iJRASET
International Journal For Research in Applied Science and Engineering Technology



# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

**Volume:** 12 **Issue:** XII **Month of publication:** December 2024

**DOI:** https://doi.org/10.22214/ijraset.2024.65690

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
*ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538*
*Volume 12 Issue XII Dec 2024- Available at www.ijraset.com*

# Enhancing CFS Scheduler Performance through Binomial Heap Integration

Rajat Dhanure[1], Vaibhav Bhosale[2], Yash Ghotekar[3], Ojus Kulkarni [4], Dipti Pandit[5], Anup Ingle[6]
*Electronics and Telecommunications Enginnering Dept., Vishwakarma Institute of Information Technology, Pune*

***Abstract:*** ***This paper proposes an enhancement to the Completely Fair Scheduler (CFS) in Linux by replacing the Red-Black Tree (RBT) with a Binomial Heap to improve scheduling efficiency. While CFS provides fair CPU time allocation, its reliance on RBT can lead to inefficiencies as the number of tasks grows, particularly in insertion and extraction operations. The Binomial Heap, with its constant time complexity for insertion operation makes it a suitable alternative. Performance evaluations demonstrate that this modification decreases the time required for insertion by up to 50% and for extraction up to 12%. In this paper, this modification is implemented and evaluated demonstrating that the Binomial Heap improves system performance in managing tasks within CFS.***

***Keywords: Operating System, Completely Fair Scheduler, Binomial Heap, Red-Black Trees, Scheduling Algorithms, Linux***

## I. INTRODUCTION

Operating Systems (OS) play a crucial role in managing system resources, including CPU time. To ensure fair and efficient utilization of the CPU, various scheduling algorithms have been developed. Modern operating systems are designed to maximize hardware efficiency by managing resources like CPU and memory through virtualization techniques. CPU virtualization is achieved by sharing the CPU between the tasks, such that each gets a small fraction of time to execute on CPU[1]. The core component responsible for managing this distribution of CPU time is the scheduler. A key requirement for any scheduling algorithm is the ability to efficiently manage the ready queue, which contains processes waiting for CPU time. Traditional data structures like linked lists or arrays can become inefficient as the number of processes increases. In multitasking operating systems like Linux, an efficient scheduler is crucial for ensuring that processes are executed fairly and system resources are utilized optimally.

The history of task scheduling in Linux dates back to its initial release in 1991, which had a simple Round-Robin (RR) approach for scheduling processes[4]. As the kernel evolved, significant developments occurred, which includes introduction of scheduling classes in Linux Kernel 2.2, allowing for distinguish between real-time and normal tasks[2]. The O(n) scheduler was introduced in Linux Kernel 2.4, enhancing performance by using a queue-based method to select processes based on priority. This was followed by the O (1) scheduler in Kernel 2.6, which improved efficiency by ensuring constant-time scheduling regardless of the number of tasks. In the kernel version 2.6.23, a totally new scheduler is introduced to replace O(1), called CFS [2]. It was implemented by Ingo Molnar, aiming to provide fair CPU time allocation among all tasks using a red-black tree and maintaining a concept of virtual runtime to gauge execution time. While CFS shows improvements, optimizing its reliance on the Red-Black Tree for managing tasks presents additional opportunities for enhancement[1]. particularly in operations involving insertion and deletion. These operations can affect the scheduler's efficiency as the number of tasks grows. To address this issue, It can be done by improving CFS by substituting the Red-Black Tree with a Binomial Heap. This paper is structured as follows: Section 2 gives a detailed look at CFS and its basic structure, Section 3 explains the features and benefits of the Binomial Heap along with how to implement the proposed method, and Section 4 offers a comparison of performance metrics between the Red-Black Tree and the Binomial Heap in CFS, concluding with our final thoughts.

