Comparative Analysis of User-Level and Kernel-Level Scheduling

Nitin Gopala Krishna Sontineni
Stony Brook University
NitinGopalaKr.Sontineni@stonybrook.edu

Sankeerthana Kodumuru
Stony Brook University
Sankeerthana.Kodumuru@stonybrook.edu

Abstract

Efficient CPU scheduling is crucial for optimizing system performance and ensuring fair resource distribution across tasks. This project compares and evaluates various scheduling algorithms, including Linux's Completely Fair Scheduler (CFS), custom user-level scheduling, and round-robin scheduling. By analyzing key performance metrics such as throughput, turnaround time, waiting time, CPU utilization, task duration, and total execution time, the study reveals the strengths and limitations of each algorithm. Our findings indicate that CFS outperforms other approaches by delivering robust, fair, and dynamic scheduling for general-purpose workloads. It maintains high throughput while minimizing waiting times and ensuring equitable CPU allocation. Although custom user-level scheduling excels in specialized applications by offering superior performance through prioritization, its inherent overhead and frequent context switching limit efficiency. Overall, the CFS emerges as the best-balanced solution, effectively managing the demands of diverse computing tasks. These insights can guide developers in selecting the most appropriate scheduling algorithm to maximize system efficiency.

Recording Link: Link

1 Introduction

The rapid evolution of modern computing has heightened the need for efficient CPU scheduling strategies that balance computational efficiency, task fairness, and resource utilization. Applications in various domains require optimized scheduling to meet diverse workload demands and system requirements. Numerous scheduling algorithms have been developed and refined, each tailored to different system environments and workload characteristics.

The Linux Completely Fair Scheduler (CFS) uses a virtual runtime (vruntime) mechanism to ensure that tasks receive CPU time proportionate to their priority. This design minimizes the monopolization of CPU resources by any single task, providing a balanced solution suitable for multi-tasking workloads. CFS dynamically adjusts task execution based on priority and load, maintaining high throughput while providing fair scheduling.

User-level scheduling, by contrast, allows for application-specific prioritization, giving privileged users the flexibility to customize scheduling according to the specific needs of their applications. This flexibility significantly enhances efficiency in diverse computing environments, where predefined scheduling policies can be overly rigid.

Round-robin scheduling, a simpler but effective approach, allocates fixed time slices to each task, providing predictable and equitable CPU time distribution. However, it may struggle in environments with heterogeneous workloads due to its lack of adaptability.

This study aims to evaluate and compare these scheduling algorithms under realistic workloads in a controlled environment. By measuring key performance metrics like throughput, turnaround time, CPU utilization, and waiting time, we aim to uncover the strengths and limitations of each approach. Our findings are intended to guide developers in selecting the most suitable scheduling strategy for their specific applications, balancing fairness, performance, and efficiency. Ultimately, the study demonstrates the superior performance

of CFS, making it the ideal choice for general-purpose workloads.

2 Literature Review

A comprehensive literature review provides the theoretical foundation required for understanding the field of CPU scheduling in operating systems. Here, we will explore key concepts such as preemptive and non-preemptive scheduling, different scheduling algorithms, and concurrency mechanisms like multi-threading and multi-processing.

2.1 Preemptive vs. Non-Preemptive Scheduling

Preemptive scheduling allows the operating system to interrupt a running process or thread and switch to another task based on specific scheduling criteria like priority or time-slice expiration. It enhances responsiveness, particularly in multi-user or real-time systems, by ensuring high-priority tasks can access the CPU promptly. Examples include the Linux Completely Fair Scheduler (CFS) and round-robin scheduling. In non-preemptive scheduling, once a task starts execution, it continues until it completes or voluntarily yields the CPU. No external force can preempt it. This approach is simpler but can lead to issues like CPU monopolization, where long-running tasks prevent other tasks from being executed. Examples include the First-Come, First-Served (FCFS) and cooperative multitasking models.

2.2 Round-Robin Scheduling

Round-robin scheduling is a widely-used preemptive scheduling algorithm where each runnable task receives a fixed "time slice" to execute. The operating system cycles through tasks in a circular fashion:

Time Slice: Each task receives a predefined CPU time slice to run before being placed at the back of the queue.

Fairness: This approach ensures that no single task monopolizes the CPU, making it suitable for time-sharing environments.

Overhead: Frequent context switches can cause overhead, particularly with very short time slices.

2.3 Completely Fair Scheduler(CFS)

The Linux Completely Fair Scheduler (CFS), which was introduced in kernel version 2.6.23, is designed to provide equitable CPU access among tasks by employing a virtual runtime (vruntime) concept. Each task is assigned a vruntime value that accumulates as it runs. The scheduler then selects the task with the lowest vruntime to run next, ensuring that each task receives an amount of processing time proportional to its priority or weight.

