**TAKE HOME ASSIGNMENT**

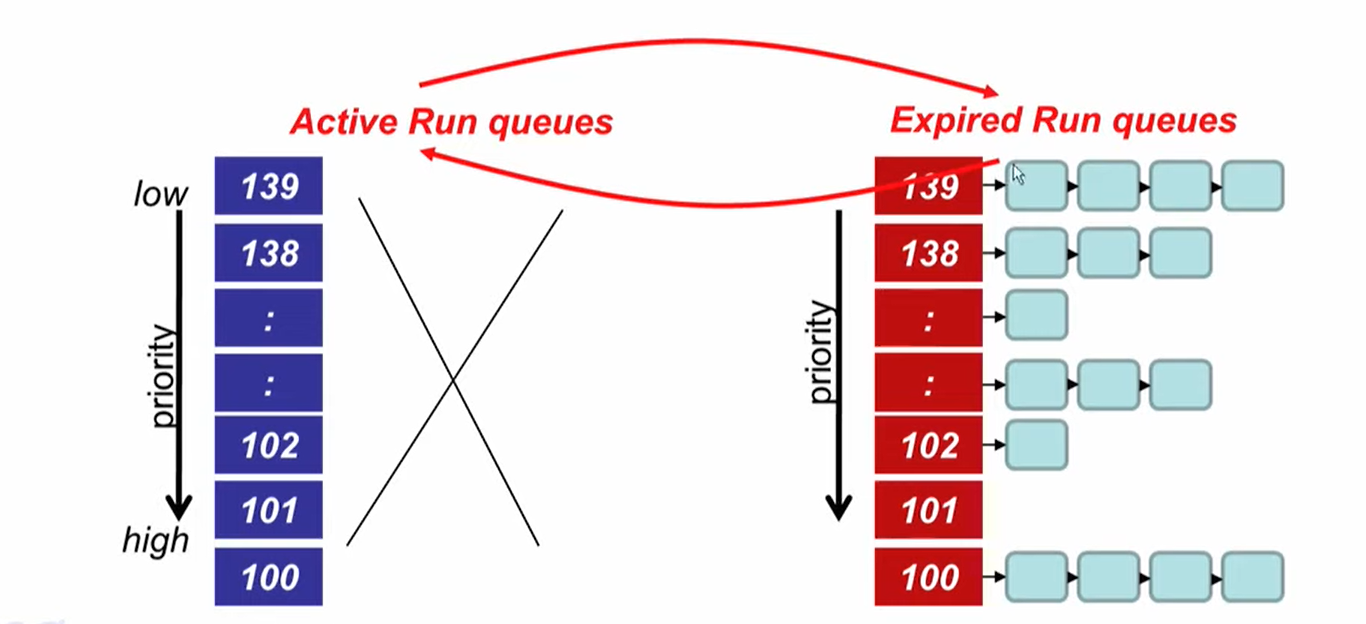
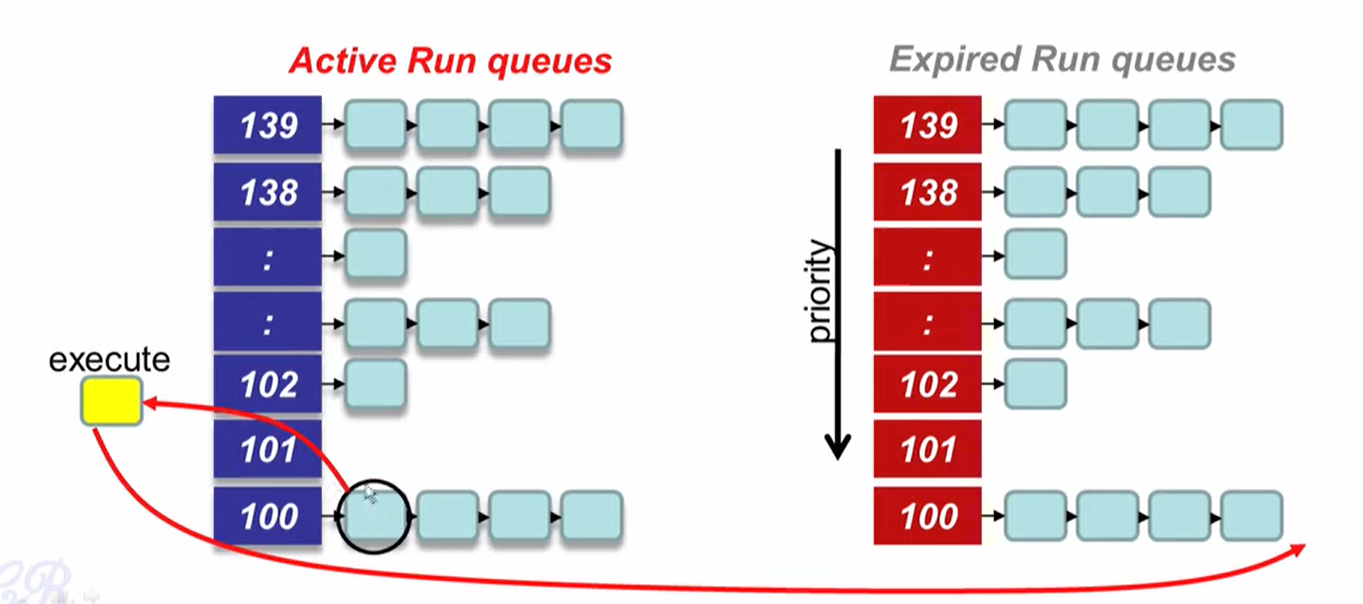
**Name: Sudipta Halder**