## II. COMPLETELY FAIR SCHEDULER

The Completely Fair Scheduler (CFS) was presented in Linux bit adaptation 2.6.23 and is planned to distribute CPU time decently to all working forms. It varies from conventional forms by taking the concept of time openings and utilizing a more productive strategy based on virtual working time. Fairness is another important design goal of CPU schedulers. More fairness in terms of CPU bandwidth allocation among tasks means each tasks will be proportionally advancing forward when executed[2]. As the execution time of the assignment increases whereas it is running, CFS can plan the assignments with less work. CFS also supports group scheduling, which was introduced to handle scenarios where multiple users or groups are competing for CPU time. Group scheduling ensures that CPU time is distributed fairly between groups rather than just individual tasks. CFS supports symmetrical multiprocessing (SMP) in which any process (whether kernel or user) can execute on any processor[6].

©IJRASET: All Rights are Reserved 

### A. *Working of CFS*

All processes and threads within Linux are represented by a structure called task struct[4]. It which contains important details like the task's name, process ID (PID), priority, and more. To keep track of scheduling information, a new structure was designed, called sched entity, which is included in the task struct structure [3].

When a new task is created, it gets added to the Red-Black Tree according to its vruntime. The Completely Fair Scheduler (CFS) keeps track of how long each task has been running and updates the tree, making sure that tasks with lower vruntime are on the left. The scheduler selects the task with the lowest vruntime, which is the leftmost node in the tree, to run next. After that task finishes, its CPU time is added to its vruntime, and it is placed back into the tree. This process allows every task to have a fair opportunity to use the CPU[4].

As the task runs, its vruntime increases. If another task enters the system or becomes runnable and has a lower vruntime than the currently running task, then the current task is pre-empted and the new task is scheduled to run. When it's time to switch tasks, the scheduler's schedule() function interrupts the current task[3]. The task's vruntime is continuously updated as it runs, and CFS repositions it in the Red-Black Tree once it is finished. This ensures that the system always schedules tasks with the least runtime first, maintaining fairness. On multi-processor systems, CFS includes load-balancing mechanisms to ensure that tasks are evenly distributed across all CPUs, preventing some CPUs from becoming overloaded while others remain idle.

### B. *Red-Black Trees*

A red black tree is a variety of binary search tree which is perfectly self-balanced. It has been dedicated its name to Rudolf Bayer and Jeff Stolf regarding its inventors. Some properties of RB trees states as follows[5]:

*1)* A node is always coloured black or red.
*2)* The colour of the root node and all leaf nodes is only black.
*3)* Red nodes must never have a red child but may have black ones.
*4)* Every simple path (not including any circles) from any node down to any of the leaf nodes has the same number of black nodes.

The balance in RB trees is maintained by the use of the uniform distribution of the number of black nodes on the path from the root to the bottom leaf. Such property is called as black-height property[5]. While inserting or deleting operations are performed the RB tree moves to optimal state by rotations of nodes acquired during the initial step.

Some operations on RB trees involve the following processes:

*a)* *Insertion:* A new leaf node added to the Red-Black Tree is initially colored red. However, if the adjacent node to which the new node is linked is also red, then the tree must apply the black-header law or perform a rotation and color change to maintain the properties of the Red-Black Tree.
*b)* *Deletion:* In a Red-Black Tree, replacement nodes or leaves are colored red when removing a node. If the parent of the replacement node is also red, the algorithm must perform a rotation or color change to maintain the black-height properties of the tree.
*c)* *Search:* The search operation in an RB tree is comparable to the search performed in a general Binary Tree. The root is searched first and then recurses into the children nodes sequentially towards a certain direction that correlates to the key being sought.
*d)* *Min and Max:* To identify the minimum and maximum nodes contained in an RB tree, one must trace the left child pointers to a leaf node for the minimum and traverse the right child nodes until reaching a leaf node.

## III. PROPOSED METHODOLOGY

In this methodology, the time complexity of operations in the Red-Black Tree and Binomial Heap is compared, followed by the implementation of the Completely Fair Scheduler (CFS) using a Binomial Heap in the Linux Kernel.

