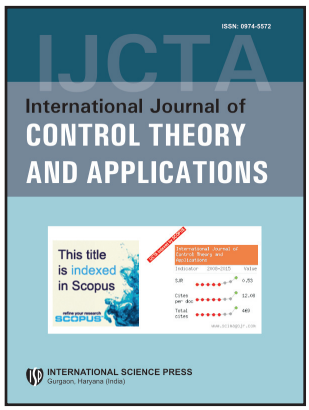

International Journal of Control Theory and Applications

ISSN : 0974-5572

Volume 9 • Number 44 • 2016

A Survey on Fairness and Performance Analysis of Completely Fair Scheduler in Linux Kernel

Pooja Tanaji Patil[a], Sunita Dhotre[b] and Rucha Shankar Jamale[c]

[a]*M.Tech, Department of Computer Engineering, Bharati Vidyapeeth Deemed University, Pune, India. Email: poojapatil2829@gmail.com*
[b]*Associate Professor, Department of Computer Engineering, Bharati Vidyapeeth Deemed University, Pune, India. Email: ssdhotre@bvucoep.edu.in*
[c]*M.Tech, Department of Computer Engineering, Bharati Vidyapeeth Deemed University, Pune, India. Email: rjrucha98@gmail.com*

Abstract: The main objective of operating system scheduler is to allocate the system resources appropriately to all the tasks or applications to obtain the multiprocessing and multithreading goal. To maintain the load on processors, schedulers perform an essential role in the embedded systems. In Linux, Completely Fair Scheduler (CFS) enhance the efficiency by ensuring the fairness among multiple tasks using thread fair scheduling algorithm. To prove the problem of unfairness and to evaluate the effectiveness of CFS algorithm there is need to analyze the performance. The CFS achieves the better performance and responsiveness as compared to earlier Linux scheduler algorithms. This paper illustrates how the fairness is achieved by CFS in Linux Kernel by calculating the vruntime, timeslice and load-weight ratio for multiple tasks and after comparing with earlier Linux schedulers it is concluded that the CFS achieves better fairness and performance than earlier scheduler algorithm

Keyword: Multiprocessing, multithreading, Scheduler, thread fair scheduling algorithm, CFS, Fairness, performance, responsiveness.

1. INTRODUCTION

An essential part of system software is the operating system (OS) [1][2]. The operating system is the software which handles the group of hardware resources and act as intermediary among the user and computer hardware[3]. A scheduler is called as the controller of operating system. The goal of scheduler [3] is to prevent starvation and deadlock between the processes at running time. Process scheduling is a key part of Multiprogramming operating systems [3]. In such operating systems, multiple processes are loaded into the executable memory at the same time, and these loaded processes shares the CPU using time multiplexing. Process scheduler schedules different processes to be executed by the CPU based on a particular scheduling algorithm. A scheduler gives an efficient context switching cost by controlling scheduling period. It is the basis of a multiprogramming operating system such as Linux. Fairness obtained the maximum importance in the scheduler, and many of scheduling algorithms have been studied in order to attain accurate fairness.

2. RELATED WORK

A. Scheduler

The main function of the scheduler[4] is to allocate the CPU (processor) time fairly[5] to all the running processes. The process of CPU scheduling is allowing the execution of one process on CPU while another process is in waiting state due to lack of resources. The main goal of CPU scheduler is to make the system fair, fast and efficient. Modern operating systems (OS) mainly designed to achieve the optimal performance by virtualizing two resources such as CPU and Memory[6].In the multitasking environment the virtualization of CPU is achieved by allocating the CPU among multiple tasks and to give the small fraction of CPU to Each running task is carried out by schedulers. Implementing scheduling algorithm is considered as a most difficult task because of dynamic priority tasks such as interactive task like Web browser which are called as higher-priority tasks and low-priority tasks such as non-interactive batch processes like program compilation. Also, scheduler has to protect against the starvation in low-priority tasks.

Operating system scheduler mainly concerned with different criteria [5]-[7] for evolution of scheduler performance:

1. Utilization – Defined as the amount of time the device is in use.
2. Throughput – it is the total number of processes completed per unit time.
3. Response Time – it is the amount of time taken by process to give first response after submitting the request.
4. Fairness – Defined as allocating equal amount of CPU time to each process.

