CS-347(M) Operating Systems

Visualising the Completely Fair Scheduler

The Process Scheduler for Linux Kernels

Project by:

Srivatsan Sridhar 150070005
Nihal Singh 150040015
Arpan Banerjee 150070011

CFS Visualizer link: https://nihal111.github.io/CFS-visualizer/

Github Repo link: https://github.com/nihal111/CFS-visualizer

Abstract

This project aims to create a visualizer for the Completely Fair Scheduler (CFS) used in Linux. The visualizer emulates the scheduler by taking as inputs different processes with their start times, durations and priorities, and demonstrates how they would be scheduled by CFS. The visualizer is implemented on a webpage using Javascript. It has been made interactive by displaying the background data structures used and the events taking place, in a zoomed-in time scale.

The main target of the project is to develop an in-depth understanding of CFS. This is achieved in a better way by enabling a graphical visualisation of the background structures used by CFS, as used in our visualizer. As developers of the visualizer, we went through a study of CFS, which helped us develop a better understanding of the method.

The project begins with a study of CFS and the earlier schedulers used in Linux. As a part of this, we have explored the primitive scheduling policies used before Linux 2.6, the O(1) and O(N) schedulers. These schedulers were based on FIFO-based queues. We observe their simplicity and their problems, leading to the motivation for the design of CFS. We then explain the details of CFS with emphasis on the new concepts it introduced, such as virtual runtime and self-balancing red-black trees.

Next, we describe the algorithm used by CFS. We describe the parameters and data structures used by the algorithm and what it does when processes are created, scheduled, preempted and terminated.

The next section describes our implementation of the visualizer. This includes the approach used, the inputs and outputs to the visualizer and the implementation of the required data structures, with code segments.

This is followed by the results observed from the visualizer. We run the visualizer for some sample test cases and demonstrate the “fairness” aspect of CFS and its method of handling priorities through these test cases.

Finally, we provide our views and critique on CFS. We have compared its results with other standard scheduling policies like FIFO and Round Robin.

Introduction

1. History

Earlier versions of Linux kernel used round robin approach which was implemented using a circular queue. It was quite simple to implement.

Linux kernel 2.2: Scheduling classes were introduced. Processes were divided in three classes namely : real-time, non-preemptible and normal or non-real time processes because these classes had to be handled differently.

In Linux Kernel 2.4: Scheduler came with O(N) complexity, N being number of tasks currently in system in runnable state. It maintains a queue of runnable tasks and whenever scheduling is to be done, it goes through all the processes in the queue and selects the best process (based on priority) to be scheduled next. It used a function called `goodness()` and the process with the highest value of goodness is scheduled first. The problem with this scheduler is that its time complexity increases with the number of tasks present. At very high loads, the processor can be consumed with scheduling and devote little time to the tasks themselves. Thus, the algorithm lacked scalability.

In Linux Kernel 2.6: Designed by Ingo Molnar, complexity of the scheduler was reduced to O(1) and hence the name of the scheduler became O(1) scheduler. There were two run queues for each priority level. As we will see later, there are 0 to 140 priority levels in Linux, there will be 280 such queues. One queue is for active processes while the second is for expired process. A process is moved to the expired queue when its time slice is completed. When all processes in the active queue exhaust their time-slice, the expired queue is made active. To identify the task to be scheduled next, the scheduler dequeues a task from each priority queue. Unlike its precursors, the Linux 2.6 scheduler allows preemption. This means a lower-priority task won't execute while a higher-priority task is ready to run. The scheduler preempts the lower-priority process, places the process back on its priority list, and then reschedules.

2. Features of the Completely Fair Scheduler

As the name suggests, the Completely Fair Scheduler tries to give all processes a fair chance to execute on the processor. It was again developed by Ingo Molnar, based on the implementation of a fair scheduling named “Rotating Staircase Deadline” by Con Kolivas.

