

Politecnico di Torino

Enhancement of the Completely Fair Scheduler

Members
Haoyi Chen
Yu Huang
Fuxin Li

Professor: Stefano Di Carlo

Operating Systems

Politecnico di Torino
November 2026

Contents

1 Introduction **1**

2 Background **2**

2.1 The Completely Fair Scheduler (CFS) 2

2.2 Virtual Runtime and the Red-Black Tree 2

2.3 Motivation: Bridging CFS with Lottery Scheduling 2

3 Customized Scheduler Based on CFS **3**

3.1 Lottery Scheduling 3

3.2 Fork Bomb Defence 3

3.3 Priority Inversion & Ticket Transfer 4

3.4 Shortest Job First 4

3.5 Boost I/O process 5

4 Test **6**

4.1 Lottery scheduling 6

4.2 Fork Bomb Defence 6

4.3 Priority Inversion & Ticket Transfer 7

4.4 Shortest Job First (SJF) 7

4.5 I/O boost 7

5 Result and Discussion **9**

1 Introduction

In this project, we implement a scheduler based on Completely Fair Scheduler (CFS) in the Linux kernel. The main strategy of our scheduler is lottery scheduling, and other implementations all start from this lottery scheduling mechanism. Our scheduler aims to address several common scenarios, such as scheduling between parent and child processes and scheduling processes that involve I/O. In terms of parent and child process scheduling, we focus on fork bomb defence, priority inversion and ticket transfer. Because the performance of processes with I/O behavior is usually limited by I/O, we focus on boosting I/O-bound processes so that they can be scheduled earlier. Finally, we implement shortest job first (SJF) on top of our lottery scheduling.

To check correctness, we write specific benchmarks to test our scheduler, which is also an important part of scheduler design. Finally, we analyze the results and evaluate our scheduler accordingly.

2 Background

2.1 The Completely Fair Scheduler (CFS)

Introduced in Linux kernel 2.6.23, the Completely Fair Scheduler (CFS) serves as the default scheduling class (`SCHED_NORMAL`). Unlike legacy $O(1)$ schedulers that relied on complex heuristics to guess interactivity, CFS elegantly simulates an "ideal multi-tasking processor." Its core philosophy is absolute fairness: ensuring that every process receives a strictly proportional share of the CPU's processing power over time, without relying on opaque priority guesswork.

2.2 Virtual Runtime and the Red-Black Tree

To quantify and track this fairness, CFS introduces a core metric called Virtual Runtime (`vruntime`). The `vruntime` accumulates as a task executes, normalized by its weight (derived from its `nice` value). Tasks with a higher weight experience a slower `vruntime` growth rate, while lower-weight tasks accumulate `vruntime` faster.

At the data structure level, CFS organizes all runnable tasks within a time-ordered Red-Black Tree, indexed by their `vruntime`. This self-balancing binary search tree allows the scheduler to efficiently insert and update tasks in $O(\log N)$ time, while fetching the next task to run—always the leftmost node with the absolute smallest `vruntime`—in $O(1)$ time. This mechanism dynamically prioritizes tasks that have been starved of CPU time.

2.3 Motivation: Bridging CFS with Lottery Scheduling

While CFS provides excellent general-purpose fairness, its reliance on static `nice` values can be rigid. In mixed workloads where aggressive CPU-bound batch tasks dominate, latency-sensitive I/O tasks can still experience unpredictable scheduling delays. Furthermore, assigning deterministic, proportional CPU shares based on dynamic system events (such as IPC dependencies) is difficult under standard CFS.

To address these limitations, this project introduces a Lottery Scheduling mechanism into the CFS framework. By completely replacing static `nice` values with dynamically assignable `Tickets`, we gain explicit, proportional control over CPU allocation. This ticket-based architecture not only guarantees deterministic CPU shares but also serves as the foundational framework for our advanced features, including the I/O Boost mechanism and the IPC-driven Ticket Transfer (to resolve priority inversion).

3 Customized Scheduler Based on CFS

In our scheduler, we mainly rely on lottery scheduling, and we implement different strategies by using different ticket assignment methods.

