Linux, macOS and Windows

Interactive Task	Background Load
X-window	None
	Video
	Burn
	-

Table 5.3 Simulation of X-window Task for Linux, macOS and Windows

Interactive Task	Background Load
Gaming	None
	Video
	X-Window
	Burn

Table 5.4 Simulation of Gaming Task for Linux, macOS and Windows

Interactive Task	Background Load
Audio	Write
	Read
	Ring
	Compile

Table 5.5 Simulation of Audio Task for Linux and macOS

Interactive Task	Background Load
Video	Write
	Read
	Ring
	Compile

Table 5.6 Simulation of Video Task for Linux and macOS

Interactive Task	Background Load
X-window	Write
	Read
	Ring
	Compile

Table 5.7 Simulation of X-window Task for Linux and macOS

Interactive Task	Background Load
Gaming	Write
	Read
	Ring
	Compile

Table 5.8 Simulation of Gaming Task for Linux and macOS

5.3 Comparison Results

5.3.1 Interactivity Performance Comparison Results for Linux, macOS and Windows

For the first interactive task in the comparison, Audio which represents a low CPU utilization interactive task and consumes 5% of CPU. Under most background loads, Windows performed the worst among the three operating systems in terms of interactivity performance. This can be proven by bar charts in Figure 5.1 and Figure 5.5 representing the average latency and maximum latency for Audio interactive task for the involved operating systems. In idle condition and Video background load, all operating systems performed similarly by showing near to zero average latencies. However, when Audio interactive task was executed concurrently with X-window and Burn background loads on Windows, the amount of average latencies are higher than Linux and macOS. Besides, Windows also showed significantly higher maximum latencies than Linux and macOS under all background loads.

Next, in condition with medium CPU utilization such as Video interactive task that consumes 40% of CPU during execution, Figure 5.2 shows that the average latencies for Windows under all background loads are higher than Linux and macOS. For maximum latencies, the values produced by macOS and Windows in idle condition are similar. However, when the interactive task is executed under variable workload and high workload represented by X-window and Burn respectively, Windows performed poorly in terms of interactivity performance by producing a higher maximum latencies than the other operating systems.

Followed by X-window interactive task that consumes CPU utilization ranging from 0% to 100% to simulate a condition where a GUI is grabbed and dragged across the screen. The results for this interactive task is different compared to the Audio and Video interactive tasks discussed previously. According to Figure 5.3, the average latency for Windows is slightly higher than Linux and macOS when X-window interactive task is being executed in idle condition. Unlike in the previous interactive tasks, the average latency produced by macOS is higher than Linux and Windows when it is executing X-window under Video background load. As for high workload simulated by Burn background load, Windows's average latency exceeded the average latencies produced by the other two operating systems. Based on Figure 5.6, greatest amount of maximum

latencies are produced by Windows under None and Burn background loads while macOS showed greatest amount of maximum latencies under Video background loads.

Lastly, Gaming interactive task which consumes as much CPU as possible was executed. According to Figure 5.4, when the interactive task is executed in idle condition, macOS's average latency is the highest among the three operating systems but it's only 2.2 milliseconds. For the average latencies under Video and X-window, the interactivity performance of macOS under medium and variable workloads are proven to be the worst between Linux, macOS and Windows. However, the same result doesn't apply to Burn background load. Under Burn, Windows produced significantly greater amount of average latency compared to Linux and macOS. For maximum latencies as shown in Figure 5.8, macOS produced the highest maximum latencies for three background loads including None, Video and X-window. As for Burn, the highest maximum latency was produced by Windows when Gaming interactive task was executed.

In conclusion, interactivity performance for system running Linux is the best under most conditions, followed by macOS producing good interactivity performance especially for interactive tasks consuming low to medium amount of CPU. Windows shows the worst interactivity performance in most of the combination pairs of interactivity tasks and background loads. The poor performance can be seen frequently when the interactive tasks were executed within conditions with high background loads.

