Umeå University 2021-06-14

Bachelor's Thesis In Computing Science

Linux CPU Schedulers: CFS and MuQSS Comparison

Name David Shakoori Gustafsson

Supervisor Anna Jonsson

Examinator Ola Ringdahl

Abstract

The goal of this thesis is to compare two process schedulers for the Linux operating system. In order to provide a responsive and interactive user experience, an efficient process scheduling algorithm is important. This thesis seeks to explain the potential performance differences by analysing the schedulers' respective designs. The two schedulers that are tested and compared are Con Kolivas's MuQSS and Linux's default scheduler, CFS. They are tested with respect to three main aspects: latency, turn-around time and interactivity. Latency is tested by using benchmarking software, the turn-around time by timing software compilation, and interactivity by measuring video frame drop percentages under various background loads. These tests are performed on a desktop PC running Linux OpenSUSE Leap 15.2, using kernel version 5.11.18. The test results show that CFS manages to keep a generally lower latency, while turn-around times differs little between the two. Running the turn-around time test's compilation using a single process gives MuQSS a small advantage, while dividing the compilation evenly among the available logical cores yields little difference. However, CFS clearly outperforms MuQSS in the interactivity test, where it manages to keep frame drop percentages considerably lower under each tested background load. As is apparent by the results, Linux's current default scheduler provides a more responsive and interactive experience within the testing conditions, than the alternative MuQSS. However, MuQSS's slightly superior performance using single process compilation may suggest that it is compatible with machines with a lower amount of logical cores.

Contents

1 Introduction

The performance of the modern operating system is deeply reliant on its process scheduling algorithms. The goal of the process scheduler is to efficiently give each concurrent process a chance to complete execution, while at the same time giving the user a smooth, uninterrupted experience [6, p.15]. A process scheduler chooses the next process to execute, how long it should run, and the order of execution. This means it must be able to handle prioritized processes, processes that are paused or stopped, as well as processes that are finished prematurely. Process scheduling is a necessary part of every multi-tasking operating system, which also holds true for the open source operating system Linux, as well as its many variants and distributions.

Although the Linux kernel already has a default scheduler, the Completely Fair Scheduler (CFS), there are alternatives available to be "patched in" as replacements. One of these alternatives is Con Kolivas's Multi Queue Skip list Scheduler (MuQSS). In short, this thesis focuses on comparing those two schedulers, the Linux kernel's present default scheduler CFS and Kolivas's more recently designed MuQSS scheduler.

1.1 Thesis Purpose

The purpose of this thesis is to test and compare the current default Linux scheduler, CFS, with Con Kolivas's MuQSS. According to Kolivas, the CFS scheduler is not designed with normal desktop PC users in mind, but rather for larger Linux servers and the like [6, p.51]. Thus he argues that his simpler design with MuQSS yields a desktop experience with better interactivity and responsiveness than CFS [8]. Interactivity focuses on how well the scheduler interacts with the user, while responsiveness is connected to multiple aspects, such as latency and turn-around time. Thus, the tests in this thesis focuses on the wake-up latency, turn-around time and interactivity of each scheduler.

In short, the thesis will primarily focus on theoretically explaining and experimentally showing the differences in performance between the schedulers. By running patched Linux kernels, the two schedulers are benchmarked and compared to see which one offers the best interactivity and responsiveness on a standard desktop computer. The following are the main research questions.

- Does MuQSS create a more interactive and responsive performance for desktop users than CFS does?
- How can the differences in performance between the two schedulers be explained by their respective designs?

1.2 Delimitations

Notably, the test environment of the schedulers is in this thesis limited to the testing on a single desktop PC. This is due to the limited availability of desktop PCs to test on. In particular, this means that a maximum of 4 logical cores are used.

Both schedulers (as well as the kernel itself) have features that can be tuned to work optimally for a specific machine, but due to the multitude of possible configurations, the schedulers are tested in their default states. The tests are only run on the latest kernel version, which is currently 5.11.18, as re-compiling the kernel for several kernel versions is too time-consuming.

2 Related Work

This thesis takes inspiration from a comparison between CFS and the progenitor to MuQSS, the Brain Fuck Scheduler (BFS), also created by Kolivas. The comparison was made by Groves, Knockel & Schulte [4], and they test the two schedulers on latency and turn-around time as well as interactivity. Their conclusion is that CFS achieves a lower turn-around time, while BFS has a lower latency. BFS also achieves a better result in their interactivity experiment, where they test how many frame drops occur during a video clip under various background loads. In this thesis, testing is done in a similar way, but using another set of software tools, which will be described.

Larabel [9] performs a similar test to Groves et al., using the benchmark software *Phoronix test suite.* They reach a different conclusion than Groves et al. did, as they find that BFS has a lower turn-around time while CFS has a lower latency. Explaining this discrepancy in test results is difficult, since Larabel describes the hardware used but not the method used for testing, while Groves et al. explain the tests but not the hardware. The difference is likely found in one, if not both, of those aspects.

