Implementing a Lottery Scheduler in Linux 2.4 Kernel

by

Asude Beyza Demirboğa

Ece Tipici

Hatice Müberra Gül

Muhammet Ramazan Dolaş

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2024

ABSTRACT

This project aims to enhance the scheduling mechanism of the Linux 2.4.27 kernel by implementing a Lottery Scheduler within the existing fair preemptive scheduler (SCHED_OTHER). The goal is to compare the performance of these schedulers with different numbers of processes and users.

We set up the project environment by adding three test users and generating processes for each user. We conducted tests by switching between the default scheduler and the Lottery Scheduler, measuring CPU usage with the 'top' command, and analyzing the results.

The findings show that both schedulers provide similar CPU usage for a single user, with the default scheduler being more precise. In contrast, in multi-user scenarios, the Lottery Scheduler ensures a more equitable CPU distribution among users, preventing monopolization by any single user.

TABLE OF CONTENTS

1. INTRODUCTION

SCHED_OTHER is the default time-sharing scheduling algorithm in the Linux 2.4 kernel, designed to ensure fair and balanced CPU allocation among processes. It assigns each process a dynamic priority based on its “nice” value and remaining time slice, with counters decreasing as CPU time is used. When all counters drop to zero, a new scheduling epoch begins, recalculating priorities and counters. Processes with lower nice values have higher priority, and the scheduling function evaluates processes using a goodness function to determine the next process to run. This mechanism ensures equitable CPU distribution, adapting dynamically to process behavior and system demands. [1]

While this scheduling approach is both equitable and precise, it is designed to allocate CPU resources based on processes rather than users. This can lead to issues when multiple users are involved, as one user could monopolize the CPU, potentially causing starvation for others. [2]

As a solution for this, we implemented the Lottery Scheduler into the existing scheduler, which enables load insulation among users. [2] The Lottery Scheduler is a randomized, proportional-share resource allocation algorithm where each process is assigned a certain number of “tickets”. When the scheduler is invoked, a random ticket is drawn, and the process holding the winning ticket is selected for execution. This approach ensures that processes with more tickets (users with fewer processes), and thus higher priority, have a greater chance of being scheduled, while still allowing for fairness among lower-priority processes. [3]

To evaluate the CPU usage of the Lottery Scheduler compared to the default scheduler (SCHED_OTHER) in the Linux 2.4.27 kernel, we created a controlled testing environment. This environment was set up by adding three test users, multiple processes for each. The number of processes was adjusted based on test scenarios. A system call, controlled by a flag was implemented to switch securely between the default scheduler (flag set to 0) and the Lottery Scheduler (flag set to 1).

The scope of the tests included single-user scenarios to compare precision, and multi-user scenarios to compare fairness in CPU distribution.

2. DESIGN and IMPLEMENTATION

2.1. METHODOLOGY

After constructing the pseudo-codes for the scheduling algorithm, the Linux environment was prepared by creating test users to simulate a multi-user environment for testing. The CPU usage statistics of the existing scheduler were examined to establish a performance baseline before implementing the Lottery Scheduler. A system call controlled by a flag was written to securely start the OS with the default scheduler and then switch to the Lottery Scheduler. Following this, the Lottery Scheduler was implemented by modifying the existing scheduler code. Tests were conducted to examine the behavior of the scheduler by manually creating processes from the test users, and simulating various load conditions. The results from both the default scheduler and the Lottery Scheduler were obtained multiple times to ensure a reliable comparison with minimal error margin.

The implementation and testing were done using the Linux 2.4 kernel, chosen for its simplicity and reliability, as well as the time and resource constraints of the project. [4]

2.2. DEFAULT SCHEDULER

SCHED_OTHER is the default scheduling algorithm in Linux 2.4, designed for fair preemptive scheduling and balanced CPU time distribution using dynamic priorities.

Dynamic Priority and Time Slice (Counter):
Processes get a dynamic priority based on their “nice” value and remaining time slice (counter).

Counters decrease as CPU time is used, and when a counter reaches 0, the process is rescheduled.

Epochs and Rescheduling:

A new epoch starts when all process counters are 0, recalculating counters and priorities. A counter of 0 signals the need for rescheduling, not process termination.

Nice Value:

Affects priority: lower nice values mean higher priority.

Ranges from -20 (high priority) to +19 (low priority), with a default of 0.

Forked processes inherit a nice value of 0.

Nice values can be increased (lower priority) using the nice() system call.

I/O and CPU Burst Adjustments:

I/O processes may have their nice values decreased (increasing priority). CPU-bound processes retain their nice values.

Task Struct and Scheduling:

The kernel tracks runnable processes using a linked list of task_struct containing fields like counter, nice, need_resched, and state. When a process is created via fork, the parent's counter is split between parent and child.

Schedule Function:

schedule() is called to select the next process to run. It uses the goodness() function to evaluate processes based on their priorities and counters.