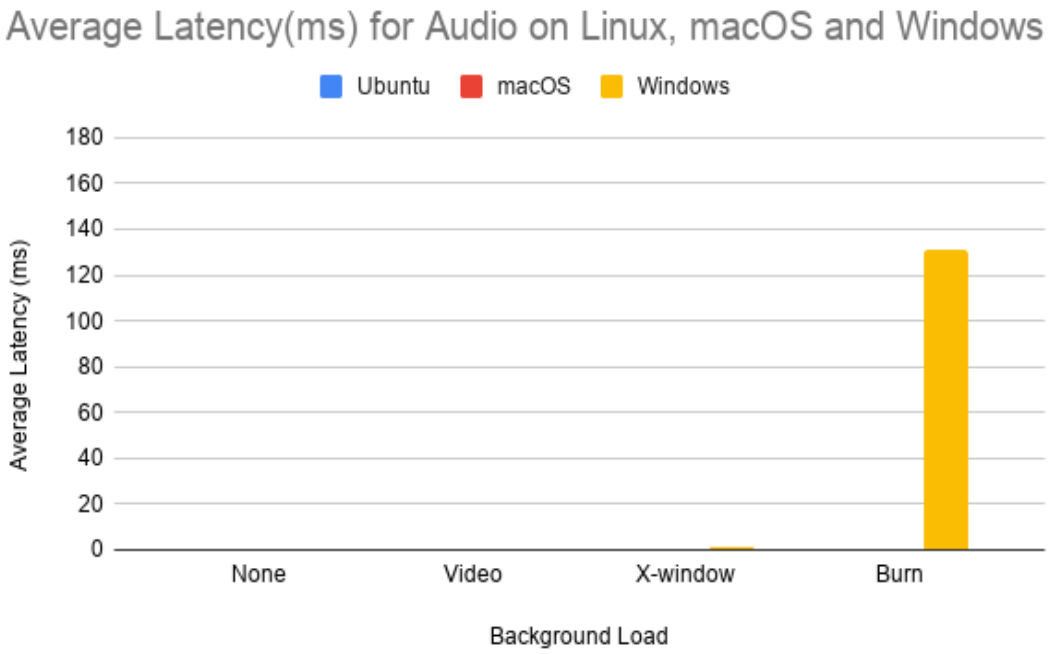


Figure 5.1 Average Latency for Audio on Linux, macOS and Windows

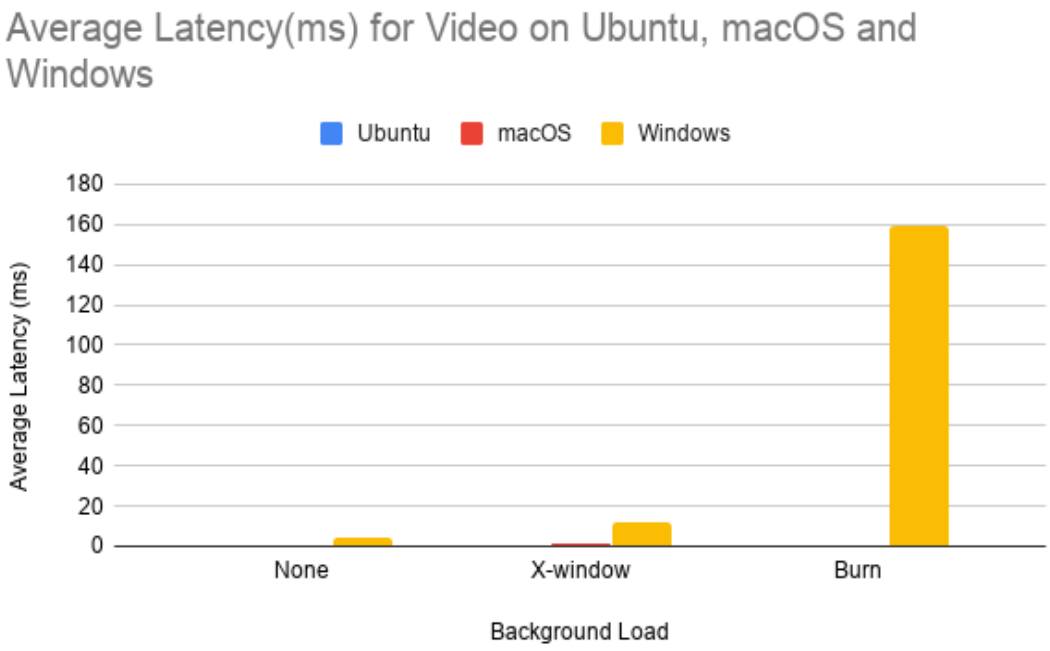


Figure 5.2 Average Latency for Video on Linux, macOS and Windows

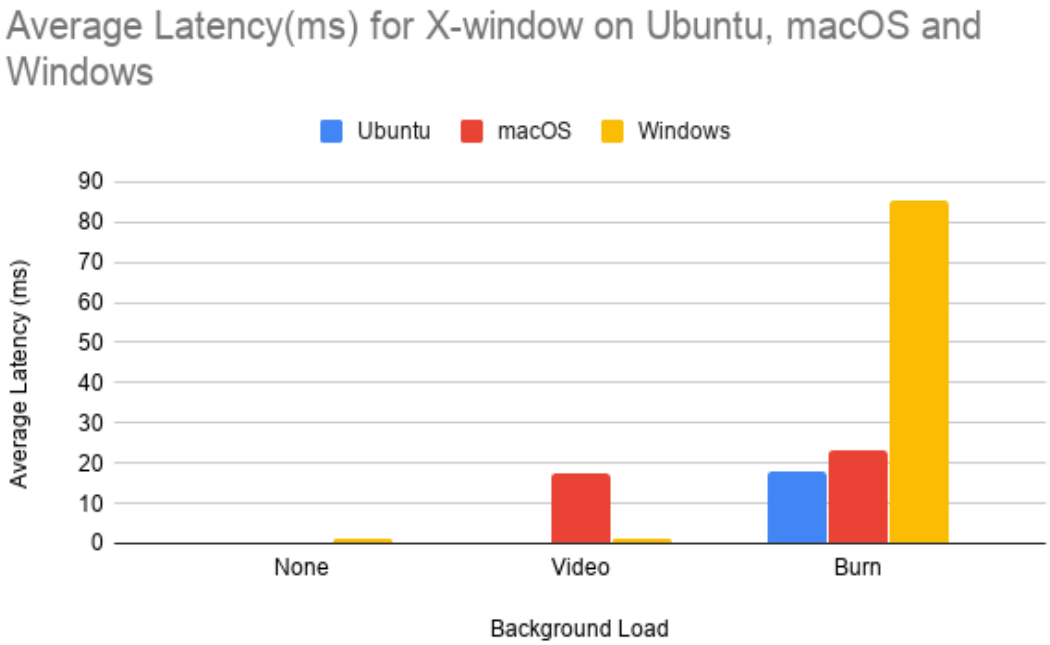


Figure 5.3 Average Latency for X-window on Linux, macOS and Windows

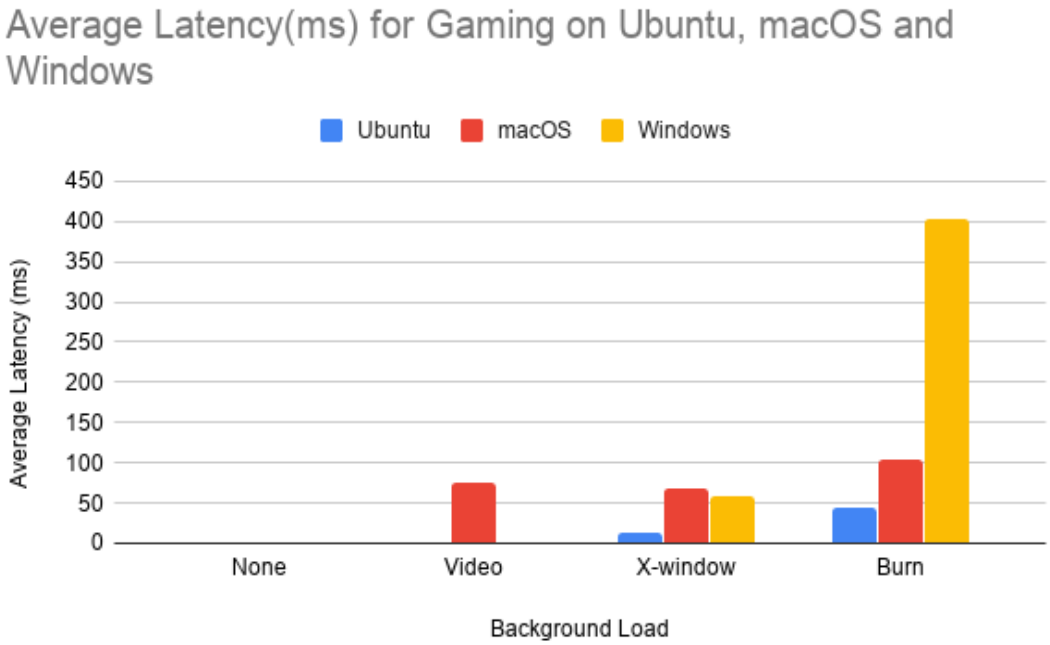


Figure 5.4 Average Latency for Gaming on Linux, macOS and Windows

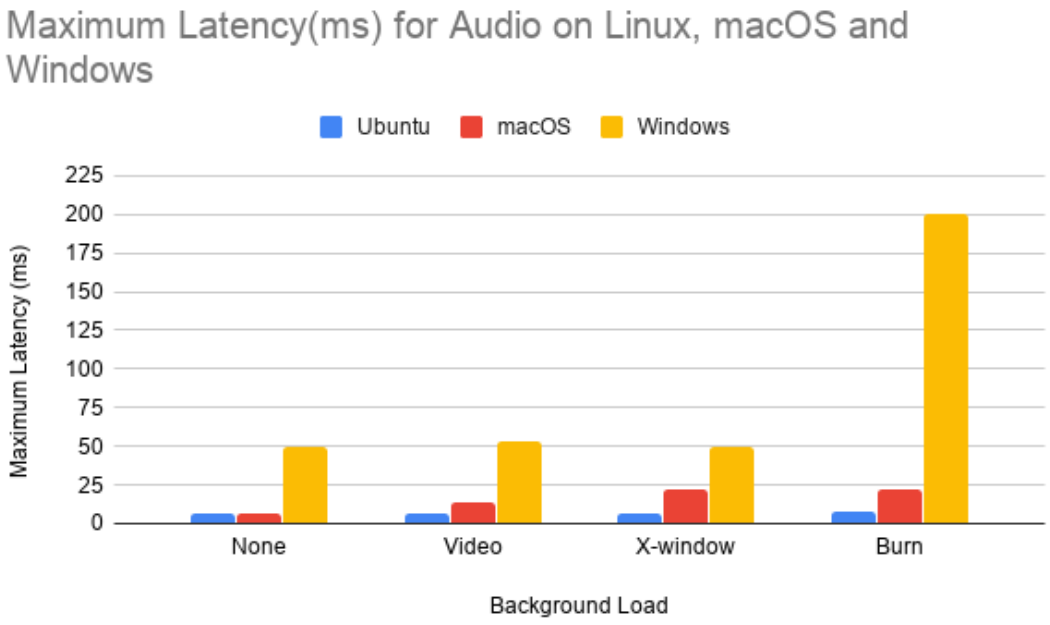


Figure 5.5 Maximum Latency for Audio on Linux, macOS and Windows

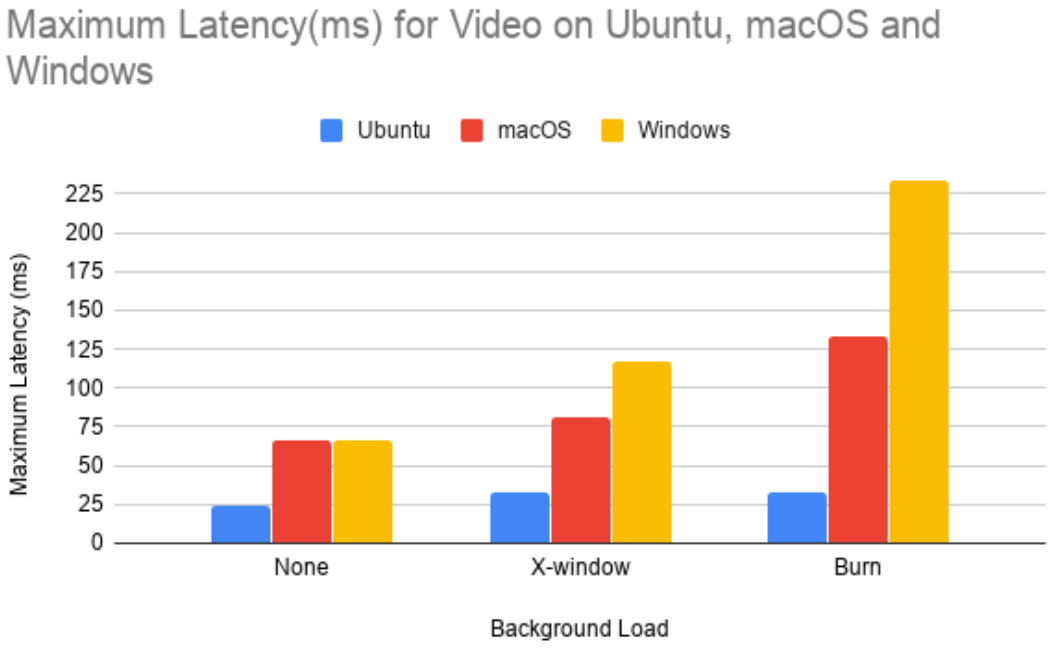


Figure 5.6 Maximum Latency for Audio on Linux, macOS and Windows

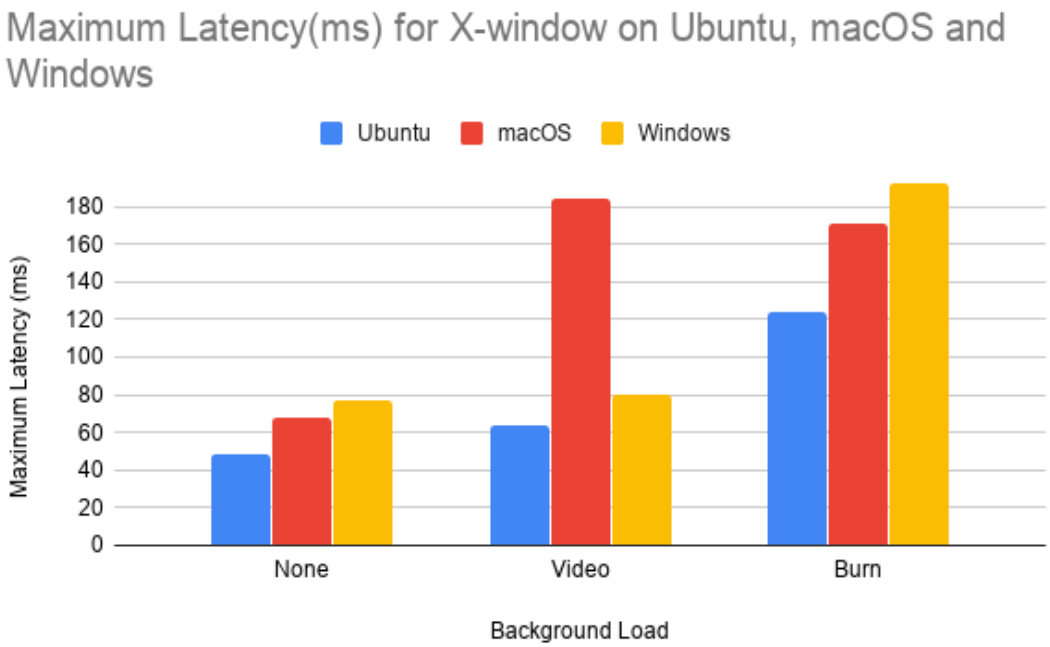


Figure 5.7 Maximum Latency for X-window on Linux, macOS and Windows

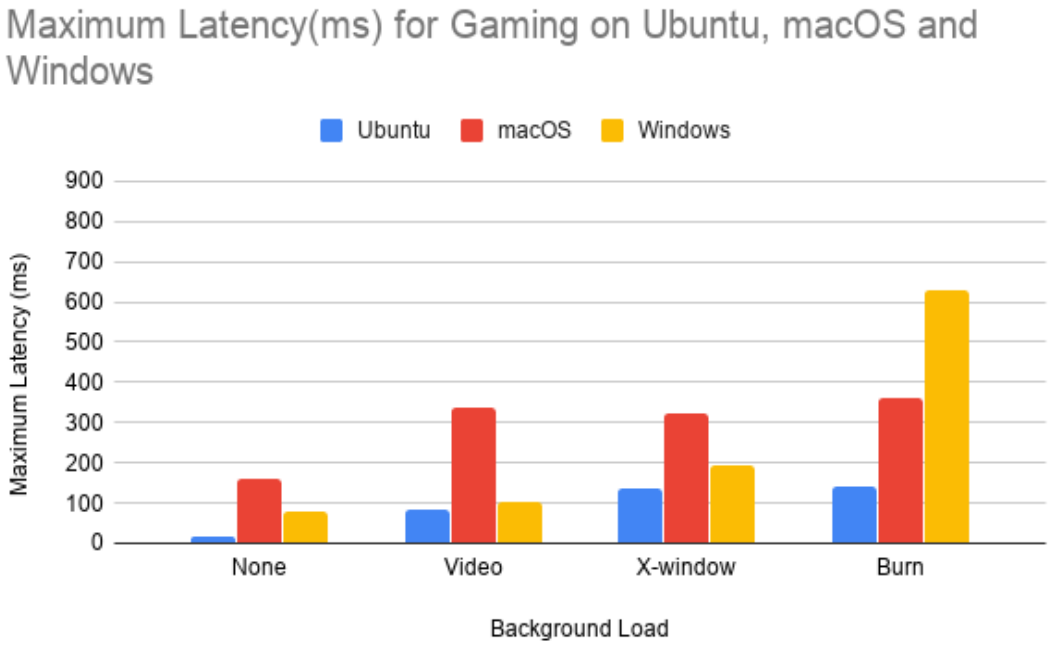


Figure 5.8 Maximum Latency for Gaming on Linux, macOS and Windows

5.3.2 Interactivity Performance Comparison Results for Linux and macOS

This section includes comparison results for the four different interactive tasks executed concurrently with Write, Read, Ring and Compile background loads on Linux and macOS only as the four background loads specified are not included in Windows Interbench.

For Audio, the average latencies for both Linux and macOS are considered to be low. The average latencies for Disk I/O background loads are slightly higher on macOS with differences of 0.4 milliseconds for Read and 0.2 milliseconds Write. For Ring background load which executes tasks in a circular manner to allow them to take turns for execution, the average latency on macOS is also slightly higher compared to Linux. When the interactive task was executed under Compile background load which is the emulation of "make –j4" compilation. "make –j4" refers to the parallel execution of 4 jobs. This background load is considered to be a heavy load because the emulation involves running of three different background loads simultaneously including Burn, Write and Read. The average latency for both operating systems are identical. As for the maximum latencies, macOS produced a greater amount of maximum latencies under all background loads for Audio interactive task.

Next, average latencies and maximum latencies for Video interactive task are discussed. Running the task concurrently with Read background load on Linux and macOS shows minimal latency difference between the two operating systems with 0.2 milliseconds of average latency on Linux and 0.3 milliseconds of average latency on macOS. Under Write and Ring background loads, both operating systems performed well in terms of interactivity by showing near to zero average latencies. The same result is no longer applicable to Compile background load, the average latency on macOS is twice the average latency on Linux. Based on Figure 5.14, a slightly higher maximum latencies were produced by Linux when Video interactive task was executed under Write and Read background loads. However, the maximum latency on macOS under Ring background load is higher than Linux. A more significant difference in terms of maximum latency can be seen when the heavy Compile background load was executed on macOS.

Followed by the third interactive task for this comparison, X-window. The average latencies for all background load conditions on macOS are higher than Linux. Higher

maximum latencies can be also seen on macOS under majority of the background loads for X-window interactive task, except for Read background load where maximum latency on Linux is higher than macOS.

Lastly, the average latencies for CPU intensive interactive task such as Gaming on macOS are up to double the average latencies on Linux. The difference became more obvious when the interactive task was executed with the existence of high background loads such as Compile. Besides, the high average latency on macOS under Compile background load had exceeded the reaction time of human to visual stimulus which is about 250 milliseconds or equivalent to 0.25 seconds (Backyard Brains, n.d.). In other words, lagging or jitters will become noticeable to humans, leading to an unpleasant experience to them. As for maximum latencies, the values produced by macOS were higher when Gaming interactive task was executed concurrently under all background loads. All the maximum latencies on macOS were beyond 250 millisecond while the maximum latencies on Linux were below 250 milliseconds except for Compile background load.

At the end of this comparison, we found out that Linux has a better interactivity performance in multiple interactive tasks with CPU utilizations ranging from low to high under Write, Read, Ring and Compile background loads.