**Roll: 2021202011**

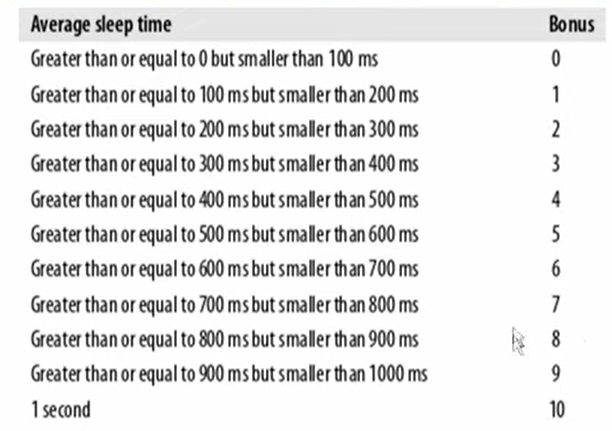
**SET - 1**

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

* **O(1) scheduler**
  + **Timeline:** Linux 2.6 to 2.6.22
  + Constant time required to pick the next process to execute for which it easily scales to large number of processes.
  + Processes are divided into two types:
    - Real Time processes:
      * Priorities from 0 to 99 (0 highest priority, 99 lowest priority).
      * 0 is the highest priority and 99 is the lowest priority for real time processes.
    - Normal Processes:
      * Interactive
      * Batch
      * Priorities from 100 to 139 (100 highest, 139 lowest).
      * 100 is the highest priority and 139 is the lowest priority for normal processes.
  + **Scheduling in Normal Processes:**
    - Two ready queues in each CPU (Active Run Queues, Expired Run Queues).
    - Each queue has 40 priority classes (100 - 139).
    - 100 has highest priority and 139 has lowest priority.
    - Algorithm used is similar to multilevel feedback queue with slight variation.
    - When a context switch occurs, the scheduler scans the active run queues starting from index 100 to 139. It picks the run queue which has a non-zero or non-empty queue present in that. After selecting the run queue, it starts executing the processes of that queue one by one. At the end of the time-slice for a process, the process is put into the expired run queue. Similarly, all the processes of that run queue are sent to the expired run queue after their time-slice. The scheduler will then proceed to the next non-empty run queue of the active run queues. Gradually, we would see that as time proceeds and time-slice is complete the scheduler keeps picking out processes from the active run queues, executes them and puts them into the expired run queue. After a while, we would have a point where the entire set of processes in the active run queue is complete. On the other hand, the expired run queue is now filled up. When this happens, the scheduler will switch between active run queues and expired run queues. The process goes on continuously. The main reason for it being used in Linux kernel scheduler because it prevents starvation. It ensures that every process gets a turn to execute in the CPU.



