**The contrast between Completely Fair Scheduling and O (1) scheduling. Which one would you prefer and why?**

Completely Fair Scheduling (CFS) replaced O (1) scheduler in Linux Kernel version 2.6.23. CFS was designed to give fair CPU resources to all the processes without sacrificing interactive natures of the processes. O (1) scheduler focuses on improving interactive nature on the processes and does so by giving bonuses to processes which have higher sleep time. Some researchers have shown that in real world responsiveness (interactivity) is more important than throughput for desktop users [[1]](#endnote-2)where O (1) scheduler tries to focus more. On the other hand, Fairness in terms of CPU bandwidth allocation is also an important design aspect I.e., each process gets a chance to advance forward equally [[2]](#endnote-3) where CFS tries to focus more. Now let us discuss O (1) and CFS in more detail.

**The O (1) Scheduler**

In O (1) scheduler there are 280 run queues for each processor. SMP affinity is also taken care. Out of these 280 run queues, 140 run queues belong to active process set and 140 belong to expired process set. Active set we place process which are yet to get their time quanta. In the expired set we place a process which have finished their time quanta. If the active set gets empty, then the process expired set and active set will switch entries. Now let us go deep into these 140 queues in each of these set. These 140 queues correspond to the priority of the processes. With 0 being the highest priority and 139 being the lowest. Scheduler will pick the highest priority run queue first. After the time quanta of a process is finished their dynamic priorities are reevaluated again. Higher priority process will get high time quanta. By default, a process starts at priority level 120. To go below (higher priority) that from user point of view we need “Sudo” privileges.

O (1) scheduler focuses more on increasing interactivity. To do so, it uses, “Interactivity=1/Sleep Time”. So, higher interactive process => More I/O Bound task. And lower interactive process => More CPU bound tasks. Based on interactivity scheduler supplies a bonus value in range [-5, +5] where –5 if the process never slept and +5 if the process reached SLEEP\_MAX\_TIME. Then dynamic priorities are calculated based on the formula,

Dynamic Priority = Static Priority – Bonus +5

Dynamic Priority = min (139, max (100, Dynamic Priority).

And based on that, time quanta for a particular process are calculated from this formula.

if Static Priority < 120: Time Quanta = (140 - Static Priority) \* 20

if static priority >= 120: Time Quanta = (140 - s Static Priority) \* 5

If a process is Interactive based on the formula: “bonus-5 >= (static priority/ 4) - 28” then we put it at the end of the active list on the same run queue after finishing the time quanta rather than the expired list. Exceptions for this is when a higher priority process is in the expired list and waiting for its turn or expired task has been waiting for more than STARVATION\_LIMIT unit time. To find the highest priority task a bit map is kept. Intel also supplied a special set of instructions for that.

At some point in time the interactive process will get more CPU bandwidth. And it might starve CPU bound intensive processes despite the fact of STARVATION\_LIMIT. The process which starts at low priority has an exceedingly rare chance to move to higher priority and hence all the process will not advance at an equal rate and CPU intensive processes may be left behind. Maintaining aging here is O(N) where N is the number of processes on a given processor.

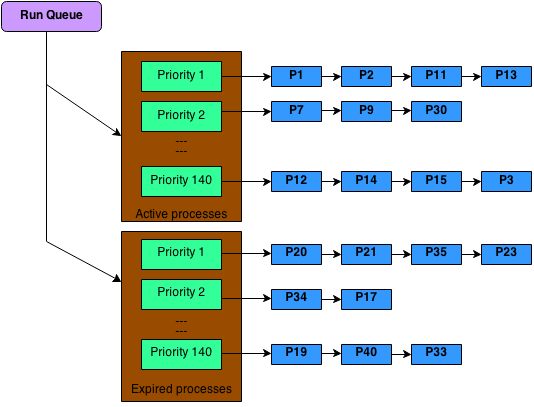


Fig. O (1) scheduler pictorial representation [[3]](#endnote-4).

**Completely Fair Scheduler**

The scheduler must schedule a lower priority process eventually so that they cannot starve for an exceptionally long time. For an interactive process context switch happens at a very rapid rate compared to the non-interactive process (CPU Bound process), so there must be a balance between interactive and non-interactive process since context switch is a costly operation.

