**SET 1**

**Q2: Contrast between Completely Fair Scheduling and O (1) scheduling. Which one would you prefer and why?**

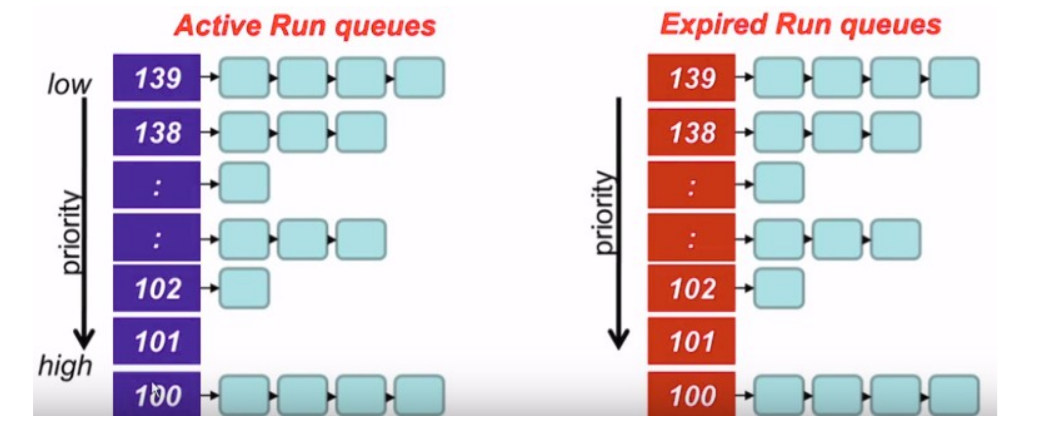
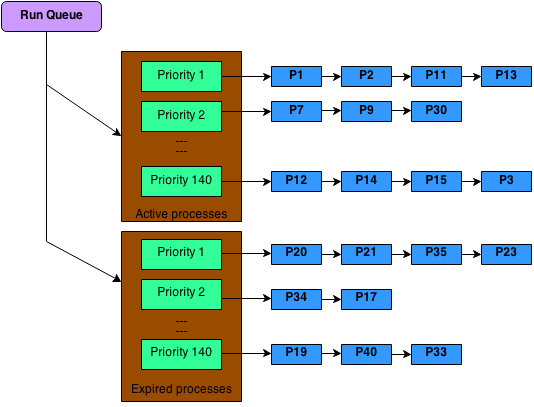
Ans: An O (1) scheduler is a kernel scheduling design which is able to schedule processes within a constant amount of time, irrespective of the number of processes that are present in the ready queue (running on the OS)[[1]](#endnote-1).For Linux 2.6 to 2.6.22, the O (1) scheduler was used. Before that in Linux 2.4, a scheduler with Linear (O(N)) complexity was used where N is the number of tasks currently in system in runnable state. Essentially It maintained a queue, whenever there was need of scheduling, it picks the best process to be scheduled next based on priority. So, it can obviously scale for a large number of processes present in the ready queue which was not possible with O (N) scheduler. So, O (1) scheduler is an improvement over previously used O(n) scheduler.

For schedulers there are basically 2 types of processes: one is Real time, and another is Normal Process. Real time processes are those with hard deadlines. They are given priorities ranging from 0 (highest real-time priority) to 99 (lowest real-time priority). On the other hand, Normal processes, which are those with soft deadlines, are given priority ranging from 100 (highest priority) to 139 (lowest priority). Normal processes are again two types: Interactive and Batch. Interactive processes are those which generally spend their time interacting with users. On the other hand, Batch processes do not require much user time. They can be scheduled to run in the background.

**O (1) Scheduling in Normal Process:**

It is like a multilevel feedback queue with some little variations. To enforce prioritization of tasks in the system, it consists of 40 priority levels from 100 to 139 (100- Highest, 139-lowest). The priority levels from 100 to 139 are called static priorities. Scheduler maintains two ready queues are used which keeps track of all running processes assigned for its associated CPU: an active run queue and an expired run queue. Processes which are present at the same priority level have the same priority. Scheduler selects the task starting from the highest priority level. For this, it maintains a bitmap of the run queue with non-zero if some entity is present otherwise zero. So, it looks for lowest numbered nonzero bit in the bitmap. Each process is given a fixed time quantum, after which it is preempted and moved to the expired run queue. When all the tasks in the active run queue are executed (i.e., run queue is empty), the pointers to the active array and expired array are toggled. So, now the active run queue becomes expired run queue and vice versa. This process continues.