As CFS has been the default scheduler for some time, Kolivas is not the first to critique its design. Wong et al. [12] describe a fairness issue found in CFS, where a multi-threaded program is given more CPU-time due to the fact that CFS does not differentiate between processes and threads (concepts explained further in 3.1). This means that a process divided into many threads is given a larger amount of CPU-time, than a single-threaded process. They suggest implementing a “process fair” scheduler, that keeps track of which threads belong to which process, to be able to divide a fair amount of CPU-time among them. Such a scheduler has not yet been implemented.

3 Theoretical Background

This section explains the essential concepts of process scheduling, as well as the two tested schedulers, CFS and MuQSS. As MuQSS is based on the BFS scheduler, BFS is also described.

3.1 Process Scheduling

As mentioned, process scheduling is an important part of every operating system. A process is a running instance of a program in the memory of a computer. Each process is made up of one or more *threads*, which are also called "lightweight processes" [6, p.2]. Each thread or process is seen as its own entity, and is therefore scheduled as such. Threads and processes are often called *tasks*, a term that is used to signify *processes* in this thesis.

It is the role of the scheduler to make sure that these processes are given CPU-time from the available processor resources. This means every process should be allowed to execute eventually. Scheduling is quite easy in a single-core system where only one task can run at a time, but it gets more advanced in multi-core systems, where processes can run in parallel. The cores mentioned here refers to *logical cores*, sometimes called *logical processors*, which are the amount of physical cores (hardware) multiplied by the amount of threads each such core can run. The amount of logical cores is thus equal to the number of tasks that can be performed simultaneously by the computer's processor.

As the Linux operating system uses *preemptive* multitasking, the scheduler's job is to decide when one process is to stop (preemption), and another is to begin. It has to be able to do this for every scenario that might occur in between. The CPU-time a process is given is often called *timeslice*, and the structure in which the processes are stored is named *the runqueue.*

Processes in the Linux operating system can also have different priorities, or *nice*-values as they are called. These values can range between -20 and 19, where a smaller number means the task requests a higher priority, and 0 is the default value [6, p.18]. A normal user can only lower the priority of a process, whereas increasing priority requires root-access. In the testing done for this thesis, the nice-values are not changed.

3.2 CFS

Since Linux kernel version 2.6.23, CFS has been the default CPU process scheduler of the Linux operating system [6, p.41]. The CFS scheduler was designed by the Linux developer Ingo Molnár, and was adopted into the kernel in 2007. A main goal within process scheduling has long been to reach

"Perfect Multitasking" [6, p.41], meaning a perfect partitioning of $1/n$ shares of CPU-time for n processes, and this is what CFS attempts to achieve. CFS is implemented using a red-black tree data structure as runqueue.

A red-black tree is a self-balancing binary tree. The tree makes use of two node colours to keep the tree balanced, meaning it should keep as low a height as possible. An example of a red-black tree can be viewed in Figure 1. It must abide by the following rules.

- Every node is either red or black.
- The root node is always black.
- No adjacent red nodes are allowed.
- Every path from any node to any of its descendant leaves must consist of the same number of black nodes.

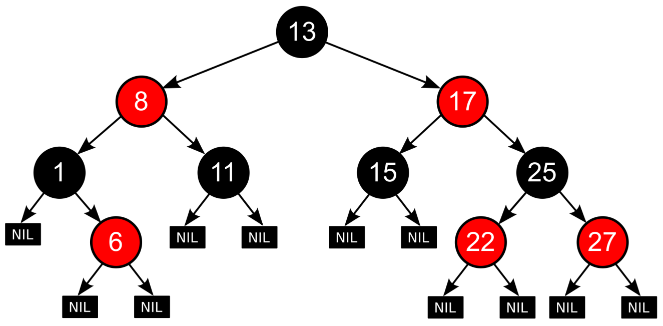

Figure 1: Visualized red-black tree [2] (Wikimedia Commons).

By following these rules, each main operation, namely insert, lookup, and delete, has a time complexity of $O(\log(n))$, where n is the amount of nodes in the tree. Being ordered like a binary search tree, the red-black tree stores the smallest values furthest to the left. To speed up the task-choosing part of the data structure, CFS always has a pointer to the leftmost element in the tree, meaning lookup in a single tree is of time complexity $O(1)$.

In CFS's tree structure, the tasks are ordered by how long each task has run for, meaning the tasks with the lowest total runtime are placed furthest to the front of the runqueue, which in the tree structure translates as being placed further to the left [6, p.46]. Thus the leftmost leaf node of the tree is always the next task to be allowed CPU-time, as it has the lowest virtual runtime. The processor picks out and executes the task which is next in line according to the tree structure. As the task is executing, its runtime increases, and it moves to the right in the tree after it has been put back into the runqueue.

Each logical processor maintains its own red-black tree runqueue. Depending on how “unfairly” treated a task is, i.e. how little runtime it has been given compared to other tasks, the sooner it will be given a timeslice of the CPU to work with. Thus each task in each processor’s runqueue must maintain the total amount of nanoseconds it has been allowed CPU-time, called “virtual runtime”.

