

Scheduling algorithms and scheduling policies in Linux, The Completely Fair Scheduler and its implementation

Fall 2019 CSCI-460
Montana State University

Ellen Marie Andersen
Benjamin Cathelineau
Olexandr Matveyev

1. Introduction	3
1.1. Brief overview of covered topics	3
2. Brief history of CFS	4
2.1. Reasons to change the scheduler	5
2.2. The previous scheduler: O(1)	5
2.2.1. Shortcomings of the previous scheduler	5
3. How CFS works : General presentation	6
3.1. Data structure of CFS and Fair share Scheduling	6
3.2. Fair share Scheduling: what is it?	6
3.3. The virtual runtime	8
3.4. Implementation of priorities	10
4. Other Policies and scheduling in Linux	10
4.1. A word on other scheduling algorithm and scheduling policies	10
4.1.1. The SCHED_OTHER policy	10
4.1.2. The SCHED_FIFO policy	10
4.1.3. The SCHED_RR policy	11
4.1.4. The SCHED_DEADLINE policy	11
4.1.5. The SCHED_BATCH policy	11
4.1.6. The SCHED_IDLE policy	11
4.2. The impact test : How useful are the policies ?	11
4.2.1. The stress command	11
4.2.2. The time command	11
4.2.3. The cpupower program	12
4.2.4. The test	12
4.2.4.1. The shell Script	12
4.2.4.2. The C program	15
4.2.4.3. The results (Plot)	15
4.2.4.3.1. Result for SCHED_OTHER	15
4.2.4.3.2. 4.2.4.3.2.Result for sched_BATCH	16
4.2.4.3.3. Result for sched_RR	17
4.2.4.3.4. Result for sched_FIFO	17
4.2.4.4. Result interpretation	18
5. Code example of CFS	18
5.1. CFS code and data-structure	18
5.2. Algorithm of CFS	19
6. The user side : Configure scheduler in Linux	19
6.1. Niceness	19
6.1.1. nice	19
6.1.2. renice	20
6.1.3. htop	21

6.2. Real time attribute	22
6.2.1. chrt command	22
6.2.2. Find most common policies with Shell script with chrt command	23
7. Performance of O(1) and CFS	24
7.1. Brief comparison of O(1) and CFS in fairness context	24
8. References	26

1. Introduction

1.1. Brief overview of covered topics

In our technical report we cover a brief history of previous Linux scheduler O(1) and the Completely Fair Scheduler (CFS), and the reason for replacing Linux O(1) scheduler onto CFS (Completely Fair Scheduler). After brief history introduction we will cover CFS in more detail. To be more specific we are going to cover how CFS works, what data-structure is used in CFS to achieve $O(n * \log n)$ time complexity. Also, we cover such topics in CFS context as what is a fair share scheduling, the virtual runtime of CFS, and implementation of priorities in CFS.

In order to have a deeper understanding of how schedulers work, we cover topic as policies in Linux. To be more specific we will talk about other scheduling algorithms and scheduling policies in Linux to make contrast with CFS. This topic will include information about following policies SCHED_OTHER, SCHED_FIFO, SCHED_RR, SCHED_DEADLINE, SCHED_BATCH, SCHED_IDLE. To conclude the policy section, we test the aforementioned policies with a bash script and a sample C program. In addition, we provide some graphs generated with libreOffice Calc to support our findings about the policies.

In this report we cover the information about scheduling configurations on the user side in Linux OS. To be more specific we cover process priority customizations with the following Linux system commands such as `nice`, `renice`, and `htop`. Also, we provide some examples of how to use priority customization. In addition to priority customizations we cover how we find the most common policies with a shell script using the `chrt` command which will conclude the scheduling configurations section.

To conclude our technical report we provide interesting graphs which shows comparison of O(1) and CFS in fairness performance.

2. Brief history of CFS

The Completely Fair Scheduler was introduced in 2007, and it was merged into kernel-2.6.23 [\[6\]](#). The first shift from $O(1)$ happened by Con Kolivas' development, with his Rotating Staircase Deadline Scheduler (RSDL). Later, Ingo Molnar developed the CFS scheduler, and he incorporated some of the ideas from Con Kolivas' RSDL scheduler to CFS. Molnar had also developed the previously used $O(1)$ scheduler [\[5\]](#).

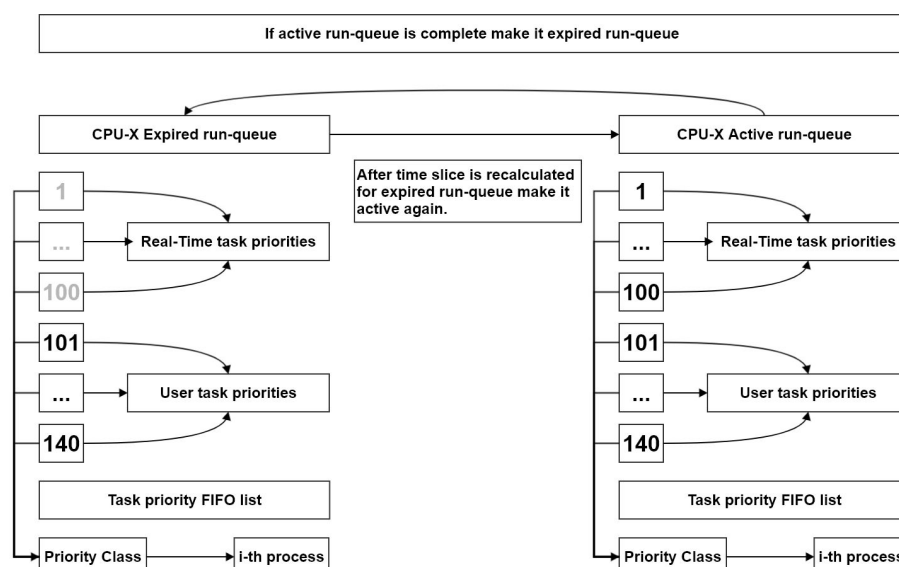
2.1. Reasons to change the scheduler

It turned out that the $O(1)$ scheduler was very difficult to maintain due to the large amount of code and algorithm complexity ([\[3\]](#), page: 462-463). The performance of interactivity with $O(1)$ was poor, and it had low throughput for background jobs [\[2\]](#). Interactive processes require short CPU bursts, and an example of an interactive process is a text editor. The CFS algorithm provides and maintains balance by providing a fair amount of processor time to a task relative to other tasks [\[5\]](#). This is explained more in depth below in section 2.

2.2. The previous scheduler: $O(1)$

The $O(1)$ scheduler is a constant time type of scheduler and it replaced the $O(n)$ scheduler. It was incorporated into Linux starting from kernel version 2.6.0 and till 2.6.22 version. In 2007 it was replaced by the Completely Fair Scheduler [\[1\]](#).

For the $O(1)$ scheduling algorithm, in order to achieve constant scheduling time it must use active and expired arrays of processes. In this algorithm, each process gets a fixed time quantum. After time quantum expires for a task in the active array it will be placed into the expired array, which means array switch will take place. Due to that fact that arrays are accessed only via a pointer, it makes array switching very fast $O(1)$. When the algorithm performs task switching it makes the active array the new empty expired array, while the expired array becomes the active array [\[1\]](#).