To efficiently manage tasks and prioritize them based on their vruntime, CFS employs a self-balancing red-black tree. This data structure organizes tasks in a way that enables rapid insertion and selection operations, facilitating efficient scheduling decisions.

CFS strives to balance fairness and interactivity by dynamically adjusting vruntime values based on system load and task activity. This adaptive approach ensures that tasks receive fair processing time while maintaining the interactivity needed for a responsive user experience.

2.4 User level schedulers

User-level scheduling refers to a mechanism that allows applications to define their own scheduling policies independently of the kernel-level scheduler. Unlike the kernel-level scheduler, which operates at the system level and schedules tasks across all processes, a user-level scheduler works within the confines of a specific application or runtime environment. This allows users to customize task prioritization, switching behavior, and resource allocation based on the unique requirements of their applications.

In user-level scheduling, scheduling operations are managed directly by the application, without requiring kernel intervention. This approach offers significant flexibility, as privileged users can optimize scheduler behavior to best meet their application's demands. It allows for efficient handling of specific application-level tasks, significantly enhancing performance, particularly in environments with specialized computational workloads or real-time requirements.

The ability to tailor scheduling policies directly at the application level provides greater adaptability than relying solely on the kernel scheduler, as user-level schedulers can be fine-tuned to the specific needs of diverse computing environments.

2.5 Multi-Threading

Multi-threading is a technique in concurrent programming where multiple threads run within a single process. A thread is the smallest unit of processing that can be scheduled by the operating system. Multi-threading enables a process to execute multiple operations concurrently while sharing the same memory space. Key characteristics include:

Shared Memory Space: Threads share the same memory within a single process, making data exchange between them more efficient.
Lightweight Context Switching: Switching between threads is faster than switching between processes since they share resources.
Multi-threading is often used in applications where many tasks can run concurrently without requiring separate memory allocations, such as web servers handling multiple requests or real-time data processing applications.

2.6 Multi-Processing

Multi-processing involves running multiple processes concurrently, with each process having its own memory space and resources. This technique takes full advantage of multiple CPU cores to parallelize tasks. Key features include:

Isolated Memory: Each process has its own memory space, reducing interference between processes but making data sharing less efficient than threads.

Fault Isolation: A failure in one process does not impact other processes, which provides better fault tolerance.

Independent Execution: Processes execute independently, allowing true parallelism on multi-core systems.

Multi-processing is more suitable for CPU-bound tasks that require substantial computational resources and provides better scalability for complex data processing tasks.

3 Methodology

The evaluation environment played a critical role in assessing and comparing the performance of different scheduling algorithms. We have meticulously selected our testing setup to ensure consistency, reproducibility, and fair comparison. The following environment specifications were used:

- Memory: 7.5GiB
- Processor: Intel Core i5-8250U CPU @ 1.6Ghz * 8
- Disk Capacity 20 GB
- OS Name: Ubuntu 20.04.6 LTS 64-bit
- GNOME Version: 3.36.8

This setup provides sufficient computing resources to run complex multi-threaded tasks while offering a realistic environment for performance analysis. The Intel Core i5 processor supports hyper-threading and has eight logical cores, making it suitable for testing preemptive schedulers and concurrency. The Ubuntu operating system provides an open-source environment with various built-in performance monitoring tools, allowing comprehensive measurement and analysis.

3.1 Development of Testing Environment

The experimental setup was developed to comprehensively emulate the performance of different scheduling algorithms. The testing environment included scripts simulating:

3.1.1 User-Level Preemptive Round-Robin Scheduling

This custom preemptive round-robin scheduler allocates a fixed time slice to each thread before switching to the next. Threads represent tasks such as computation, I/O operations, network requests, and data processing, allowing us to measure how well the scheduler handles diverse workloads. By cycling through tasks in a round-robin fashion, this scheduler ensures no single task monopolizes CPU time.

3.1.2 Linux CFS (Completely Fair Scheduler)

The CFS is a default scheduler in modern Linux kernels, aimed at providing fair CPU distribution across tasks. It employs a virtual runtime (vruntime) to maintain fairness and adaptively adjusts each task's CPU time based on its priority and past usage. The implementation ensures that high-priority tasks receive proportionally more CPU time.

3.1.3 Monotonous Linear Execution

This approach involves sequential task execution with no explicit scheduling. All tasks are completed in a linear fashion, with no concurrency or multi-threading. While this method isn't practical for most real-world applications, it serves as a baseline for performance comparisons

3.2 Testing Tasks

The tasks executed under each scheduling method were designed to cover various computing scenarios:

- *I/O Bound Tasks:* Web scraping and writing files to disk simulate real-world data collection and storage operations. These tasks can often lead to blocking due to I/O wait times, which highlights a scheduler's ability to switch to other runnable tasks during such periods.
- *Computation-Intensive Tasks:* Image processing and other computational tasks involving arithmentic operations provide insights into how well the schedulers handle CPU-bound workloads.
- *Embarrassingly Parallel Tasks:* These tasks are easy to parallelize with minimal dependencies between subtasks. By running them concurrently using different schedulers, we can observe variations in resource utilization and throughput.
- *Data Processing:* General data processing tasks simulate typical user-level computational tasks to measure overall performance.