Another important factor in the ordering of the tree is also the task’s priority. The total virtual runtime increases more slowly for a task with higher priority [6, p.47], meaning such a task will be given a timeslice more often. Thus a task with a higher priority moves more slowly to the right in the red-black tree, which constitutes the runqueue. Priority is also used to decide how big of a timeslice the task will receive, which will be described further.

CFS does not provide each process with a set timeslice of CPU-time, but rather calculates a timeslice based on the current total amount of runnable tasks. CFS divides the *default target latency*, which is 20 ms, among the current tasks within the same processor runqueue [7]. A larger number of runnable tasks means a smaller timeslice, but the default target latency value increases when the number of tasks grows too large [12, p.36]. The timeslice equation used is

$$\texttt{Timeslice} = \frac{\texttt{taskweight}}{\texttt{total\_rq\_taskweight}} \cdot \texttt{target\_latency},$$

where `taskweight` is the the task’s weight based on its nice-value (1 as default) and `total_rq_taskweight` is the total accumulation of weights in the runqueue [12, p.36]. Thus, a higher priority task will receive a larger timeslice than a lower priority task.

The CFS scheduler does another important thing; load balancing between the cores. Load balancing is used to divide process load between the runqueues in the logical cores, to minimize the risks of idle cores doing no work [10]. If a core finds that another core is especially “busy”, it will grab tasks from that core’s runqueue.

3.3 BFS

Around the same time as Molnár presented his CFS scheduler design, another Linux developer, Con Kolivas, left the Linux community after some disagreements [4, p.3]. He later resurfaced in 2009 with his own task scheduler, designed based on, according to him, more important principles than those used in the design of CFS. This was BFS, which was intended to be a more simple, easily maintainable scheduler [6, p.51].

The data structure used by BFS is a single queue of runnable processes. This choice was done in the name of simplicity, which was Kolivas's goal. BFS uses *virtual deadline* for fairness, which means each task in the queue is given and is ordered by a deadline. That deadline is the longest time any two tasks of the same priority (nice-value) will have to wait before being given a CPU-timeslice [4, p.7]. The tasks are of course not guaranteed to be given a timeslice before the deadline, but the parameter is what is used for ordering in the queue. A task is sent back into the queue when its timeslice is completed, and given a new deadline. If the process is prematurely paused, it keeps the remainder of its timeslice, and its deadline. Processes with higher priority are given earlier deadlines. The given timeslice is a tunable integer, `rr_interval`, which is set to 6 ms as default. The deadline is calculated using the equation

$$\texttt{Deadline} = \texttt{niffies} + (\texttt{prio\_rate} \cdot \texttt{rr\_interval}),$$

where `niffies` is a nanosecond specific timer counter obtained from the Time Stamp Counter in the processor, which counts CPU cycles since the last reset, and `prio_rate` is a ratio based on the priority of the task (nice-value) [5].

3.4 MuQSS

In 2016, Kolivas presented a new scheduler based on BFS, namely MuQSS [5]. Kolivas had noticed that the single runqueue implementation of BFS was not scalable enough, at least not for computers with a large number of cores. This was due to the fact that the runqueue had to be locked every time it was accessed by each individual logical core, which caused lock contention (cores waiting in line). Additionally, the $O(n)$ time complexity of using a single queue to house every task obviously increased the overhead of the scheduler under large task loads [8]. To combat these inadequacies, MuQSS introduces multiple runqueues. These queues are implemented using a skip list data structure.

A skip list is a probabilistic data structure, with an average $O(\log(n))$ search and insertion time complexity, where n is the amount of elements in the list. It works like a multi-layered ordered linked list, where the base layer contains all currently inserted elements, and each subsequent layer contains a smaller set of the same elements. Each time an element is added to a layer in the skip list, it has a certain probability to be added to the next layer as well (most often 1/2). The data structure is visualized in Figure 2. A binary search in the skip list starts on the topmost layer, which likely has the fewest elements, and jumps to a lower layer if the element cannot be found, and so on.

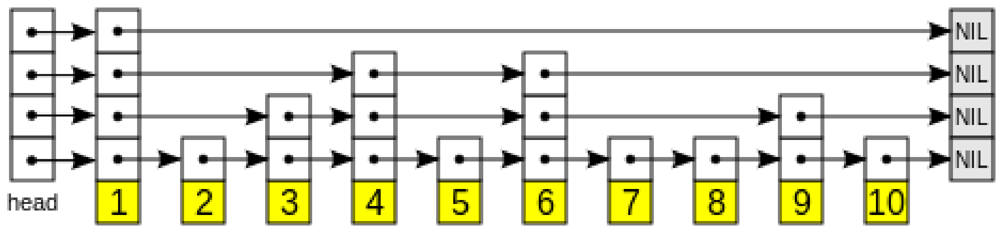

Figure 2: Visualized skip list with 4 layers [11] (Wikimedia Commons).

Unlike BFS with its one runqueue, MuQSS has one skip list per logical CPU. Thus every time a new task is to be chosen by a CPU, the first element in each CPU's skip list is checked to see which is the most suitable task to execute (looking at their virtual deadline). This means that task lookup in MuQSS is $O(k)$ where k is the number of logical processors [8]. The calculation of the virtual deadline in MuQSS is equivalent to the equation used in BFS, mentioned in 3.3. Notably, neither BFS nor MuQSS does any explicit load-balancing like CFS does.