Multitasking operating systems are of two types: preemptive and cooperative. Linux and most modern operating systems provide preemptive multitasking [6]. In preemptive multitasking, the scheduler decides when to stop the execution of running process and when to resume the new process to complete its execution.

B. Linux kernel Scheduler

The key component of Operating system (OS) is Kernel [1]-[5]. The kernel is piece of code which is responsible for resource management by receiving requests for resources, granting the request and servicing several interrupts[6][7].Linux is open-source , multiprogramming OS. The Linux was earlier developed as desktop OS, as it is evolved further it can be uses as servers, mainframe computers, and embedded devices. Linux scheduler provides the multiprogramming, multiprocessing, user responsiveness and overall Fairness. For Linux scheduler[7] each process and thread is task and it mainly deals with tasks. The scheduler goes on evolving as the kernel developed and modified [8][9]. Earlier the Linux scheduler was not focused on complex architecture with multiple processors.

3. LITERATURE SURVEY

The Linux scheduler 1.2 was very simple [7][10] and faster one. It used the round robin scheduling policy[11] and managed the runnable task using circular queue. This scheduler was efficient for adding and deleting the processes in queue. The version 2.2 of Linux scheduler[11][12] support for symmetric multiprocessing (SMP) by introducing new concepts such as enabling scheduling policies for real-time tasks, non-real-time tasks and scheduling classes. The 2.4 Linux kernel version has introduced the scheduler which is better than the previous schedulers. This scheduler is named as O(N) [15] because it used the algorithm with O(n) complexity which iterated over every task during each single scheduling event. This scheduler divided total CPU time into epoch,

and each epoch is assigned to every task which was permitted to complete its execution up to its assigned time slice. If the executing task did not complete its execution in given epoch, then the remaining time slice was added to the new time slice which was allowed to execute the task and complete its execution in the next epoch. In such a way this scheduler iterates over the tasks by deciding which task to schedule next on CPU. Even though this scheduler is simpler than previous schedulers but it is cannot handle multiple tasks simultaneously in case of multiple runnable processes.

Linux kernel 2.4 designed the new scheduler called as O(1) to overcome the limitations of O(N) scheduler by removing the ability of the scheduler to iterate over all tasks in the system to identify next task for scheduling [14][17]. O(1) scheduler is designed in such a way that, instead of going through all the process in system it consist of two running queues to identify the next task to execute. First running queue keep the tract of all active processes which are ready to execute, while second running queue consist of the list of all expired processes. Whenever there is need to select the next task to schedule, O (1) scheduler selects the task from queue that consist of active processes. Due to this unique functionality the O(1) scheduler is more scalable and efficient than the O (N) scheduler[17]. O(1) can also determine whether the task is CPU-bound or I/O-bound. But this scheduler becomes unmanageable in the kernel due to its huge code.

In the paper [10] author C.S. Wong, R.D. Kumari, and J.W. Lam has compared the two Linux kernel scheduler such as O (1) and Completely fair scheduler (CFS) in terms of fair sharing policy and interactive performance. In Linux kernel 2.6 O(1) scheduler is used while in 2.6.23 uses the CFS. O(1) replaced by CFS. Design goals of CFS are to provide the fair amount of CPU among all runnable tasks without immolating their interactive performance. Both scheduler shares some characteristics in terms of fairness and interactive performance. Author has measured these design goals by using benchmarks that measures the system performance in terms of throughput. The results from the test conclude that the CFS is more fairer than O(1) in case of CPU bandwidth distribution and interactive performance.

In paper[11] author J. Wei, R.Ren, Juarez, F. Pescador provides the Energy based Fair Queuing scheduling algorithm(EFQ) which consume the energy on many devices. EFQ algorithm can achieve proportional sharing of power by consuming power on both CPU and I/O tasks based on their energy consumptions. Author also proposed that sharing of power of specific application can be protected by EFQ which is not achieved by CFS. So, the CFS is extended by adding the new scheduling policy SCHED_EFQ. Four variables are included to the structure such as sched_entity and struct cfs_rq and initial weight, reserved share, energy packet size and warp parameters are added. Then the Linux nice values are calculated from 40 to 100. The nice values ranges from [-20 to 19] which is further modified to [-50 to 40].