CFS basically models an "ideal, precise multi-tasking CPU" on real hardware. "Ideal multi-tasking CPU" is a (non-existent) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/n running speed. On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime."

The concept of virtual runtime is central to CFS. The virtual runtime of a task effectively keeps track of its elapsed runtime, normalized to the total number of running tasks. Each time a task has to be scheduled, the task with the minimum virtual runtime is selected to be scheduled. Thus, the task that has run for the least time so far gets the chance to be scheduled first, making the scheduling “fair”. The virtual runtime of the task is increased when it runs on the processor for a while. In addition, virtual runtime helps to implement different priorities for tasks elegantly. Virtual runtime increases slowly for a task with a high priority, thus allowing it more actual execution time on the CPU.

A major change brought by CFS is the data structure used. Instead of using queue-based structures like the earlier schedulers, CFS uses a red-black tree to maintain the current tasks in the order of their virtual runtimes. A red-black tree is a self-balancing tree, i.e. it maintains the property that no path from the root to a leaf is more than twice as long as another path. In a red-black tree, tasks are arranged in increasing order of their virtual runtimes from left to right. Thus the task with the least virtual runtime is easily found as the leftmost node in the tree. Any task can be inserted or deleted from the red-black tree in O(log(N)) time, while ensuring that the tree is still balanced.

Another interesting aspect of CFS is group scheduling (introduced with the 2.6.24 kernel). Sometimes, it may be desirable to first provide fair CPU time to each user on the system and then to each task belonging to a user. This is useful for a server that spawns many tasks to parallelize incoming connections (a typical architecture for HTTP servers).

Approach

1. Overview of the CFS Algorithm

This section describes the basics of the data structures and algorithms used in CFS.

1) The scheduler maintains following important variables:

- ***sched_latency_ns*** (6 ms): Minimum time period in which all tasks are scheduled at least once. (With many processes, the latency may be more than this)
- ***sched_min_granularity_ns*** (0.75 ms): Minimum time a process is allowed to run without being preempted.
- ***sched_wakeup_granularity_ns*** (1 ms): Time within which a newly woken up process will be scheduled.
- ***period_ns***: Computed as $max\{sched\_latency\_ns,\ n * sched\_min\_granularity\_ns\}$ with *n* as the number of processes currently active.

  These variables may be observed on your machine (in ns) by typing this command in the terminal: `cat /proc/sys/kernel/<variable_name>`
  On the machine we checked, the values of the first three variables were 18 ms, 2.25 ms and 3 ms respectively.

2) Each process, when created, gives three pieces of information to the scheduler:

- The time of creation of the process (in system time)
- The total time duration that the process requires on the processor (before it terminates or gets blocked for an I/O operation) known as the CPU burst
- The priority of the process, given as a “nice” value ranging from -20 to +19. (Greater the nice value, lower is the priority of the process)

3) When the process is created, it is added to the red-black tree

- Virtual runtime (*vruntime*) of the process is set as the minimum virtual runtime among processes already present (*min_vruntime*). This is done so that newly created processes get an early chance to be executed.
- Real runtime of the new process (*truntime*) is set to zero.
- Weight of the process computed as $weight = 1024 * 1.25^{-nice}$

- Inserting in the red-black tree maintains the property that the process with the minimum virtual runtime is the leftmost in the tree.

4) When a process has to be scheduled,

- The scheduler chooses the one with the lowest vruntime (the leftmost in the tree).
- This process is removed from the red-black tree. (The tree re-balances itself to find the process with the next least vruntime and update the value of min_vruntime).
- The process is given a time slice computed as $slice = \frac{weight}{\sum weights} * period$

5) As the process runs on the processor,

- Real runtime (*truntime*) increases according to the real amount of time run. $truntime \ += \ exectime$
- Virtual runtime (*vruntime*) increases as $vruntime \ += \ \frac{exectime}{weight} * 1024$,i.e. *vruntime* increases more slowly for processes with a higher weight.