The MuQSS-specific implementation of the skip list limits the amount of layers to 8, and the total amount of tasks to 256. This means the MuQSS scheduler allows for a total of $256 \cdot n$ tasks, where n is the number of logical cores [8].

3.5 Comparison of CFS and MuQSS

The three main differences between CFS and MuQSS lies in:

- What data structures they use to store the current tasks.
- What metric is used to order the tasks.
- How the timeslice is calculated.

As mentioned, CFS uses a red-black tree structure to store tasks, and orders tasks using their virtual runtime. When a task is chosen, it runs for a calculated timeslice which depends on parameters such as priority and total amount of tasks in the tree. MuQSS on the other hand, uses skip lists to store the tasks, and each task is given a virtual deadline for task execution to be ordered by. A chosen task is given a timeslice, `rr_interval`, which as mentioned is set at 6 ms.

The metric used to order the tasks, virtual runtime contra virtual deadline, likely has the least impact on performance out of the three main differences. Both of them only take a few calculations at most, before the tasks are inserted into their respective runqueue data structures. The timeslice size on the other hand can create quite the difference in some circumstances. Where a single task in MuQSS can only get a timeslice of 6 ms, a task in CFS can get anywhere between 1 and 20 ms depending on the total amount of tasks in the runqueue. This difference might have a significant impact on the test results.

As for the data structures used for runqueues in the schedulers, there are some interesting differences and similarities. Both of the structures feature an average $O(\log(n))$ insertion complexity, and as CFS's red-black tree includes a pointer to the leftmost element, they both also have a $O(1)$ lookup time in the case of a single runqueue. The red-black tree will however, by its ingenious design, never have a worse insertion time complexity. Being a probabilistic structure, the skip list always have a probability to be built up inefficiently, meaning it might take longer to insert a task into the runqueue. On the other hand, the red-black tree is a very advanced structure, where the trees always has to be balanced as well as following the given rules, whereas MuQSS's skip lists lack that overhead [5]. There is also the added overhead of CFS's load balancing, which is not a feature of MuQSS.

3.6 Scheduler Testing Factors

The thesis's experiments are primarily focused on the wake-up latency, turn-around time and interactivity of the two schedulers. These three aspects are well connected with the responsiveness and general interactivity of the scheduler, and measuring these can therefore be used to answer the first of the two research questions. They can be explained as follows:

Latency – The time between the waking of a process and its actual execution. To achieve a low latency, the scheduler must be efficient in its lookup of the next task, as well as its ability to actually hand over the ready task to be run by the processor.

Turn-around time – The total time a process needs to complete. As turn-around time is an aspect concerned with how long it takes a scheduler to finish a single process, testing it is an effective way of measuring the schedulers' performance. The turn-around time test measures not only how fast the scheduler can start running the task, but also putting it back in the runqueue data structure after running its timeslice, and starting it over again until complete. It is a general test of the overall speed of the scheduler's data structures, as well as its ability to divide CPU-time.

Interactivity – A more abstract aspect, which focuses on how well the scheduler can service the user, such as responding to keystrokes and other interaction. To achieve a good interactivity, a scheduler must have a consistently low latency in combination with a good ability to handle a large amount of tasks at the same time.

4 Method

The different tests were run on a dual-booted PC with OpenSUSE Leap 15.2 as well as two installed versions of Linux kernel version 5.11.18, one with the MuQSS scheduler version 5.11 patched in, and one with only the default CFS scheduler. OpenSUSE was installed on a mechanical harddrive (HDD) alongside Windows 10 Home. Using a HDD might have an impact on tests where reading and writing to disk is required, as the disk's cache will store some data until the next run of the test. The desktop is running on a Intel Core i5-4570 processor, 3.20 GHz, with 4 logical cores.

4.1 The Latency Test

The latency test makes use of a benchmark and stress test software called *Hackbench*. It was designed by Rusty Russel, with contributions from CFS's creator Ingo Molnár among others [3]. Hackbench sets up a number of tasks which can communicate in pairs, and measures the wake-up latency of the communication. The communication is done over either pipes, using local memory-based file descriptors, or sockets, which are most often used for communication between processes on different hosts. Since sockets are used for communication between hosts, and therefore have more overhead in setting up, the faster alternative, pipes, will be used. The test can run using multiple processes, or multiple threads. In this case, multiple threads will be used, in order to avoid the overhead of process creation and inter-process communications. After running, the test software returns the average latency in seconds. Hackbench is available as part of the *perf-bench* tool for running tests on the kernel.

In this test, the following command was run 50 times, and the results recorded:

```
perf bench sched messaging -p -t -g 20 -l 2000
```

where the first two flags indicate that the test should be done using pipes for communication rather than sockets, and that threads should be used. The second to last flag indicates how many pairs of communicating file descriptors should be used, and the last flag decides how many times the the messaging should loop. As the default configuration (without the last two flags) of the command gave very small differences between runs, they were set to higher values (default is 10 pairs and 1 loop).