### A. *Binomial Heap*

A binomial heap is a type of data structure that extends binary heaps and makes merging two heaps more efficient. It's made up of multiple binomial trees, and each of these trees follows a specific structure. A binomial tree of order k contains exactly 2k nodes and has a height of k. These trees are made by connecting two smaller binomial trees of order k−1[7]. To form a binomial tree of height k, one complete binomial tree of height k−1 is added as the leftmost child to another tree. Each binomial tree in the heap follows the min-heap property, meaning each node's value is at least as large as that of its parent, ensuring the smallest value is at the root[1].

In any binomial heap with n nodes, the maximum number of binomial trees is log(n) + 1, since the structure of binomial trees is based on powers of two..

1) *Insertion:* Insertion works by creating a new binomial heap with the single element and then merging it with the existing heap. It takes constant time on average (O(1)).
2) *GetMin:* The find minimum operation involves checking the roots of all binomial trees and returning the smallest key, which takes O(log n) time due to the logarithmic number of trees.
3) *Merge:* Merging two binomial heaps is similar to the process of adding binary numbers. Trees of the same order are linked together in a process that's similar to carrying over in binary addition. This operation has a time complexity of O(log n).
4) *Delete:* Deleting the minimum element involves finding the tree with the smallest root, removing this root, breaking its children into separate binomial trees, and then merging these trees back with the remaining heap. This also takes O(log n) time.

*B. Implementation*

A Binomial Heap-based process scheduling simulation has been implemented in C on Windows. First, a Task structure representing a process was defined, with fields such as task_id and virtual_runtime given as input. Next, the Binomial Heap was created using structs and pointers to manage tasks, supporting operations like insert, extractMin, and delete. In the simulation, processes (tasks) were created and inserted into the heap, and the scheduler simulated task execution by repeatedly extracting the task with the smallest virtual runtime. The C program was compiled using MinGW on Windows, allowing for testing and observation of how tasks are scheduled based on their virtual runtime without needing access to the underlying OS scheduler. The output revealed a sequence of how the inserted tasks were executed, demonstrating that the task with the lowest virtual runtime was executed first.

Implementing a Binomial Heap-based Completely Fair Scheduler (CFS) in Linux requires modifications to the existing Linux kernel. Set up a kernel development environment by installing essential tools like build-essential, ncurses-dev, and libelf-dev. Download the Linux kernel source from kernel.org or clone the repository using git. Study the current CFS implementation in kernel/sched/fair.c, which uses a Red-Black Tree for task management.

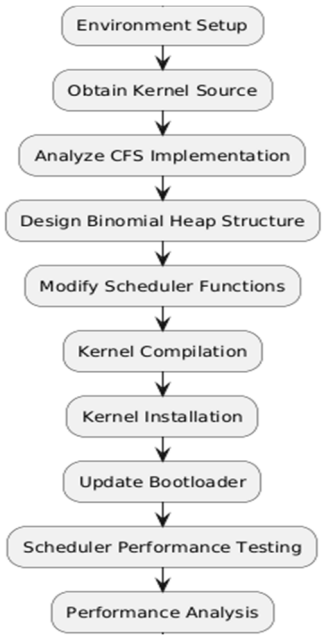



Figure 1: Flowchart

Replace this Red-Black Tree with a Binomial Heap by defining the heap structure and implementing key functions such as insert, extract_min, and delete. Modify core functions like enqueue_task_fair(), dequeue_task_fair(), and pick_next_task_fair() to incorporate Binomial Heap operations. Once the changes are complete, compile the kernel using make -j$(nproc) to utilize all CPU cores, and install it with make modules_install and make install. After installation, update the bootloader with update-grub and reboot into the modified kernel. Finally, test the custom scheduler using stress-testing tools like sysbench or stress-ng to evaluate the performance of the new Binomial Heap-based scheduler.