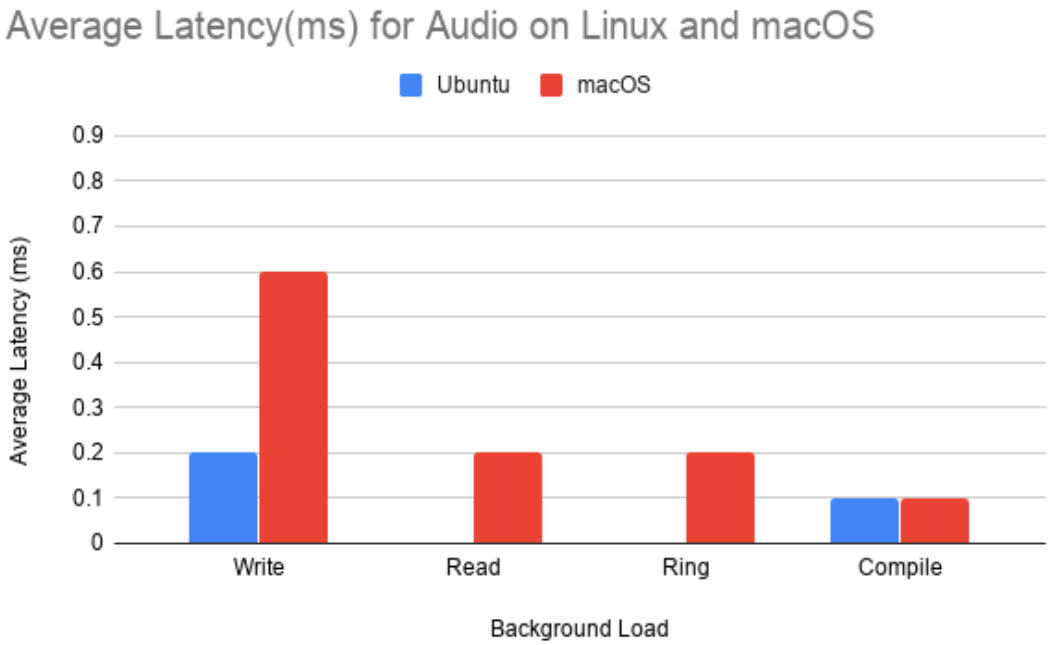


Figure 5.9 Average Latency for Audio on Linux and macOS

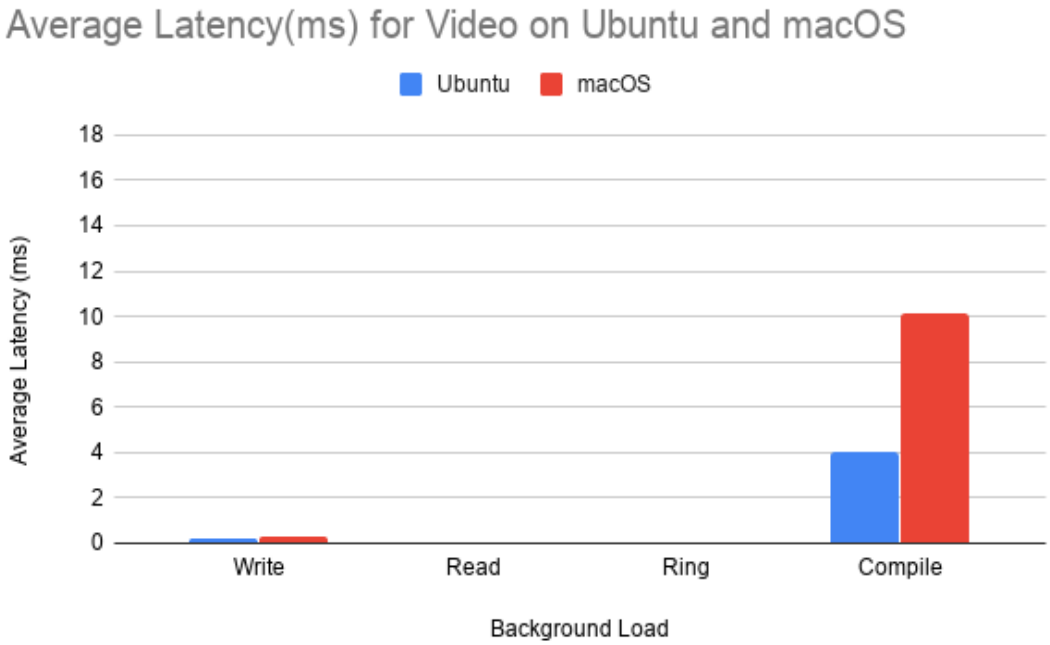


Figure 5.10 Average Latency for Video on Linux and macOS

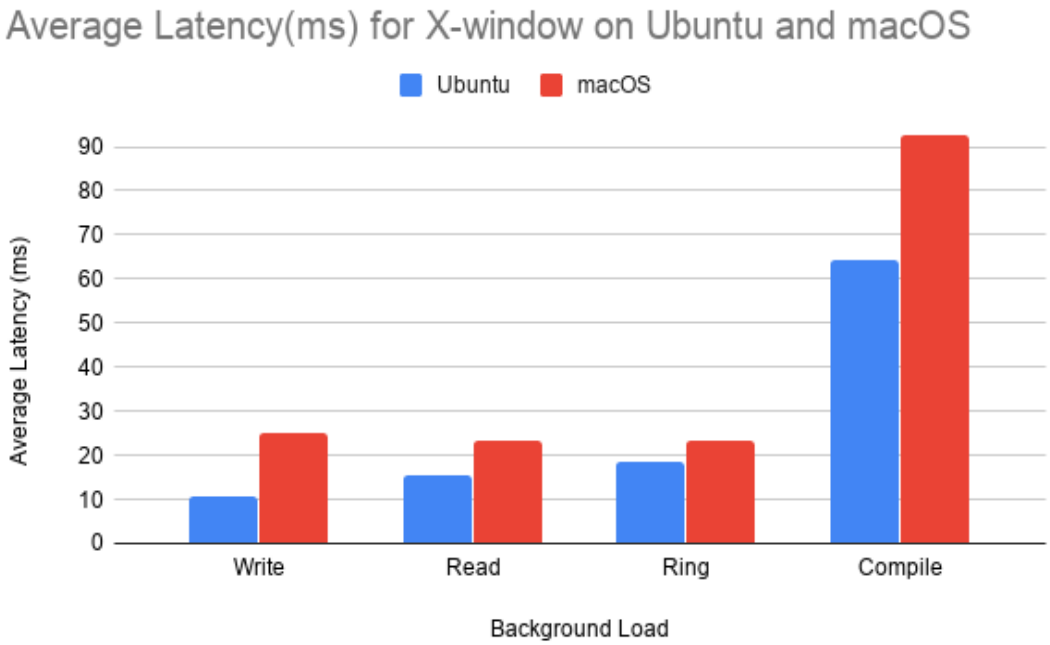


Figure 5.11 Average Latency for X-window on Linux and macOS

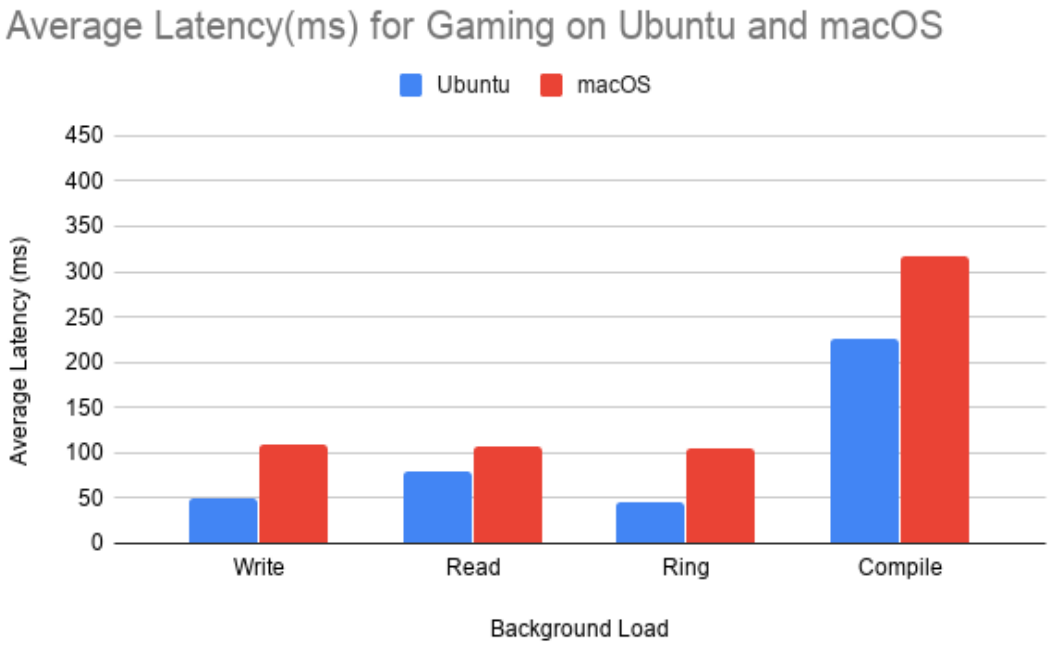


Figure 5.12 Average Latency for Gaming on Linux and macOS

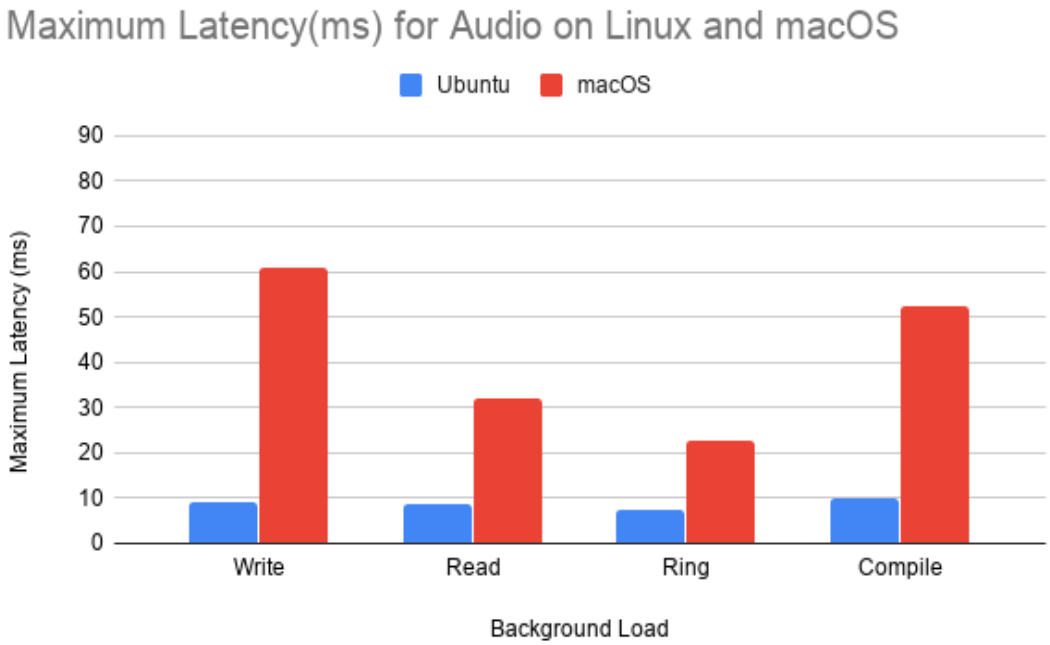


Figure 5.13 Maximum Latency for Audio on Linux and macOS

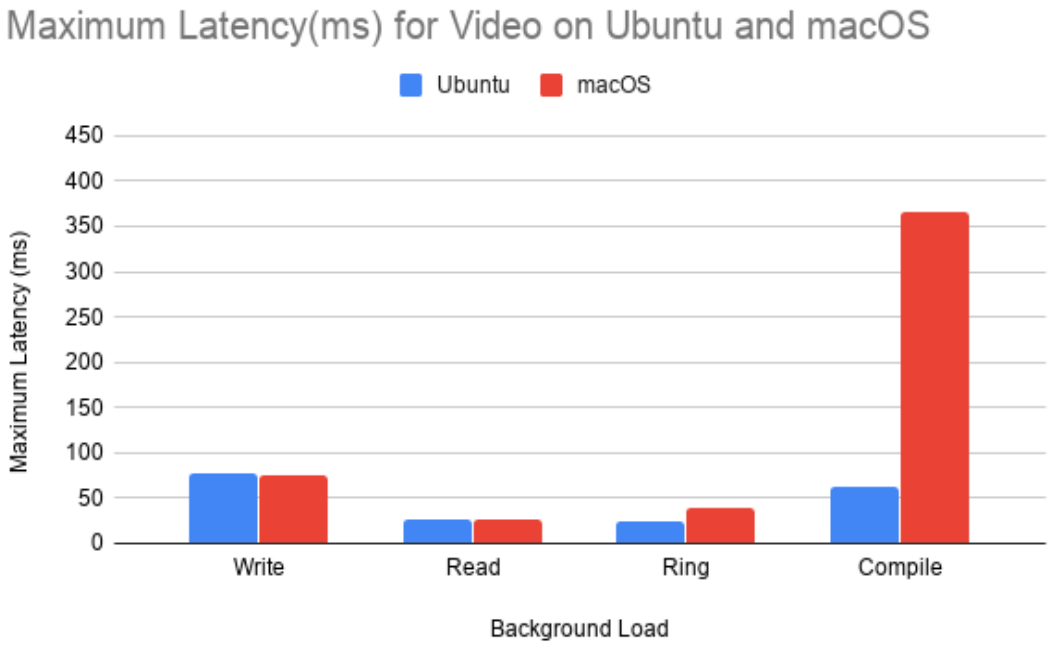


Figure 5.14 Maximum Latency for Video on Linux and macOS

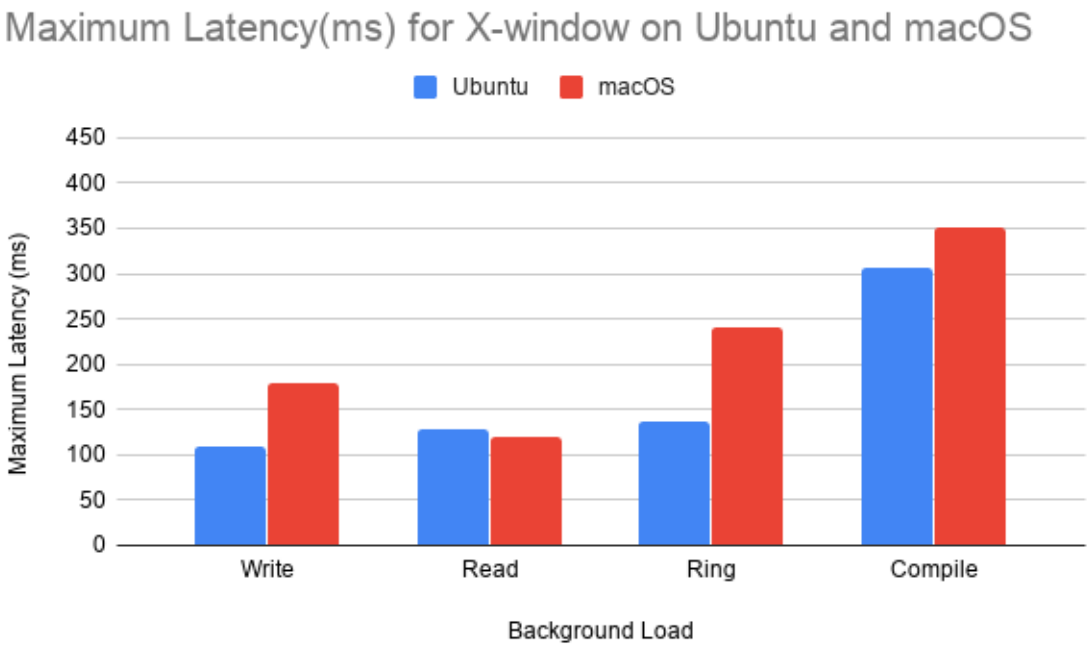


Figure 5.15 Maximum Latency for X-window on Linux and macOS

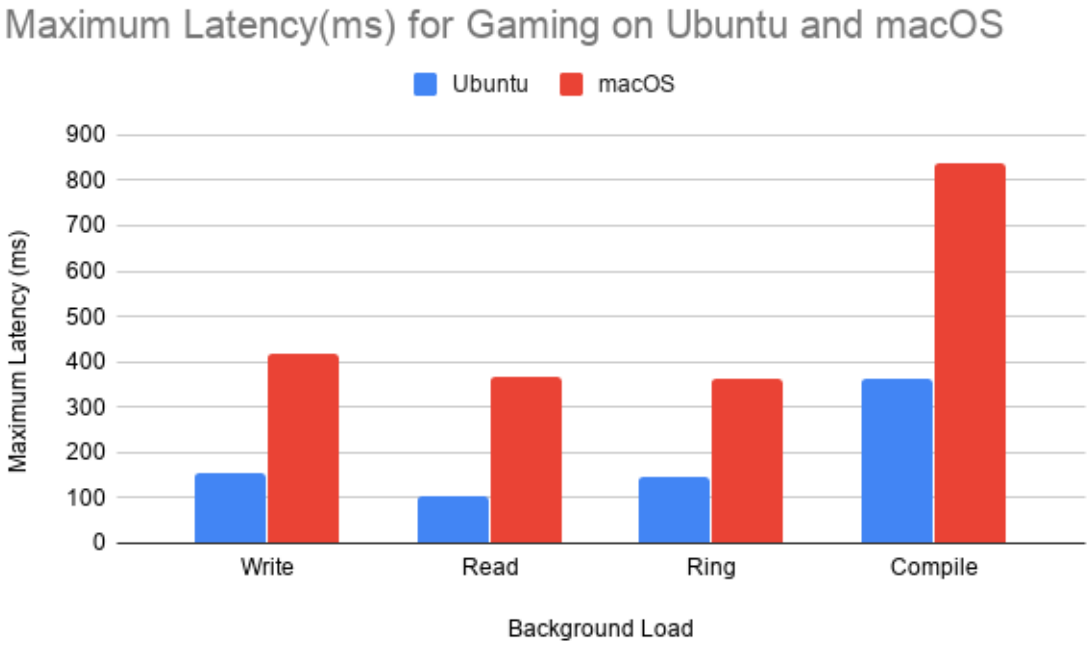


Figure 5.16 Maximum Latency for Gaming on Linux and macOS

CHAPTER 6 - CONCLUSION

In conclusion, Interbench which is originally used to determine the interactivity performance of Linux OS kernel schedulers is ported to macOS using C programming language in order to solve the lack of support for interactivity performance benchmark program in different operating systems. Next, the ported benchmark program has the potential to assist OS developers to perform fair and accurate comparisons between different OS kernel schedulers. Besides OS developers, normal users will be able to make a better decision in choosing a system that best fits their requirements and needs based on the benchmark results.