4.2 The Turn-Around Time Test

This test makes use of the common command `make`, which is frequently used in compiling software in the Linux operative system. The test is composed of running `make` to compile some software, and timing how long it takes to complete. As a default, `make` runs one process at a time, but with the added j-flag, it can run several jobs simultaneously. The test is first run 10 times without any flags, and then run another 10 times as `make -j4`, which means it will attempt to divide the work on the 4 logical cores. The results are then recorded.

The chosen software for this experiment is the open source relational database software, PostgreSQL. Any software could realistically have been chosen, but PostgreSQL was of a decent enough size in memory to let `make` run for more than a few seconds. The uncompiled source files has a size of around 111 MB, and consists of almost 6000 files.

4.3 The Interactivity Test

Interactivity is, as mentioned, a bit more abstract than the other two aspects, and thus a bit harder to formulate a test for. This test however is inspired by a similar test done by Groves et al. [4]. The test uses a video player with built-in benchmark capabilities to record how many frames are dropped under various background loads. A one minute clip, from the opening of the short movie *Cosmos Laundromat* [1], available under Creative Commons, was chosen for the test. The clip is in 1920x804 resolution, and recorded in 24 frames per second. The clip was played using the MPlayer software, which has built in benchmark capabilities that allow recording of frame drops during the video. Using the following command, the benchmark was run a number of times, and the frame drop percentage recorded:

```
mplayer -benchmark -hardframedrop Cosmos_Laundromat.webm
```

where the `hardframedrop`-flag enables frame dropping. This means that the player does not stop playback to buffer late frames, which would otherwise be the player's way of handling frame drops. Thus the flag is needed to be able to get correct frame drop percentages from the benchmark.

Each run of the mplayer-test was accompanied by an increasing background load on the system. This was done using the rt-app software, which is a real-time workload simulator originally created by Giacomo Bagnoli, that can be used to create different numbers of background tasks to increase current system load [3]. The software takes a JSON file that describes the details of the background task execution. A JSON file is a formatted file in which data can be store, which in this case could be used to alter rt-app's functionality, such as duration and amount of threads. In this test, the JSON file was quite simple, only specifying that a certain number of tasks should be running and causing load/stress in the background. The test ran using 10, 20, 30, 40, 50, and 60 background tasks, respectively.

4.4 Test Reliability and Generalisations

It is, of course, impossible to test all possible hardware combinations and scheduler configurations. Likewise, it is impossible to perform every variation of each test that was performed, such as using different values in the latency test. Thus, such an approach to this problem is not feasible. This thesis, however, and the tests performed in order to write it, could reasonably be used as a sort of generalisation of how similar desktop hardware would perform while using the two process schedulers.

5 Results

In this section, the results of the three tests are presented. Each test have graphs and plots representing their respective experimental results, and the data shown is briefly discussed.

The Latency Test – The very first test puts the CFS scheduler at an advantage, with Figure 3 showing CFS having a consistently lower latency than MuQSS. Even though the lowest latency value from MuQSS is almost comparable to to CFS's highest latency values, those are both outliers.

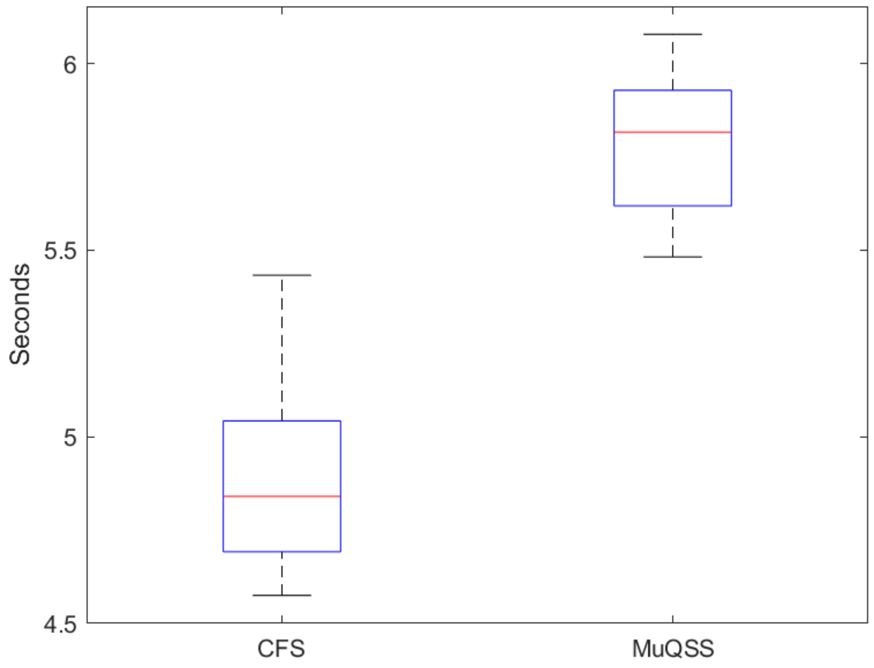

Figure 3: Box plot over latency test results in seconds from each scheduler.