Many research studies have been carried out on the Fairness of CFS in Linux Kernel. However few studies are focuses on the frequency scaling scheme using CFS. To achieve the CFS enabled Frequency Scaling scheme there is need to prove the fairness of CFS and also to compare the performance of CFS with earlier schedulers.

4. COMPLETELY FAIR SCHEDULER (CFS)

The latest Linux kernel scheduler is Completely Fair scheduler (CFS)[7][9][14] which was introduced in Linux Kernel 2.6.23 and extended in 2.6.24. CFS is "Desktop" process scheduler which was implemented by Ingo Molnar. Its core design can be summed up in single sentence: "CFS basically models an 'ideal, precise multitasking CPU' on real hardware [7][14]." This scheduler's key role is to eliminate the unfairness from the system by allocating a fair amount of CPU to each runnable process. In CFS the 'ideal, precise multitasking CPU' means the CPU which runs multiple processes concurrently by dividing the power of processor (Fair share of processing time) among all runnable processes. That means if single process is running in the system then it will get 100%

CPU's power,if there are two runnable processes then each process will execute on 50% of processor's power in this way the multiple runnable processes can execute simultaneously by sharing the fair amount of CPU. It is impossible to get the ideal CPU in reality, but the CFS tries to imitate such ideal processor in system [16]. for scheduling the process CFS uses the process priority and timeslice [11][17]. timeslice is defined as the total amount of time taken by process to run and the process which is having the large timeslice is considered as higher priority process. The nice value given to each process according to user's perspective determines the priority of process. The proportion of the time that any processor receives is determined by the difference between the nice values of runnable process and the nice value of process itself.

The CFS Scheduler supports the following scheduling policies:

SCHED_NORMAL/SCHED_OTHER: It is used for regular tasks.

SCHED_FIFO: Uses the First-In-First- Out Scheduling Policy

SCHED_BATCH: It is used for running the tasks for longer time without preempting it .

SCHED_IDLE: it is used to avoid the tasks to get into the priority.

SCHED_RR: Similar to SCHED_FIFO, but uses the Round Robin scheduling algorithm.

In CFS, processes (tasks) are given processing time, when time for any task is out of balance (if tasks have not given the fair amount of CPU) as compared to other task, then those out of balance tasks should given the processing time to execute, in this way CFS maintains the Fairness [19]. So, to determine the balance among multiple tasks CFS introduces the concept of "virtual runtime (vruntime)" [17]. Virtual runtime defines the total amount of time provided to given task. The task which is having small virtual time means it has higher priority and will schedule first. This scheduler also maintains the fairness for those processes which are waiting for I/O events to occur. Instead of maintaining these processes in run queue, the Completely Fair Scheduler maintains the time order Red-Black tree (RBTree) in order to decide the task to schedule next on processor. CFS maintains the time order RBTree. Red-black tree [17][19] is generally the self balancing search tree which has following features:

1. Each node in the RBtree is either black or red.
2. Every leaf node is black.
3. If node is red then it means that both its children are black.
4. Every simple path from node to leaf node contains the equal number of black nodes.

The benefits of using RBtree in CFS are:

1. It is self Balancing tree, which means that there is no path from the root to leaf node is more than twice as long as any other.
2. For searching the RBTree takes O(log n) time.

The Figure 1. Shows the Red-black tree and each node in tree is task/process in the system and key value of the node represent the virtual runtime of the specific task. According to the definition of Red-black tree, left most node has smallest key value, which means that this task has highest priority with smallest virtual runtime and vice versa. Hence CFS has to take left most tasks for processing and once the task is processed then it is removed from tree.