Real-Time Priority tasks from 1 to 100 and Normal Priority tasks from 101 to 140.

Source: [\[4\]](#)

2.2.1. Shortcomings of the previous scheduler

The design goals of the Linux kernel scheduler version 2.6 O(1) is to achieve fairness and boost interactivity performance, but because O(1) did try to achieve fairness, it actually resulted in decreasing interactive performance [12]. The main reason to replace the O(1) scheduler was the large mass of code needed to calculate heuristics, to mark a task as interactive or non-interactive which makes it fundamentally difficult to increase interactivity performance [5].

3. How CFS works : General presentation

3.1. Data structure of CFS and Fair share Scheduling

Red-Black Tree:

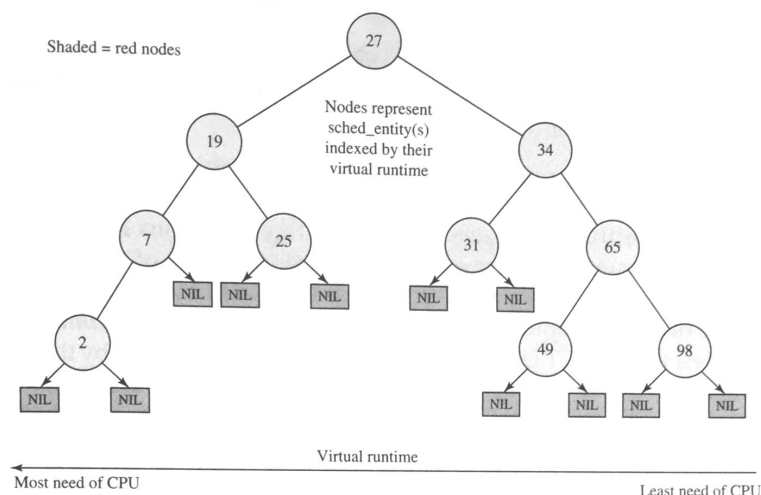


Figure 10.11 Example of Red Black Tree for CFS

Source: ([3]: pg 463)

CFS uses a Red-Black tree as its data structure. This tree represents processes that are runnable and that are to be chosen to run. The leftmost task is always picked by the scheduler to run next. The Red-Black Tree is ordered by vruntime where processes on the right side have higher vruntime than the processes placed on the left side. vruntime is explained in [section 3.3](#).

3.2. Fair share Scheduling: what is it?

Before looking at the CFS algorithm as a whole, we want to introduce a few concepts that make up the algorithm. Fair scheduling is the concept of processes sharing the processor in a fair way. As we will see, the term “fair” is defined and based on the algorithm’s implementation since other factors play into what makes the equation fair for each process.

Also, the concept of something being “fair” is subjective, so it depends on the implementation.

Let’s say we have 4 processes with the corresponding running times:

Process	Required Running time
Process A	3ms
Process B	2ms
Process C	4ms
Process D	3ms

Source: [\[8\]](#)

How does the concept of Fair share scheduling work for these processes?

If these four processes are runnable and ready, and there are no other processes interfering with them, each process will get assigned a fair share of time to run on the CPU. Therefore, the processes will alternate running on the CPU in fair time quantas while none of them are finished. If we simply look at time as a factor in this fair share example, and nothing else, we can demonstrate fair share by the following:

This is shown in the example by calculating the following: N processes and $(100/N)\%$ of CPU time. At time 0, N is 4 and each of these four processes will get a fair share of the processor, as shown in figure below:

	2ms quanta, N = 4	Remaining running time	2ms quanta, N =4	Remaining running time	2ms quanta, N = 4	Remaining running time
Process A	0.5ms	2.5ms	0.5ms	2ms	0.5ms	1.5ms
Process B	0.5ms	1.5ms	0.5ms	1ms	0.5ms	0.5ms
Process C	0.5ms	3.5ms	0.5ms	3ms	0.5ms	2.5ms
Process D	0.5ms	2.5ms	0.5ms	2ms	0.5ms	1.5ms

Source: [\[8\]](#)

Here we observe that each process gets a 25% share of the CPU. In the example below, processes that are finished running are marked as green for remaining running time. Since this will decrease N, each process left will gain a bigger share than before of the CPU.

Continuing below:

	2ms quanta, N = 4	Remaining running time	2ms quanta, N = 3	Remaining running time	2ms quanta, N = 3	Remaining running time
Process A	0.5ms	1ms	0.6667ms	0.3333ms	0.6667ms	0ms
Process B	0.5ms	0ms	n/a	n/a	n/a	n/a
Process C	0.5ms	2ms	0.6667ms	1.3333ms	0.6667ms	0.6667ms
Process D	0.5ms	1ms	0.6667ms	0.3333ms	0.6667ms	0ms

Source: [\[8\]](#)

Continuing below:

	2ms quanta, N = 1	Remaining running time
Process A	n/a	n/a
Process B	n/a	n/a
Process C	2ms	0ms
Process D	n/a	n/a

Source: [\[8\]](#)

In this example, the fair share started off by each of the four processes getting an equal share of the processor. This is not typically fair however, and depends on the implementation and if other factors are taken into consideration, and not just looking at time. How does this differ if one of the processes have higher priority than another? Or if a process spawns many processes related to itself that are placed on the leftmost side of the red-black tree which means these are to be chosen to run next, to capture more CPU resources? This is described in more detail below, in [section 3.3](#), with virtual runtime and [section 3.4](#) the concept of grouping scheduling [\[8\]](#).

3.3. The virtual runtime

The CFS algorithm is implemented in a way such that the algorithm chooses the next process to run based on the process with the lowest virtual runtime(vruntime). Virtual runtime is calculated by t (time spent running) multiplied with the weight of the process' niceness/priority (explained below). The vruntime increases as the process' time spent running increases. Runnable processes are represented by a red-black tree as shown in the data structure displayed above. Processes are inserted into this tree based on their vruntime value with the processes with the least vruntime on the left(smaller value) side of the tree. Therefore, we see that processes on the right side of the tree have higher vruntime than the ones on the left side of the tree, which can (again depending on implementation of different factors) mean that these processes on the right side of the tree have already spent more

time executing than the processes on the left side of the tree. This data structure is efficient in the sense that search, delete and inserts have complexity of $O(\log n)$ [29], [8].

The virtual runtime is simply incremented differently according to the priority of the process. Processes with low priority number get their virtual runtime increased **less**. Process with high priority number gets their virtual runtime increased **more**.

3.4. Our view on I/O bound processes vs Processor bound processes in light of CFS:

As we know from studying different scheduling algorithms, some tend to favor CPU bound processes and I/O bound processes may then risk starvation. As vruntime is affected by different factors such as time spent running, niceness or priority, these may favor both processes depending on their values. However, let's have a look at vruntime when we ignore the other factors except time spent running. Some I/O processes may spend less time executing before being blocked. This may be caused due to the process awaits user input from keyboard or mouse before being able to continue running. At that time, the process is not running and another process is running. Let's say the process that is now running is a processor bound process. This process can execute for longer CPU bursts while the I/O bound process is blocked awaiting the event. This will increase the CPU bound process' vruntime more than the I/O bound process' vruntime, for time running, since the CPU bound process got to execute longer (if we ignore priority) [6].