The Turn-Around Time Test – Using the default `make` without any flags, MuQSS managed to deliver a generally lower compilation time than CFS did. In the graph presented in Figure 4, the first entry for both schedulers is around 190 seconds, before going down significantly for the rest of the runs. This is most likely due to caching done by the HDD to speed up compilation time. In the box plot of Figure 5, the first entry of the test data is omitted, which gives a better visual on the time comparison between the two schedulers.

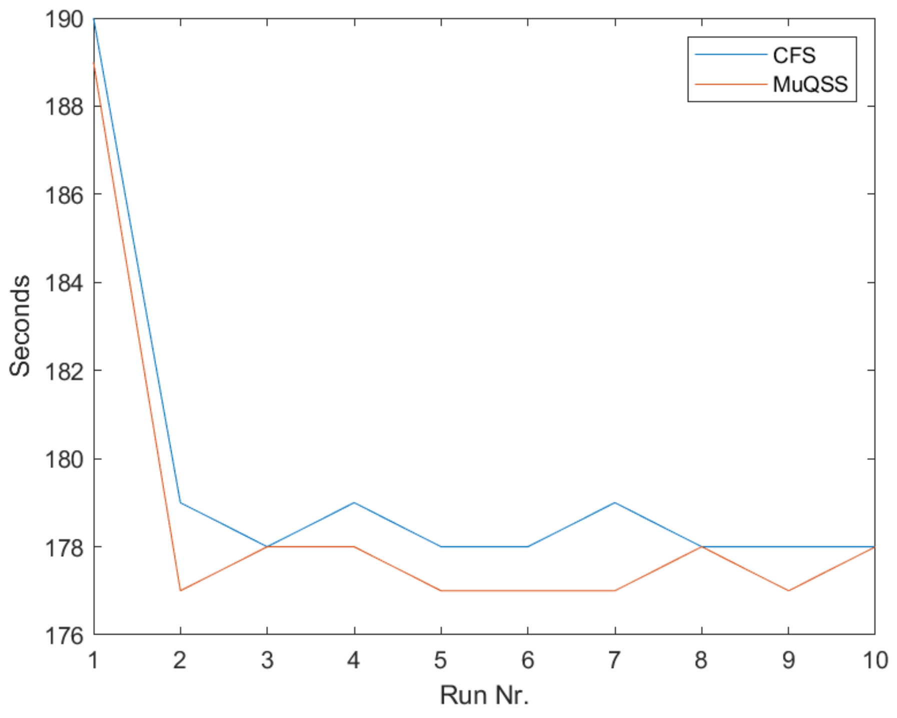

Figure 4: Line diagram over compilation time in seconds from the turn-around time test, for both schedulers.

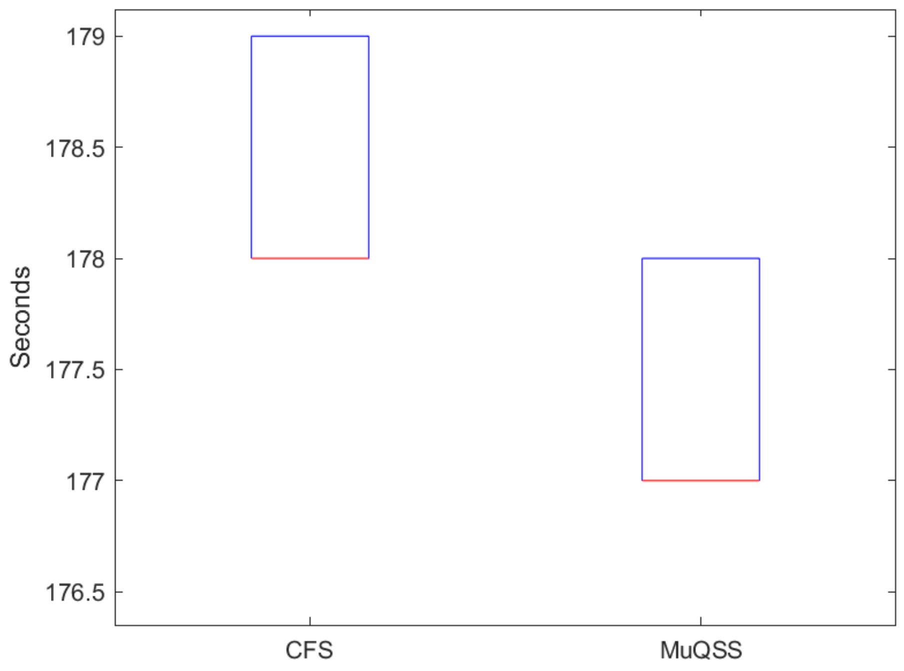

Figure 5: Box plot over compilation times from the turn-around time test, with the first entry omitted.

The results from using `make -j4`, in Figure 6, show less differences between the schedulers. The two higher outliers are once again the first run of the test, before caching speeds the process up. Using simultaneous jobs, the two schedulers manage to compile the software in similar times, although MuQSS has some small variations.