## IV. RESULTS

The implementation of the Completely Fair Scheduler (CFS) using a Binomial Heap instead of a Red-Black Tree (RB-Tree) was conducted to analyze the performance and efficiency of task scheduling operations. The primary focus was on three key operations: insertion, deletion, and extraction of the minimum task based on virtual runtime. Empirical runtime measurements were taken during the simulation, revealing significant performance improvements in time complexity and runtime performance when using the Binomial Heap. In terms of time complexity, the Red-Black Tree exhibits logarithmic complexity for all operations: insertion, extraction of minimum, and deletion, each taking O(log n). Conversely, the Binomial Heap offers an amortized constant time complexity of O(1) for insertion, while extraction of the minimum and deletion still operating at O(log n). This fundamental difference in insertion complexity provided a advantage in scenarios with frequent task creation. TABLE I. summarizes the complexity comparison of both the data structures.

Table 1. Time Complexity Comparision

| ***Operations*** | ***RB Trees*** | ***Binomial Heap*** |
|---|---|---|
| Insertion | O(log n) | O(1) Amortized |
| Deletion | O(log n) | O(log n) |
| GetMin | O(log n) | O(log n) |

The results highlighted the effectiveness of the Binomial Heap in reducing operation times. For n=1000 tasks, the average insertion time for the Red-Black Tree was approximately 0.000355 seconds, while the Binomial Heap averaged around 0.000150 seconds. This translates to a reduction of approximately 57% in insertion time when using the Binomial Heap. Similarly, the average time for extracting the minimum task from the Red-Black Tree was about 0.000108 seconds, compared to 0.000095 seconds for the Binomial Heap, resulting in a reduction of around 12%. Previous results show that insertion in a Red-Black Tree takes 304.79 ns compared to 107.90 ns in a Binomial Heap, and ExtractMin takes 104.52 ns in a Red-Black Tree versus 98.25 ns in a Binomial Heap. Simulation in C was utilized to evaluate how well the operations of both data structures perform.. Table II presents the runtime performance of both data structures. The findings indicate that the Binomial Heap can greatly enhance scheduling in Linux-based systems that use CFS.