When these processes are runnable and placed in the red-black tree, the I/O bound process is then favored when it comes to vruntime and strictly looking at time spent executing. Since the I/O bound process is likely to have a smaller vruntime than the CPU bound process in this scenario, the I/O bound process is paced accordingly in the red-black tree, and is likely running before the CPU bound process (as long as it is still not blocked awaiting an event). This demonstration shows the fairness concept in the CFS implementation when it comes to how the time aspect influences the process likeliness to run next or in the future among all the other processes. For the I/O bound process in this scenario, we see that it is not starved since it is likely running before the other process. For I/O bound processes that are awaiting some event, ie sleeping, since it needs something, it would not be considered fair if the time it spent waiting (let's say on your friend who types extremely slow) would account for time spent running and increasing vruntime. Therefore, through the concept of "sleeper fairness", CFS does not take into account the time a process spends sleeping, so the vruntime is kept at a minimum which allows the process to be more likely to run next. [6].

Continuing on our example above, what if the CPU bound process was behaving in a way that it forked many new processes just to take advantage of the CPU? Would it "take over" the CPU for some time? The answer is no, due to auto grouping.

Autogrouping, or grouping scheduling, is a feature available in CFS that collects related processes into one group and essentially treats this group as one process. This is fair in the sense that any process would not be able to take over the CPU by spawning new processes

which would have the least runtime and be chosen by the scheduler to run next [9], [13], page 464). As we see, the CFS algorithm maintains fairness in this aspect as well.

Through our research, we discovered that one can enable and disable the grouping scheduling features in the scheduling algorithm in a `/proc/` file. Below shows that when 0 is in the file, autogrouping is disabled and when 1 is in the file, autogrouping is enabled [19].

We tested this on our computers. Here is an example of the autogrouping (grouping scheduling) disabled and enabled on our computers.

```
0
/proc/sys/kernel/sched_autogroup_enabled (END)

1
/proc/sys/kernel/sched_autogroup_enabled (END)
```

3.5. Implementation of priorities

Two types of priorities in CFS exist such as static and dynamic priority, which is also known as niceness. The `sched_OTHER` and `sched_BATCH` policies use dynamic priority, while `sched_rr`, `sched_fifo` use static priority. As the name implies, the static priority never changes and if a task that has a higher priority is ready to run, then a lower priority task will be preempted. However with dynamic priority it's different. The niceness is used to compute the virtual runtime used in CFS, which is explained [in section 3.3 of this report](#). Also, user can customize priorities and it is explained [in section 6 of this report](#).

4. Other Policies and scheduling in Linux

4.1. A word on other scheduling algorithm and scheduling policies

CFS is not the only scheduling algorithm used in the Linux Kernel. There are also simpler schedulers used for certain processes (mainly real-time processes).

In fact there are 6 different scheduling policies available on Linux as shown here:

```
benjamin@debian-benjamin-laptop:~$ chrt --max
SCHED_OTHER min/max priority : 0/0
SCHED_FIFO min/max priority  : 1/99
SCHED_RR min/max priority    : 1/99
SCHED_BATCH min/max priority  : 0/0
SCHED_IDLE min/max priority   : 0/0
SCHED_DEADLINE min/max priority : 0/0
```

Only the `SCHED_OTHER` policy uses CFS. The other policies use different scheduling algorithms. As shown later in this report, the policies that are mainly used on a desktop system is `SCHED_OTHER`, so most processes use CFS [14].

Later in this report, we explain how to manipulate these scheduling policies. Here we explain what each of them do, and also what their real life impacts are.

4.1.1. The SCHED_OTHER policy

The SCHED_OTHER policy uses CFS, it's detailed earlier in this report.

4.1.2. The SCHED_FIFO policy

This has been studied in class and it works exactly like that, except that there are several queues; one for each priority [\[14\]](#).

4.1.3. The SCHED_RR policy

This has been studied in class and it works exactly like that, except that there are several queues; one for each priority [\[14\]](#).

4.1.4. The SCHED_DEADLINE policy

sched_DEADLINE is an implementation of the fairly simple, early deadline first, algorithm [\[14\]](#).

4.1.5. The SCHED_BATCH policy

This policy is working very similarly to sched_OTHER, except that there is a small penalty in wakeup behavior. This is useful for non-interactive work [\[14\]](#).

4.1.6. The SCHED_IDLE policy

This policy is reserved for the lowest priority processes such as tracker in the gnome shell desktop environment [\[36\]](#). This policy is very useful to run extremely CPU intensive tasks while maintaining a very good degree of interactivity [\[14\]](#).

4.2. The impact test : How useful are the policies ?

To test the impact of the scheduling policies we wrote a shell script to load a system completely with "stress", and then run a process at the same time, with varying policies or niceness (with the sched_OTHER policy), to see how well it performs. For that we need the stress, the time and the cpupower programs [\[28\]](#).

4.2.1. The stress command

The stress command simply loads the system [\[38\]](#). In that case we were using cpu load (spinning on `sqrt()`), but you can also load IO (spinning on `sync()`), virtual memory (spinning on `malloc()/free()`) and others. It is a very useful tool to see how your system will behave under heavy load. A stress-ng program also exists but it's much more complex. The "normal" stress program seemed enough for this usage [\[32\]](#).

4.2.2. The time command

The time command is very simple to understand. It measures the time that a command / program took to execute.

It gives 3 results:

“real” : which is the time elapsed, in the real world, as measured by the system clock, between the moment the user started the program, and the moment the program finished, or was killed

“user” : which is the amount of time the process really spent on the cpu executing, in user space

“sys”: which is the amount of time the process really spent on the cpu executing in kernel space.

There are several versions of the time command, one can be downloaded as a separate package [\[31\]](#), one is built into bash. We used the one which is built into bash [\[30\]](#).

```
benjamin@debian-benjamin-laptop:~$ type time
time is a shell keyword
```

The type command, which is also built into bash, allows us to see that we are using a feature of bash and not an external program [\[30\]](#).

4.2.3. The cpupower program

The cpupower program is needed because the frequency of processors on moderns systems change over time as it is not fixed [\[34\]](#). With a lower frequency, a deterministic program will take longer to execute (even if it doesn't share the processor). This could impact our test as the frequency will go lower as the test goes on and the temperature of the processor rise.

The cpupower program is part of the linux kernel and allows administrator to fix the frequency of the processor of a given system [\[35\]](#).

It is simply used this way :

```
sudo cpupower -c all frequency-set -u 1GHz
```

Source: [\[15\]](#)

Here we change the maximum frequency of all core to 1GHz.

4.2.4. The test

The sum of user + sys “time” command returns should not vary much if at all (only measured errors should occur), for a given deterministic program. Indeed, a given deterministic process always spend the same amount of time on the processor. The only thing that should vary is the “real” return, depending on how the scheduling was done for the process. Another thing to note is that the “sys” return is negligible or null for many programs as they do not need to perform system call or have privileged access [\[17\]](#).

The core idea of this test is to compare the “real” time of execution of a process against its “user” time. This can be done for any process, but most processes are interactive and do not have a determinist execution time, they wait for the user to close them (such as a web browser). For the sake of simplicity, we decided to “develop” a simple C program that is deterministic, does not do any system call or any privileged function. This way we only need to compare the “user” return against the “real” return [\[17\]](#).