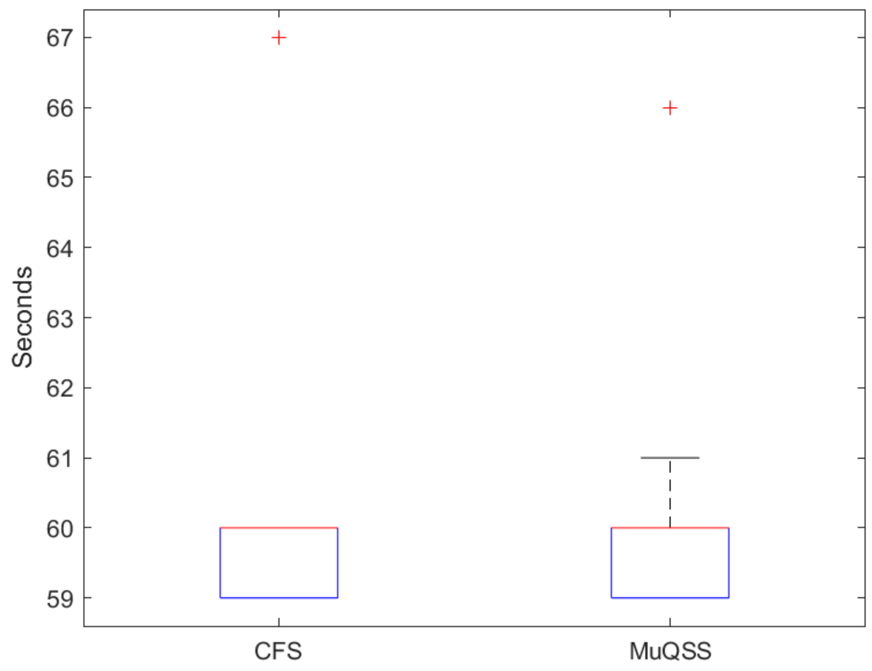

Figure 6: Box plot over compilation times from the turn-around time test, using `make -j4`.

The Interactivity Test – This test likely shows the biggest discrepancy between the two schedulers out of the three tests. As can be seen in Figure 7, CFS is a lot better at avoiding frame drops than MuQSS is. The video clip is instantly effected by the increased background load when using MuQSS, while CFS manages to keep a consistently low frame drop percentage until the number of background tasks surpass 40.

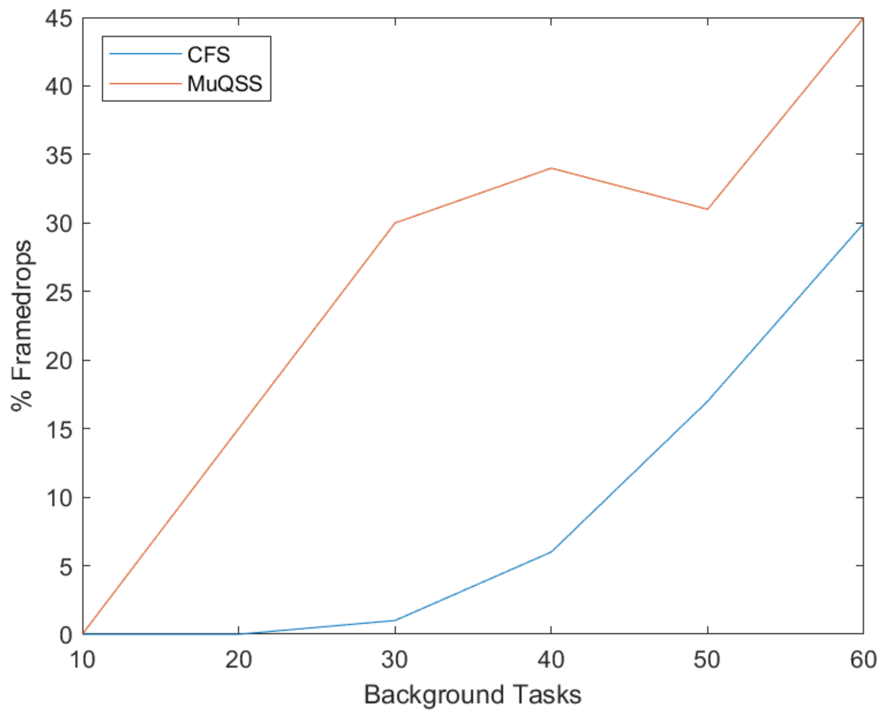

Figure 7: Line diagram over frame drop percentages per number of background tasks from the interactivity test.

6 Discussion

In this final section, the experimental results are analysed and discussed in relation to the theoretical details of the two schedulers, as that was the goal of the second research question. This is followed by a conclusion along with some comments on the limitations of the work, and potential future research.