In order to port Interbench into macOS successfully, some modifications have been made in the headers required by the program. For instance, “semaphore.h” in the original Interbench is replaced by Grand Central Dispatch’s “dispatch/semaphore.h” as POSIX semaphore is not applicable in macOS. Besides, semaphore implementations were also changed accordingly to ensure that operations related to semaphores can be done similar to the semaphore operations using POSIX semaphores. As some Linux-specific system files is not available in macOS, shell scripting is applied in the ported benchmark program to obtain some system information such as total RAM to simulate Read and Write background loads.

Besides, reverification of Windows Interbench was performed in the project’s verification plan together with the original Linux Interbench and ported macOS Interbench. Windows Interbench was proven to perform as expected in different hardware configurations and Windows OS versions. The verification plan also showed that both ported versions of Interbench benchmark program were able to behave similarly to the original Linux Interbench, allowing reliable comparisons of interactivity performance for all operating systems involved.

At the end of this project, two sets of comparison were produced by executing the original and ported benchmark program to determine the interactivity performance of the involved operating systems including Linux, macOS and Windows.

The first comparison involved all three operating systems. Audio, Video, X-window and Gaming interactive tasks were executed concurrently with background loads such as None, Video, X-window and Burn. As for the results, Linux performed the best by producing the lowest average and maximum latencies in most of the task

and background load conditions. macOS performed well in terms of interactivity in tasks utilizing low and medium amount of CPU resources. The interactivity performance in Windows is the worst compared to the other two operating systems by producing the highest average and maximum latencies in majority of the background loads for each interactive task.

The second comparison is comprised of results for Write, Read, Ring and Compile background loads and they were executed with the same set of interactive task on Linux and macOS. The final results showed that Linux produced lower average and maximum latencies in majority of the background loads in each interactive task.

In the future, the remaining Memload background load will be made available to the Interbench on macOS to determine the average latencies and maximum latencies when the RAM is fully occupied and requires swapping to and from the virtual memory. Besides, Write, Read, Ring, Compile and Memload background loads will be implemented in Windows Interbench to broaden the coverage of the interactivity performance comparison. Lastly, benchmarks for multiprocessor environment can be done to gain insights on the operating systems' interactivity performance compared to the results obtained from a uniprocessor environment.

BIBLIOGRAPHY

Abaffy, J & Krajcovic, T 2009, 'Latencies in Linux and FreeBSD kernels with different schedulers – O(1), CFS, 4BSD, ULE', pp 1-6.

Apple Developer 2013, *Synchronization Primitives*, Available from: <https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/synchronization/synchronization.html>. [3 November 2019].

Apple Developer 2019, *DispatchSemaphore*, Available from: <https://developer.apple.com/documentation/dispatch/dispatchsemaphore>. [10 November 2019].

Backyard Brains n.d., *Experiment: How Fast Your Brain Reacts To Stimuli.* Available from < https://backyardbrains.com/experiments/reactiontime>. [22 April 2020].

Cheng, SW 2015, 'Interactivity Performance Benchmark on Windows OS', pp.2-38.

ElysiumAcademy Private Limited 2017, *What are the Software Development Life Cycle (SDLC) phases?*. Available from: <https://www.linkedin.com/pulse/what-software-development-life-cycle-sdlc-phases-private-limited>. [10 August 2019].

GeeksforGeeks n.d., *Software Engineering | Incremental process model*. Available from: <https://www.geeksforgeeks.org/software-engineering-incremental-process-model/>. [10 August 2019].

Gnuplot, computer software 2020. Available from: <http://www.gnuplot.info>. [15 April 2020].

Intel Corporation n.d., *Intel Turbo Boost Technology 2.0.* Available from: <https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html>. [31 March 2020].

Kamal, R 2011, *Microcontrollers: Architecture Programming, Interfacing and System Design, 2nd Edition.* Pearson Education, India. [22 April 2020].

Kolivas, C 2006, The homepage of Interbench The Linux interactivity benchmark. Available from: < http://www.users.on.net/~ckolivas/interbench/>. [20 July 2019].

Linfo 2005, *Kernel Definition*. Available from <http://www.linfo.org/kernel.html>. [20 July 2019].

Luo, Y & Wu, Y 2011, 'A Comparison on Interactivity of Three Schedulers in Embedded System', pp. 494 – 497.

Monitor Types, n.d. Available from : <https://pages.mtu.edu/~shene/NSF-3/e-Book/MONITOR/monitor-types.html> [20 November 2019].

Patel, K n.d, *pipe() System call*. Available from: <https://www.geeksforgeeks.org/pipe-system-call/>. [3 November 2019].

Phoronix Test Suite 2020, *Open-Source, Automated Benchmarking*. Available from < https://www.phoronix-test-suite.com>. [20 April 2020].

Silberschatz, A, Galvin, PB & Gagne, G 2009. *Operating System Concepts.* John Wiley & Sons Inc, New York. [20 April 2020].

Stallings, W n.d., *Operating Systems, Internals and Design Principles.* Pearson Prentice Hall, New Jersey. [20 April 2020].

Stupak, N 2009, *Time delays and system response time in human-computer interaction.* Rochester Institute of Technology. [22 April 2020].

Techopedia n.d., *Benchmarking*. Available from: < https://www.techopedia.com/definition/17053/benchmarking>. [20 July 2019].

Turbo Boost Switcher, computer software 2019. Available from <http://tbswitcher.rugarciap.com>. [20 March 2019].

Wang, S, Chen, Y, Jiang, W, Li, P, Dai, T & Cui, Y 2009, 'Fairness and Interactivity of Three CPU Schedulers in Linux', Proceedings of the fifteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 172-177.

Wong, CS, Tan, IKT, Kumari, RD & Lam, JW 2008. 'Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers', pp 1-8.

APPENDIX A – RESULTS DATA FOR INTERBENCH BENCHMARK

A.1 Audio Interactive Task Results Data

None						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.1	3.0	0.0	1.7	0.2	50.0
2	0.1	1.9	0.0	2.7	0.0	0.0
3	0.1	2.9	0.0	0.6	0.0	0.0
4	0.1	3.1	0.0	1.1	0.0	0.0
5	0.1	3.0	0.0	0.2	0.0	0.0
6	0.1	2.9	0.0	0.2	0.1	50.0
7	0.0	0.9	0.0	0.6	0.0	0.0
8	0.0	0.7	0.0	0.7	0.0	0.0
9	0.1	3.0	0.0	0.2	0.0	0.0
10	0.1	2.8	0.0	0.1	0.0	0.0
11	0.1	0.8	0.0	0.6	0.0	0.2
12	0.0	0.4	0.0	0.7	0.0	0.0
13	0.1	1.0	0.0	0.2	0.0	0.0
14	0.1	3.0	0.0	2.8	0.0	0.0
15	0.1	3.1	0.0	6.3	0.0	0.0
16	0.1	2.3	0.0	0.2	0.0	0.3
17	0.1	2.9	0.0	0.5	0.0	0.0
18	0.1	2.3	0.0	0.4	0.1	50.0
19	0.1	2.6	0.0	0.3	0.0	0.0
20	0.0	0.4	0.0	0.2	0.0	0.2
21	0.1	2.3	0.0	0.1	0.0	0.0
22	0.1	7.0	0.0	0.2	0.0	0.0
23	0.0	0.4	0.0	0.2	0.0	0.0
24	0.1	2.3	0.1	3.1	0.0	0.0
25	0.1	3.0	0.0	0.2	0.0	0.9
26	0.1	3.1	0.0	1.3	0.0	0.0
27	0.1	0.9	0.0	0.2	0.0	0.0
28	0.1	2.9	0.1	4.0	0.1	50.0
29	0.1	3.2	0.0	0.4	0.0	0.0
30	0.1	6.0	0.1	3.9	0.0	0.0
	Average : 0.1	Max : 7.0	Average : 0.0	Max : 6.3	Average : 0.0	Max : 50.0

Video						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	2.1	0.0	2.7	0.0	0.1
2	0.0	2.4	0.1	7.0	0.2	53.1
3	0.0	2.3	0.0	0.5	0.0	0.0
4	0.0	2.6	0.1	7.8	0.0	0.1
5	0.0	2.8	0.6	13.1	0.0	0.0
6	0.0	5.0	0.0	0.1	0.0	0.1
7	0.0	2.8	0.0	2.7	0.0	0.0
8	0.0	2.1	0.0	3.8	0.1	50.0
9	0.0	2.5	0.0	0.6	0.0	0.0
10	0.0	2.8	0.2	7.9	0.0	0.0
11	0.0	2.8	0.0	1.3	0.0	0.0
12	0.0	2.5	0.0	0.1	0.0	0.0
13	0.0	2.8	0.0	0.3	0.0	0.0
14	0.0	0.4	0.1	7.2	0.0	0.0
15	0.0	2.3	0.0	0.1	0.0	0.0
16	0.0	2.4	0.1	2.7	0.0	0.0
17	0.0	6.7	0.1	7.2	0.0	0.3
18	0.0	6.9	0.1	8.1	0.0	0.0
19	0.0	1.8	0.0	0.1	0.1	50.0
20	0.0	2.8	0.0	0.2	0.0	0.1
21	0.0	2.1	0.0	0.1	0.0	0.0
22	0.0	2.4	0.0	0.6	0.0	0.1
23	0.0	2.6	0.0	6.8	0.0	0.0
24	0.0	2.6	0.2	8.2	0.0	0.1
25	0.0	2.8	0.0	2.7	0.0	0.1
26	0.0	1.8	0.0	0.2	0.0	0.1
27	0.0	2.8	0.0	0.5	0.0	0.1
28	0.0	5.8	0.0	0.1	0.0	0.0
29	0.0	1.8	0.0	0.9	0.0	0.0
30	0.0	2.1	0.1	8.1	0.1	50.0
	Average : 0.0	Max : 6.9	Average : 0.1	Max : 13.1	Average : 0.0	Max : 53.1

X						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.1	3.2	0.6	11.2	1.2	50.0
2	0.1	3.6	0.1	10.0	2.8	50.1
3	0.1	3.3	0.2	11.3	1.9	50.1
4	0.1	3.1	0.5	11.2	1.4	50.0
5	0.1	6.5	0.9	21.6	1.3	50.0
6	0.1	3.0	0.4	11.2	2.0	50.0
7	0.1	2.6	0.1	10.0	1.2	50.0
8	0.1	4.6	0.4	11.3	1.1	50.0
9	0.1	4.4	0.8	20.0	1.5	50.0
10	0.1	2.6	0.4	11.2	1.5	50.0
11	0.1	3.0	0.2	11.4	1.3	50.0
12	0.1	4.3	0.1	10.0	1.6	50.0
13	0.1	3.2	0.4	10.1	1.1	50.0
14	0.1	7.0	0.4	11.3	2.1	50.0
15	0.1	3.2	0.3	11.2	1.6	50.0
16	0.1	5.2	0.3	22.6	1.4	50.0
17	0.1	2.7	0.3	11.4	1.0	50.0
18	0.1	2.6	0.3	11.5	1.3	50.0
19	0.1	3.3	0.1	10.0	1.5	50.0
20	0.1	2.6	0.5	11.3	0.9	50.0
21	0.1	3.8	0.2	10.0	1.7	50.0
22	0.1	2.6	0.5	20.0	1.0	50.1
23	0.1	3.4	0.2	20.8	1.7	50.1
24	0.1	2.9	0.4	20.0	1.8	50.0
25	0.1	3.2	0.7	20.0	1.5	50.0
26	0.1	3.1	0.1	10.0	1.9	50.0
27	0.1	4.2	0.3	10.0	1.4	50.0
28	0.1	3.5	0.6	21.2	1.0	50.0
29	0.1	4.5	0.6	20.0	1.4	50.0
30	0.1	2.6	0.3	10.0	1.9	50.0
	Average : 0.1	Max : 7.0	Average : 0.4	Max : 22.6	Average : 1.5	Max : 50.1

Burn						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	6.2	0.2	20.1	135.0	200.0
2	0.0	1.3	0.0	10.0	130.0	150.0
3	0.0	7.4	0.8	20.0	130.0	150.0
4	0.0	0.2	0.1	21.2	131.0	150.0
5	0.0	0.2	0.2	20.0	132.4	150.1
6	0.0	1.8	0.0	0.1	132.4	150.2
7	0.0	4.9	0.2	21.2	128.9	150.2
8	0.0	0.5	0.1	20.0	131.0	150.2
9	0.0	0.5	0.1	21.2	129.2	150.1
10	0.0	0.2	0.0	4.0	129.2	150.2
11	0.0	0.1	0.2	20.0	132.0	150.2
12	0.0	0.1	0.0	0.1	131.0	200.2
13	0.0	1.8	0.1	20.0	130.0	150.2
14	0.0	0.1	0.0	0.4	131.3	200.0
15	0.0	5.7	0.1	21.2	128.9	150.2
16	0.0	0.5	0.0	0.2	132.4	150.2
17	0.0	0.1	0.2	21.2	133.3	150.0
18	0.0	0.1	0.0	2.6	132.4	150.0
19	0.0	6.5	0.2	21.2	129.5	150.2
20	0.0	1.1	0.3	22.5	129.5	150.0
21	0.0	2.8	0.2	20.0	131.8	150.2
22	0.0	0.1	0.0	0.6	131.0	150.2
23	0.0	0.1	0.0	0.7	129.5	150.2
24	0.0	0.2	0.1	20.0	131.3	150.1
25	0.0	0.1	0.1	20.1	130.7	200.0
26	0.0	2.7	0.0	10.1	127.8	150.0
27	0.0	1.4	0.2	20.2	131.8	150.2
28	0.0	0.1	0.1	20.0	132.4	150.0
29	0.0	6.6	0.0	0.1	130.3	150.2
30	0.0	4.2	0.5	21.2	133.6	200.0
	Average : 0.0	Max :7.4	Average : 0.1	Max : 22.5	Average :131.0	Max : 200.2

Write				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.1	6.8	0.5	50.2
2	0.2	6.5	0.5	50.3
3	0.2	4.5	0.6	50.2
4	0.1	5.5	1.0	60.4
5	0.1	4.2	0.4	50.2
6	0.1	6.8	0.5	22.1
7	0.1	6.6	0.9	50.3
8	0.2	7.7	1.0	50.3
9	0.2	6.3	0.7	60.8
10	0.2	7.5	0.7	50.2
11	0.1	6.8	0.6	50.3
12	0.1	8.0	0.6	50.2
13	0.1	6.1	0.5	21.3
14	0.1	6.7	0.8	60.7
15	0.2	6.7	0.7	21.7
16	0.2	6.8	0.7	53.2
17	0.2	7.7	1.2	59.4
18	0.1	5.3	0.6	21.3
19	0.1	5.0	0.6	22.1
20	0.2	8.2	0.5	50.2
21	0.2	7.8	0.8	60.8
22	0.2	6.4	0.6	60.7
23	0.2	8.7	0.1	20.9
24	0.2	7.6	0.5	50.2
25	0.1	8.0	0.8	50.3
26	0.1	6.8	0.3	17.3
27	0.1	8.9	0.8	50.4
28	0.2	7.5	0.8	50.2
29	0.2	6.9	0.8	50.2
30	0.1	5.4	0.1	10.8
	Average : 0.2	Max : 8.9	Average : 0.6	Max : 60.8