6) When the process completes the time slice allocated to it,

- If the process has not completed its required time duration, it is added back to the red-black tree with its increased virtual runtime, such that the tree remains balanced.
- The scheduler goes back to step 4 and selects another process to run.

2. Implementation of the visualizer

1) The visualizer takes input in the following format: The first line gives the number of processes N and the total execution time of all processes T. The next N lines give the process ID, the starting time, the execution time and the nice value of each of the processes. The input can be entered in a text box or given as a file.
2) Time queue for the processes: The processes are initially added in a time queue in the order of their starting times. When the starting time of a process is reached, that process is dequeued and added to the red-black tree. Note that this time queue is not involved in the scheduling in any way. This time queue is only used to temporarily store the process before they are added into the red-black tree at their starting time.
3) A red-black tree name *timeline* is used to store the processes. The implementation of the red-black tree was picked up from elsewhere (see References) and will not be explained in this section.
4) Initializing the scheduler:

```
time_queue = tasks.task_queue;  // queue of tasks sorted in start_time order
time_queue_idx = 0; // index into time_queue of the next nearest task to start
min_vruntime = 0;  // min_vruntime is set to the smallest vruntime of tasks on the timeline
running_task = null; // current running task or null if no tasks are running.
curTime = 0;  // current time in millis
 total_weight = 0;
 num_of_tasks_until_curTime = 0;  // Total number of tasks that are currently in timeline +
running task
min_granularity = 0.75*1000; //Value in microseconds
latency = 6*1000; //value in microseconds
```

5) The function nextIteration() adds tasks from the time queue when it starts, inserts a task back to the red-black tree when its time slice is over, and finds the current running task. It does these tasks at an interval of every millisecond.

```
function nextIteration(tasks, timeline, callback) {
    if (curTime < tasks.total_time) {
        updateMessageDisplay("CPU Time = " + curTime);
        // Periodic debug output
        updateCurTimeDisplay(curTime);

        setTimeout(function(){
            addFromTaskQueue(tasks, timeline, callback);
        }, DELAY/3);
```

```
        setTimeout(function(){
            insertRunningTaskBack(tasks, timeline, callback);
        }, 2*DELAY/3);
        setTimeout(function(){
            findRunningTask(tasks, timeline, callback);
        }, 3*DELAY/3);

        if (callback) {
            callback(curTime, results);
        }

        return new Promise(resolve => {
          setTimeout(() => {
            resolve(nextIteration(tasks, timeline, callback));
          }, DELAY);
        });
    } else {
        return;
    }
}
```

6) The function addFromTaskQueue() adds a task from the time queue to the red-black tree at its start time, initializes its parameters and computes the allowed time slices.

```
function addFromTaskQueue(tasks, timeline, callback) {
    // Check tasks at the beginning of the task queue. Add any to
    // the timeline structure when the start_time for those tasks
    // has arrived.
    while (time_queue_idx < time_queue.length &&
           (curTime >= time_queue[time_queue_idx].start_time)) {
        num_of_tasks_until_curTime++;
        var new_task = time_queue[time_queue_idx++];
        // new tasks get their vruntime set to the current
        // min_vruntime
        new_task.vruntime = min_vruntime;
        new_task.truntime = 0;
        new_task.actual_start_time = curTime;
        timeline.insert(new_task);
        curTree.insert('n', new_task.vruntime, new_task.id);
        updateMessageDisplay("Adding " + new_task.id + " with vruntime " +
new_task.vruntime);
        //updateMessageDisplay("Adding " + new_task.id);
        update(curTree);

        updateSummationWeights(new_task.weight);
    }

    updateSlices(time_queue, Math.max(latency, min_granularity*num_of_tasks_until_curTime));
}
```

