ISSN:0975-9646

CFS for Addressing CPU Resources in Multi-Core Processors with AA Tree

Prajakta Pawar, S.S.Dhotre, Suhas Patil

Computer Engineering Department
BVDUCOE Pune-43(India)

***Abstract*- CPU scheduler plays crucial role in operating system as scheduling is its primary job. The design of operating system scheduler is proposed to hand out its resources accurately among applications. The main goal of multi-core systems is load balancing across cores. Thus methods are employed to set tasks on cores try to balance runnable tasks across available resources. All this is made to ensure fair distribution of CPU and minimize the idling of core. Current multiprocessing operating systems like Linux use a scheduling approach to enable efficient resource sharing. The Completely Fair Scheduler (CFS) design of Linux ensures equal opportunity among tasks using thread fair scheduling algorithm. This research work discuss about strategies used by CFS of Linux. In this paper we will talk about CFS algorithm in detail. Also this paper proposes a new technique for CFS which can be implemented using AA Tree. We wrap up this paper highlighting the usefulness of proposed system and the future work in certain direction.**

***Keywords*- Operating system, Linux, Process scheduling, Completely fair scheduler, Interactivity, Fairness.**

I. INTRODUCTION

The operating system is software that acts as an interface between computer hardware and its user. The principal aim of OS is to make computer system convenient to make use of and to use hardware resources in proficient manner. A CPU scheduling is basis of multiprogrammed operating systems. The aim of multiprogramming is to have some task running all times to exploit CPU utilization. The operating system know how to make the computer system more productive by switching the CPU among processes [7]. Main goal of multicore system is load balancing across cores. Thus some strategies are employed to place threads on cores which aim to balance runnable threads across available resources. This ensures fair distribution of CPU time and minimizes the idling of cores. Modern multiprocessing Operating systems like Linux 2.6 use two level scheduling approaches to enable efficient resource sharing. First level uses a distributed run queue model with per core queues and fair scheduling policies to manage each core. Further at second level it makes use of a load balancer which redistributes tasks across available cores [17].

A short-term scheduler is a type of scheduler that decides which of the ready and in-memory processes are to be owed a CPU after some interrupt in system. This scheduler can be preemptive or non-preemptive. It is one of the core components of a multitasking operating system such as Linux which is responsible for optimum utilizing system resources to guarantee that several processes are being executed simultaneously. Linux is a principal operating system being developed in the open source community. Over few years because of increasing number of Linux users, the CPU scheduler in Linux kernel has been improved to enhance its performance. And lots of good schedulers have been implemented by the Linux kernel developers. Furthermore they are extensively revised to achieve performance enhancement in terms of interactivity, fairness and scalability [8].

The Linux 1.0 used simple linked list of runnable processes and scheduler decision O (N). Then Linux 2.0 included SMP support .Next to it Linux 2.5 uses O (1) scheduler. There come Linux 2.6.23 CFS with many improvements [16]. The O (1) and CFS are most popular Linux schedulers. In Linux the default scheduler used is Completely Fair Scheduler which was combined into mainline Linux version 2.6.23. This paper presents detailing of CFS and its work flow using data structures red black tree and AA tree.

This paper contributes following aspects:

- To understand the Linux CFS in detail using usual method of Red black tree.
- To implement AA Tree for entity management in CFS instead of RB Tree.

The rest of this paper is organized as follows: In Section 2 analysis of various scheduling approaches is presented under background heading. The section 3 discuss about CFS in more detail. Next section 4 proposes alternative approach for implementation of CFS with alternative implementation method. Section 5 presents algorithm for proposed system workflow. Lastly a conclusion is made in section 6.

II. BACKGROUND

In Linux scheduler 1.2 used a circular queue for run able task management which operated with a round-robin scheduling strategy. With this scheduler it was efficient to add and remove tasks. Also the scheduler wasn't complex, instead was simple and fast. The next Linux version 2.2 introduced the design of scheduling types, presenting scheduling policies for real-time tasks, non-real-time tasks and non-preemptive tasks. The scheduler had support for symmetric multiprocessing (SMP) too. The former scheduler 2.4 used O (N) scheduler which operated in O (N) time as it iterated over each process during a scheduling action. This scheduler separated time into epochs and in every epoch, each process was allowed to perform up to its time slice. When a process did not use its entire time slice, at that time half of the enduring time slice was added to the new time slice to let it execute longer in the next epoch. It uses a goodness function as metric to determine which process to execute next. In spite of the simplicity of this approach, it was relatively inefficient, lacked scalability, and was pathetic for real-time systems. This one also failed to exploit new hardware architectures such as multi-core processors. A Linux 2.6, O (1) scheduler solved many of the problems with the 2.4 scheduler and it is not iterated the entire task list to identify the process to schedule next. The O (1) scheduler used a two run queues to keep track of runnable processes, one for active and other for expired processes . The scheduler basically dequeue the next task off the active per-priority run queue to recognize the process to execute next. Thus O (1) scheduler was much more scalable and efficient. But in the O (1), large group of code required to estimate heuristics which was not easy.