Read				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.0	7.4	0.0	0.3
2	0.0	0.7	0.1	20.1
3	0.0	4.1	0.1	20.1
4	0.0	3.3	0.1	10.7
5	0.0	0.5	0.1	20.1
6	0.0	6.5	0.4	30.1
7	0.0	6.8	0.1	20.1
8	0.0	0.3	0.2	22.0
9	0.0	4.1	0.3	21.7
10	0.0	1.2	0.4	20.1
11	0.0	0.2	0.0	11.7
12	0.0	2.0	0.1	20.1
13	0.0	0.2	0.5	31.7
14	0.0	0.2	0.0	0.3
15	0.0	4.6	0.0	0.1
16	0.0	3.5	0.1	20.1
17	0.1	8.5	0.1	21.8
18	0.0	1.5	0.5	22.9
19	0.0	2.1	0.2	20.1
20	0.0	0.2	0.1	20.1
21	0.0	4.2	0.1	20.4
22	0.0	2.1	0.5	31.8
23	0.0	0.2	0.2	21.6
24	0.0	0.9	0.4	21.7
25	0.0	0.2	0.0	0.1
26	0.0	6.8	0.0	10.0
27	0.0	1.6	0.1	21.7
28	0.0	0.4	0.1	20.1
29	0.0	0.2	0.2	23.0
30	0.0	2.0	0.0	0.4
	Average : 0.0	Max : 8.5	Average : 0.2	Max : 31.8

Ring				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.0	6.6	0.0	0.1
2	0.0	5.8	0.3	21.3
3	0.0	3.4	0.2	21.3
4	0.0	1.7	0.3	22.6
5	0.0	0.8	0.0	0.2
6	0.0	0.1	0.0	10.0
7	0.0	0.4	0.1	20.0
8	0.0	1.9	0.0	10.0
9	0.0	0.1	0.1	20.0
10	0.0	0.2	0.1	20.0
11	0.0	6.4	0.1	20.0
12	0.0	0.1	0.7	21.6
13	0.0	5.4	0.2	21.4
14	0.0	3.0	0.2	20.0
15	0.0	1.7	0.1	21.4
16	0.0	4.3	0.2	20.0
17	0.0	7.2	0.1	22.4
18	0.0	0.5	0.1	20.0
19	0.0	1.6	0.3	22.1
20	0.0	3.9	0.1	20.1
21	0.0	0.0	0.2	21.4
22	0.0	6.0	0.1	20.8
23	0.0	1.7	0.1	20.0
24	0.0	6.5	0.5	21.3
25	0.0	1.9	0.0	11.4
26	0.0	4.2	0.2	20.0
27	0.0	1.7	0.1	21.4
28	0.0	6.8	0.1	20.0
29	0.0	1.2	0.2	21.6
30	0.0	0.1	0.1	20.0
	Average : 0.0	Max : 7.2	Average : 0.2	Max : 22.6

Compile				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.1	2.3	0.0	1.8
2	0.1	5.7	0.1	50.7
3	0.1	4.3	0.1	50.2
4	0.1	7.7	0.0	0.6
5	0.1	3.1	0.0	1.2
6	0.0	2.3	0.1	50.3
7	0.1	2.8	0.0	9.2
8	0.1	6.9	0.0	0.7
9	0.0	3.1	0.0	0.7
10	0.1	2.5	0.1	51.9
11	0.1	4.2	0.1	50.0
12	0.1	4.9	0.2	50.7
13	0.0	3.1	0.0	0.8
14	0.1	4.2	0.1	50.7
15	0.1	4.1	0.0	0.6
16	0.1	3.4	0.0	0.7
17	0.0	2.6	0.0	0.6
18	0.1	4.2	0.0	0.7
19	0.1	5.0	0.0	0.7
20	0.1	3.2	0.0	0.7
21	0.1	2.0	0.1	50.3
22	0.1	2.8	0.2	50.8
23	0.1	4.7	0.0	0.8
24	0.1	3.1	0.0	0.7
25	0.0	4.3	0.0	0.7
26	0.1	9.8	0.1	50.3
27	0.1	3.2	0.1	50.3
28	0.1	3.0	0.2	52.4
29	0.1	3.6	0.0	0.7
30	0.1	3.6	0.0	0.7
	Average : 0.1	Max : 9.8	Average : 0.1	Max : 52.4

A.2 Video Interactive Task Results Data

None						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	2.9	0.0	16.7	4.2	16.9
2	0.1	16.7	0.0	16.7	4.2	16.7
3	0.0	2.6	0.0	0.8	4.2	16.7
4	0.0	2.5	0.0	16.7	4.2	16.7
5	0.0	16.7	0.0	16.8	4.3	66.7
6	0.0	2.9	0.0	2.7	4.2	16.7
7	0.0	2.2	0.0	16.7	4.2	16.7
8	0.0	2.2	0.0	2.7	4.2	16.7
9	0.0	2.6	0.0	16.7	4.2	16.9
10	0.0	2.4	0.0	2.7	4.2	16.7
11	0.1	16.7	0.1	66.8	4.2	16.7
12	0.0	2.7	0.0	16.7	4.2	16.7
13	0.0	2.3	0.0	16.7	4.2	16.7
14	0.0	6.8	0.0	16.7	4.2	16.7
15	0.1	18.4	0.0	25.5	4.2	16.7
16	1.0	16.7	0.1	50.0	4.2	66.7
17	0.1	16.7	0.0	16.7	4.2	16.7
18	0.0	2.4	0.0	16.7	4.2	16.7
19	0.0	2.8	0.0	16.7	4.2	16.7
20	0.0	3.0	0.0	16.7	4.2	16.7
21	0.0	2.2	0.0	16.7	4.2	16.7
22	0.1	23.5	0.0	16.9	4.2	16.7
23	0.1	24.0	0.0	16.7	4.2	16.7
24	0.0	2.1	0.0	2.8	4.2	16.8
25	0.0	2.3	0.0	16.7	4.2	16.7
26	0.0	2.7	0.0	16.7	4.2	16.7
27	0.0	1.8	0.0	2.7	4.3	66.7
28	0.1	20.9	0.0	25.8	4.3	17.0
29	0.0	2.9	0.0	16.7	4.2	16.7
30	0.0	2.6	0.0	16.7	4.2	16.7
	Average : 0.1	Max : 24.0	Average : 0.0	Max : 66.8	Average : 4.2	Max : 66.7

X						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.4	23.5	1.5	66.2	12.4	50.2
2	0.4	16.7	1.5	70.3	12.5	50.1
3	0.3	23.6	1.3	45.7	12.3	74.2
4	0.4	24.3	1.5	50.0	12.2	50.1
5	0.5	23.6	1.5	68.6	12.3	109.4
6	0.4	23.5	1.4	68.9	12.0	50.1
7	0.3	24.1	1.6	68.2	12.5	96.5
8	0.4	16.7	1.6	78.3	12.7	50.2
9	0.4	29.7	1.4	69.3	12.0	50.0
10	0.4	21.0	1.7	80.8	12.3	50.1
11	0.4	16.8	1.5	48.7	12.4	50.0
12	0.4	22.6	1.4	73.4	12.1	50.1
13	0.4	22.7	1.7	66.9	12.7	50.0
14	0.3	16.7	2.0	70.6	12.3	50.0
15	0.4	23.6	1.3	66.4	12.7	50.0
16	0.4	16.7	1.6	68.0	12.7	50.1
17	0.4	23.4	1.4	73.9	12.1	50.1
18	0.4	17.0	1.5	65.7	12.4	50.1
19	0.4	28.5	1.6	70.1	12.2	50.0
20	0.3	23.1	1.4	69.5	12.4	50.1
21	0.4	23.4	1.3	48.5	12.3	50.1
22	0.4	33.3	1.8	80.4	12.4	50.1
23	0.4	16.7	1.6	70.4	12.2	50.0
24	0.3	22.2	1.4	68.1	12.6	111.3
25	0.4	23.4	1.5	70.9	12.3	50.1
26	0.4	23.8	1.6	68.9	12.4	64.6
27	0.4	18.0	1.5	70.7	12.1	112.1
28	0.3	16.7	1.5	67.4	12.2	50.0
29	0.4	24.0	1.6	47.3	12.4	116.7
30	0.4	24.1	1.5	67.3	12.2	113.8
	Average : 0.4	Max : 33.3	Average : 1.5	Max : 80.8	Average : 12.3	Max : 116.7

Burn						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.1	33.3	0.2	28.2	159.0	183.4
2	0.0	23.7	0.1	28.2	159.4	183.4
3	0.0	22.4	0.1	28.2	158.2	183.3
4	0.0	0.1	0.2	133.3	159.3	183.4
5	0.0	23.7	0.0	16.7	159.1	183.3
6	0.0	22.7	0.1	26.7	159.3	183.3
7	0.0	0.1	0.1	26.7	169.5	216.7
8	0.0	21.6	0.0	16.7	159.0	183.3
9	0.0	1.7	0.0	1.6	159.4	216.7
10	0.0	1.7	0.1	36.7	158.9	183.4
11	0.0	21.2	0.0	16.7	158.8	183.3
12	0.0	2.7	0.0	16.7	158.8	183.3
13	0.0	18.7	0.0	16.7	158.7	183.4
14	0.0	22.4	0.0	16.7	158.6	183.4
15	0.0	21.3	0.1	29.7	158.8	183.3
16	0.0	21.6	0.1	28.2	159.0	183.3
17	0.0	30.6	0.0	16.7	159.2	183.4
18	0.0	1.1	0.1	32.0	159.5	183.4
19	0.0	16.7	0.0	0.7	158.7	183.3
20	0.0	24.0	0.0	6.3	159.1	233.3
21	0.0	1.8	0.0	16.7	159.1	183.4
22	0.0	23.8	0.0	1.5	158.6	183.3
23	0.0	16.7	0.0	16.7	158.6	183.3
24	0.0	2.4	0.1	36.7	158.3	183.3
25	0.0	22.2	0.1	26.7	160.0	200.0
26	0.0	20.5	0.1	28.5	158.9	183.3
27	0.0	0.2	0.0	16.7	159.1	183.4
28	0.0	23.1	0.1	26.7	159.0	183.4
29	0.0	20.3	0.0	16.7	158.3	183.3
30	0.0	1.7	0.1	28.4	158.9	183.3
	Average : 0.0	Max : 33.3	Average : 0.1	Max : 133.3	Average : 159.3	Max : 233.3

Write				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.2	23.3	0.3	50.5
2	0.1	6.9	0.4	72.1
3	0.1	5.5	0.3	62.6
4	0.1	23.4	0.4	50.7
5	0.2	18.4	0.2	53.6
6	0.1	16.7	0.5	60.9
7	0.2	21.2	0.3	50.6
8	0.2	21.9	0.3	50.4
9	1.3	78.1	0.3	61.4
10	0.1	16.7	0.4	53.4
11	0.1	16.7	0.2	50.7
12	0.1	22.8	0.4	61.3
13	0.2	16.8	0.3	50.6
14	0.1	16.7	0.3	60.8
15	0.1	22.2	0.4	75.6
16	0.2	26.8	0.4	50.5
17	0.2	19.6	0.4	74.3
18	0.2	24.7	0.5	43.4
19	0.1	22.8	0.1	50.2
20	0.2	28.5	0.3	50.4
21	0.2	22.1	0.4	60.8
22	0.2	23.4	0.3	60.9
23	0.1	16.7	0.3	50.3
24	0.1	26.0	0.2	50.4
25	0.1	16.7	0.4	50.2
26	0.1	16.7	0.4	50.4
27	0.2	22.4	0.3	50.6
28	0.1	16.7	0.3	50.3
29	0.2	23.9	0.4	50.0
30	0.1	22.8	0.2	50.4
	Average : 0.2	Max : 78.1	Average : 0.3	Max : 75.6

Read				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.0	1.7	0.1	26.7
2	0.0	21.8	0.0	0.8
3	0.0	23.3	0.0	5.9
4	0.1	22.0	0.1	26.7
5	0.0	24.2	0.0	16.7
6	0.0	23.8	0.1	26.7
7	0.0	2.1	0.0	7.2
8	0.0	23.7	0.0	0.9
9	0.0	24.9	0.0	0.8
10	0.0	1.4	0.0	16.7
11	0.1	22.5	0.0	16.7
12	0.0	23.0	0.1	26.7
13	0.0	2.1	0.1	26.7
14	0.0	23.1	0.0	16.7
15	0.0	16.7	0.1	26.7
16	0.0	1.8	0.1	26.9
17	0.0	22.9	0.0	16.7
18	0.0	21.9	0.0	0.5
19	0.0	2.1	0.0	16.7
20	0.0	22.3	0.0	0.7
21	0.0	1.8	0.0	3.8
22	0.0	5.3	0.1	26.7
23	0.0	22.1	0.0	16.7
24	0.0	0.7	0.0	26.7
25	0.1	24.0	0.0	16.7
26	0.0	16.7	0.0	4.2
27	0.1	27.3	0.1	26.7
28	0.1	23.5	0.0	16.7
29	0.0	16.7	0.1	26.8
30	0.0	1.9	0.0	16.7
	Average : 0.0	Max : 27.3	Average : 0.0	Max : 26.9

Ring				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	0.0	19.2	0.0	16.7
2	0.0	23.5	0.0	0.8
3	0.0	1.7	0.0	16.7
4	0.0	23.6	0.0	16.7
5	0.0	3.8	0.0	0.9
6	0.0	3.1	0.0	24.8
7	0.0	24.0	0.1	36.7
8	0.0	2.2	0.0	0.8
9	0.0	1.8	0.0	0.8
10	0.0	21.8	0.0	16.7
11	0.0	19.2	0.0	16.7
12	0.0	23.0	0.1	40.1
13	0.0	23.0	0.1	26.7
14	0.0	18.9	0.0	0.7
15	0.0	25.0	0.0	6.4
16	0.0	22.1	0.1	26.7
17	0.0	1.6	0.0	16.7
18	0.0	4.5	0.0	0.8
19	0.0	23.5	0.1	27.0
20	0.0	1.7	0.0	0.7
21	0.0	16.7	0.0	5.1
22	0.0	22.5	0.1	36.9
23	0.0	1.7	0.1	26.7
24	0.0	24.0	0.0	4.4
25	0.0	18.9	0.1	26.7
26	0.0	0.3	0.0	16.7
27	0.0	16.7	0.0	16.7
28	0.0	1.9	0.0	6.5
29	0.0	1.7	0.1	25.5
30	0.0	21.1	0.0	16.7
	Average : 0.0	Max : 25.0	Average : 0.0	Max : 40.1

