Scheduling Algorithms Today

Modern operating systems employ a range of process scheduling algorithms to efficiently manage CPU usage and ensure smooth execution of concurrent processes. These scheduling algorithms differ in their approach to task prioritization, responsiveness, and resource allocation, but each aims to maximize CPU utilization and minimize waiting and turnaround times.

One widely used algorithm in contemporary operating systems is First-Come, First-Served (FCFS). This non-preemptive scheduling method executes processes in the order they arrive in the ready queue. While simple and fair in its approach, FCFS can lead to the "convoy effect," where short processes are delayed by longer ones, ultimately increasing average waiting time (Scaler, n.d.).

Another fundamental algorithm is Shortest Job Next (SJN), also known as Shortest Job First (SJF). This algorithm selects the process with the smallest estimated running time. It significantly reduces average waiting time compared to FCFS. However, the primary limitation of SJF lies in accurately predicting the duration of processes. Moreover, it can lead to starvation of longer tasks if shorter ones keep entering the queue (Scaler, n.d.).

To address fairness and responsiveness, many systems use Round Robin (RR) scheduling. This preemptive algorithm assigns each process a fixed time slice or quantum. If a process doesn't finish within its time slice, it is moved to the end of the queue. Round Robin is particularly effective in time-sharing systems as it ensures that all processes receive CPU attention at regular intervals, thus improving response time (Scaler, n.d.).

In systems that require prioritization, the Priority Scheduling algorithm is employed. Each process is assigned a priority level, and the CPU executes the process with the highest priority next. This can be implemented in both preemptive and non-preemptive forms. While efficient in prioritizing critical tasks, this approach can also suffer from starvation of low-priority processes. Techniques like aging, which gradually increases the priority of waiting processes, are used to mitigate this issue (Scaler, n.d.).

More advanced operating systems, especially those managing complex enterprise workloads, often adopt Multilevel Queue Scheduling and Multilevel Feedback Queue Scheduling. The multilevel queue approach separates processes into different queues based on priority or process type (e.g., interactive vs. batch), each with its own scheduling algorithm. The feedback version allows processes to move between queues based on their behavior and requirements, adapting to workload dynamics and improving responsiveness (Scaler, n.d.).

From an enterprise job scheduling perspective, the Fair Share algorithm is gaining traction. It allocates CPU time based not just on process priority but also on resource consumption history, ensuring equitable distribution across users and applications (Redwood, n.d.). Additionally, algorithms like Backfill Scheduling, commonly used in grid or cluster

environments, improve CPU utilization by allowing smaller jobs to "fill in" available CPU slots as long as they don't delay higher-priority tasks (Redwood, n.d.).

Research in CPU Scheduling

Recent research in CPU scheduling focuses on two main areas: (1) optimizing traditional scheduling algorithms for better performance in general computing systems, and (2) developing specialized algorithms suited for unique environments like data centers. Both directions reflect ongoing efforts to improve system responsiveness, reduce latency, and enhance CPU utilization.

One example of optimizing existing scheduling techniques is the study by Dash et al., which proposes the Dynamic Average Burst Round Robin (DABRR) algorithm. Traditional Round Robin (RR) scheduling suffers from performance inefficiencies when the time quantum is poorly chosen—too small and it causes excessive context switches; too large and it becomes close to FCFS. DABRR addresses this by dynamically calculating the time quantum as the average of the burst times of all processes in the ready queue, rather than using a fixed time slice. The algorithm re-evaluates this average after each scheduling round. Experimental results demonstrate that DABRR significantly reduces average waiting time, turnaround time, and context switches compared to traditional RR and its variants. This research underscores a trend in CPU scheduling studies where the algorithm's behavior is adapted dynamically based on workload characteristics to improve fairness and throughput in conventional time-shared environments (Dash et al., 2015)

In contrast, Rikos et al. focus on scheduling in data center environments, where unique challenges such as scale, heterogeneity, and real-time requirements necessitate novel approaches. They propose a finite-time distributed coordination mechanism for task scheduling that ensures optimal CPU allocation across server nodes. Unlike traditional centralized solutions, their algorithm enables each server node to communicate only with its neighbors, exchanging quantized (discrete) values and performing event-triggered updates. This design improves scalability and reduces communication overhead. The key innovation lies in its guaranteed convergence to the optimal solution in a finite number of steps, along with a distributed stopping mechanism that allows nodes to determine independently when to terminate computation. Simulation results reveal that the algorithm effectively balances CPU workloads across thousands of nodes, making it suitable for cloud infrastructures and high-performance computing environments (Rikos et al., 2021)
.