Thus to solve the issues in the O (1) scheduler something required to change. So, Con Kolivas came with a kernel patch, with Rotating Staircase Deadline Scheduler (RSDL) included his earlier work on the staircase scheduler and it included fairness with bounded latency. Then based around some of the thoughts from Kolivas' work, Ingo Molnar developed the CFS with fairness in CPU bandwidth allocation among processes and better interactivity [9].

III. COMPLETELY FAIR SCHEDULER

The previous O (1) Scheduler required one priority array for RT and non RT tasks and decision algorithm based on the position in the priority array. With O (1) it is difficult to calculate CPU share and not easy to achieve fairness. To treat interactive tasks in O (1), special heuristics are required. CFS Scheduler is absolute new design written by Ingo Molnar is the descendant of the O (1) scheduler in Linux [10]. Notable things about CFS are: (1) CFS is free of heuristics. (2) Fairness algorithm is straightforward mathematics. (3) Extendible framework of CFS, that makes it easy to set up new scheduler algorithms or even a pluggable scheduler implementation [18].

General principle of this scheduler is to offer maximum fairness to each task in system in terms of computational power it is given. The CFS design ensures fairness among processes using the thread fair scheduling algorithm, which does the allocation of resources based on the number of threads in the system instead number of threads in executing programs.Unlike other schedulers and preceding Linux implementations, CFS does not maintain any array with runques for each level [16]. Instead, it maintains the time ordered red black tree. The distinct goals that CFS is designed to bring about are: (1) CFS shall make available finer interactive performance even as maximizing overall CPU utilization.(2) To make sure balance in allowance of CPU time to every entity. (3)To pick up the efficiency, by removal of components like array of runqueue, an interactive processes identification and heuristics estimation. (4)The entire scheduler is implemented utilizing the modular scheduler or group scheduler framework by introducing Scheduling Classes. CFS uses seperate queue mechanisms and scheduler decision functions for RT and non RT tasks (scheduler classes).

A. Basic CFS Algorithm

In multicore systems where primary goal is to fairly distribute workload across available cores. Thus to balance runnable threads across available resources, threads are mapped to cores and stored in core's run queue. For each core a respective run queue is created. CFS does not requires concept of time slice while it only considers waiting time of task and task with highest need of CPU time is always scheduled next. This is why CFS is called completely fair.

The CFS was designed to offer higher interactive performance while keeping high overall CPU utilization. So without sacrificing the interactive performance it tries to provide fairness in each task. This is possible by using proportional share algorithm in which a share is assigned to each process and it is related with the weight of the task [1] [2]. Ingo Molnar describes the original design of CFS which can be stated in single statement as "CFS models an ideal, precise and multitasking CPU on real hardware". This mean that CFS tries to follow such CPU that can run numerous processes in parallel while giving each process equal share of CPU power instead equal share of CPU time. According to this statement when a single process is running, it receives 100% CPU power and if two processes running, each would receive 50% CPU power. In the same way, if four processes are running then each would get 25% of CPU power all together. In this manner CPU would be fair to all processes running in the system, but in reality such ideal CPU is nonexistent, but CFS tries to look for such processor in software. On genuine real processor only one process can be assigned at particular time and all other processes are waiting during this period which is not fair because presently running task gets 100% of the CPU power while all other remaining tasks get 0% of the CPU power. To remove such unfairness from a system, CFS keeps track of equal share of the CPU that would have been available to each process in system. CFS tracks the amount of time a process waits for the CPU over the ideal processor and uses this wait time to rank the processes for scheduling. The process having extensive wait time is considered to have gravest need of CPU and it is allotted to CPU. When this selected process is running, its wait time decreases and eventually the time for other process increases. Thus after some time there will be some another process with largest wait time and the presently executing process will be pre-empted. Using this principle, CFS tries to be fair to all processes and all the time tries to have system having zero wait time for every process. Thus every process has a fair share of CPU.