Compile Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	3.6	29.7	10.1	144.0
2	3.6	33.3	10.0	204.9
3	3.6	40.9	10.0	212.3
4	4.1	31.1	10.1	173.4
5	4.4	46.8	10.0	167.7
6	4.3	33.4	10.3	192.5
7	4.3	46.1	10.1	142.0
8	4.1	47.0	10.3	241.7
9	3.7	45.2	9.8	138.8
10	3.6	39.2	10.2	145.3
11	3.7	41.6	10.4	365.8
12	4.2	39.2	10.2	178.1
13	4.2	30.0	9.9	139.8
14	4.1	46.7	10.0	132.4
15	4.3	36.7	10.3	204.9
16	4.4	45.5	10.1	142.6
17	3.9	35.9	10.0	136.8
18	3.8	60.6	9.9	135.7
19	3.8	29.8	10.0	146.6
20	4.2	33.4	10.2	141.3
21	3.8	29.8	9.9	135.5
22	4.4	36.8	10.2	146.4
23	4.4	33.3	10.2	168.5
24	4.3	30.7	10.0	139.2
25	3.7	62.8	10.1	143.9
26	4.4	42.8	9.9	111.3
27	3.6	30.9	9.9	134.7
28	3.7	33.4	10.0	147.4
29	4.5	43.8	10.1	142.2
30	3.8	30.2	10.0	141.2
	Average : 4.0	Max : 62.8	Average : 10.1	Max : 365.8

A.3 X-window Interactive Task Results Data

None						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	0.0	0.2	18.0	0.8	15.0
2	0.0	0.0	0.1	10.0	1.0	18.0
3	0.0	0.1	0.1	10.0	0.9	16.0
4	0.0	0.0	0.1	12.0	0.9	14.0
5	0.0	0.0	0.1	6.0	1.0	14.0
6	0.0	0.0	0.2	10.0	1.0	17.0
7	0.0	0.0	0.4	68.0	0.9	18.0
8	0.0	0.0	0.5	38.0	0.9	14.0
9	0.2	48.0	0.1	10.0	0.9	16.0
10	0.0	0.0	0.2	24.0	0.9	14.0
11	0.0	0.0	0.1	14.0	1.2	54.0
12	0.0	0.1	0.2	15.0	0.9	24.0
13	0.0	0.0	0.1	10.0	1.2	18.0
14	0.0	0.2	0.1	12.0	0.9	16.0
15	0.0	0.0	0.1	10.0	1.1	16.0
16	0.0	0.0	0.1	10.0	1.1	18.0
17	0.0	0.0	0.1	10.0	1.0	16.0
18	0.0	0.1	0.1	10.0	0.9	15.0
19	0.0	0.0	0.1	12.0	0.8	15.0
20	0.0	0.1	0.3	27.0	1.2	19.0
21	0.0	0.0	0.5	39.0	0.8	15.0
22	0.0	0.0	0.1	20.0	1.3	77.0
23	0.0	0.1	0.1	9.0	1.0	17.0
24	0.0	0.0	0.1	12.0	1.0	14.0
25	0.0	0.1	0.1	12.0	0.9	14.0
26	0.0	0.0	0.1	8.0	0.9	16.0
27	0.0	0.0	0.1	15.0	0.9	16.0
28	0.0	0.0	0.1	10.0	0.9	15.0
29	0.0	0.1	0.1	6.0	0.9	15.0
30	0.0	0.0	0.2	20.0	0.8	14.0
	Average : 0.0	Max : 48.0	Average : 0.2	Max : 68.0	Average : 1.0	Max : 77.0

Video						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	0.0	16.7	83.0	1.4	22.0
2	0.1	10.0	18.1	80.0	1.5	22.0
3	0.1	10.0	17.9	84.0	1.3	22.0
4	0.1	14.0	18.4	80.0	1.6	80.0
5	0.4	15.0	16.7	85.0	1.3	22.0
6	0.2	15.0	17.5	84.0	1.4	24.0
7	0.1	9.0	17.5	80.0	1.5	23.0
8	0.0	0.0	19.0	136.0	1.1	20.0
9	0.0	10.0	17.6	80.0	1.4	22.0
10	0.0	6.0	18.4	93.0	1.1	18.0
11	0.0	9.0	17.5	114.0	1.3	25.0
12	0.1	9.0	17.4	84.0	1.3	24.0
13	0.0	5.0	18.0	85.0	1.4	22.0
14	0.1	21.0	16.6	85.0	1.5	28.0
15	0.1	11.0	17.3	80.0	1.3	20.0
16	0.3	64.0	16.9	79.0	1.6	25.0
17	0.1	36.0	17.6	80.0	1.4	22.0
18	0.0	8.0	18.0	84.0	1.3	20.0
19	0.1	11.0	17.1	88.0	1.2	23.0
20	0.0	0.0	18.0	80.0	1.1	20.0
21	0.1	15.0	19.2	184.0	1.2	22.0
22	0.0	6.0	17.0	80.0	1.3	20.0
23	0.1	14.0	17.4	78.0	1.3	24.0
24	0.1	10.0	17.1	85.0	1.2	43.0
25	0.0	2.0	17.8	80.0	1.2	22.0
26	0.0	0.0	17.3	80.0	1.3	22.0
27	0.2	20.0	17.6	96.0	1.2	20.0
28	0.1	15.0	17.1	79.0	1.2	24.0
29	0.1	10.0	17.6	79.0	1.4	24.0
30	0.1	8.0	17.9	90.0	1.2	21.0
	Average : 0.1	Max : 64.0	Average : 17.6	Max : 184.0	Average : 1.3	Max : 80.0

Burn						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	17.3	82.0	23.6	104.0	84.4	152.0
2	18.1	84.0	22.5	114.0	85.5	166.0
3	17.7	88.0	22.9	108.0	85.7	166.0
4	17.7	86.0	23.1	107.0	84.8	152.0
5	25.9	124.0	23.2	113.0	86.0	166.0
6	17.5	85.0	22.5	107.0	83.6	160.0
7	17.9	92.0	23.4	110.0	84.6	172.0
8	17.4	87.0	24.0	171.0	86.2	160.0
9	17.9	86.0	23.3	106.0	89.6	172.0
10	18.2	96.0	23.3	108.0	85.1	158.0
11	18.1	87.0	24.0	107.0	83.2	150.0
12	17.3	87.0	22.8	104.0	86.3	164.0
13	18.0	91.0	22.9	108.0	83.0	172.0
14	17.3	90.0	23.2	110.0	84.0	172.0
15	17.9	88.0	23.4	105.0	85.1	192.0
16	17.9	87.0	22.8	108.0	86.8	166.0
17	20.3	106.0	22.4	108.0	86.0	170.0
18	18.2	84.0	23.3	107.0	84.3	144.0
19	18.1	87.0	23.3	105.0	85.2	154.0
20	17.8	92.0	22.2	110.0	85.1	162.0
21	17.0	84.0	22.7	108.0	86.7	174.0
22	18.6	89.0	28.7	168.0	83.1	166.0
23	17.3	87.0	23.1	109.0	85.5	172.0
24	17.8	87.0	22.9	114.0	85.5	172.0
25	18.7	84.0	24.1	132.0	83.9	166.0
26	17.5	86.0	22.8	105.0	85.1	170.0
27	17.6	96.0	23.0	110.0	87.9	166.0
28	17.6	85.0	23.3	109.0	83.5	164.0
29	17.4	87.0	22.9	103.0	83.7	166.0
30	18.1	89.0	23.0	108.0	86.5	166.0
	Average : 18.1	Max : 124.0	Average : 23.3	Max : 171.0	Average : 85.2	Max : 192.0

Write				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	10.4	78.0	24.5	132.0
2	9.9	74.0	24.9	121.0
3	10.4	70.0	24.9	132.0
4	10.5	75.0	24.7	120.0
5	12.4	95.0	25.6	180.0
6	10.0	75.0	24.6	138.0
7	11.0	75.0	25.0	136.0
8	10.2	80.0	25.2	126.0
9	10.9	81.0	24.4	132.0
10	10.5	75.0	24.3	120.0
11	10.5	72.0	25.6	126.0
12	10.8	72.0	25.5	140.0
13	10.6	81.0	25.6	137.0
14	10.8	82.0	24.7	119.0
15	10.8	80.0	24.0	114.0
16	11.0	77.0	24.9	130.0
17	11.8	90.0	24.8	116.0
18	10.4	90.0	24.8	126.0
19	10.5	70.0	25.3	120.0
20	10.9	76.0	26.4	147.0
21	10.6	77.0	26.2	120.0
22	10.9	80.0	24.0	132.0
23	11.2	90.0	23.6	126.0
24	10.3	78.0	23.6	126.0
25	10.5	70.0	25.0	114.0
26	11.6	72.0	25.6	129.0
27	10.8	77.0	25.0	112.0
28	10.6	78.0	25.3	133.0
29	11.3	82.0	24.8	128.0
30	12.4	110.0	25.3	117.0
	Average : 10.8	Max : 110.0	Average : 24.9	Max : 180.0

Read				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	13.7	85.0	23.5	117.0
2	13.7	91.0	23.6	119.0
3	12.6	80.0	23.1	108.0
4	16.9	87.0	23.2	109.0
5	22.3	128.0	23.6	115.0
6	16.9	87.0	23.4	105.0
7	17.1	84.0	22.9	107.0
8	17.4	87.0	23.3	108.0
9	15.4	84.0	23.4	109.0
10	14.2	86.0	23.3	104.0
11	14.1	84.0	23.3	106.0
12	17.1	91.0	23.1	104.0
13	16.6	88.0	22.9	105.0
14	17.0	84.0	23.0	108.0
15	17.1	87.0	23.4	114.0
16	17.3	87.0	23.6	111.0
17	12.1	77.0	23.0	112.0
18	14.0	84.0	22.9	108.0
19	13.4	81.0	23.6	105.0
20	17.3	86.0	22.9	116.0
21	13.8	86.0	23.4	108.0
22	17.4	85.0	23.1	108.0
23	16.7	86.0	23.2	108.0
24	17.6	90.0	23.3	106.0
25	11.9	82.0	23.3	105.0
26	16.7	90.0	23.4	108.0
27	13.7	84.0	23.1	111.0
28	12.6	86.0	23.4	105.0
29	16.7	87.0	23.4	108.0
30	13.6	96.0	22.5	104.0
	Average : 15.6	Max : 128.0	Average : 23.2	Max : 119.0

Ring				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	17.3	88.0	23.0	104.0
2	17.8	89.0	22.5	108.0
3	17.8	86.0	23.3	102.0
4	17.5	90.0	23.1	104.0
5	31.4	128.0	23.1	106.0
6	17.6	86.0	22.6	107.0
7	17.8	91.0	27.1	240.0
8	17.4	88.0	23.1	112.0
9	17.7	91.0	23.5	108.0
10	18.1	84.0	23.3	106.0
11	17.2	86.0	22.5	108.0
12	17.7	89.0	22.6	102.0
13	18.0	88.0	23.4	108.0
14	18.2	86.0	23.5	104.0
15	18.2	94.0	23.4	109.0
16	17.9	88.0	23.1	108.0
17	18.7	88.0	23.4	107.0
18	17.9	91.0	23.0	108.0
19	17.4	89.0	23.4	108.0
20	18.3	85.0	22.6	108.0
21	17.5	86.0	23.2	112.0
22	17.9	90.0	23.0	104.0
23	17.8	89.0	23.6	104.0
24	17.5	86.0	22.9	108.0
25	18.5	86.0	23.0	144.0
26	17.9	87.0	22.6	108.0
27	16.9	92.0	22.6	108.0
28	17.7	136.0	22.6	108.0
29	18.3	90.0	22.7	108.0
30	18.4	95.0	22.8	112.0
	Average : 18.3	Max : 136.0	Average : 23.2	Max : 240.0

Compile Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	55.2	182.0	92.2	328.0
2	56.2	215.0	93.4	323.0
3	54.2	175.0	93.4	324.0
4	72.7	262.0	92.2	324.0
5	58.6	306.0	92.8	325.0
6	70.2	271.0	93.0	315.0
7	72.1	264.0	92.4	315.0
8	71.7	249.0	93.1	320.0
9	55.2	204.0	91.4	312.0
10	56.4	225.0	93.4	320.0
11	55.1	245.0	92.0	330.0
12	71.8	258.0	92.7	315.0
13	73.3	270.0	91.7	320.0
14	72.8	270.0	92.6	315.0
15	72.6	270.0	91.9	324.0
16	72.1	266.0	93.0	317.0
17	57.9	231.0	93.5	330.0
18	57.2	248.0	92.8	330.0
19	53.5	204.0	91.4	307.0
20	73.1	264.0	92.9	336.0
21	54.7	207.0	91.7	314.0
22	72.3	255.0	92.7	351.0
23	72.9	260.0	92.0	316.0
24	72.1	255.0	91.6	328.0
25	54.1	248.0	92.6	308.0
26	72.2	254.0	92.8	318.0
27	56.7	220.0	92.4	323.0
28	58.8	224.0	92.8	313.0
29	72.1	270.0	92.8	322.0
30	60.4	246.0	92.9	345.0
	Average : 64.3	Max : 306.0	Average : 92.5	Max : 351.0

A.4 Gaming Interactive Task Results Data

None						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.0	0.0	2.1	17.2	1.4	8.5
2	0.0	0.0	2.0	15.8	1.5	16.4
3	0.0	0.0	2.0	15.8	1.8	16.7
4	0.0	0.0	2.5	90.6	1.8	21.4
5	0.0	0.0	2.1	15.7	1.2	6.6
6	0.0	4.9	2.0	17.9	1.5	19.9
7	0.0	0.0	2.0	18.8	1.6	8.1
8	0.0	0.0	2.1	19.3	1.9	76.4
9	0.0	0.0	3.0	133.2	1.6	16.4
10	0.0	0.0	2.0	16.7	1.6	19.6
11	0.0	0.0	2.1	17.8	1.3	9.6
12	0.0	0.0	2.0	15.6	2.3	9.6
13	0.0	0.0	2.5	52.3	1.5	16.3
14	0.0	0.0	2.0	16.0	1.8	14.6
15	0.0	7.8	2.0	15.9	2.0	7.8
16	0.0	0.0	3.1	159.5	2.1	10.5
17	0.0	0.0	2.1	16.0	1.2	12.1
18	0.0	0.0	2.1	18.4	1.1	8.6
19	0.0	17.4	2.0	15.9	1.9	63.6
20	0.0	0.0	2.0	17.0	1.7	15.1
21	0.0	2.9	2.1	17.4	3.2	12.1
22	0.0	0.0	2.0	16.2	1.4	6.6
23	0.0	0.0	2.1	15.9	1.3	7.5
24	0.0	0.0	2.4	59.8	2.3	17.2
25	0.0	0.0	2.0	15.9	1.4	12.1
26	0.0	0.0	2.1	16.4	1.6	8.4
27	0.0	0.0	2.1	16.3	1.2	16.6
28	0.0	0.0	2.2	34.1	1.4	21.0
29	0.0	0.0	2.1	16.1	1.0	7.1
30	0.0	0.0	2.1	16.1	1.8	77.7
	Average : 0.0	Max : 17.4	Average : 2.2	Max : 159.5	Average : 1.6	Max : 77.7