CFS as the name suggests will try to divide the CPU bandwidth into equal parts to all the processes. It uses no heuristics. It is based on ideal fair scheduling where each process will get (100/N) % of CPU time. For every process CFS maintain a virtual runtime (vruntime) and if a process gets CPU bandwidth for ‘t’ time then vruntime of that process will be incremented by ‘t.’ So vruntime keeps on monotonically increasing. For all the processes a common red black tree is kept for a processor. The *implementation* is as follows:

The process represented by a structure “task\_struct.” However, scheduler does not use this structure completely instead uses its one attribute named “sched\_entity.” This holds the vruntime variable and pointer to the red black tree [[4]](#endnote-5).

After the minimum vruntime process is picked time quanta is calculated based on the system load as well as the next process vruntime. After context switch next minimum vruntime process is selected (leftmost node in red black tree). A pointer can be kept named min\_vruntime to store the next process with minimum vruntime. Modifying everything is O (log n) time slice the tree is balanced. The process tends to move from left side to right side of the red black tree after execution is performed.

To keep *Priority* two factors are used. One is “nice value” which is in the range of 20 to –19, default being 0. Lower nice value means higher priority. Second priority is the main priority. Ranging from 0 to 100. There is an attribute name “prio” in task\_struct which stores effective priority. Now the vruntime calculation changed into “vruntime+=t\*(prio).” So, a lower priority processes time move at a faster rate than of higher priority.

Since the I/O bound process will have less vruntime as they mostly get suspended waiting for I/O their vruntime will be smaller so by default they have less vruntime value compared to CPU bound processes. In this way interactive processes get high priority by design and CPU bounded processes get lower priority. If a new process is added it is added in red black tree with min\_vruntime. So, the response time will be extremely fast for any new process.

There are separate arrangements for diverse types of class scheduling process. Like real time process were scheduled into SCHED\_FIFO class. Similarly for SCHED\_RR and SCHED\_NORMAL for normal processes.

If a process creates a fork or thread, then too they are considered as a part of same group and treated as one unit. It will make sure that a process creating multiple forks does not capture complete resources of the CPU under Group Scheduling CFS. One can check scheduling related file in /usr/src/linux-source-5.4.0/linux-source-5.4.0/kernel/sched, after installing the linux-source, using the common sudo apt-get source linux-source, and extracting the files.

**Which one would I prefer and why?**

Linux Kernel Development is a large community consisting of many talented open-source developers and researchers they choose CFS and are still using some variation of CFS. So obviously CFS will be better than O (1), otherwise why should it be deployed into Linux Kernel which is powering so many systems worldwide. According to me I will choose CFS due to the following reasons:

1. The main issue with O (1) is the heuristic that is used to figure out interactiveness of the process. It tries to find interactivity based on sleep time. O (1) give bonus to process having higher sleep time while penalizing process with less sleep time. The calculation based on only one parameter is ambiguous and may lead to issues like non interactive behavior from an interactive process. For example. If a process has some CPUS bound instruction and then after it will interactive so at first few iterations the priority of the process will decrease, and it will be difficult to rise priority again.
2. CFS does not do any such heretical calculations and simply works on the fact that all processes should get equal CPU bandwidth. CFS tries to be ideal multitasking CPU. If a process suddenly gets interactive, then after a certain time its vruntime value is less compared to other processes and then it can get a chance to be interactive irrespective of its past behavior.
3. CFS can handle more elegantly I/O and CPU bound processes.
4. CFS perform better in fairness test and interactivity test [[5]](#endnote-6).
5. In CFS we give fairly divide slice to all the process and hence more fairness. And because of no heuristic it performs based on the current state of the system and the type of process.
6. If priority in O (1) scheduler decreases, it is extremely hard to increase the priority.
7. O (1) can perform better in interactions but perform poorly in case of fair CPU resource distribution where CFS ace (and not losing too much ground in interactivity).

1. B. Schneiderman, "Designing the User Interface: Strategies for Effective Human-Computer Interaction," 3rd Edition, Addison Wesley Longman, 1998. [↑](#endnote-ref-2)
2. D.P. Bovet, et. al., "Understanding the Linux Kernel,” 3rd Edition, O'Reilly Press, ISBN 0-596-00565-2, 2004. [↑](#endnote-ref-3)
3. https://www.algorithmsandme.com/scheduling-o1-and-completely-fair-scheduler-cfs/ [↑](#endnote-ref-4)
4. https://www.linuxjournal.com/node/10267 [↑](#endnote-ref-5)
5. Fairness and interactive performance of O (1) and CFS Linux kernel schedulers. C.S. Wong; I.K.T. Tan; R.D. Kumari; J.W. Lam; W. Fun [↑](#endnote-ref-6)