3.1 Lottery Scheduling

This part is the core of our scheduler. We introduce an important variable (tickets) in the task struct (in sched.h). The tickets will be used in the delta time calculation (calc_delta_fair function in fair.c file), which will affect the value of the virtual time. According to the Equation 1, the more tickets a process has, the more slowly the corresponding vruntime increases. In order to modify this tickets variable, we need to create a system call (sys.c) to set corresponding tickets. We also need to add this new system call in the entry file (syscall_64.tbl). For initialization, the tickets field is set in the function __sched_fork in core.c file.

$$\text{vruntime} \mathrel{+}= \text{time_delta} \times \left(\frac{1024}{\text{Weight(nice)}}\right) \times \left(\frac{\text{Base_Tickets}}{\text{Process_Tickets}}\right) \tag{1}$$

3.2 Fork Bomb Defence

In a standard proportional share scheduler, a malicious or malfunctioning process can monopolize CPU resources by rapidly spawning a massive number of child processes, a classic attack known as a Fork Bomb. If the scheduler blindly assigns a default number of tickets to every newly created process, the total number of tickets held by the malicious process family will grow exponentially. This global ticket inflation severely dilutes the relative CPU share of legitimate tasks, leading to CPU starvation.

To mitigate this vulnerability at the scheduling level, we implemented a strictly enforced "Ticket Conservation" policy within the kernel's process creation routine (e.g., `__sched_fork`). The core principle is that a process cannot artificially inflate its CPU share simply by creating clones of itself.

When a parent process forks a child, the kernel does not mint new tickets. Instead, it equally divides the parent's current tickets between the parent and the newly created child. The ticket distribution follows this mathematical rule:

$$T_{\text{child}} = \lfloor \frac{T_{\text{parent}}}{2} \rfloor, \quad T_{\text{parent}} = \lceil \frac{T_{\text{parent}}}{2} \rceil \tag{2}$$

By enforcing Equation 2, the sum of tickets held by the entire malicious process tree is strictly bounded by the initial ticket count of the parent process. The malicious family can recursively divide its internal CPU share among its children, but it can never increase its total weight against other independent tasks in the system. This purely mathematical constraint guarantees that independent tasks maintain their expected CPU allocation regardless of the topology or depth of an adversarial process tree.

3.3 Priority Inversion & Ticket Transfer

The design of our ticket transfer mechanism is deeply inspired by the classic priority inversion anomaly frequently encountered in modern Real-Time Operating Systems (RTOS). In strict priority-based RTOS environments, such as FreeRTOS or ThreadX, a critical high-priority task can be indefinitely blocked if it waits for a shared resource (e.g., a Mutex) held by a low-priority task, while medium-priority tasks continuously preempt the low-priority one. Industry-standard RTOS natively resolve this by implementing the Priority Inheritance Protocol (PIP) within their synchronization primitives. In this project, we successfully adapted this real-time engineering concept into the domain of proportional share scheduling.

In a proportional share scheduling environment, an analogous priority inversion occurs when a high-weight task (holding a large number of tickets) is blocked waiting for a shared resource or synchronization provided by a low-weight task. Because the low-weight resource holder receives a minimal share of the CPU, its execution is frequently preempted by independent medium-weight background tasks. Consequently, the high-weight task's effective progress is severely bottlenecked by the low-weight task's CPU starvation, fundamentally violating the intended proportional resource allocation policy.

To resolve this anomaly, we introduced a deterministic ticket transfer mechanism, acting as a proportional-share analog to the RTOS Priority Inheritance Protocol. This mechanism is facilitated by a custom system call (`SYS_TRANSFER`), which allows a blocked client process to temporarily donate its tickets to a server process that is fulfilling its request.

3.4 Shortest Job First

The Shortest Job First (SJF) algorithm is mathematically proven to be optimal for minimizing the average turnaround time of a given set of processes. However, implementing a strict SJF scheduler in a general-purpose operating system is fundamentally impossible. A true SJF policy requires a priori knowledge of the exact execution time (CPU burst length) of a task before it runs. Because the operating system cannot predict the future behavior of arbitrary user-level programs, true SJF remains a theoretical ideal rather than a practical implementation.

Although exact future execution times are unknowable, our scheduler effectively approximates SJF by dynamically adjusting a task's ticket allocation based on its historical CPU usage. We leverage the heuristic that a task's accumulated execution time (`sum_exec_runtime`) is a reliable predictor of its future CPU demands.