Video						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	0.2	17.4	73.8	101.7	1.7	24.8
2	0.3	20.3	73.6	95.5	1.6	8.5
3	0.5	83.9	91.8	338.4	1.3	9.0
4	0.2	12.2	74.0	99.5	1.5	16.2
5	0.6	13.5	73.9	105.3	1.9	17.5
6	0.0	2.7	73.9	99.5	1.2	10.5
7	0.3	15.7	74.0	98.3	1.3	12.6
8	0.2	25.8	73.6	99.9	3.1	103.9
9	0.1	5.0	73.9	89.1	1.3	8.3
10	0.2	26.5	73.9	99.9	2.0	77.4
11	0.2	14.5	65.3	199.1	1.6	17.1
12	0.5	59.3	74.1	100.1	2.1	22.3
13	0.2	19.7	73.7	99.2	2.2	10.4
14	0.2	12.5	73.8	95.7	1.1	9.3
15	0.0	6.7	73.9	98.0	1.3	18.1
16	0.2	26.1	74.0	101.6	1.6	17.9
17	0.2	17.1	74.0	97.3	1.4	8.4
18	0.1	11.1	73.7	100.1	1.3	15.7
19	0.3	26.9	73.9	92.8	1.5	46.7
20	0.1	14.8	73.7	90.3	1.2	9.2
21	0.1	15.6	73.9	98.3	1.6	81.4
22	0.2	41.5	74.0	103.5	1.6	22.0
23	0.1	10.3	76.7	216.9	1.3	17.3
24	0.2	16.8	73.9	98.7	1.1	9.2
25	0.2	22.8	73.8	100.2	1.2	8.5
26	0.4	43.1	73.7	100.2	1.6	22.6
27	0.2	26.6	74.6	96.8	1.2	10.6
28	0.1	15.2	73.8	107.7	1.9	10.0
29	0.3	10.1	74.0	90.0	1.2	17.6
30	0.3	32.8	76.8	218.2	1.5	17.7
	Average : 0.2	Max : 83.9	Average : 74.4	Max : 338.4	Average : 1.5	Max : 103.9

X						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	13.3	95.1	68.0	200.8	59.7	140.3
2	12.9	78.4	66.9	139.0	58.5	133.7
3	13.6	95.3	68.0	175.3	57.5	139.9
4	14.7	118.3	67.2	133.0	58.1	142.1
5	16.7	83.5	67.9	173.1	60.0	139.6
6	14.4	87.1	66.9	126.6	59.6	140.0
7	14.8	92.5	67.3	153.6	58.1	139.8
8	14.4	84.6	66.6	130.7	57.7	138.9
9	12.9	83.1	67.1	128.5	57.8	138.9
10	13.1	82.9	67.3	149.7	57.9	133.7
11	13.0	84.3	67.0	151.7	57.2	138.9
12	13.6	77.5	66.4	128.9	59.6	141.0
13	13.6	86.9	67.1	160.0	59.4	136.4
14	13.5	92.7	68.3	151.0	58.7	146.1
15	13.5	84.8	68.2	321.8	58.5	140.6
16	12.9	71.8	67.4	124.2	57.9	134.8
17	12.3	89.1	67.1	149.0	58.5	138.5
18	13.1	88.7	68.7	203.6	59.3	141.9
19	13.1	86.7	67.2	121.6	58.1	137.8
20	12.1	84.8	67.0	129.0	58.4	138.6
21	13.0	94.3	67.2	144.2	61.4	137.8
22	13.2	90.5	67.7	170.0	57.8	140.2
23	12.5	90.0	67.3	159.7	59.2	138.0
24	12.9	82.7	66.5	139.3	59.4	140.5
25	13.5	85.2	67.4	145.6	58.0	141.5
26	13.2	90.5	66.2	133.4	58.0	140.5
27	13.6	93.8	67.0	133.9	57.7	139.2
28	14.4	87.1	67.3	165.4	58.1	140.8
29	15.8	135.5	67.1	143.1	58.1	182.9
30	13.7	94.5	67.2	133.6	58.3	143.3
	Average : 13.6	Max : 135.5	Average : 67.3	Max : 321.8	Average : 58.6	Max : 182.9

Burn						
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)	Windows (Average)	Windows (Max)
1	41.7	108.6	104.4	119.7	423.1	595.1
2	45.8	93.5	104.2	121.2	404.4	549.9
3	51.0	97.1	109.1	363.0	404.6	607.7
4	49.5	140.6	104.7	171.6	404.9	553.5
5	47.2	91.7	103.4	119.6	397.1	548.9
6	48.9	94.3	104.4	130.8	404.7	550.7
7	46.0	91.8	103.5	125.4	404.3	555.9
8	36.8	91.7	104.2	132.5	404.9	547.6
9	41.1	91.2	103.4	126.5	404.6	564.3
10	53.5	95.7	104.3	118.8	404.0	567.5
11	50.1	96.6	104.4	124.1	412.6	558.2
12	45.8	91.7	104.6	140.7	404.7	563.5
13	36.5	91.6	104.3	125.7	397.3	548.3
14	37.8	91.7	105.8	196.5	405.7	628.1
15	44.4	94.4	104.5	122.9	397.9	548.0
16	38.0	105.0	104.4	118.9	397.6	549.3
17	43.6	93.0	103.3	125.7	405.9	563.1
18	42.4	94.9	104.2	119.5	404.1	549.8
19	42.5	96.7	104.6	121.4	397.7	552.0
20	43.0	91.8	103.4	123.6	397.7	562.0
21	50.2	93.8	105.8	195.7	403.9	562.0
22	39.1	91.5	104.6	128.5	404.5	551.7
23	39.6	92.2	104.6	132.4	404.3	562.2
24	38.5	108.4	104.3	128.8	396.6	549.0
25	45.1	91.6	103.3	119.7	404.7	546.3
26	37.7	91.7	103.3	120.3	404.8	562.5
27	46.0	97.4	104.4	125.2	397.9	545.6
28	45.0	91.7	103.5	119.1	397.4	562.2
29	45.2	139.8	104.4	119.2	405.9	547.6
30	41.7	95.1	104.2	125.2	404.3	561.7
	Average : 43.8	Max : 140.6	Average : 104.4	Max : 363.0	Average : 403.4	Max : 628.1

Write				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	49.2	108.0	108.6	151.8
2	49.4	99.6	109.3	156.6
3	48.2	109.1	124.8	419.0
4	54.0	125.3	109.2	165.1
5	50.4	110.0	109.0	189.0
6	47.8	102.7	114.1	375.3
7	47.3	103.0	108.4	225.6
8	47.9	155.2	108.6	158.5
9	50.7	118.2	109.2	154.4
10	49.2	113.6	108.0	164.2
11	50.5	103.8	108.5	180.0
12	48.4	109.3	108.7	133.3
13	47.0	96.2	108.8	138.2
14	47.5	102.5	107.3	152.5
15	47.4	104.9	108.4	151.4
16	50.0	100.9	109.1	138.1
17	52.5	125.9	109.5	227.6
18	49.5	105.3	108.7	140.4
19	49.2	102.7	110.2	243.0
20	46.9	97.0	108.5	145.5
21	51.1	104.8	108.8	148.5
22	46.9	96.0	107.9	157.3
23	47.6	96.6	110.4	195.1
24	45.8	98.6	109.4	157.2
25	49.2	100.9	108.7	147.0
26	49.3	101.1	110.5	150.9
27	50.3	109.2	109.5	154.2
28	50.3	109.3	111.4	187.5
29	55.9	141.0	108.0	206.0
30	51.6	116.4	110.4	195.2
	Average : 49.4	Max : 155.2	Average : 109.7	Max : 419.0

Read				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	81.6	91.9	104.7	136.0
2	82.6	101.8	104.8	122.4
3	81.7	98.5	146.4	368.2
4	82.5	96.6	105.3	126.8
5	81.6	101.3	106.3	206.9
6	81.7	96.1	109.6	333.7
7	81.7	93.6	105.0	119.2
8	81.5	92.0	104.9	123.0
9	82.1	93.2	105.1	122.1
10	81.5	94.5	104.4	119.4
11	82.0	93.7	104.4	127.3
12	82.1	93.5	105.3	128.7
13	47.0	96.2	105.0	122.0
14	47.5	102.5	104.4	117.8
15	47.4	104.9	104.3	125.8
16	81.4	96.1	104.5	119.5
17	82.3	95.7	105.1	122.7
18	82.1	94.4	104.7	117.5
19	82.4	94.6	104.8	121.2
20	82.3	93.4	104.9	117.6
21	81.5	92.0	104.8	123.8
22	81.4	94.6	104.8	121.4
23	81.6	93.0	104.9	121.0
24	80.4	93.5	104.9	129.0
25	82.2	91.9	104.2	119.2
26	81.4	97.1	104.7	121.2
27	82.1	92.0	104.7	124.4
28	81.5	96.1	104.9	122.7
29	84.0	99.1	105.0	119.9
30	81.6	95.6	104.8	120.6
	Average : 78.4	Max : 104.9	Average : 106.4	Max : 368.2

Ring				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	47.3	96.7	104.2	120.3
2	45.8	94.3	103.5	118.8
3	43.8	95.6	125.9	362.3
4	54.8	145.9	104.3	122.6
5	48.4	92.2	104.3	118.3
6	49.4	134.8	103.3	120.2
7	46.5	93.6	103.0	120.5
8	43.0	95.6	103.3	119.0
9	47.2	94.4	104.3	119.1
10	48.7	92.5	104.3	118.5
11	51.7	91.7	103.4	119.9
12	41.9	91.6	103.3	128.7
13	37.1	91.6	103.4	122.2
14	43.2	95.2	103.4	118.3
15	44.3	91.7	104.3	118.6
16	42.3	91.7	104.3	118.8
17	50.4	93.5	104.3	117.1
18	42.9	91.6	103.3	122.5
19	52.7	93.0	105.7	232.0
20	40.2	91.6	103.4	115.9
21	49.3	91.6	103.4	118.5
22	50.2	144.2	106.2	359.0
23	42.6	94.2	103.3	121.1
24	44.2	91.7	104.3	118.8
25	45.9	94.2	103.3	120.4
26	43.7	91.5	105.8	244.3
27	46.6	91.9	104.2	127.6
28	45.1	91.2	103.0	119.8
29	57.2	145.6	103.3	137.3
30	44.4	91.6	104.2	120.3
	Average : 46.4	Max : 145.9	Average : 104.7	Max : 362.3

Compile				
Bench Sample Number	Ubuntu (Average)	Ubuntu (Max)	macOS (Average)	macOS (Max)
1	227.4	283.7	314.3	353.7
2	230.0	293.5	314.5	365.9
3	224.3	283.0	344.6	837.5
4	224.4	341.8	314.2	371.2
5	229.8	285.3	314.6	356.4
6	226.4	283.3	315.1	356.5
7	216.7	270.7	314.9	385.7
8	220.1	301.0	314.8	376.9
9	230.6	283.7	316.9	374.2
10	222.0	281.8	314.9	347.3
11	224.9	292.9	317.0	379.9
12	218.2	282.7	314.6	349.0
13	217.4	290.4	317.8	399.5
14	223.1	282.6	314.9	359.6
15	220.4	283.0	319.3	363.3
16	234.0	361.2	317.2	356.9
17	235.6	297.9	316.9	359.8
18	233.7	283.6	314.5	356.9
19	231.5	284.7	314.5	358.1
20	219.5	284.3	315.1	350.1
21	224.7	286.4	314.3	367.5
22	219.7	281.1	315.0	366.3
23	217.2	270.4	318.0	358.7
24	212.9	283.6	317.7	368.1
25	227.6	292.3	316.9	369.5
26	226.7	293.0	314.7	361.6
27	230.2	296.1	314.9	365.1
28	231.9	282.9	314.6	349.5
29	234.4	309.5	314.9	350.7
30	240.0	357.0	314.8	363.2
	Average : 225.8	Max : 361.2	Average : 316.5	Max : 837.5

APPENDIX B – WEEKLY LOG

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 1
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

All the interactive tasks were executed by excluding emulate_memload( ) function from the background loads as Memload cannot be simulated successfully for now. Preparation for verification was done by disabling extra physical and virtual CPU cores in the system. The number of CPU cores was set to 1 through XCode's Instrument Preferences and hardware multithreading was disabled. Besides that, Intel Turbo Boost Technology was also disabled.

2. WORK TO BE DONE

Proceed with verification for Audio interactive task.

3. PROBLEMS ENCOUNTERED

Faced difficulties in disabling Intel Turbo Boost Technology as Apple computer products do not have a BIOS where the option of disabling/enabling Intel Turbo Boost Technology can be found in other computer systems. Turbo Boost Switcher, which allows user to disable or enable Intel Turbo Boost Technology did not work for the first installation. To solve this problem, a clean macOS Mojave 10.14.6 installation was performed.

4. SELF EVALUATION OF THE PROGRESS

Try to spend some time looking for alternatives instead of sticking to only one solution in the project.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 2
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

Verification on the CPU utilization was started by monitoring the CPU utilization for Audio interactive task and the background loads used for concurrent execution. At the end of the verification, Audio interactive task's was proven to be able to generate 5% CPU utilization which is similar to the original Linux Interbench.

2. WORK TO BE DONE

Proceed with verification for Video interactive task.

3. PROBLEMS ENCOUNTERED

Faced difficulties in monitoring CPU utilization graphs as WindowServer process which is responsible in rendering the Graphical User Interface (GUI) consumes high amount of CPU, causing inconsistencies in CPU utilization graphs in Activity Monitor. As a solution, CPU utilization data was collected into a data file and illustrated using gnuPlot.

4. SELF EVALUATION OF THE PROGRESS

Try to repeat the verification process to ensure that the interactive tasks and background loads in the macOS Interbench are able to behave consistently.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 3
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

Verification on the CPU utilization was continued by monitoring the CPU utilization for Video interactive task and the background loads used for concurrent execution. At the end of the verification, Video interactive task's was proven to be able to generate 40% CPU utilization which is similar to the original Linux Interbench.