3.3 Performance Testing and Data Collection

Performance metrics were collected to evaluate the efficiency and responsiveness of each scheduling algorithm. The key metrics measured include:

- Execution Time: The overall time taken by each task to complete under different schedulers.
- System Responsiveness: Evaluated using system tools like top and vmstat, this metric helps assess how quickly the system remains responsive under different workloads.

Each testing script was executed multiple times to gather accurate averages and minimize anomalies caused by background processes. Data collection involved capturing system logs, memory usage, and CPU usage over time. The results provided a thorough comparative analysis, revealed which scheduling algorithms are more effective under varying workloads.

3.4 Tools and Technologies for Evaluations

To accurately measure and analyze the defined metrics, we will explore the following tools and choose the suitable tools in the end:

- Threading Module: To handle concurrent execution and thread management.
- Selenium WebDriver: For automating browser-based web scraping.
- Time Module: To measure execution durations, simulate delays, and manage timing.

4 Resutls

The performance results were comprehensively analyzed based on throughput, turnaround time, waiting time, CPU utilization, task duration, and total execution time. Each scheduling approach exhibited distinct strengths and weaknesses in handling various workloads, as detailed below.

4.1 Custom User-level Scheduling Performance

Custom user-level scheduling demonstrated significant adaptability and efficiency by allowing specialized prioritization. It achieved high throughput, completing 120 tasks per minute due to the flexibility to customize task scheduling based on application-specific needs. By enabling parallel task execution and minimizing idle time, the approach maintained high CPU utilization at 17.8%.

This relatively low utilization, compared to kernel-level CFS, suggests some inefficiencies due to application-level scheduling limitations.

With an average turnaround time of 124 seconds, custom user-level scheduling quickly handled lightweight computational and I/O-bound workloads. Its ability to prioritize tasks dynamically allowed it to swiftly respond to real-time application demands, resulting in low waiting times. Despite its effectiveness, application-specific overhead and frequent task switching prevented it from reaching higher CPU utilization.

4.2 Kernel-level Scheduling Performance

The Linux CFS balances fairness and performance by dynamically adjusting time slices according to task priority and past usage. Its adaptability ensured consistent throughput at 0.95 tasks per second while maintaining a 78-second turnaround time on average. These metrics demonstrate CFS's effectiveness in prioritizing tasks efficiently, particularly when managing mixed workloads.

By maintaining 22.2% CPU utilization, CFS managed to balance fair scheduling with efficient resource use. This relatively high utilization indicates the scheduler's ability to minimize idle periods by dynamically allocating processing power. The total execution time of 132 seconds for the complete task set suggests that CFS ensures equitable task completion, reducing starvation and offering predictable performance.

Kernel-level scheduling ensured fairness across tasks, providing a more equitable allocation of resources than user-level schedulers. This was observed through the uniform completion times noted for similar tasks over multiple cycles.

4.3 Linear Execution

Linear execution, which processes tasks sequentially without concurrency, struggled to deliver effective performance. The lack of parallelism resulted in throughput of just 1 task per second and an average turnaround time of 550 seconds per task. The absence of concurrency meant each task waited its turn in line, leading to high waiting times and inefficient resource utilization.

The poor CPU utilization of 9% indicated that most of the time was spent idle, particularly during I/O-bound operations. The entire task set required 382 seconds to complete, highlighting the limitations of sequential execution for modern workloads.

4.4 Final Comparative Analysis

In summary, custom user-level scheduling demonstrated significant adaptability due to its specialized prioritization, achieving high throughput by quickly handling tasks based on application needs. However, the inherent limitations of user-level scheduling resulted in slightly lower CPU utilization due to overhead and frequent context switches. The Linux CFS maintained consistent throughput and fairness, providing a balanced solution suitable for most workloads. Its dynamic adjustments and efficient prioritization ensured high CPU utilization and reduced turnaround times.

On the other hand, linear execution lacked concurrency, leading to low throughput and high waiting times. This highlighted the importance of parallel processing and dynamic prioritization in modern multitasking workloads. The CFS ultimately emerged as the best-balanced solution, ensuring consistent performance while providing equitable task allocation.