B. Run Queue

When many processes run at the same time in a system, all active processes are positioned in an array called a run queue. Given several threads mapped to a core, with the help of CFS runqueue management component this scheduler decides which thread from run queue will run next. A run queue possibly will hold priority values for each task and it will be used by the scheduler to find out which process to run next. This way it acts as a fundamental data structure in the scheduler implementation. The CFS algorithm needs following things to imitate "idle, precise and multitasking CPU": (1) A way to work out what is the fair CPU share per task and it is achieved by using run queue variable (cfs_rq→fair_clock). (2)Also a behaviour to keep track of time for which each task was waiting when CPU was allotted to currently running task and wait time is gathered in wait_runtime variable (process→wait_runtime) [11]. A run queue for Linux is defined inside kernel/sched.c as struct rq [12].

C. Fair Distribution of CPU Bandwidth

To distribute CPU power, each task is assigned a weight which establishes the share of CPU bandwidth that tasks will receive. The share given to a process is ratio of its weight to sum of weight of all active processes in runqueue. Following expression is used for this:

$$share = \frac{\text{schd\_entity→load.wt}}{\text{cfs\_rq→load.wt}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(Eq1)}$$

Where,
Schd_entity->load.wt is weight of schedulable entity and cfs_rq->load.wt is total weight of all entities under ruqueue of CFS.
The time slice that process should get is given by,

$$slice = \frac{\text{schd\_entity→load.wt}}{\text{cfs\_rq→load.wt}} \times \text{ period}\ldots\ldots\ldots\ldots\ldots \text{(Eq2)}$$

Where,
period is time slice the scheduler tries to execute all tasks. The time slice received by every task is not a constant and it is dependent to period, which has assumed a minimum value of 20ms. Also it is required to prevent unnecessary scheduling when the number of processes is much larger than the number of CPU in the system. Virtual runtime is used by CFS to track progress of every entity and it is weighted time slice given to every schedulable entity which is expressed using equation:

$$vir\_run = \frac{\text{delta\_exect}}{\text{schd\_entity} \rightarrow \text{load.wt}} \text{ NICE\_0\_load} \quad \ldots\ldots\ldots\ldots\ldots \text{(Eq3)}$$

Where,
delta_exect is execution time of process and NICE_0_load is unity value of weight (1024) [9], [10].

D. CFS Scheduler Classes

CFS includes expandable hierarchical set of scheduler classes as: (1)rt_sched_class that handles FIFO and RR tasks with O (1) priority array. (2) fair_sched_class which handles tasks other than real time tasks with O (log(N)) red black tree.(3) idle_sched_class that handles idle task [17].

CFS fair_sched_class
Its working is analogous to "fair queuing" for packet networks where red black tree for task management keeps a virtual timeline of tasks to schedule. The scheduler decision is O (1) and reinsertion of a task is O (log (N)) nanoseconds based accounting. Here task with the longest wait time in the red black tree is selected next. The nice levels are not depending on the timeslice and they are multiplicative. Also sleep time of interactive task is privileged.

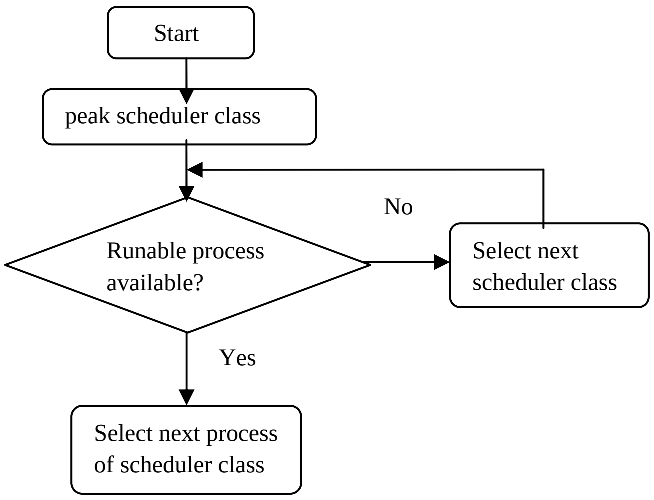
Start
peak scheduler class
Runable process available?
No
Select next scheduler class
Yes
Select next process of scheduler class

Figure 1: CFS Scheduler decision

IV. PROPOSED SYSTEM

This section presents a proposed approach for CFS that uses the CFS for scheduling and so it includes all the features of existing CFS algorithm along with some supplementary data structure implementation [3],[4]. The analysis results from the interactivity and fairness tests proves that the CFS has the advantage of being fairer in CPU bandwidth allotment without compromising interactivity performance a lot [5]. The CFS uses wait_runtime to rank the processes as well as to find out amount of time for which process is permitted to execute ahead of being preempted [6]. This paper presents the new proposal of using new maintenance algorithm for balanced tree structures called AA Tree. The scheme suggested here is an ongoing project which implements CFS in both ways, by using red black tree as well as AA tree. There after implementing it both ways, finally it compares results of the two. Following system flow diagram depicts the work flow of proposed system.