If we get a real over user of 1 it means the process as been prioritised completely. If we get a big real over user however it means that the process was not prioritized at all, as it had to wait a lot.

The shell script and the sample C program are available on github [\[39\]](#).

4.2.4.1. The shell Script

We wrote a shell script to conduct our test. It is mostly original work except how to grep the output of the time command [\[33\]](#),[\[15\]](#). Part of the script may be commented or uncommented to test the different policies.

```
#!/bin/bash
frequency=1.7
NormalFrequency=2.7

resultfile="result_batch_other"

time_to_s(){
    executiontime=$1
    calcstring="$(echo $executiontime | grep -Eo '[0-9]{1,3}m' | grep -Eo '[0-9]')"
    calc=' * 60' # one minute is 60 seconds
    calcstring=$calcstring$calc
    seconds_standart_executiontime="$(bc <<< $calcstring) + $(echo $executiontime | grep -Eo '[0-9]{1,2}.[0-9]{3}')"
    seconds_standart_executiontime=$(bc <<< $seconds_standart_executiontime)
    echo $seconds_standart_executiontime
}

sudo cpupower frequency-set --max $frequency\GHz > /dev/null

timereturn=$( { time ./increment 999999999 >/dev/null; } |& grep -E real | grep -Eo "[0-9]{1}m[0-9]{1,2}.[0-9]{3}")
```

```

seconds_standart_executiontime=$(time_to_s \''$timereturn\'')
echo $seconds_standart_executiontime
echo ""> $resultfile

sudo echo
stress -c 4 & #loading the system massively with stresses (4 because I
have a 4 core processor)
for pid in $(pidof stress)
do
sudo chrt --idle -p 0 $pid
# :
done

# # RR loop
# for staticpriority in $(seq 1 99);
# do
#   commandrr='sudo chrt --rr -p '$staticpriority' $BASHPID
#   { time ./increment 999999999 >/dev/null; } |& grep -E real | grep
-Eo "[0-9]{1}m[0-9]{1,2}.[0-9]{3}"
#   result=$(bash <<< $commandrr) #new bash session to avoid setting
realtime stresses which would block the system
#   second=$(time_to_s \''$result\'')
#   ratio=$(bc -l <<< "$second / $seconds_standart_executiontime")
#   echo $ratio >> $resultfile
# done

# # FIFO loop
# for staticpriority in $(seq 1 99);
# do
#   commandfifo='sudo chrt --fifo -p '$staticpriority' $BASHPID
#   { time ./increment 999999999 >/dev/null; } |& grep -E real | grep
-Eo "[0-9]{1}m[0-9]{1,2}.[0-9]{3}"
#   result=$(bash <<< $commandfifo) #new bash session to avoid setting
realtime stresses which would block the system
#   second=$(time_to_s \''$result\'')
#   ratio=$(bc -l <<< "$second / $seconds_standart_executiontime")
#   echo $ratio >> $resultfile
# done

# # Other LOOP
# for niceness in `seq -19 20`;
# do
#   commandother='sudo chrt --other -p 0 $BASHPID

```

```

#  sudo renice --priority '$niceness' --pid $BASHPID > /dev/null
#  { time ./increment 999999999 >/dev/null; } |& grep -E real | grep
-Eo "[0-9]{1}m[0-9]{1,2}.[0-9]{3}"
#  result=$(bash <<< $commandother) #new bash session to avoid setting
realtime stresses which would block the system
#  second=$(time_to_s \'$result\')
#  ratio=$(bc -l <<< "$second / $seconds_standart_executiontime")
#  echo $ratio >> $resultfile
# done

# BATCH LOOP
for niceness in `seq -19 20`;
do
    commandother='sudo chrt --batch -p 0 $BASHPID
sudo renice --priority '$niceness' --pid $BASHPID > /dev/null
{ time ./increment 999999999 >/dev/null; } |& grep -E real | grep -Eo
"[0-9]{1}m[0-9]{1,2}.[0-9]{3}"
    result=$(bash <<< $commandother) #new bash session to avoid setting
realtime stresses which would block the system
    second=$(time_to_s \'$result\')
    ratio=$(bc -l <<< "$second / $seconds_standart_executiontime")
    echo $ratio >> $resultfile
done

killall stress # removing stresses so that your computer doesn't become
a airplane
sudo cpupower frequency-set --max $NormalFrequency\GHz > /dev/null