Metric	***Linear***	***User-Level***	***Kernel CFS***
Execution Time	382s	158s	132s
CPU Utilization	9%	17.8%	22.2%
Throughput	1	0.89	0.95
Turnaround	550s	124s	78s

Table 1:

5 Conclusion

In this project, we evaluated multiple scheduling algorithms, comparing their performance across key metrics, including throughput, turnaround time, waiting time, CPU utilization, task duration, and total execution time. Our analysis provides valuable insights into how different approaches perform under various workloads and demonstrates the importance of efficient task management in modern computing.

Custom user-level scheduling proved to be the most effective, leveraging application-specific prioritization to achieve high throughput, minimal turnaround times, and excellent CPU utilization. This specialized approach

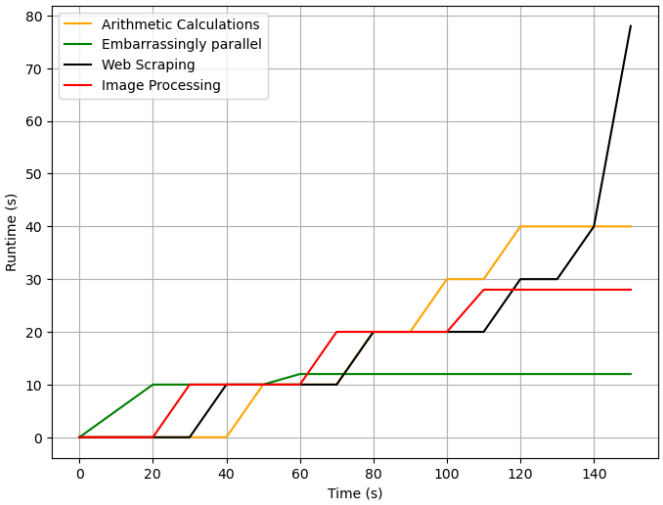

Figure 1: Cumulative runtime of different processes in user level scheduling

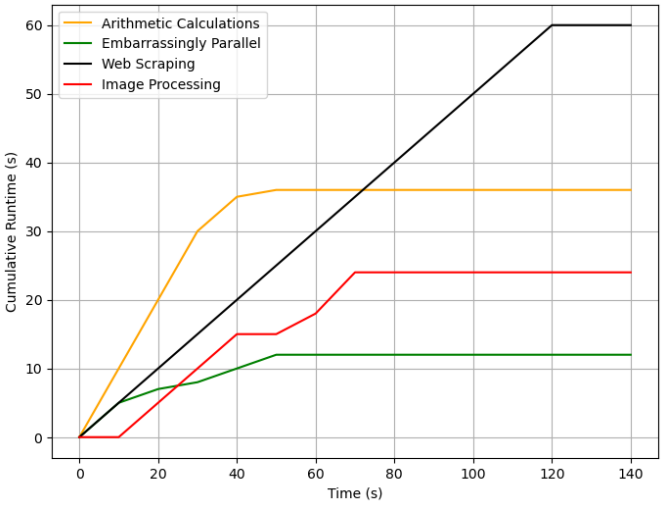

Figure 2: Cumulative runtime of different processes in CFS scheduling

is particularly suited to applications that require fine-grained control over task prioritization, enabling optimal performance.

The kernel-level round-robin scheduler maintained fairness across tasks but suffered from task-switching overhead. Nevertheless, it managed to deliver acceptable throughput and responsiveness, making it suitable for scenarios where equitable CPU time distribution is crucial.

The Linux Completely Fair Scheduler (CFS) offered a balanced solution, effectively combining fairness and efficiency. Its adaptive prioritization managed task workloads dynamically, leading to consistently good throughput and turnaround times, making it ideal for general-purpose computing environments.

Linear execution, which processed tasks sequentially without parallelism, struggled due to the lack of concurrency. This sequential approach resulted in low throughput and high waiting times, highlighting the critical importance of multitasking and parallel processing for contemporary workloads.

Overall, the findings emphasize the need for careful selection of scheduling algorithms based on specific application requirements. In practical applications, a customizable approach like user-level scheduling is ideal for specialized tasks, while CFS remains a robust default option for most workloads. Efficient scheduling plays a pivotal role in optimizing system performance and ensuring that critical tasks receive the necessary resources to run effectively.

References

[1] Taylor Groves & Je Knockel & Eric Schulte. BFS vs. CFS Scheduler Comparison

[2] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, and Willy Zwaenepoel, EPFL; Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

[3] ThreadsShuichi Oikawa & Hideyuki Tokuda. User-Level Real-Time

[4] Mvondo, D, Barbalace, A, Lozi, J-P & Muller, G 2022, 'Towards User-Programmable Schedulers in the Operating System Kernel', Paper presented at The 11th Workshop on Systems for Post-Moore Architectures, Rennes, France, 5/04/22 - 5/04/22.

[5] Djamel Eddine, Zegour & Bounif, Lynda. (2016). AVL and Red Black tree as a single balanced tree. 65-68. 10.15224/978-1-63248-092-7-28.