Fig. 1.1. Run Queues in O (1) scheduler[[2]](#endnote-2)



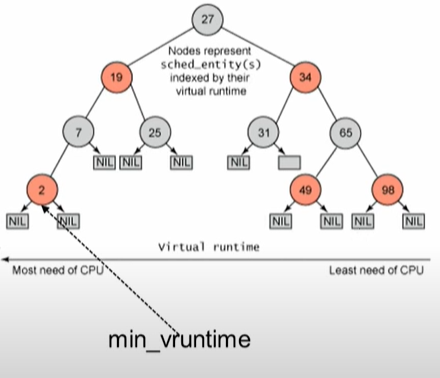
For differentiating between interactive and batch process, dynamic priority is also used which is assigned based on a formula which considers the average sleep time. After the process execution, it is placed in the expired run queue at some priority level which is determined by dynamic priority. Further heuristics are involved in calculating the time slice for ensuring that Interactive processes with high priority completes its burst without being preempted.

From Linux 2.6.23 onward **Completely Fair Scheduler** (CFS) is adopted. It was made by Ingo Molnar in place of the O (1) scheduler[[3]](#endnote-3). O (1) scheduler components such as run queues, heuristic for interactive processes, dynamic priority and priority-based time slices concepts are cut down to increase efficiency. This scheduler handles I/O bound process and CPU bound process efficiently.

The main idea behind it is to divide the processor time equally among processes by promoting the processes, which have got a small amount of time, to the processor, and those who got a bigger amount of time are resisted. This is done by using Virtual time. At every scheduling point, Virtual runtime is increased by the amount of time it has been executed with some weightage to priority. When context switch occurs, a process with minimum Virtual runtime is selected and is executed in the CPU. This process continues.

This implementation uses Red-Black Tree data structure to maintain runnable task. It is used because of its self-balancing nature and less time required for insertion and deletion. Each node represents a runnable process which are ordered according to their virtual runtime. All nodes which have got small time on processor are stored on left side of root, while those who have got larger time on processor are stored on right hand side. A pointer is used for pointing leftmost node having minimum time and is accessed whenever context switch occurs (O (1) time). After its execution its virtual runtime is calculated, and it is inserted into Red-Black Tree (O (logn)). When a new process is added, it begins with minimum Virtual runtime and so it gets to execute quickly.

Fig. 1.2 Red-Black Tree for CFS[[4]](#endnote-4)



Since CFS uses virtual runtime, so it does not need to do anything extra for I/O Bound and CPU Bound processes because I/O bound processes get higher priority as their CPU burst is low resulting in low Virtual Runtime. If the process spends a lot of its time sleeping, then its spent time value is low, and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running. Another interesting feature of CFS is that it uses the concept of group scheduling (introduced with the 2.6.24 kernel). Group scheduling is another way to bring fairness to scheduling. It is useful for the tasks that spawn many other tasks. CFS also brings the idea of introducing scheduling classes. Its main feature is that each task belongs to a scheduling class, which is a deciding factor that how a task will be scheduled. A scheduling class defines a common set of functions that define the behavior of the scheduler. Scheduler classes are linked together via linked lists.

**Preference**

I would prefer Completely Fair Scheduling over O (1) scheduling. O (1) scheduling algorithm makes Linux more scalable but the main issue with this algorithm is the complex heuristic that is used to select a task as interactive or non-interactive. All these are tedious tasks and hence are prone to miscalculations. Also, there is dependency between time slice and priority and priority and time slice values are non-uniform among the different priority ranges. Completely Fair Scheduling achieves tasks fairness[[5]](#endnote-5). By fairness in terms of CPU allocation among processes we mean that each task will be proportionally advancing forward when executed. There are many reasons behind the better task fairness achieved by CFS. CFS reduces a lot of things that O (1) carried out:

* No time slices and tedious dynamic priority calculations
* No sleep time tracking
* No process type identification

I/O bound processes and CPU bound processes are distinguished quite inherently by the CFS algorithm. Instead, CFS tries to behave like an ideal and accurate multitasking CPU one that could run multiple processes simultaneously, by giving each process fairly equal weightage that they deserve. It also enhances the tasks fairness by introducing the concepts of group scheduling and scheduling classes. So CFS overshadows O (1) in terms of fairness and interactivity performance which are one of the goals of Operating systems.

1. [**https://en.wikipedia.org/wiki/O(1)\_scheduler**](https://en.wikipedia.org/wiki/O(1)_scheduler) [↑](#endnote-ref-1)
2. [**https://www.youtube.com/watch?v=bsjOY7pjQII**](https://www.youtube.com/watch?v=bsjOY7pjQII) [↑](#endnote-ref-2)
3. <https://en.wikipedia.org/wiki/Completely_Fair_Scheduler> [↑](#endnote-ref-3)
4. iv [**https://www.youtube.com/watch?v=MkJfuI5\_hjc**](https://www.youtube.com/watch?v=MkJfuI5_hjc) [↑](#endnote-ref-4)
5. v Wong, C. S., I. K. T. Tan, R. D. Kumari, J. W. Lam, and W. Fun. "Fairness and interactive performance of O (1) and CFS Linux kernel schedulers." In *2008 International Symposium on Information Technology*, vol. 4, pp. 1-8. IEEE, 2008. [↑](#endnote-ref-5)