```

4.2.4.2. The C program

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    if(argc < 2){
        printf("One argument : number to increment\n");
        exit(128);
    }
    long double incrementmax=atoi(argv[1]);
    long double i = 0;
    while (i<incrementmax){
        // printf("%Lf",i);
        i = i+1;
    }
}

```

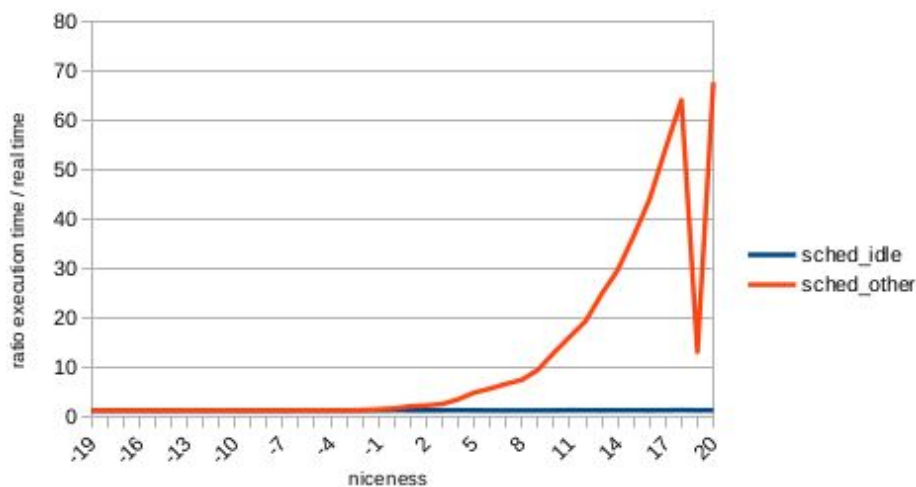
Source: [\[15\]](#)

4.2.4.3. The results (Plot)

Please note for the results below that our tools (laptop) were limited, so the tests have been run on a laptop that is also running other processes in the background. Therefore, the results are probably biased, but we wanted to include them since the graphs display interesting patterns.

4.2.4.3.1. Result for SCHED_OTHER

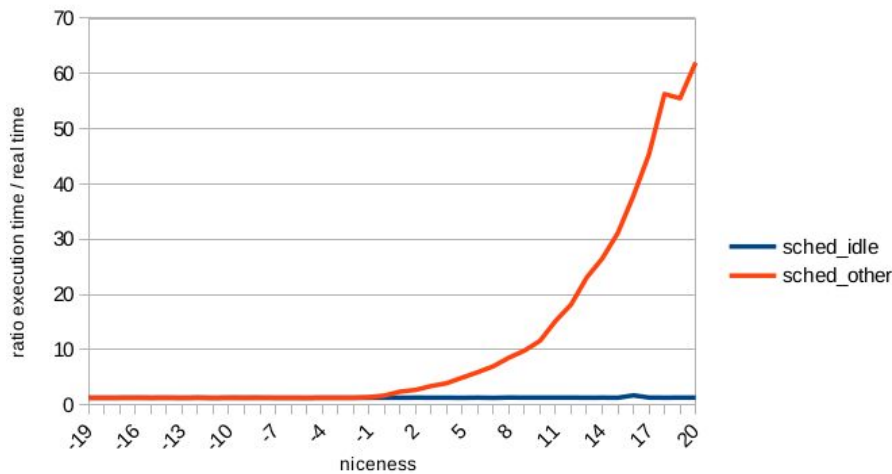
Here we see the sample C program is running with the sched_OTHER policy, with increasing niceness. The orange line is the result while the “stress” process is running with the sched_OTHER, the blue line is the result where the “stress” process is running as sched_IDLE.



Source: [\[15\]](#)

4.2.4.3.2. Result for sched_BATCH

Here we see the sample C program is running with the sched_BATCH policy, with increasing niceness. As the above graph, the orange line is the result while the “stress” process is running with the sched_OTHER, the blue line is the result where the “stress” process is running as sched_IDLE.

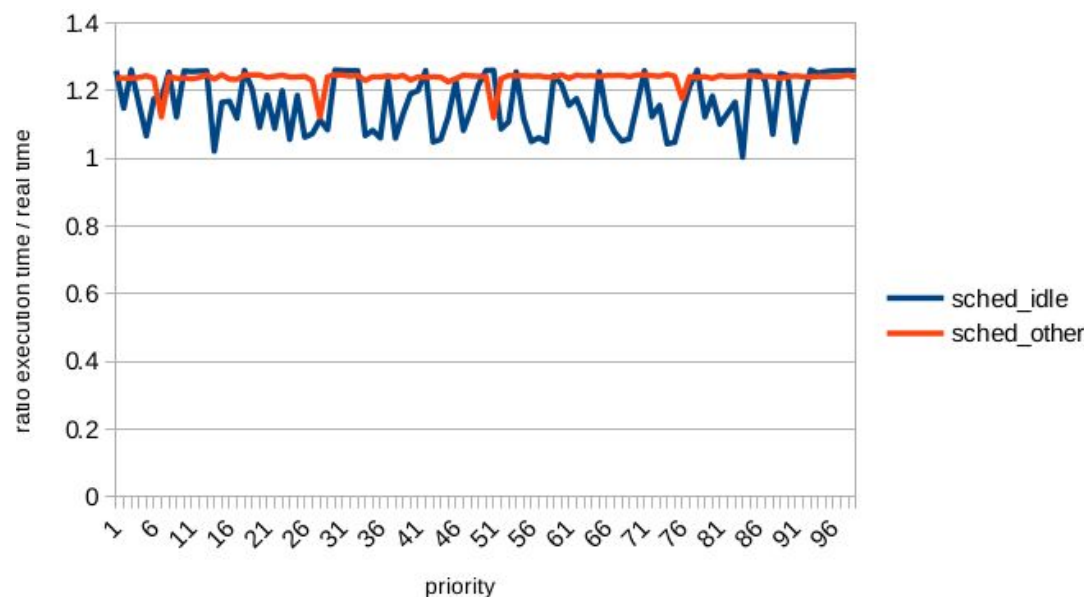


Source: [\[15\]](#)

4.2.4.3.3. Result for sched_RR

For the next two examples, please notice the scale is changed and shows a much smaller difference however we still see a trend.

Here we see the sample C program is running with the sched_RR policy, with increasing priority. As the above graph, the orange line is the result while the “stress” process is running with the sched_OTHER, the blue line is the result where the “stress” process is running as sched_IDLE.

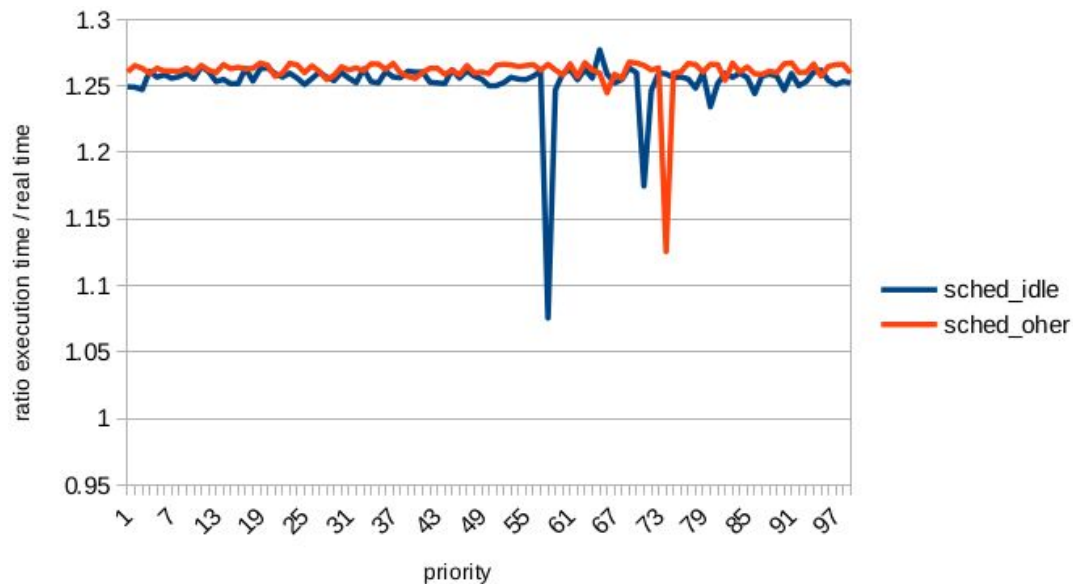


Source: [\[15\]](#)

4.2.4.3.4. Result for sched_FIFO

For the next example, please notice the scale is changed and shows a much smaller difference however we still see a trend.

Here we see the sample C program is running with the sched_FIFO policy, with increasing priority. As the above graph, the orange line is the result while the “stress” process is running with the sched_OTHER, the blue line is the result where the “stress” process is running as sched_IDLE.



Source: [\[15\]](#)

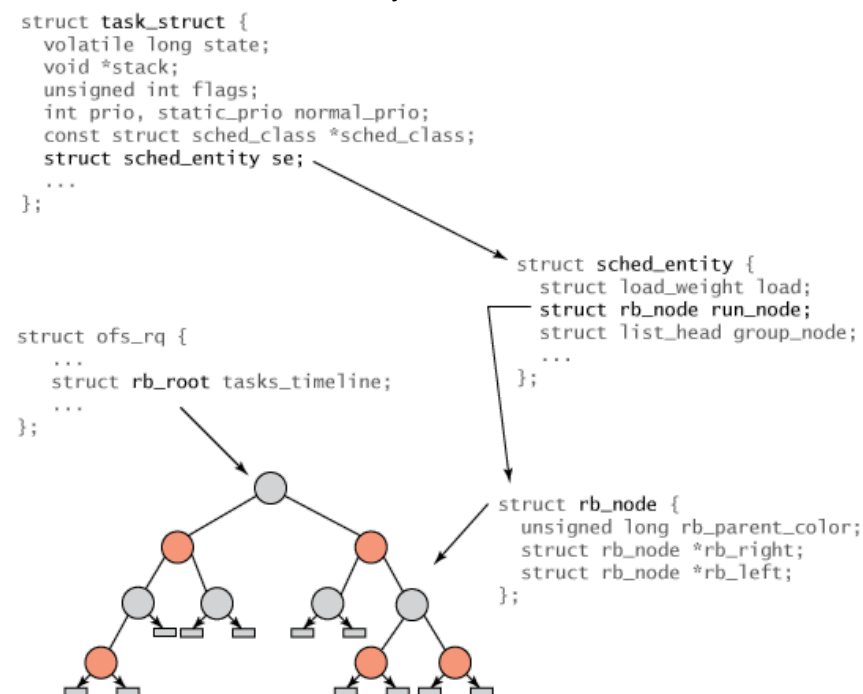
4.2.4.4. Result interpretation

We observed that the higher priority policies that are sched_RR and sched_FIFO give better results for the process. We also found that the niceness has a very big impact, just as claimed in the sched man page : “In the current implementation, each unit of difference in the nice values of two processes results in a factor of 1.25 in the degree to which the scheduler favors the higher priority process” [\[14\]](#).

5. Code example of CFS

5.1. CFS code and data-structure

Here is the structure hierarchy for tasks of the Red-Black Tree:



Source: [\[5\]](#)

5.2. Algorithm of CFS

Based on the CFS documentation we can see that CFS has about 10000 lines of code [\[10\]](#).

6. The user side : Configure scheduler in Linux

6.1. Niceness

On Linux, the user can, to a certain degree, customize the priority of a process. This customisation is made through the use of niceness. The niceness of a process can range from -20 to 19. The lower the niceness, the higher the priority.

Only positive (lower) priority number are accessible to non-root user. To set a negative priority you need to be root. However, like many things in Linux, this can be changed in the `/etc/security/limits.conf` file, to allow non privileged user to set negative niceness [\[7\]](#).

Still, it is important to keep in mind that these priority adjustments only concern non-real time process, as explained [in section 3.5 of this report](#)

6.1.1. nice

Nice is the standard utility to start a program with a different niceness than the default one (0). Here is an example of that:

```
benjamin@debian-benjamin-laptop:~$ nice -n 10 ls
Android  Dev      Downloads  mbox      opt        Public  Templates  virt-manager
Desktop  Documents  github     Music     Pictures   Steam   Videos
benjamin@debian-benjamin-laptop:~$
```

Example usage of nice

Source: [\[15\]](#)

6.1.2. renice

Renice is the utility that allows to change the niceness of an already started process.

```
benjamin@debian-benjamin-laptop: ~
File Edit View Search Terminal Help
benjamin@debian-benjamin-laptop:~$ ping debian.org -i 20 &
[1] 16010
benjamin@debian-benjamin-laptop:~$ PING debian.org (130.89.148.77) 56(84) bytes
of data.
64 bytes from klecker-misc.debian.org (130.89.148.77): icmp_seq=1 ttl=49 time=16
9 ms
64 bytes from klecker-misc.debian.org (130.89.148.77): icmp_seq=2 ttl=49 time=14
5 ms

benjamin@debian-benjamin-laptop:~$ man renice
benjamin@debian-benjamin-laptop:~$ sudo renice -n 19 16010
16010 (process ID) old priority 0, new priority 19
benjamin@debian-benjamin-laptop:~$
```

```
[2] 0: bash* "debian-benjamin-lapto" 16:53 10-Nov-19
```

Source: [\[15\]](#)

There is also a `ionice` program but it is outside the scope of this report because it's related to IO, so we have chosen not to include it.

6.1.3. htop

HTOP is a general purpose command line task manager, inspired by top but better [\[18\]](#).

1	[]	17.1%	Tasks: 155, 830 thr; 2 running
2	[]	9.3%	Load average: 1.10 0.94 0.93
3	[]	17.2%	Uptime: 1 day, 21:42:21
4	[]	8.6%	
Mem	[]	6.36G/7.69G	
Swp	[]	361M/7.46G	

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2766	benjamin	20	0	1263M	170M	27256	S	0.0	2.2	0:00.00	/usr/bin/gnome-software -
2803	benjamin	20	0	4150M	849M	348M	S	4.6	10.8	1h02:36	/usr/lib/firefox-esr/fire
2807	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.09	/usr/lib/firefox-esr/fire
2808	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.05	/usr/lib/firefox-esr/fire
2810	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	5:24.94	/usr/lib/firefox-esr/fire
2811	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:01.29	/usr/lib/firefox-esr/fire
2812	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:08.52	/usr/lib/firefox-esr/fire
2813	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:08.99	/usr/lib/firefox-esr/fire
2814	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:08.61	/usr/lib/firefox-esr/fire
2815	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:08.98	/usr/lib/firefox-esr/fire
2816	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:20.52	/usr/lib/firefox-esr/fire
2817	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.01	/usr/lib/firefox-esr/fire
2818	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	1:20.13	/usr/lib/firefox-esr/fire
2819	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:10.09	/usr/lib/firefox-esr/fire
2820	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:04.33	/usr/lib/firefox-esr/fire
2821	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:35.17	/usr/lib/firefox-esr/fire
2822	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:22.26	/usr/lib/firefox-esr/fire
2826	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:30.36	/usr/lib/firefox-esr/fire
2827	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.00	/usr/lib/firefox-esr/fire
2830	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.00	/usr/lib/firefox-esr/fire
2831	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.00	/usr/lib/firefox-esr/fire
2832	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	3:07.13	/usr/lib/firefox-esr/fire
2833	benjamin	20	0	4150M	849M	348M	S	0.7	10.8	19:01.11	/usr/lib/firefox-esr/fire
2834	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.13	/usr/lib/firefox-esr/fire
2835	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.07	/usr/lib/firefox-esr/fire
2840	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:00.00	/usr/lib/firefox-esr/fire
2841	benjamin	20	0	4150M	849M	348M	S	0.7	10.8	3:09.27	/usr/lib/firefox-esr/fire
2842	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:07.23	/usr/lib/firefox-esr/fire
2850	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:01.58	/usr/lib/firefox-esr/fire
2851	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:01.63	/usr/lib/firefox-esr/fire
2852	benjamin	20	0	4150M	849M	348M	S	0.0	10.8	0:01.61	/usr/lib/firefox-esr/fire
1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice +F9Kill F10Quit											

Source: [\[15\]](#)

The priority can be increased (f7 decrease niceness) or decreased (f8 increase niceness). This tool is easier to use than renice because it displays the name of the program.

6.2. Real time attribute

6.2.1. chrt command

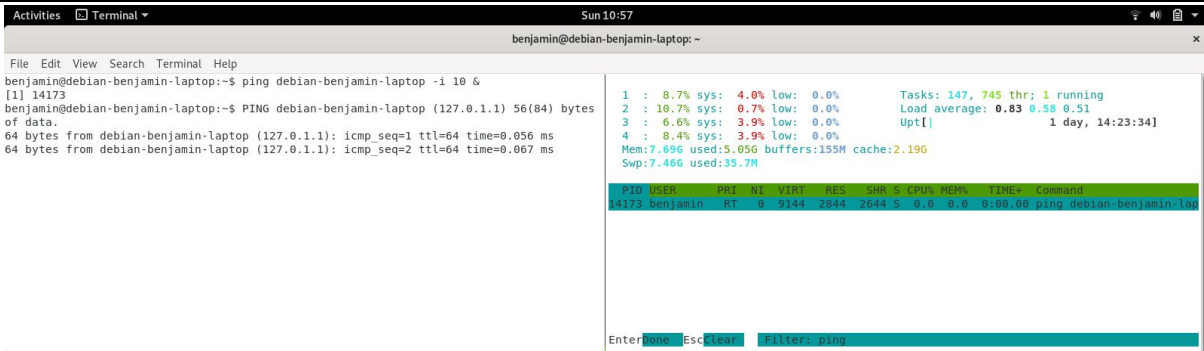
The default scheduling policies for processes is `SCHED_OTHER`, or `SCHED_NORMAL` (same thing). Processes under these policies are scheduled with CFS, and they use niceness.

However as for the other scheduling policies, such as `SCHED_RR` and `SCHED_FIFO`, they do not use CFS and niceness. Instead they use static priorities.

With `chrt` you can change the policy of your process, and it's static priority [\[14\]](#).

Here is an example usage doing a ping and changing static priority to 99(highest possible priority).