* + **Why constant time (O(1) time)?**
    - There are two steps in the scheduling
      * Find the lowest numbered queue with at least 1 task. [Stores bitmap of 40 bits of run queues with non-zero entities. Uses special instruction ‘find-first-bit-set’ bsfl on intel.]
      * Choose the first task from the queue [extracting data from head is obviously O(1) time.]
  + **Static Priority:**
    - 120 is the base priority for normal processes.
    - nice: commands line to change default priority of a process. $nice -n N ./a.out
    - N is a value from +19 to -20
      * Most selfish ‘-20’; (I will go first).
      * Most generous ‘+19’; (I will go last).
  + **Dynamic Priority:**
    - To distinguish between batch and interactive processes.
    - Use a ‘bonus’, which changes based on a heuristic.
    - Dynamic priority = max(100, min(static priority – bonus + 5), 139)
    - bonus has a value between 0 and 10
    - If bonus < 5, implies less interaction with the user, thus more of a CPU bound process. The dynamic priority is therefore decreased (toward 139).
    - If bonus > 5, implies more interaction with the user, thus more of an interactive process. The dynamic priority is increased (toward 100).
    - Based on average sleep time
      * An I/O bound process will sleep more therefore should get a higher priority.
      * A CPU bound process will sleep less, therefore should get lower priority.
    - Dynamic priority used to determine which run queue to put the task.
    - No matter how ‘nice’ you are, you still need to wait on run queues – prevents starvation.



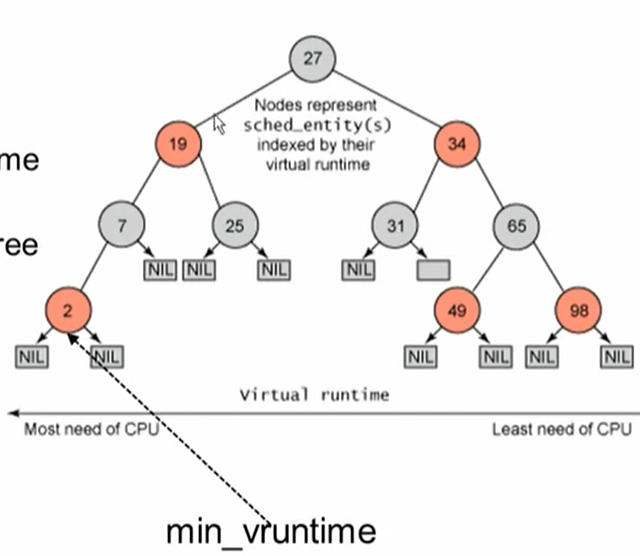
* + **Setting the Time slice:**
    - Interactive processes have higher priorities but likely to not complete their time slice. Give it the largest time slice to ensure that it completes its burst without being pre-empted.
    - If priority < 120, time slice = (140 – priority) \* 20 milliseconds
    - Else time slice = (140 – priority) \* 5 milliseconds
  + **Limitations of O(1) Scheduler:**
    - Too complex to distinguish between interactive and non-interactive processes.
    - Dependence between time-slice and priority.
    - Priority and time-slice values not uniform.[[1]](#endnote-1)
* **Completely Fair Scheduler(CFS)**
  + **Timeline:** Since 2.6.23
  + By Ingo Molnar based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
  + Incorporated in the Linux Kernel since 2007.
  + No heuristics.
  + Elegant handling of I/O and CPU bound processes.
  + Aims at dividing the CPU time equally among the processes by Ideal Fair Scheduling.
  + **Ideal Fair Scheduling:**
    - If there are N processes in the system, each process should have got (100/N)% of the CPU time.