In our implementation, a task's accumulated runtime is converted to milliseconds (t_{ms}) and passed into an inverse proportional decay model to dynamically assign tickets:

$$\text{Tickets} = \begin{cases} 500, & \text{if } t_{\text{ms}} < 150 \\ \max\left(20, \min\left(500, \frac{20000}{t_{\text{ms}}}\right)\right), & \text{otherwise} \end{cases} \tag{3}$$

Under this model, short or newly spawned tasks receive the maximum 500 tickets, so their

`vruntime` grows slowly (per Equation 1) and they quickly preempt long-running tasks. Conversely, as CPU-heavy tasks continue to execute, their ticket allocation smoothly decays toward a minimum floor of 20 tickets. This dynamic, history-based penalty effectively mimics the aggressive preemption of true SJF while eliminating the need for a predictive "oracle".

3.5 Boost I/O process

I/O-bound tasks spend much of their time waiting for I/O requests (or sleeping). Because of CFS optimization, the vruntime of these processes does not increase during this waiting period so that they can have priority after waking up. If a process contains I/O operations, its performance will be limited by the bound of I/O operations, which is much slower compared with CPU operations.

As a result, in our lottery based scheduler, we give more tickets to I/O-related processes, which temporarily boosts them.

In order to know when to boost I/O processes, we need to know which flags are set when I/O processes wake up. At the same time, we need to set a threshold on the sleep duration to determine which I/O processes are boosted. In our scheduler, we mainly modify the fair.c file to implement I/O boost. In this file, we modify the functions enqueue_task_fair, dequeue_task_fair and check_preempt_tick.

In enqueue_task_fair function, we use the flag ENQUEUE_WAKEUP to detect that an I/O process is waking up. After the process wakes up, we calculate the sleep time as the wake-up time minus the sleep start time recorded in the dequeue_task_fair function. If the sleep duration is larger than the threshold, the I/O boost factor is activated (in our scheduler it is 3), which increases the final tickets assigned to this process, so its vruntime increases more slowly and it stays close to the leftmost position of the rbtree. In check_preempt_tick function, if a task has an I/O boost factor greater than 1, its time slice limit is ignored and it is allowed to keep running uninterrupted during this timer tick, which ensures that I/O processes have higher priority.

4 Test

During testing, we use the corresponding statements and system calls in the benchmarks to trigger related processes to run, and then use the top command in terminal to check whether the CPU utilization meets our expectations.

4.1 Lottery scheduling

In this benchmark, we use the corresponding system call to set the ticket for a process, and then check whether the corresponding percentage of CPU time (%CPU) of every process meets our design. In the test script, we use different initial tickets for two processes. If the implementation is correct, the %CPU ratio between the two processes is equal to the ratio of their tickets.

4.2 Fork Bomb Defence

In this test, we need to compare CPU time (%CPU) consumed by the child process and the parent process to check whether the %CPU is consistent with Equation 2 under the defence strategy. In the benchmark, we use the fork() function to create a child process in the parent process, and in the parent we use fork() again to create another child process. Then a child process is created in this child process. For comparison, we first run the benchmark without our defence method, then move to the test with the defence strategy. The test structure is shown as Figure 1. In theory, the three processes will have the same %CPU without the defence method, and a %CPU distribution consistent with Equation 2 under the fork bomb defence method.

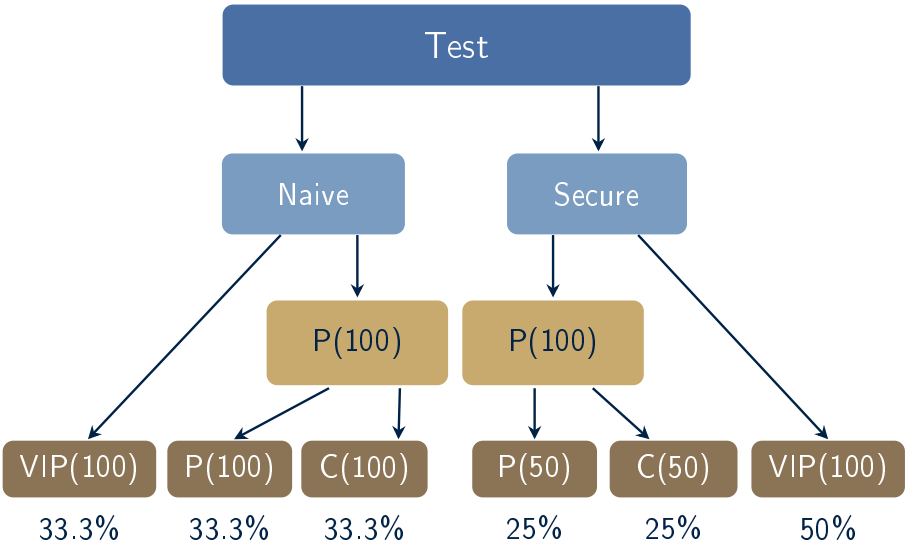


Figure 1: The test structure of Fork Bomb.