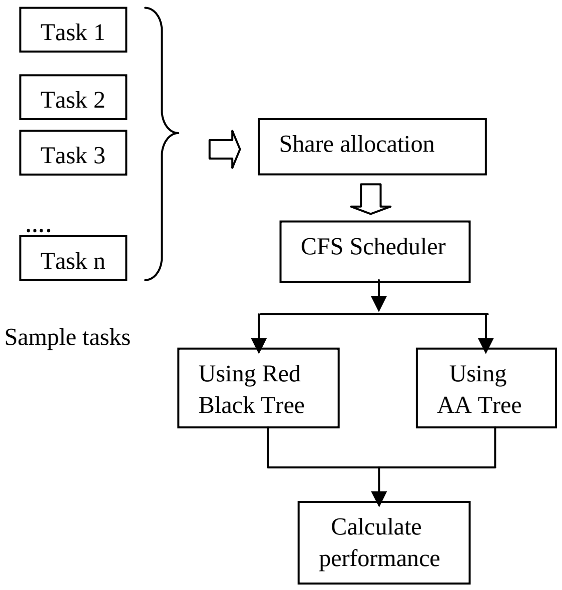

Figure 2: Workflow of proposed system

A. *Red Black Tree*

Red black tree data structure is a form of self balancing binary search tree which is used in CFS to sort runnable tasks. Leftmost leaf in this tree has smallest value and larger value is in right child. A leftmost node in Red black tree represents a node having gravest need of CPU, thus CFS selects the leftmost process. The leaf nodes in red black tree do not contain data [13]. As the system progresses forward, newly awakened processes are set in the tree beyond and farther to the right gradually but at the same time for sure it is giving every process an opportunity to become leftmost process [11].

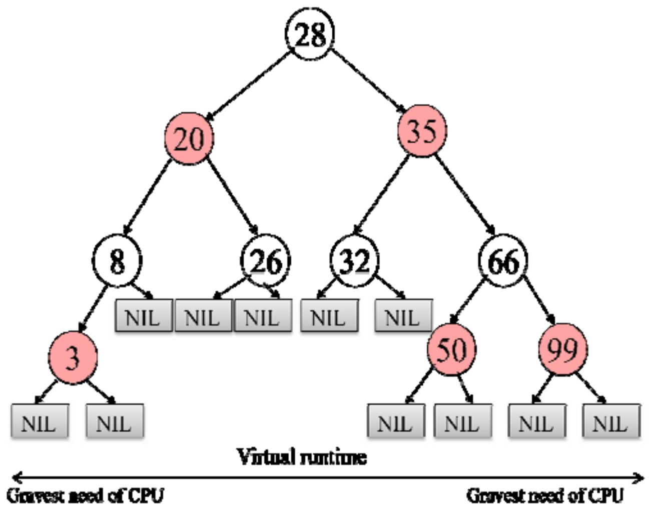

Figure 3: Example of Red black tree

B. *AA Tree*

In red black trees the implementation and number of rotation cases is complex as compared to AA tree. AA trees are named after its inventor Arne Andersson and it's an optimization over original definition of Binary B-trees. AA-trees has fewer rotation cases so easier to code, particularly deletions eliminates about half of the rotation cases. AA-trees still have O (log n) searches in the worst case.

Contrasting to red-black trees, the red nodes on an AA tree can be added only as a right sub child. Thus, no red node can be a left sub-child, which results in the imitation of a 2-3 tree instead of a 2-3-4 tree. It helps to simplify the maintenance operations. And for a red-black tree this maintenance algorithms need to consider seven different shapes to properly balance the tree, while AA tree on contrast only desires to consider two shapes due to the strict condition that only right links can be red [14],[15].

The red-Black tree has high complexity as compared to AA tree data structure which helped for modification in the CFS approach used by implementing AA tree instead of Red-Black Tree. In AA Tree instead of color the level of a node is used as balancing information. Red nodes are simply nodes that located at the same level as their parents. The level of a node in an AA-tree is: (1) Leaf nodes are at level 1. (2) Red nodes are at the level of their parent. (3) Black nodes are at one less than the level of their parent.
The AA tree fulfills the properties of Red-Black trees along with one addition to it:

- The color of each node is either red or black.
- The root is black.
- If a color of node is red then its children have to be black.
- All paths from any node to a descendent leaf must contain the same number of black nodes.
- Left children may not be red.