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| **A** | 8 ms |
| **B** | 4 ms |
| **C** | 16 ms |
| **D** | 4 ms |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ideal Fairness** | | | | | | | | |
| **A** | 1 | 2 | 3 | 4 | 6 | 8 |  |  |
| **B** | 1 | 2 | 3 | 4 |  |  |  |  |
| **C** | 1 | 2 | 3 | 4 | 6 | 8 | 12 | 16 |
| **D** | 1 | 2 | 3 | 4 |  |  |  |  |

-----------------------------------🡪 Execution w.r.t time

* + **Virtual Runtimes**
    - In each processes block there is an entity called Virtual runtime(vruntime).
    - At every scheduling point, if process has run for t ms, then (vruntime += t)
    - Therefore, the vruntime for a process increases monotonically.
  + **The main idea behind CFS**
    - Whenever timer interrupt occurs
      * Choose the task with the lowest vruntime. It is maintained by a variable called min\_vrintime.
      * Compute its dynamic time-slice.
      * Program the high resolution timer with the time-slice.
    - The process begins to execute in the CPU.
    - When interrupt occurs again
      * Context switch if there is another task with a smaller runtime.

****

* + **Picking the Next Task to Run**
    - CFS uses a red-black tree
      * Each node in the tree represents a runnable task.
      * Nodes ordered according to their vruntime.
      * Nodes on the left have lower vruntime compared to nodes on the right of the tree.
      * The left most node is the task with the least vruntime. (This is cached in min\_vruntime).
    - At a context switch
      * Pick the left most node of the tree
        + This has the lowest runtime.
        + It is cached in min\_vruntime. Therefore it is accessed in O(1).
      * If the previous process is runnable, its is inserted into the tree depending on its new vruntime. Time complexity for this is O(log(n)).
        + So, after execution of a process, the vruntime of the process increases and so when it is inserted back in the RB tree, it would be inserted somewhere in the right or the middle. So, tasks move from left to right of the tree after its execution completes. This ensures that every process gets a chance to execute because at one or the other, every process is going to get the min\_vruntime in the RB tree and therefore gets executed. Therefore starvation is avoided.
  + **Why Linux Kernel chose Red Black Tree** 
    - Self-Balancing (No path in the tree will be twice as long as any other path.)
    - All operations are O(log(n)). Thus inserting/ deleting tasks from the tree is quick and efficient.
  + **Priorities and CFS**
    - CFS does not use any exclusive priority queue like O(1) scheduler.
    - If a process has run for t ms, then vruntime += t \* (weight based on nice of process)
    - A lower priority implies time moves at a faster rate compared to that of a high priority task.
  + **Distinguish between I/O and CPU bound processes**
    - We need to ensure that I/O bound processes should get higher priority and get a longer time to execute compared to CPU bound.
    - CFS achieves efficiently
      * I/O bound processes have small CPU bursts and hence have a low vruntime. They would appear towards the left of the tree. So, they are given higher priorities.
      * I/O bound processes will typically have larger time slices, because they have smaller vruntime.
  + **How New processes are treated**
    - A new process is added to the RB-tree with an initial value of min\_vruntime. Therefore, it gets placed at the leftmost node of the RB-tree. This ensures that it get to execute quickly. [[2]](#endnote-2)
* **Which one would you prefer and why?**

I would definitely choose Completely Fair Scheduler(CFS) over O(1) Scheduler because it keeps non-interactive processes running along with the interactive ones as well. This makes the CPU feel much faster even for the background processes. CFS really better allocates better CPU time and scatters tasks around the cores. Also, the advantages of CFS include a significantly shorter response time than O (1). In O(1) schedule, the normal processes are kept in he last 40 priority queues(100 to 139) which can increase the runtime of normal processes whereas CFS doesn’t use any priority queue for priority. It just uses weighted vruntime for adjusting priority which is quite simple and effective at the same time. Also, In O(1) it is too complex to distinguish between interactive and non-interactive processes because it uses bonus value which is a complex heuristics. On the other hand, CFS decides this just by weighted vruntime. Lastly, the time-slice is also distributed very efficiently in CFS in comparison to O(1) scheduler. Thus, I can understand the reason behind dropping the O(1) scheduler and introducing the CFS scheduler from Linux 2.6.23.[[3]](#endnote-3)

1. <https://www.youtube.com/watch?v=vF3KKMI3_1>s [↑](#endnote-ref-1)
2. <https://www.youtube.com/watch?v=scfDOof9pww> [↑](#endnote-ref-2)
3. <https://sudonull.com/post/222495-CFS-vs-O-1-scheduler> [↑](#endnote-ref-3)