The virtual runtime can be considered as a weighted time slice, which is represented by following equation-[11][12]

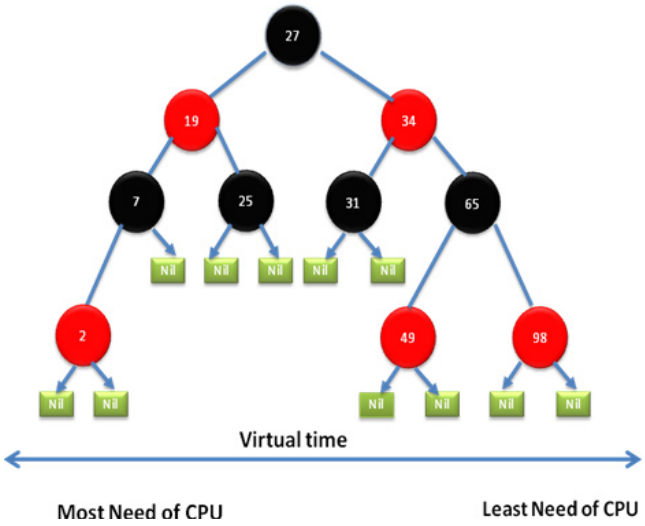

Figure 1: RBTree for CFS

$$\text{virtualruntime} \mathrel{+}= \frac{(\text{delta\_exec})(\text{default weight of process})}{\text{se(load.weight)}} \tag{1}$$

From the equation (1) of virtualruntime , delta_exec is the total amount of execution time of task, default weight of process means the unit value of weight and load.weight is weight of task/entity[19]. The weight of runnable processes is decided by their priority. By assigning the proper weights to processes the CFS maintains the fairness [17][18]. In CFS share is calculated as:

$$\text{share} = \frac{\text{weight schedulable entity}}{\text{total weight of all entities in CFS runqueue}} \tag{2}$$

Time slice is defined as:

$$\text{Timeslice} = \text{share} \times \text{period} \tag{3}$$

in equation (3), the period is the total time slice that is used by scheduler for all tasks. The minimum period is 20ms. To calculate the priorities of tasks, CFS uses task weight represented by load.weight and divides runtime by tasks weight which is saved as vruntime in a RBtree.

On an ideal hardware only single task can run at once. So each task gets a particular time slice. The weight of a task depends upon the nice level which ranges from –20 to +19 and the scheduling policy. The time slice for a process becomes smaller as the load increases. The time slices of the range [–20…0 … 19] are mapped with an array of time [800ms….100ms…5ms]. The total task at any given time are considered and the fraction time for each task is calculated by equation (4).

$$\text{load} - \text{weight ratio}(i) = \frac{\text{time\_slice}(i)}{\sum \text{V}_j \ \text{time\_slice}(j)} \tag{4}$$

Table 1 above represents Virtual Runtime for each process is equal which proves the fairness property of CFS. For simplicity, the Period of execution is taken as 100 ms. Each value of nice is mapped with its equivalent load.The total load weight for each runqueue process is calculated by calculating the load-weight ratio. The reference load weight, it the inverse load weight consider the value of 1024 and finally the vruntime is computed as the product of period to reference load weight. The equal vruntime in table shows that CFS achieves fairness among all runnable processes.

Table 1
Virtual runtime

Priority	*Load Weight*	*Load Weight Ratio*	*Period(100ms)*	*Reference Load Weight Ratio*	*Vruntime*
–20	88761	0.67198	67.198	0.012	0.7752
–15	29154	0.22071	22.071	0.035	0.7752
–10	9548	0.07228	7.228	0.107	0.7752
–5	3121	0.02363	2.363	0.328	0.7752
0	1024	0.00775	0.775	1.000	0.7752
5	335	0.00254	0.254	3.057	0.7752
10	110	0.00083	0.083	9.309	0.7752
15	36	0.00027	0.027	28.444	0.7752
	132089	1	100.000		

5. CONCLUSION

The paper is about the survey of fairness and performance analysis of Completely Fair scheduler is defined by calculating the vruntime of all tasks in CFS runqueue. To define the fairness of CFS it is compared with other Linux scheduler theoretically and it is concluded that the CFS maintains the fairness and gives better performance than earlier Linux scheduler such as O(1) and O(N). As Embedded processors are getting more numerous and complex day by day. So, power management is the biggest issue in embedded systems. To maintain the performance of embedded system analysis of frequency change is an essential task. Hence the further research will be carried out on CFS scheduler-driven DVFS scheme to maintain the performance of embedded system. Also, the Response-time Estimation will be carried out by running the Compute-Intensive Task in Scheduler-driven DVFS scheme.