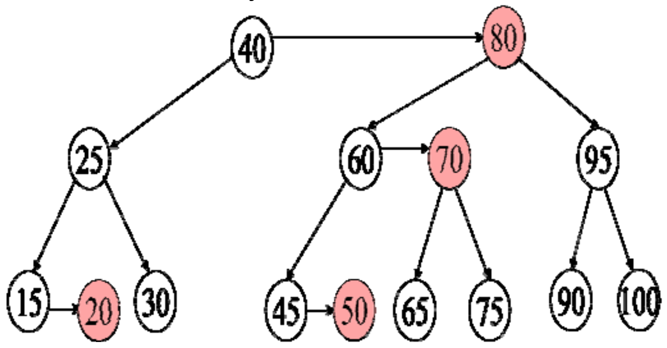

Figure 4: Example of AA Tree

This way AA trees help to simplify the algorithms as:
1. This removes half the restructuring cases
2. This also simplifies deletion eliminating an annoying case
   - Whenever it has only one child at internal node then that child must be a red right child
   - One can always replace a node with the smallest child in the right sub tree

V. CFS Algorithm Using RB Tree And AA Tree

The steps of proposed system are as follows.

- Create the sample tasks which will run in the background. Code to get that task for scheduling and assign the weights for each task.
- Find out the share allotment for each task. Find out the time-slice that a task should receive in a period of time. Also find out the virtual runtime for every schedulable task.
- Initially use red-black tree data structure to track all the runnable tasks. And fairly assign the CPU for each task running.
- Compute the performance in terms of equality and interactivity of the CFS scheduler.
- Now using realization of AA tree as an alternative of Red-Black tree in CFS and calculate the performance in terms of equality and interactivity of the CFS scheduler.
- Finally compare the results of both approaches for CFS.

VI. Conclusion

This paper has explored various CPU scheduling techniques. The paper talks about the most popular Linux CPU schedulers, O (1) and CFS. It also discusses how to achieve interactivity and fairness in CFS. This is explained by implementation of fairly divided time slice given to every task and the nanoseconds accurate accounting. It shows that how CFS algorithm is more efficient than O (1) scheduler as CFS does not require a complex algorithm to identify interactive tasks. As a contribution work, this paper proposed alternative method called implementation of AA tree for CFS. This proposed work is an ongoing project which implements red black tree as well as AA tree for CFS and compares the performance of the two methods.

REFERENCES

[1] I. Stoica et. al., "A Proportional Share Resource Allocation Algorithm for Real-time and Time-shared Systems," *in Proc. 17th IEEE Real-Time Systems Symp., Dec.1996.*

[2] Steve Goddard Jian Tang, "EEVDF Proportional Share Resource Allocation Revisited".

[3] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, Fun Wey "Towards Achieving Fairness in the Linux Scheduler".

[4] C.S. Wong, I.K.T. Tan, R.D. Kumari, J.W. Lam W. Fun, "Fairness and Interactive Performance of O(1) and CFS Linux Kernel Schedulers", *IEEE 2008.*

[5] MA Wei-feng, Wang lia-hai, "Analysis of the Linux 2.6 kernel scheduler", *International Conference on Computer Design and Applications (ICCDA 2010).*

[6] Shen Wang, Yu Chen Wei Jiang Peng Li Ting Dai Yan Cui, "Fairness and Interactivity of Three CPU Schedulers in Linux", *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications 2009.*

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

[8] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas 2005 Silicon Graphics, Inc. (SGI) 17th February 2005.

[9] Jones, T.: Inside the Linux 2.6 Completely Fair Scheduler_ Copyright IBM Corporation (2009).

[10] Molnar, I. 2007, Modular Scheduler Core and Completely Fair Scheduler [CFS], http://lwn.net/Articles/230501/

[11] http://www.linuxjournal.com/magazine/completely-fair-scheduler

[12] The Linux Kernel Archives Website: www.kernel.org

[13] http://en.wikipedia.org/wiki/Red%E2%80%93black_tree.

[14] http://en.wikipedia.org/wiki/AA_tree

[15] http://www.cepis.org/upgrade/files/full-2004-V.pdf

[16] http://web.eecs.umich.edu/~sugih/courses/eecs281/f11/lectures/12-AAtrees+Treaps.pdf

[17] Sergey zhuravlev, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors", *ACM Computing Surveys, Vol. V, No. N, September 2011, Pages 1–31.*

[18] https://www.linuxfoundation.jp/jp.../seminar20080709/lfjp2008.pdf