Research in Operating Systems

Operating systems research continues to evolve to address the increasing demands of modern computing. Two prominent areas of focus are improving memory tiering performance and enhancing concurrency testing within the kernel.

One key research area involves optimizing how operating systems manage memory across different tiers. Modern systems often use a mix of memory technologies, such as DRAM and Non-Volatile Memory (NVM), each with varying speeds and costs. "Memory tiering systems aim to achieve memory scaling by adding multiple tiers of memory wherein different tiers have different access latencies and bandwidth" (Kanellis et al., 2025). The challenge lies in efficiently placing data, keeping frequently accessed ("hot") data in faster tiers and less frequently accessed ("cold") data in slower tiers. Existing systems often rely on heuristics and pre-configured thresholds, which "fail to adapt to different workloads and the underlying hardware, so perform sub-optimally" (Kanellis et al. 2025). Current research seeks to improve this by using application behavior knowledge to dynamically adjust system parameters, or "knobs." For example, researchers are exploring the use of Bayesian Optimization to discover high-performing configurations that can adapt to specific application needs and hardware characteristics, potentially leading to significant performance improvements.

Another critical research area is concurrency testing, particularly within the Linux kernel. Concurrency, which enables systems to perform multiple tasks simultaneously, is essential for modern performance, but it also introduces the risk of concurrency bugs. These bugs, "are notoriously difficult to detect and reproduce" (Xu et al. 2025). Traditional methods for finding these bugs, such as stress-testing and kernel fuzzers, have limitations. Stress-testing is labor-intensive, and kernel fuzzers primarily focus on input search space, often missing concurrency-specific issues. Researchers are developing new techniques, such as LACE, a lightweight concurrency testing framework that leverages eBPF. LACE aims to provide more controlled concurrency testing by enabling control over thread interleavings and systematically exploring the interleaving space. This approach seeks to address the limitations of existing tools, such as high runtime overhead and lack of forward compatibility, by offering a more efficient and adaptable solution for finding concurrency bugs.

REFERENCES

Dash, A. R., Sahu, S. K., & Samantra, S. K. (2015). An Optimized Round Robin CPU Scheduling Algorithm with Dynamic Time Quantum. *International Journal of Computer Science, Engineering and Information Technology*, *5*(1), 07-26. http://dx.doi.org/10.5121/ijcseit.2015.5102

Kanellis, K., Yadalam, S., Chen, F., Swift, M., & Venkataraman, S. (2025). *From Good to Great: Improving Memory Tiering Performance Through Parameter Tuning*. arXiv. https://arxiv.org/abs/2504.18714

Redwood. (n.d.). Job Scheduling Algorithms. Retrieved from https://www.redwood.com/article/job-scheduling-algorithms/

Rikos, A. I., Grammenos, A., Kalyvianaki, E., Hadjicostis, C. N., Charalambous, T., & Johansson, K. H. (2021). *Optimal CPU Scheduling in Data Centers via a Finite-Time Distributed Quantized Coordination Mechanism*. arXiv. https://arxiv.org/abs/2104.03126

Scaler. (n.d.). Process Scheduling in Operating System. Retrieved from https://www.scaler.com/topics/operating-system/process-scheduling/

Xu, J., Wolff, D., Han, X. Y., Li, J., & Roychoudhury, A. (2025). *Concurrency Testing in the Linux Kernel via eBPF*. arXiv. https://arxiv.org/abs/2504.21394