Acknowledgment

The proposed survey paper on "Fairness and Performance Analysis of Completely Fair Scheduler in Linux Kernel" has been prepared by Pooja Tanaji Patil under the guidance of Prof. Sunita Dhotre.

REFERENCES

[1] A. Silberschatz, P.B. Galvin, G. Gagne, "Operating System Concepts," 7th Edition, John Wiley & Sons Inc.,2005

[2] Richard Petersen, "The Complete Reference" Linux, Second Edition, Tata McGraw Hill.

[3] Daniel P. Bovet & Marco Cesati". Understanding the Linux Kernel, OReilly October 2000

[4] J. Lozi, J. Funston, F. Gaud, V. Qu, and A. Fedorova, "The Linux Scheduler : a Decade of Wasted Cores."

[5] S. M. Mostafa, H. Amano, and S. Kusakabe, "FAIRNESS AND HIGH PERFORMANCE FOR TASKS IN GENERAL PURPOSE MULTICORE SYSTEMS," Vol. 29, No. December, pp. 74–86, 2016.

[6] Wenbo Wu, Xinyu Yao, Wei Feng, Yong Chen, "Research on Improving Fairness of Linux Scheduler", Proceeding of the IEEE International Conference on Information and Automation, China, pp. 409-414,August 2013

[7] "Completely Fair Scheduler _ Linux Journal." http://www.linuxjournal.com/magazine/completely-fair-scheduler.

[8] "Tuning the Task Scheduler _ System Analysis and Tuning Guide _ openSUSE Leap 42." https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html.

[9] G. Cheng, "A Comparison of Two Linux Schedulers," Master thesis, pp. 1–89, 2012.

[10] C. S. Wong, R. D. Kumari, and J. W. Lam, "Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers," No. 1, 2008.

[11] J. Wei, R.Ren, Juarez, F. Pescador, "A linux Implementation of the Energy-Based Fair Queuing Scheduling Algorithm for Batter Limited Mobile Sysems", IEEE Transactions on Consumer Electronics, Vol. 60, No. 2, pp. 267 – 275, May 2014.

[12] S. Wang, "Fairness and Interactivity of Three CPU Schedulers in Linux," pp. 2–7, 2009.

[13] "Inside the Linux 2." https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/.

[14] C. S. Wong, I. Tan, and R. Deena, "Towards Achieving Fairness in the Linux Scheduler," pp. 34–43.

[15] Yigui Luo, Bolin Wu, "A Comparison on Interactivity of Three Linux Schedulers in Embedded System", Communications, Computers and Signal Processing(PacRim), 2011 IEEE Pacific Rim Conference, pp. 494-498, August 2011.

[16] Wei-feng MA, WANG Jia-hai, " Analysis of the Linux 2.6 Kernel Scheduler" 2010 IEEE International conference on computer Design and Applications, pp.71-74, 2010

[17] Prajakta Pawar, SS Dhotre, Suhas Patil, "CFS for Addressing CPU Resources in Multi-Core Processors with AA Tree", International Journal of Computer Science and Information Technologies, Vol. 5 (1), 913-917, 2014

[18] J. Kobus and R. Szklarski, "Completely Fair Scheduler and its tuning," pp. 1–8, 2009.

[19] Poonam Karande, SS Dhotre, Suhas Patil, "Illustration of Task Scheduling in Heterogeneous Quad-Core Processors", International Journal of Engineering and Technology Research, Vol 03, Issue 08, Pages:1389-1393, May 2014

[20] Dilipkumar, Vora Shivani, M. Tech, and S. S. Dhotre. "Runtime CPU Scheduler Customization Framework for Real Time Operating System."

[21] Kabugade, Rohan R., S. S. Dhotre, and S. H. Patil. "A Modified O (1) Algorithm for Real Time Task in Operating System."