```
sudo chrt -p 99 10540
```



```
benjamin@debian-benjamin-laptop:~$ ping debian-benjamin-laptop -i 10 &
[1] 14173
benjamin@debian-benjamin-laptop:~$ PING debian-benjamin-laptop (127.0.1.1) 56(84) bytes of data:
64 bytes from debian-benjamin-laptop (127.0.1.1): icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from debian-benjamin-laptop (127.0.1.1): icmp_seq=2 ttl=64 time=0.067 ms

1 :  8.7% sys:  4.0% low:  0.0%    Tasks: 147, 745 thr: 1 running
2 : 10.7% sys:  0.7% low:  0.0%    Load average: 0.83 0.58 0.51
3 :  6.6% sys:  3.9% low:  0.0%    Uptime: 1 day, 14:23:34
4 :  8.4% sys:  3.9% low:  0.0%
Mem:7.69G used:5.05G buffers:155M cache:2.19G
Swp:7.46G used:35.7M

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 1173 benjamin   PT    0  9144 2844 2644 S   0.0  0.0  0:00.00 ping debian-benjamin-lap

benjamin@debian-benjamin-laptop:~$ sudo chrt -p 99 14173
benjamin@debian-benjamin-laptop:~$
```

Source: [\[15\]](#)

6.2.2. Find most common policies with Shell script with `chrt` command

This shell script is heavily inspired and slightly modified by [\[15\]](#),[\[13\]](#).

```
#!/bin/bash
ps -aux |grep [0-9]|awk '{print $2}' > test.txt
cat test.txt |while read line
do
chrt -p $line 2> /dev/null
done
```



```
benjamin@debian-benjamin-laptop:~/Documents/montana
CSCI460/project$ ./shellscrip |grep OTHER |wc
    218    1308   10833

benjamin@debian-benjamin-laptop:~/Documents/monta
CSCI460/project$ ./shellscrip |grep FIFO |wc
     12      72    572