6.1 Analysis

As is apparent by the test results, CFS seems to outdo MuQSS in two out of three tests, namely the latency and the interactivity test. There are many possible reasons behind CFS's lower latency. Because of CFS's use of load balancing, its logical processors do not have to look outside their own runqueues when choosing a task, granting a $O(1)$ time complexity. In comparison, MuQSS's processors check each logical core's runqueue for the most suitable task, which, on the hardware used for testing, gives it a latency that is 4 times larger than that of CFS (despite having the same time complexity of $O(1)$). This design choice was likely made by Kolivas in order to avoid the need of load balancing. This, in combination with CFS's more reliable tree data structure, might have led to shorter runqueue insertion times, and therefore a lower wake-up latency for tasks.

As mentioned, CFS managed to keep a consistently lower frame drop percentage than MuQSS did, under every tested background load. The interactivity test assesses multiple aspects and capabilities of the scheduler, meaning both turn-around time and latency might have some precedence, but the scheduler's most important task is to make sure that the interactive process, in this case the video, is consistently given enough time to run to avoid stutters, such as frame drops and the like.

CFS, with its unique timeslice calculation, might have an edge in this case. As CFS divides CPU-time based on the amount of tasks in the runqueue, the video player will always be given its fair share of CPU-time, even if its a generally small amount. This gives CFS good handling of up to around 30-40 concurrent background tasks, as can be seen in Figure 7, after which it gets harder due to the sheer number of tasks that the 4 cores have to share. MuQSS, on the other hand, always gives each process the same timeslice, namely 6 ms. This means the video player will have to wait in the runqueue longer and longer as background load increases, thus creating more severe frame drops.

MuQSS managed to keep a generally lower turn-around time using the default `make`, while the results were very similar when using `make -j4`. With the default `make`, where only one logical core is used at a time, MuQSS likely saved some time through its more simple functionalities. It has no overhead from CFS's timeslice calculation or load balancing, but rather gives the task its 6 ms timeslice, lets it run, puts it back in the runqueue, and repeats until the task is done.

Running the turn-around time test using `make -j4`, the two schedulers seem to be equally proficient, despite MuQSS being faster in the default single-core case. The lower latency of CFS likely makes up for the lost overhead from load balancing and timeslice, when running multiple jobs in parallel.

6.2 Conclusion

In the test environment used for this thesis, the current default Linux scheduler, CFS, seemingly outperforms Kolivas's MuQSS in most of the tested situations. MuQSS only managed to do better in one part of the turn-around time test, compiling on a single core. Thus, the experiments do not consistently support Kolivas's claim that MuQSS provides a more interactive and responsive desktop experience. In the way of responsiveness, i.e. latency and turn-around time, MuQSS's performance is not excessively inferior to CFS's, while in terms of interactivity, it manages very poorly in comparison.

6.3 Limitations and Future Work

As has been mentioned, this test was only done on a single desktop PC, with only 4 available logical cores. It is not unreasonable to assume that there might exist hardware combinations in which MuQSS outperforms CFS in more aspects. As MuQSS's performance using default `make` was the one thing it excelled at, MuQSS could possibly be useful in cases where less logical cores are available, such as older processors or smaller devices. In short, we cannot dismiss MuQSS as a viable scheduler without more work and testing, in different environments and possibly different configurations.

References

[1] Blender-Foundation, *Cosmos laundromat: First cycle.* https://www.youtube.com/watch?v=Y-rmzh0PI3c, 2015. CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[2] C. M. Burnett, *Red-black tree example.* https://commons.wikimedia.org/wiki/File:Red-black_tree_example.svg, 2006. CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[3] M. Fleming, *A survey of scheduler benchmarks.* https://lwn.net/Articles/725238/, 2017. (Accessed: 2021-06-14).

[4] T. Groves, J. Knockel, and E. Schulte, *BFS vs CFS - Scheduler Comparison.* https://www.cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf, 2009. University of New Mexico.

[5] N. Hussein, *The MuQSS CPU scheduler.* https://lwn.net/Articles/720227/, 2017. (Accessed: 2021-06-14).

[6] N. Ishkov, *A complete guide to Linux process scheduling*, Master's thesis, University of Tampere, 2015. https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf?\sequence=1&isAllowed=y.

[7] M. Kalin, *CFS: Completely fair process scheduling in Linux*, Opensource.com, (2019). https://opensource.com/article/19/2/fair-scheduling-linux (Accessed: 2021-06-14).

[8] C. Kolivas, *MuQSS - The Multiple Queue Skiplist Scheduler by Con Kolivas.* http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt, 2016. (Accessed: 2021-06-14).

[9] M. Larabel, *BFS Scheduler Benchmarks*, Phoronix, (2009). https://www.phoronix.com/scan.php?page=article&item=bfs_scheduler_benchmarks (Accessed: 2021-06-14).

[10] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, *The Linux scheduler: a decade of wasted cores*, EuroSys '16: Proceedings of the Eleventh European Conference on Computer Systems, 1 (2016), pp. 1–16. https://doi.org/10.1145/2901318.2901326.

[11] W. Muła, *Skip list.* https://commons.wikimedia.org/wiki/File:Skip_list.svg, 2013. CC0 1.0 Public Domain.

[12] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, *Towards achieving fairness in the Linux scheduler*, ACM SIGOPS Operating Systems Review, 42 (2008), pp. 34–43. https://doi.org/10.1145/1400097.1400102.