7) The function insertRunningTaskBack() inserts a process back into the red-black tree once its time slice is completed.

```
function insertRunningTaskBack(tasks, timeline, callback) {
    // If there is a task running and its vruntime exceeds
    // min_vruntime then add it back to the timeline. Since
    // vruntime is greater it won't change min_vruntime when it's
    // added back to the timeline.
    if (running_task && (running_task.vruntime > min_vruntime) && (running_task.this_slice >
running_task.slice)) {
        timeline.insert(running_task);
        curTree.insert('n', running_task.vruntime, running_task.id);
        updateMessageDisplay("Inserting " + running_task.id + " with vruntime " +
running_task.vruntime);
        //updateMessageDisplay("Inserting " + running_task.id);
        update(curTree);
        running_task = null;
        updateCurTaskDisplay("-");
    }
}
```

8) The function findRunningTask() checks if a task is already running. If no task is currently running, it removes the task with minimum *vruntime* from the tree, makes it the running task and changes the value of *min_vruntime.* If a task is already running, it increases *vruntime* and *truntime* for that task, and checks if its required time duration is over.

```
function findRunningTask(tasks, timeline, callback) {
    // If there is no running task (which may happen right after
    // the running_task is added back to the timeline above), find
    // the task with the smallest vruntime on the timeline, remove
    // it and set it as the running_task and determine the new
    // min_vruntime.
    if (!running_task && timeline.size() > 0) {
        var min_node = timeline.min();
        running_task = min_node.val;
        running_task.this_slice = 0;
        timeline.remove(min_node);
        curTree.remove(curTree.min());
        updateMessageDisplay("Removing " + running_task.id + " with vruntime " +
running_task.vruntime);
        //updateMessageDisplay("Removing " + running_task.id);
        updateCurTaskDisplay(running_task.id);
        update(curTree);
        if (timeline.size() > 0) {
            min_vruntime = timeline.min().val.vruntime
            updateMessageDisplay("Updating min_vruntime to " + min_vruntime);
        }
```

```
    }

    // Results data for this time unit/tick
    var tresults = {running_task: null,
                    completed_task: null};

    // Update the running_task (if any) by increasing the vruntime
    // and the truntime. If the running task has run for it's full
    // duration then report it as completed and set running_task
    // to null.
    var task_done = false;
    if (running_task) {
        running_task.vruntime += roundTo(1024/(1000*running_task.slice), 2);
        running_task.truntime++;
        running_task.this_slice++;
        tresults.running_task = running_task;
        //updateMessageDisplay(curTime + ": " + running_task.id);
        if (running_task.truntime >= running_task.duration) {
            running_task.completed_time = curTime;
            tresults.completed_task = running_task
            task_done = true; // Set running_task to null later
            //updateMessageDisplay("Completed task:", running_task.id);
            num_of_tasks_until_curTime--;
            updateSummationWeights(-1*running_task.weight);
            updateMessageDisplay(running_task.id + " is over")
            updateCurTaskDisplay("-");
        }
    }

    tresults.num_tasks = timeline.size() + (running_task ? 1 : 0);

    results.time_data[curTime] = tresults;

    if (task_done) {
        running_task = null;
    }

    curTime++;
}
```

9) The visualizer shows what is happening at every millisecond, at intervals of 4 seconds. It shows the process currently in the red-black tree. It displays a log of events taking place in the current millisecond, such as adding/removing nodes from the tree. It also displays the ID of the process currently running. Finally the visualizer generates a report showing the currently executing process and the number of processes in the tree every millisecond.