4.3 Priority Inversion & Ticket Transfer

In this benchmark, we evaluate the ticket transfer mechanism designed to resolve the priority inversion anomaly.

In our test, we spawn three processes: a high-ticket client (Task A), a low-ticket server (Task B), and a medium-ticket background noise task (Task C). Task A is designed to block and wait for an IPC response from Task B, explicitly donating its tickets to Task B before entering the sleeping state.

To thoroughly investigate the system's scheduling behavior during ticket restoration and task wakeup, we designed an iterative two-stage testing strategy:

- **Initial Test (Task A = 300 Tickets):** We first ran the benchmark assigning a massive 300 tickets to Task A. While the ticket donation successfully allowed Task B to overcome the noise of Task C, we observed an intriguing side effect using the `top` command: upon waking up, Task A monopolized the CPU (%CPU spiking to 100%) for an extended period due to its accumulated virtual time deficit.
- **Follow-up Test (Task A = 150 Tickets):** Motivated by the prolonged CPU monopoly observed in the first test, we explicitly added a second configuration where Task A's initial tickets were reduced to 150. Our goal here is to check whether the reduced ticket weight accelerates Task A's virtual time catch-up. If the implementation of our ticket-based virtual time is strictly correct, we expect to observe the %CPU of Task A transforming from a static 100% monopoly into a rapid dynamic oscillation (alternating between 100% and ∼60%) as it leapfrogs with Tasks B and C.

4.4 Shortest Job First (SJF)

In this test, we assign tickets to the SJF task according to Equation 3. To prevent this task from occupying all CPU time, we run another basic lottery task with 100 tickets, which is also used for comparison. In the SJF test, we need to switch the scheduler to SJF mode.

4.5 I/O boost

In order to verify this implementation, we need to write a benchmark which includes I/O behaviors or I/O like behaviors (like usleep) to trigger our I/O boost mechanism. In the test script, we set the initial tickets of the two processes to be the same, and then we compare their results. Here we can use two strategies:

- **Comparing time:** we run three processes, first is CPU hog, second is I/O process without boost, and the last one is I/O process with boost mechanism. In theory, the boosted I/O process consumes less time compared with the one without boost because more tickets are assigned to it.

- **Comparing %CPU:** we run two processes, first is CPU hog, and the second is boosted I/O process. In our design, the ratio of %CPU between these two processes is approximately equal to the boost factor (3 here).

To keep this test consistent with the other tests, we decide to use the %CPU test method. Because I/O related processes spend much time on sleeping, we introduce some dummy work (e.g., loops) after waking up to consume the CPU time. Then we compare %CPU of boosted I/O process with the CPU hog.

5 Result and Discussion

After finishing the benchmarks, we use them to test our scheduler. Because the tickets assigned to a process affect the CPU time allocation, we use the percentage of CPU time to check whether the obtained results correspond to our design.

Table 1: Test Results of Basic Lottery, Fork Bomb Defence and I/O Boost

Test feature		Avg %CPU	Task
Basic Lottery		74.9	Tickets_300
		25.1	Tickets_100
fork bomb defence	Naive mode	34.0	VIP
		33.2	N_parent
		33.5	N_Child
	Secure mode	50.6	VIP
		23.7	S_parent
		24.5	S_Child
I/O boost		75.6	boost_io
		24.7	CPU_HOG

As Table 1 shows, in the basic lottery scheduling test, we set the initial tickets of the two processes to 300 and 100, respectively. The %CPU ratio between two tasks is close to 3, which is the same as the ratio of the two initial tickets. As for the Fork bomb defence test, the result is consistent with Figure 1. In terms of I/O boost, the ratio of %CPU is similar to I/O boost factor (3 in our scheduler). These results meet our expectations.