Table 2. Runtime performance comparision

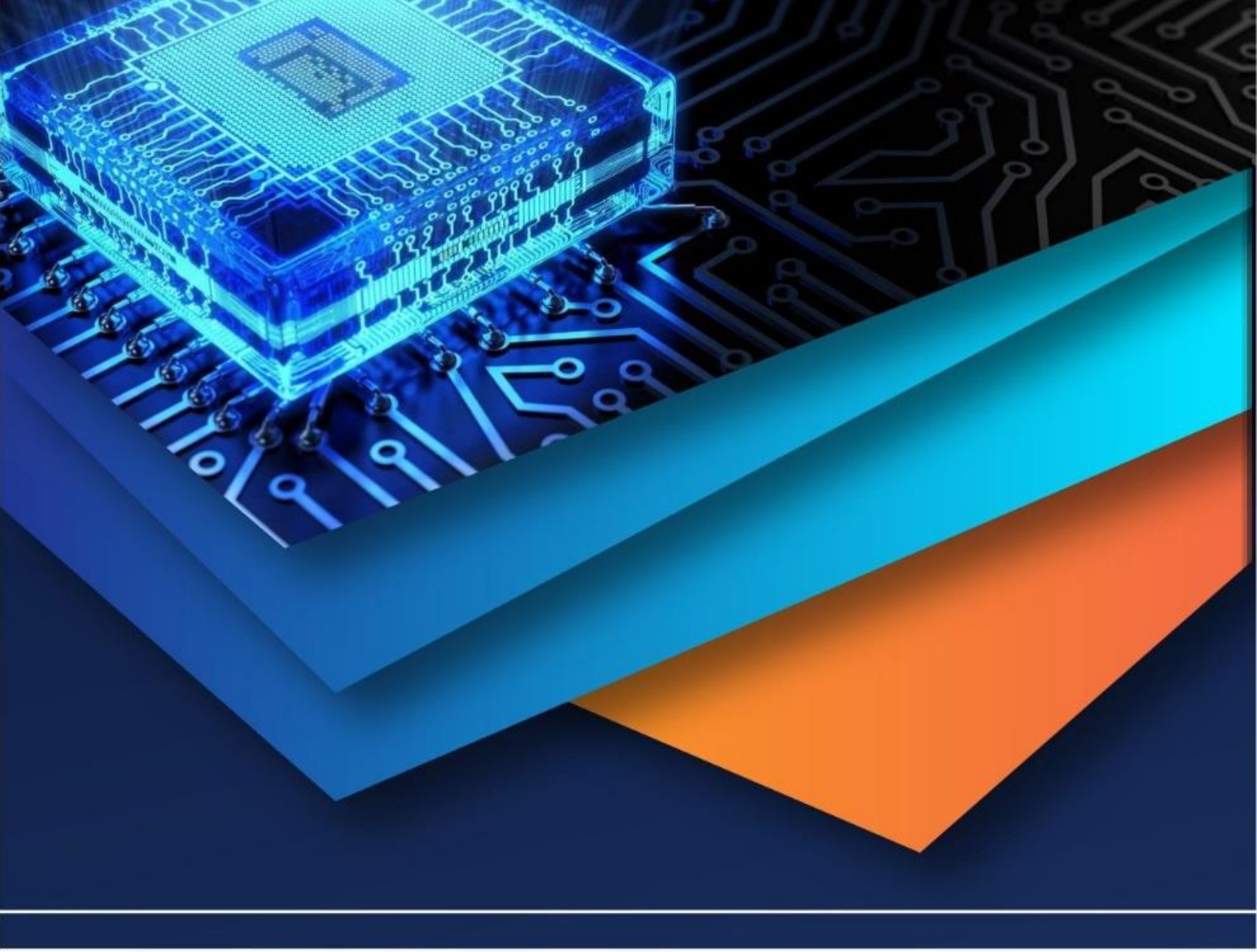
| ***Operations*** | ***RB Trees*** | ***Binomial Heap*** |
|---|---|---|
| Insertion | 355 us | 150 us |
| GetMin | 108 us | 95 us |

## V. CONCLUSION

Overall, the switch to a Binomial Heap for managing task scheduling in the CFS led to faster insertions and efficient minimum extractions. The ability to insert tasks in constant amortized time significantly enhanced the scheduler's responsiveness, especially in environments where task creation is frequent. The efficient handling of task insertions by the Binomial Heap allows the scheduler to maintain high performance under various workloads, which is crucial for real-time applications. Looking ahead, there are many opportunities for future research. One potential area of research is the optimization of the Binomial Heap operations further, particularly focusing on concurrent modifications to enhance performance in multi-threaded environments. Furthermore, investigating hybrid approaches that combine the strengths of both the Binomial Heap and other data structures may yield even better results. By continuing to improve and update scheduling algorithms, the Linux kernel can stay strong and efficient, meeting the needs of today’s computing environments.

## REFERENCES

[1] S. Singh and P. Kumar, "CFS performance improvement using Binomial Heap," 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 2015, pp. 1822-1824, doi: 10.1109/ICACCI.2015.7275881.
[2] C. S. Wong, 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," 2008 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 2008, pp. 1-8, doi: 10.1109/ITSIM.2008.4631872.
[3] CFS Scheduler — The Linux Kernel documentation https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, accessed October 2024
[4] Process Scheduling https://developer.ibm.com/tutorials/l-completely-fair-scheduler/, accessed October 2024
[5] Red Black Trees https://en.wikipedia.org/wiki/Red%E2%80%93black_tree, accessed October 2024
[6] https://opensource.com/article/19/2/fair-scheduling-linux, accessed October 2024
[7] HINZE R. Explaining binomial heaps. Journal of Functional Programming. 1999;9(1):93-104. doi:10.1017/S0956796899003317



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)