Results

1. The visualizer in Action

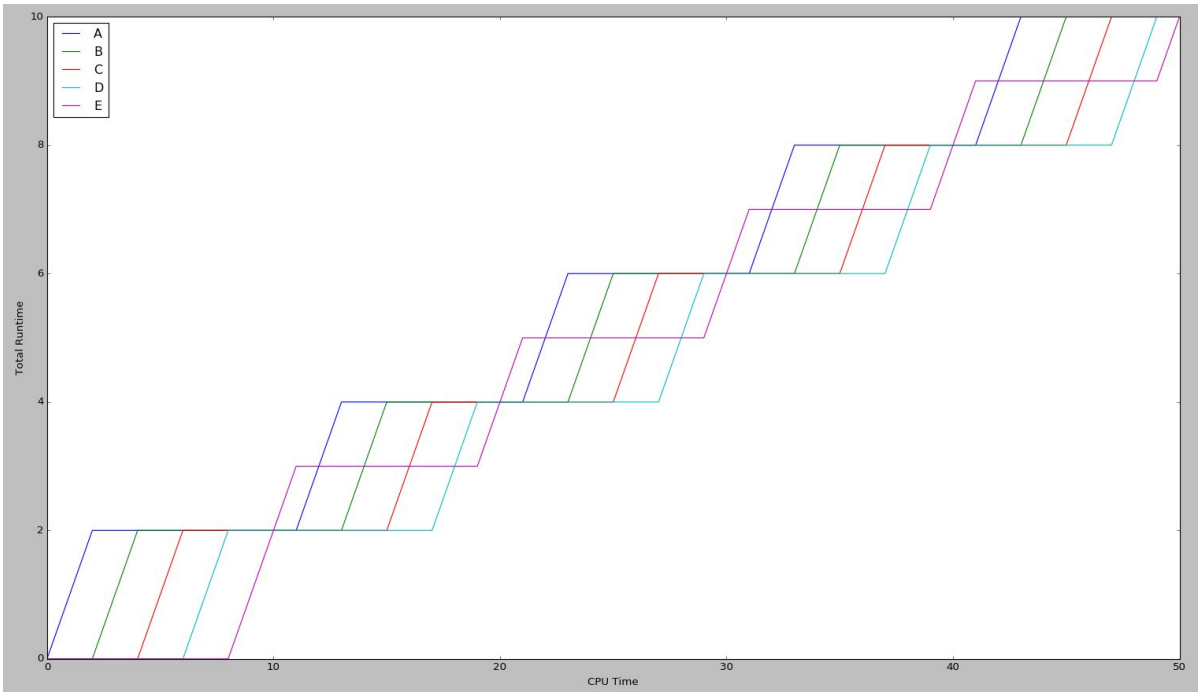
The following are the plots of real runtimes of each process versus the CPU time, for various test cases, obtained after running the scheduler.

Test Case 1 (Processes starting at the same time)

Number of Processes = 5

Total time = 51

Process Name	Start Time	Duration	Nice Value
A	1	10	0
B	1	10	0
C	1	10	0
D	1	10	0
E	1	10	0