Table 2: Test Results of Priority Inversion & Ticket Transfer

Phase	**Task**	**Samples**	**Avg %CPU (150t)**	**Avg %CPU (300t)**
Phase 1 Normal	Task_A	#1–#3	47.1	64.5
	Task_B		15.7	11.2
	Task_C		32.7	22.4
Phase 2 Transfer	Task_A	#4–#7	2.0	3.4
	Task_B		65.2	74.3
	Task_C		33.3	22.3
Phase 3 Return	Task_A	#8–end	78.0	83.2
	Task_B		6.0	5.0
	Task_C		11.8	7.0

Table 2 presents the test results for the Priority Inversion and Ticket Transfer benchmark. The empirical data conclusively aligns with our theoretical design for resolving priority inversion:

- **Phase 1 (Normal) & Phase 2 (Transfer):** The results strictly follow the proportional ticket weights. In Phase 1, the tasks share the CPU according to their initial tickets (e.g., in the 150t case, Task A, B, and C distribute CPU at roughly a 3:1:2 ratio). During Phase 2, when Task A blocks and donates its tickets to Task

B, Task B's %CPU successfully surges, effectively overcoming the background noise of Task C and preventing priority inversion.

- **Phase 3 (Return):** Upon Task A's wakeup and ticket restoration, we observe a significant spike in its CPU usage (78.0% and 83.2%), temporarily starving Tasks B and C. This monopolization is an inherent artifact of the CFS `vruntime` mechanism: while Task A was blocked in Phase 2, its `vruntime` remained frozen. When awakened with a massive ticket weight, it experiences a massive historical deficit, causing the scheduler to heavily compensate it until its `vruntime` catches up with the rest of the system.

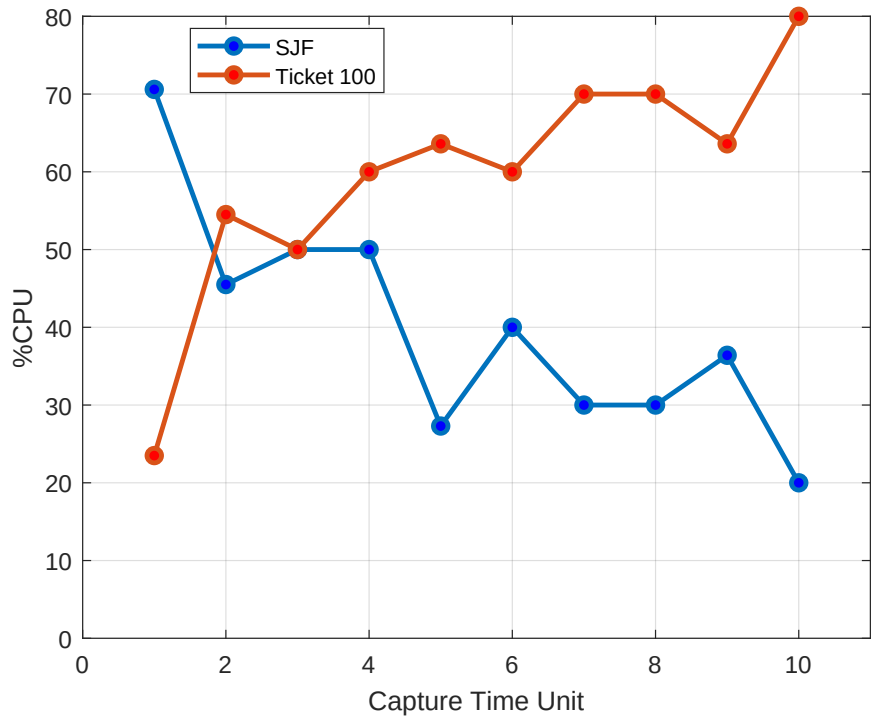


Figure 2: The result of the shortest job first vs. ticket 100.

Figure 2 shows the trend of %CPU change over time in the shortest job first test. At the beginning, SJF task has high %CPU, but its vruntime accumulates as it consumes CPU time. As a result, the tickets assigned to the SJF task decrease over time, which causes its %CPU to decrease.

To sum up, in our scheduler, we focus on lottery scheduling starting from CFS, and implement four scheduling strategies based on this basic lottery scheduling. The results approximately satisfy our expectations. Although our benchmarks only test a few processes, which may differ from real applications (where many processes run), they reflect the core principles behind the customized scheduler.