benjamin@debian-benjamin-laptop:~/Documents/m
CSCI460/project$ ./shellscrip |grep IDLE|wc
      1       6     49
```

Source: [\[15\]](#)

We see that most processes are running as SCHED_OTHER on this system.
Some are running as SCHED_FIFO.
And only one is using SCHED_IDLE, which is the lowest priority policy.

7. Performance of O(1) and CFS

7.1. Brief comparison of O(1) and CFS in fairness context

In order to conclude our technical report for the Operation System class we are including some interesting information about performance comparison between O(1) and CFS. These graphs were produced by (Wong, C., Tan, I., Kumari, R., Lam, J., & Fun, W. in 2008) [\[12\]](#):

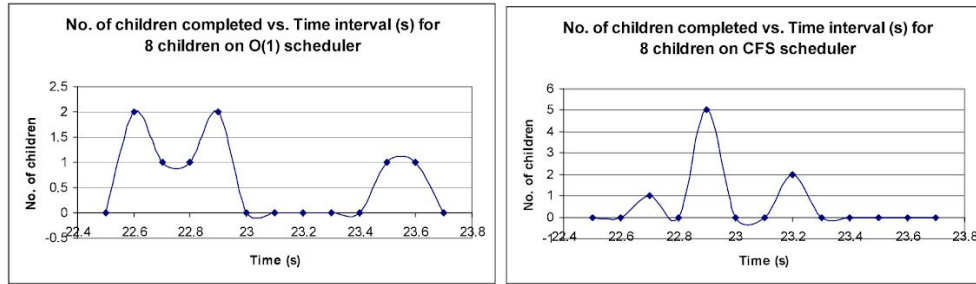


Figure 1 (a): Results of fairness measurement for 8 number of children on O(1) and CFS schedulers.

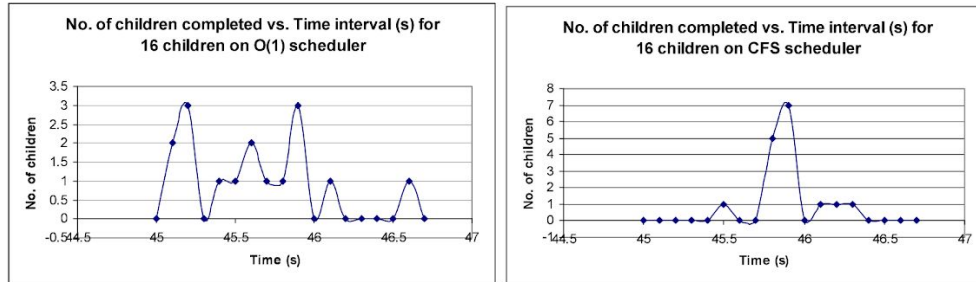


Figure 1 (b): Results of fairness measurement for 16 number of children on O(1) and CFS schedulers.

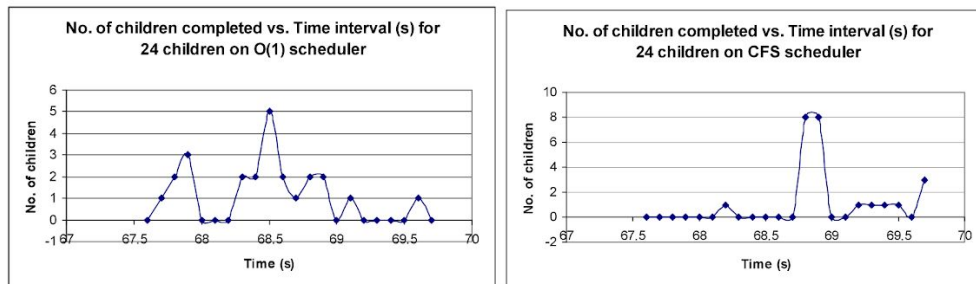


Figure 1 (c): Results of fairness measurement for 24 number of children on O(1) and CFS schedulers.

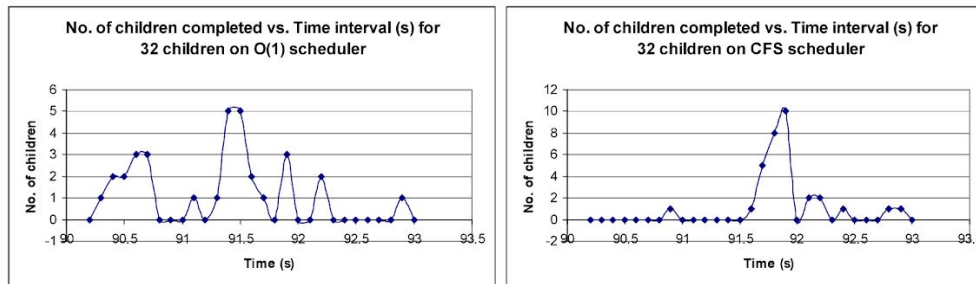


Figure 1 (d): Results of fairness measurement for 32 number of children on O(1) and CFS schedulers.

Source: ([12], page 6)

By observing the graphs above, we notice that there is a trend for O(1) completing tasks at a different rate as compared to the CFS ones. The standard deviation of task completion is likely higher for O(1) than in CFS for each task since CFS shows a tendency of completing the tasks at around the same time (which may be interpreted as more fair).

8. References

- [0]: Cover page 'penguin'.
Tux, originally drawn as raster image by Larry Ewing in 1996.
- [1]: O(1) scheduler. (2019, October 13).
Retrieved from [https://en.wikipedia.org/wiki/O\(1\)_scheduler](https://en.wikipedia.org/wiki/O(1)_scheduler).
- [2]: J. Jose, O. Sujisha, M. Giles and T. Bindima, "On the Fairness of Linux O(1) Scheduler," *2014 5th International Conference on Intelligent Systems, Modelling and Simulation*, Langkawi, 2014, pp. 668-674. doi: 10.1109/ISMS.2014.120
URL: <https://ieeexplore.ieee.org/document/7280991>.
- [3]: Stallings, W. (2018). *Operating systems: internals and design principles*. Harlow, Essex: Pearson.
- [4]: Jones, T. (2006, June 30). Inside the Linux scheduler. Retrieved November 10, 2019.
Retrieved from <https://www.ibm.com/developerworks/library/l-scheduler/index.html>.
- [5]: Jones, . (2009, December 15). Inside the Linux 2.6 Completely Fair Scheduler.
November 10, 2019,
Retrieved from <https://developer.ibm.com/tutorials/l-completely-fair-scheduler/>.
- [6]: Completely Fair Scheduler. (2019, October 1).
Retrieved from https://en.wikipedia.org/wiki/Completely_Fair_Scheduler.
- [7]: Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*. Hoboken, NJ: Wiley.
- [8]: Operating System #22 Completely Fair Scheduling (CFS).
Retrieved from <https://www.youtube.com/watch?v=scfDOof9pww>.
- [9]: Retrieved from https://www.eit.lth.se/fileadmin/eit/courses/eitf60/Rapporter/Ludwig_Hellgren_Winblad_4547_assignsubmission_file_ludwig.hellgren.winblad.pdf.
- [10]: Retrieved from <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>.
- [11]: Nice (Unix). (2019, May 7).
a: Retrieved from [https://en.wikipedia.org/wiki/Nice_\(Unix\)](https://en.wikipedia.org/wiki/Nice_(Unix)).
b: Retrieved from <https://askubuntu.com/questions/656771/process-niceness-vs-priority>.
- [12]: Wong, C., Tan, I., Kumari, R., Lam, J., & Fun, W. (2008). Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. *2008 International Symposium on Information Technology*. doi: 10.1109/itsim.2008.4631872.
- [13]: Retrieved from <https://unix.stackexchange.com/questions/407497/how-to-find-scheduling-policy-and-active-processes-priority>.
- [14]: Kerrisk, M. (2019, November 19). Linux Programmer's Manual.
Retrieved from <http://man7.org/linux/man-pages/man7/sched.7.html>.
- [15]: Cathelineau, B
- [16]: Andersen, E
- [17]: CPU Scheduling. (n.d.). Retrieved December 7, 2019, from
https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/6_CPU_Scheduling.html.
- [18]: Author of htop Hisham Muhammad, <hisham@gobolinux.org>.
Kerrisk, M. (2019, November 19). Linux Programmer's Manual.
Retrieved from <http://man7.org/linux/man-pages/man1/htop.1.html>.
- [19]: Kerrisk, M. (2019, November 19). Linux Programmer's Manual.
Retrieved from <http://man7.org/linux/man-pages/man7/sched.7.html>.

- [20]: Troan, E. W., & Johnson, M. K. (2005, July 8). Informat.
Retrieved from <http://www.informat.com/articles/article.aspx?p=397655&seqNum=6>.
- [21]: Answered by sumouli.choudhary on 2017-05-11 00:57:46
Retrieved from: <https://practice.geeksforgeeks.org/problems/which-type-of-scheduling-is-done-in-windows-10-os>
- [22]: Linux source code: kernel/sched/fair.c (v5.4.1). (n.d.).
Current linux version : CA10000 lines of code.
Retrieved from <https://elixir.bootlin.com/linux/latest/source/kernel/sched/fair.c>.
- [23]: Linux source code: kernel/sched_fair.c (v2.6.23). (n.d.).
First version introduced : 1200 lines of code.
Retrieved from https://elixir.bootlin.com/linux/v2.6.23/source/kernel/sched_fair.c.
- [24]: Linux source code: kernel/sched.c (v2.6.22.9). (n.d.).
O(1) scheduler : 7000 lines of code.
Retrieved from <https://elixir.bootlin.com/linux/v2.6.22.9/source/kernel/sched.c>.
- [25]: Linux Scheduler. (n.d.).
Retrieved from <https://www.kernel.org/doc/html/latest/scheduler/index.html>.
- [26]: Kernel Programming Guide. (2013, August 8).
Retrieved from <https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html>.
- [27]: The Linux Process Scheduler. (n.d.).
Retrieved from <http://www.informat.com/articles/printfriendly/101760>.
- [28]: Nikita Ishkov, A complete guide to Linux process scheduling. University of Tampere School of Information Sciences Computer Science M.Sc.Thesis Supervisor: Martti Juhola February 2015.
URL: <https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf>.
- [29]: Red-Black Tree: Set 1 (Introduction). (2019, November 27).
Retrieved from <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>.
- [30]: Bash Shell, Brian Fox and contributors. Retrieved from www.gnu.org/software/bash/.
- [31]: GNU time, Assaf Gordon. Retrieved from <https://www.gnu.org/software/time/>.
- [32]: Amos Waterland. Retrieved from <https://bazaar.launchpad.net/~ubuntu-branches/ubuntu/precise/stress/precise/view/head:/src/stress.c>.
- [33]: How to grep time command output,First answer.
Retrieved from <https://stackoverflow.com/a/17257787>.
- [34]: Dynamic frequency scaling. (2019, October 9).
Retrieved from https://en.wikipedia.org/wiki/Dynamic_frequency_scaling.
- [35]: Thomas Renninger, Len Brown. Retrieved from <https://www.kernel.org/>.
- [36]: Tracker. (n.d.). Retrieved from <https://wiki.gnome.org/Projects/Tracker>.
- [37]: Deadline Task Scheduling. (n.d.). Retrieved from <https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html>.
- [38]: Canonical. (n.d.). Retrieved from <https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html>.
- [39]: Cathelineau, B,
Retrieved from <https://github.com/olimar718/SchedulerPoliciesAndNicenessTest>.