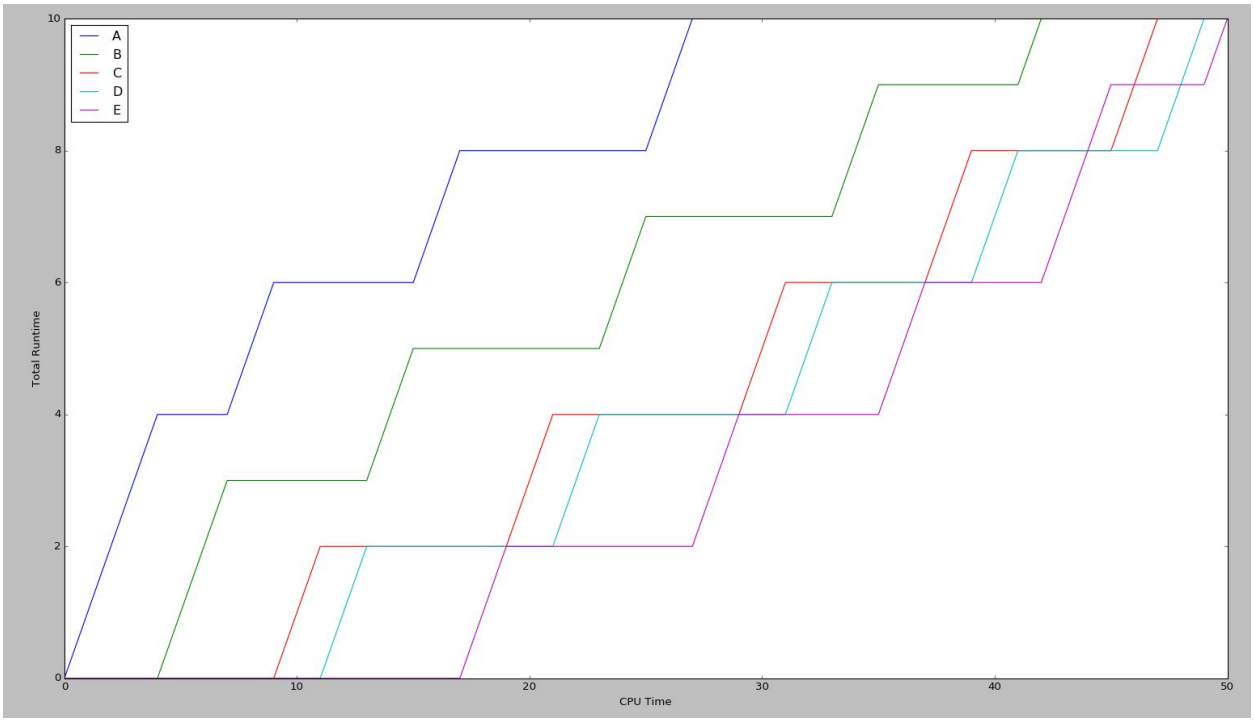


Test Case 2 (Processes starting at different times)

Number of Processes = 5

Total time = 51

Process Name	Start Time	Duration	Nice Value
A	1	10	0
B	4	10	0
C	6	10	0
D	10	10	0
E	15	10	0