2. WORK TO BE DONE

Proceed with verification for X-window interactive task.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to repeat the verification process to ensure that the interactive tasks and background loads in the macOS Interbench are able to behave consistently.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 4
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

Verification on the CPU utilization was continued by monitoring the CPU utilization for X-window interactive task and the background loads used for concurrent execution. According to the execution of the same interactive task on Linux Interbench, X-window interactive task should be able to emulate CPU utilization ranging from 0% to 100% to simulate a condition where a Graphical User Interface (GUI) window is being grabbed and dragged. In conclusion, the behavior of X-window on macOS is similar compared to the one in the original Linux Interbench.

2. WORK TO BE DONE

Proceed with verification for Gaming interactive task.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to repeat the verification process to ensure that the interactive tasks and background loads in the macOS Interbench are able to behave consistently.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 5
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

The last interactive task verification for macOS Interbench was conducted. Gaming interactive task utilizes as much CPU as it can in Linux. Similar behavior was obtained from macOS Interbench. All the interactive tasks and background loads in macOS Interbench were verified with the interactive tasks and background loads in Linux Interbench.

2. WORK TO BE DONE

Perform verification for Windows Interbench ported by previous researcher before conducting the actual comparison of interactivity performance between Linux, macOS and Windows.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to make sure that the CPU configuration for the verification process is maintained for all operating systems so that the verification will be reliable before proceeding to the final comparison.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 6
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

Verification environment for Windows Interbench was set up. Started with a clean installation of Windows 10 Single Language on the test computer. Limiting the number of active CPU cores to 1 by making changes in the boot option located within System Configuration program. Then, Intel Turbo Boost Technology was disabled with the help of ThrottleStop program which is intended for CPU power adjustments.

After all the preparation steps were completed, the Windows Interbench from the previous research was executed for verification purposes. Just like the Interbench on macOS and Linux, each interactive task was executed concurrently with the background loads. Task manager was used to monitor the Windows Interbench CPU utilization while the benchmark program was executing at the same time.

After several executions, the behavior of Windows Interbench was similar to the Linux Interbench and macOS Interbench for every interactive tasks and background loads involved.

2. WORK TO BE DONE

Collect the average and maximum scheduling latency values by executing Windows Interbench for 30 iterations.

3. PROBLEMS ENCOUNTERED

Error occurred on the first attempt of executing Windows Interbench. Problem solved by recompiling Windows Interbench using Microsoft Visual Studio 2013.

4. SELF EVALUATION OF THE PROGRESS

Try to use batch scripts to automate the Interbench benchmarks for all operating systems to save time and maintain consistencies in the benchmarks.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 7
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

For the execution of Windows Interbench and collection of scheduling latencies, a batch script is written for the repeated execution of the benchmark program for 30 times. When the batch script was running, the test computer was left uninterrupted in order to avoid any unnecessary workload that can affect the final results. After running Windows Interbench for 30 times, the results were written to a log file. Lastly the collected results were transferred to a spreadsheet for comparison after all the benchmark results are obtained.

2. WORK TO BE DONE

Prepare benchmark environment for Linux Interbench. Collect the average and maximum scheduling latency values by executing Linux Interbench for 30 iterations.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to use batch scripts to automate the Interbench benchmarks for all operating systems to save time and maintain consistencies in the benchmarks.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 8
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

The test computer was formatted and included with a clean installation of Linux OS (Ubuntu). Similar to Windows, number of active cores was changed and Intel Turbo Boost Technology was disabled from the terminal. A shell script was written for automation of Linux Interbench. The shell script was executed to run the benchmark program repeatedly for 30 times. The test computer was left uninterrupted to increase the accuracy of the final benchmark results. After the completion of the benchmark, the scheduling latencies were recorded into a spreadsheet for future comparison.

2. WORK TO BE DONE

Prepare benchmark environment for macOS Interbench. Collect the average and maximum scheduling latency values by executing macOS Interbench for 30 iterations.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to use shell scripts to automate the Interbench benchmarks for all operating systems to save time and maintain consistencies in the benchmarks.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 9
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

For macOS Interbench, the test computer was formatted and clean installation was performed just like the previous benchmark preparation. A shell script was also written for repeated execution of macOS Interbench for 30 times. Number of active CPU cores was set to 1 with the help of XCode. Then, Turbo Boost Switcher was used to switch off Intel Turbo Boost Technology. The shell script was executed and the test PC was left uninterrupted. Lastly, the benchmark results were recorded into a spreadsheet for comparison purpose.

2. WORK TO BE DONE

Repeat benchmark for macOS with boot arguments changed.

3. PROBLEMS ENCOUNTERED

Even with the extra physical and virtual CPU cores disabled through XCode, ud_cpuload variable still remains as 4. This causes 4 threads used for simulation of Burn, Ring and Compile background loads instead of 1 thread. This leads to an unfair benchmark.

4. SELF EVALUATION OF THE PROGRESS

Try to figure out reasons that produces the benchmark results to be included in the discussion of the final comparison.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 10
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

The execution of macOS Interbench was repeated by changing the number of active CPU cores through 'sudo nvram boot-args="cpus=1"' instead of using XCode. The same macOS Interbench was executed for 30 times to obtain the latest values for scheduling latencies. The latest macOS benchmark results showed a significant decrease in average and maximum scheduling latencies for Burn, Ring and Compile background loads in all interactive tasks compared to the previous macOS benchmark.

2. WORK TO BE DONE

Analyze the benchmark results recorded in the spreadsheet for Linux, macOS and Windows. Generate graphs to illustrate their interactivity performance.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Try to figure out reasons that produces the benchmark results to be included in the discussion of the final comparison.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 11
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

All benchmarks for Linux, macOS and Windows were completed. The average latencies for background loads executed together with each of the interactive tasks were calculated by totaling the average latencies from 30 benchmark samples and dividing them by 30. The maximum latencies were determined by obtaining the greatest value within the 30 benchmark samples. Bar charts were created for the calculated values for every background loads within every interactive tasks.

2. WORK TO BE DONE

Add necessary comments into the source code of macOS Interbench and remove temporary codes used for debugging previously. Proceed to discussion of comparison results.

3. PROBLEMS ENCOUNTERED

-

4. SELF EVALUATION OF THE PROGRESS

Provide sufficient details and explanations for the comparison results.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project I / Project II)

Trimester, Year: Semester 3, Year 3	**Study week no.:** 12
Student Name & ID: Fan Wei Cong (16ACB02681)	
Supervisor: Mr. Wong Chee Siang	
Project Title: Interactivity Performance Benchmark for Windows and Mac OS	

1. WORK DONE

Code cleanup performed by removing all unnecessary codes used in troubleshooting and debugging. New comments added to allow better understanding of codes in future use. The benchmark program was executed once to make sure that the changes made does not cause failure of macOS Interbench.

2. WORK TO BE DONE

-

3. PROBLEMS ENCOUNTERED -
4. SELF EVALUATION OF THE PROGRESS -

Supervisor's signature

Student's signature

FINAL YEAR PROJECT 2

INTERACTIVITY PERFORMANCE BENCHMARK FOR WINDOWS AND MAC OS

INTRODUCTION

Operating system (OS) refers to a software that acts as an interface between programs and computer hardware. The OS consists of a kernel, which is a fundamental component that provides services such as process management within a computer session. At any moment in a computer session, there will be multiple tasks in an OS requesting to execute. Thus, the Central Processing Unit (CPU) scheduler, interrupt handler and process manager are introduced to handle the tasks' requests and to ensure fair execution of tasks. Scheduler is responsible to swap the tasks from time to time to ensure that the same task will not be monopolizing the CPU.

The performance of a scheduler can be determined in terms of interactivity. Interactivity is defined as the response time for a task to switch from Ready State to Running State. One of the most popular benchmark program that can be used to determine interactivity performance is Interbench. The main goal of Interbench is to measure the latencies and jitters that exist in Linux kernel schedulers under various simulated conditions. Interactivity is important especially in user-oriented systems such as desktops as a bad interactivity performance caused by large amount of latencies will lead to noticeable lags and unpleasant user experience. However, Interbench is only available for Linux. Other OS which are designed for user-oriented applications such as macOS and Windows were not supported by the benchmark program. In this project, porting of Interbench from Linux to macOS is done to find out the operating systems' interactivity performance.

PROBLEM STATEMENT

a. LACK OF SUPPORT OF INTERACTIVITY BENCHMARK PROGRAM FOR DIFFERENT OPERATION SYSTEMS

Interbench only supports Linux. Because the benchmark program was originally intended to compare different schedulers in Linux, the performance of the same set of benchmark tests on other OS such as Windows and macOS is limited.

b. INCREASING NEED FOR COMPARISON OF BENCHMARK PERFORMANCE IN TERMS OF INTERACTIVITY BETWEEN DIFFERENT OS KERNEL SCHEDULERS

OS developers are not able to obtain sufficient and accurate information regarding the interactivity performance of different OS kernel schedulers. Improvements of scheduling performance will become more challenging to achieve on newly rolled out operating systems or updates. Besides OS developers, normal users will also face a hard time finding the most efficient OS schedulers as there is no way to accurately determine their scheduling performance.

OBJECTIVES

- To modify existing benchmark application to enable the ability to run interactivity benchmarking on macOS.
- To reverify Windows Interbench's ability to work in different system hardware configurations.
- To differentiate the interactivity performance of different OS kernel schedulers from different operating systems by using a benchmark program with the simulation of multiple computer related tasks under different types of load.

METHODS

The whole research and development process is based on the Incremental Model.

DISCUSSION

In the benchmark program, emulation threads and timekeeping threads will gain access to shared resources by using semaphores. Semaphores for both Linux and macOS are based on the traditional counting semaphores. However, for macOS, Grand Central Dispatch's (GCD) dispatch semaphores are used in the ported Interbench because the semaphore.h in macOS which is similar to Linux's semaphore.h header contains deprecated functions. Table below shows the mapping of the original functions from Linux semaphores to GCD semaphores.

Linux	macOS
sem_init(sem_t *sem, int pshared, unsigned int value)	dispatch_semaphore_create(int value)
sem_wait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_FOREVER)
sem_trywait(sem_t *s)	dispatch_semaphore_wait(dispatch_semaphore_t *s, DISPATCH_TIME_NOW)
sem_post(sem_t *s)	dispatch_semaphore_signal(dispatch_semaphore_t *s)

Besides, Windows Interbench ported by the previous researcher was verified with Linux Interbench in the verification plan in the past research. However, the ported benchmark program was only executed on one set of system hardware which does not prove its ability to work on different hardware configuration. Therefore, reverification of Windows Interbench is completed before proceeding to the actual comparisons involving Linux, macOS and Windows. The outcome of the verification showed that the behavior of Windows Interbench is similar to Linux Interbench and macOS Interbench.

RESULTS

After the completion of macOS Interbench and verification plan for all three versions of Interbench, Two different comparisons for interactivity performance were conducted. Same set of interactive tasks is used for both comparisons. The first comparison covered Linux, macOS and Windows. The interactive tasks were executed concurrently with various background loads which include None, Video, X-window and Burn.

As for the second comparison, the background loads involved are Write, Read, Ring and Compile. This comparison only involves Linux and macOS because the four background loads specified are not available in Windows Interbench.

For the comparison results of the first comparison, Linux showed the best interactivity performance under most conditions, followed by macOS producing good interactivity performance especially for interactive tasks consuming low to medium amount of CPU. Windows shows the worst interactivity performance in most of the combination pairs of interactivity tasks and background loads. The poor performance became more obvious when the interactive tasks were executed within conditions with high background loads.

In the second comparison, Linux has a better interactivity performance in multiple interactive tasks with CPU utilizations ranging from low to high under Write, Read, Ring and Compile background loads.

CONCLUSION

In this project, Interbench has been successfully ported to macOS with the implementation of GCD semaphores and other changes. Reverification of Windows Interbench proved the benchmark program's ability to work in different hardware configurations. Comparisons done in this project shows that Linux is the best in terms of interactivity performance in most conditions.

Prepared by : Fan Wei Cong
Bachelor of Computer Science (HONS)
Faculty of Information and Communication Technology

PLAGIARISM CHECK RESULT

Document Viewer

Turnitin Originality Report

Processed on: 23-Apr-2020 17:44 +08
ID: 1305408725
Word Count: 11054
Submitted: 1

Interactivity Performance Benchmark For Windo... By Wc Fan

Similarity Index	Similarity by Source	
6%	Internet Sources:	2%
	Publications:	3%
	Student Papers:	3%

include quoted | include bibliography | excluding matches < 2 words | mode: show highest matches together | Change mode | print | download

1% match (publications)
C.S. Wong. "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers", 2008 International Symposium on Information Technology, 08/2008

1% match (publications)
Shen Wang. "Fairness and Interactivity of Three CPU Schedulers in Linux", 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 08/2009

<1% match (student papers from 05-Feb-2019)
Submitted to Asia Pacific Instutute of Information Technology on 2019-02-05

<1% match (publications)
"Distributed Embedded Control Systems", Springer Nature, 2008

<1% match (publications)
Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, Fun Wey. "Towards achieving fairness in the Linux scheduler", ACM SIGOPS Operating Systems Review, 2008

<1% match (Internet from 13-Mar-2016)
http://etheses.bham.ac.uk

<1% match (student papers from 24-Oct-2015)
Submitted to University of Johannsburg on 2015-10-24

<1% match (publications)
Nigel Jacob, Carla Brodley. "Offloading IDS Computation to the GPU", 2006 22nd Annual Computer Security Applications Conference (ACSAC'06), 2006

Universiti Tunku Abdul Rahman			
Form Title : Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Fan Wei Cong
ID Number(s)	16ACB02681
Programme / Course	Bachelor of Computer Science (HONS)
Title of Final Year Project	Interactivity Performance Benchmark For Windows And Mac OS

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)**
Overall similarity index: 6 **%** **Similarity by source** Internet Sources: 2 **%** Publications: 3 **%** Student Papers: 3 **%**	
Number of individual sources listed of more than 3% similarity: 0	

Parameters of originality required and limits approved by UTAR are as Follows:
- **(i) Overall similarity index is 20% and below, and**
- **(ii) Matching of individual sources listed must be less than 3% each, and**
- **(iii) Matching texts in continuous block must not exceed 8 words**

Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.

Note: Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

Signature of Supervisor

Name: Wong Chee Siang

Date: 23 April 2020

Signature of Co-Supervisor

Name: _____

Date: _____

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY (KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	16ACB02681
Student Name	Fan Wei Cong
Supervisor Name	Wong Chee Siang

TICK (√)	DOCUMENT ITEMS Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
√	Front Cover
√	Signed Report Status Declaration Form
√	Title Page
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
√	Appendices (if applicable)
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report. _____ (Signature of Student) Date: 23 April 2020	Supervisor verification. Report with incorrect format can get 5 mark (1 grade) reduction. _____ (Signature of Supervisor) Date: 23 April 2020