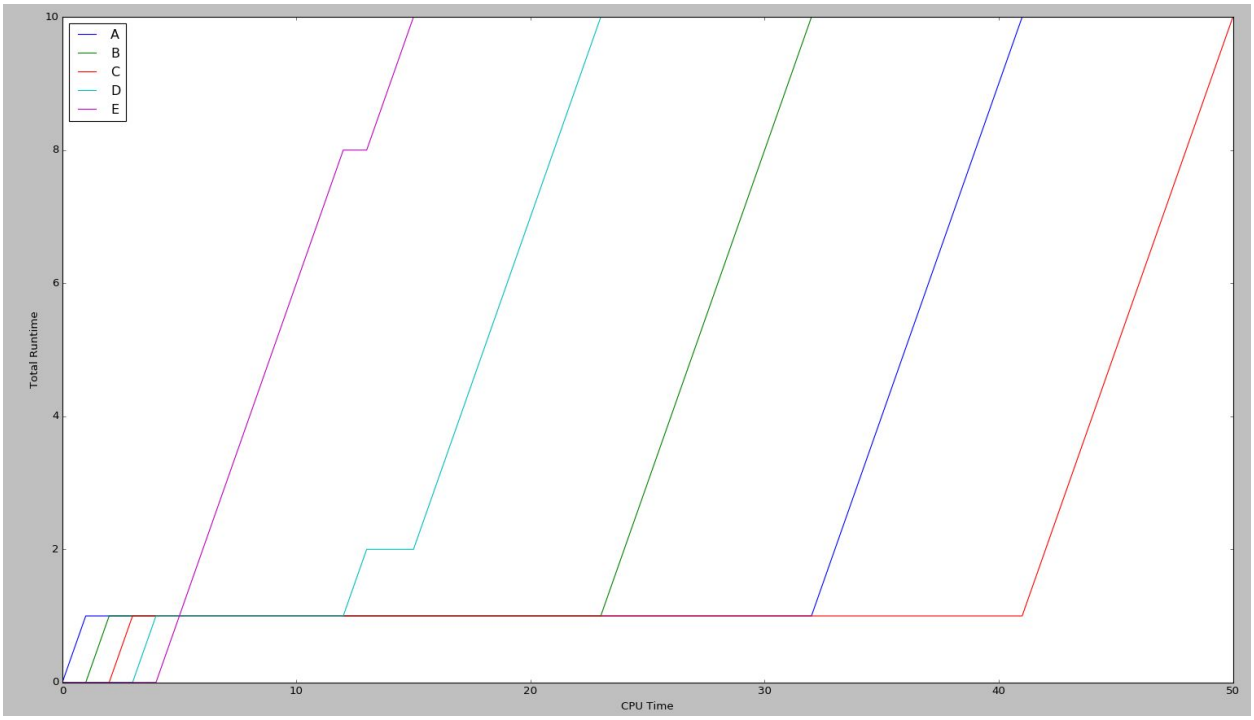


Test Case 3 (Processes with different nice values over a wide range)

Number of Processes = 5

Total time = 51

Process Name	Start Time	Duration	Nice Value
A	1	10	2
B	1	10	-4
C	1	10	3
D	1	10	-10
E	1	10	-19

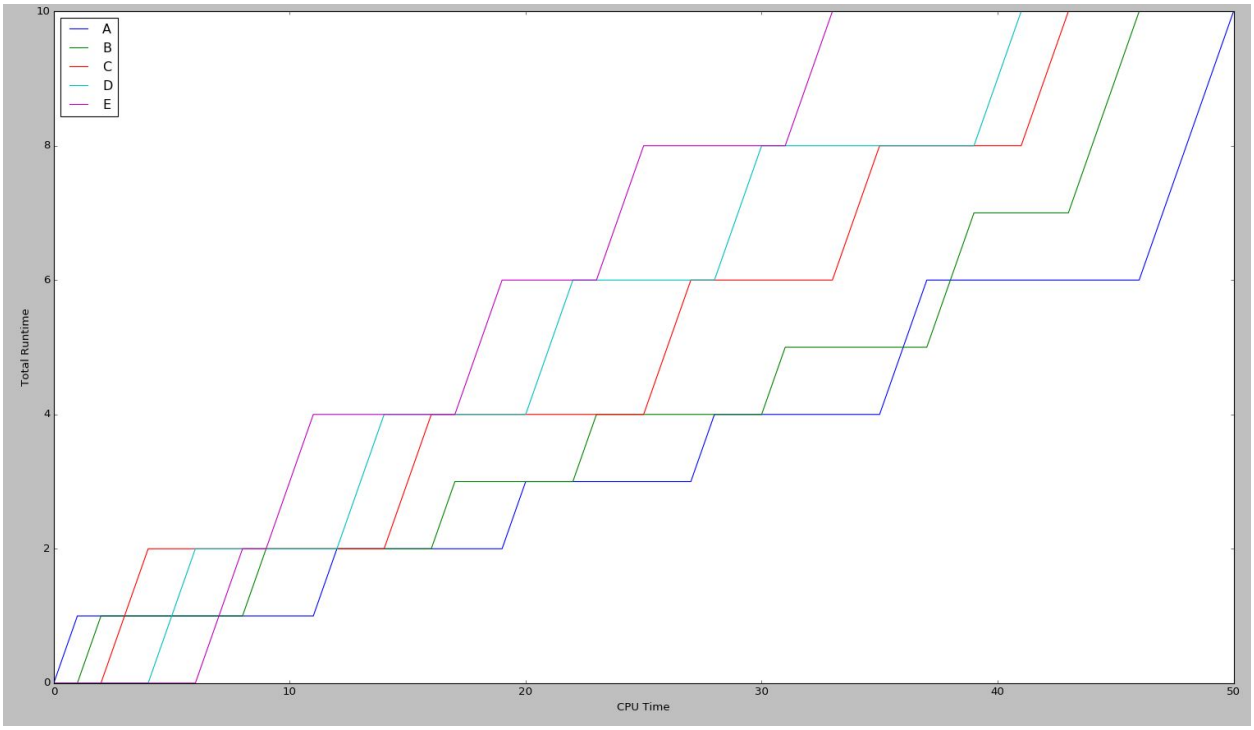


Test Case 4 (Processes with different nice values over a small range)

Number of Processes = 5

Total time = 51

Process Name	Start Time	Duration	Nice Value
A	1	10	0
B	1	10	-1
C	1	10	-2
D	1	10	-3
E	1	10	-4



2. Demonstration of Fairness

In the above plots of process virtual runtimes vs CPU time, we can observe that the virtual runtimes of two process differ by at most a very small amount (corresponding to one period in which each process must be scheduled once). In general, CFS guarantees good fairness as no process will run for too long and starve the others. As its virtual runtime increases, it will be switched out for another process.

3. Demonstration of Process Priorities

In test case 3, with a wide range of nice values, the processes with the least nice values take almost all of the processor time wanting to finish early. Process E, with the least nice value, finishes first with others following a similar trend.

In test case 4, with the processes having nearly alike nice values, process time is shared a bit more equally than in case 3. Process A with the highest nice value, having started earliest, finishes last. On the other hand, process E, starts last and finishes first.

Analysis and Critique

1. Fairness vs Round Robin

It may not be very intuitive as to how the CFS achieves a better form of fairness than the simple Round Robin queue. However, the key difference lies in the concept of “sleeper fairness”.

With RR, each of the processes on the ready queue gets an equal share of CPU time. However, the processes that are blocked/waiting for I/O may sit on the I/O queue for a long time, but they don't get any built-up credit for that once they get back into the ready queue. With CFS, processes do get credit for that waiting time, and will get more CPU time once they are no longer blocked (due to the concept of virtual runtime). That helps reward more interactive processes (which tend to use more I/O) and promotes system responsiveness.

To see why this happens, note that the scheduler selects the process with the minimum virtual runtime. The virtual runtime of a process increases when a process runs on the CPU. When a process gets blocked for I/O, it stops running on the CPU and its virtual runtime does not increase in that duration. When it wants to resume execution, its virtual runtime remains much lower than the other process present in the tree. This allows the process to run for a longer time till its virtual runtime comes at par with the other process. (If the process had been blocked for a very long time, it can potentially get a very large time slice at once. This is prevented by not letting the resuming process to lag by more than Δ behind the minimum virtual runtime.)

2. Processing and Memory Overheads of CFS

As seen in the section on “History”, the Linux 2.4 scheduler was an O(N) scheduler. This meant that the time overhead incurred by operating system in deciding which process must execute, increases linearly with the number of process in the queue. This overhead becomes tremendously large when the number of processes increases. Therefore this system was not suitable for high degrees of multiprogramming and multithreading.

The alternative implemented in Linux 2.6 was an O(1) scheduler. This means that the time overhead is constant even if the number of processes increases. However, this scheduler maintained two separate queues for each of the 140 levels of priority. This means 280 queues of processes required for implementation of priorities. This presents an overhead in terms of the memory required to store the queues.

CFS uses a red-black tree to keep the processes. The self-balancing property of red-black trees ensures that the minimum virtual runtime is found in O(1) time and any insertion/deletion of a process in the tree takes O(logN) time. Thus the red-black tree structure helps to efficiently implement priorities by maintaining the overhead time as O(logN) which is scalable for a large number of processes.

References

1. Documentation of the Linux Scheduler: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
2. On history of development of CFS: https://algorithmsandme.in/2014/03/16/scheduling-o1-and-completely-fair-scheduler-cfs/
3. On red-black trees: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
4. Fairness Test and Response Time Analysis for CFS with greedy threaded programs: https://www.researchgate.net/figure/220623439_fig4_Figure-8-Process-Fair-Scheduler-prevents-greedy-threaded-program-from-dominating-CPU
5. Code for the implementation of red-black trees: https://github.com/kanaka/rbt_cfs