Goodness Function:

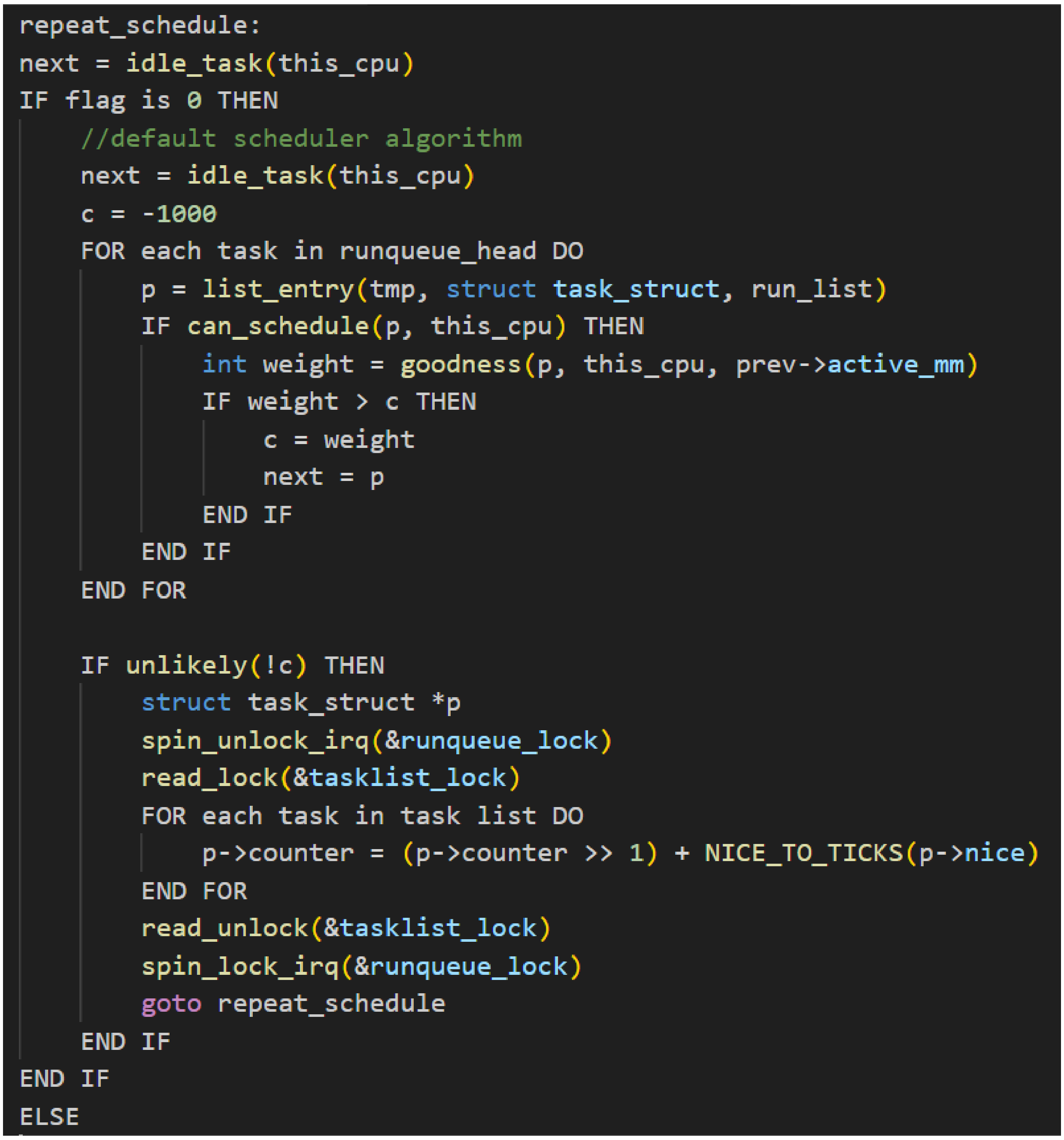
Evaluates process suitability for execution, returning a value (c) indicating "goodness":

$c = -1000$: Process must not be selected.

$c = 0$: Process has exhausted its quantum.

$0 < c < 1000$: Process has remaining quantum.

$c >= 1000$: Real-time task with high priority. [1]



```
repeat_schedule:
next = idle_task(this_cpu)
IF flag is 0 THEN
    //default scheduler algorithm
    next = idle_task(this_cpu)
    c = -1000
    FOR each task in runqueue_head DO
        p = list_entry(tmp, struct task_struct, run_list)
        IF can_schedule(p, this_cpu) THEN
            int weight = goodness(p, this_cpu, prev->active_mm)
            IF weight > c THEN
                c = weight
                next = p
            END IF
        END IF
    END FOR

    IF unlikely(!c) THEN
        struct task_struct *p
        spin_unlock_irq(&runqueue_lock)
        read_lock(&tasklist_lock)
        FOR each task in task list DO
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice)
        END FOR
        read_unlock(&tasklist_lock)
        spin_lock_irq(&runqueue_lock)
        goto repeat_schedule
    END IF
END IF
ELSE
```

Fig. 1. Default scheduler's (SCHED_OTHER) pseudo-code [5]

The pseudocode represents the scheduling loop for the SCHED_OTHER algorithm. Initially, next is set to the idle task. If a specific flag is 0, the default scheduling algorithm is used. The scheduler iterates through all tasks in the run queue, evaluating each task's goodness value using the goodness function. The task with the highest goodness value is selected to run next. If no suitable task is found (c remains -1000), the scheduler recalculates the counters for all tasks by halving their current counters and adding a value based on their nice values. The loop then repeats to select the next task for execution, ensuring fair CPU time distribution among processes.

2.3. LOTTERY SCHEDULER IMPLEMENTATION

Part 1: Initializing Users and User Count

We start by setting up an array to hold user information and initializing the count of users to zero.

Part 2: Finding User Processes

We iterate through all tasks in the run queue.

For each task, if it is eligible for scheduling and belongs to a user with a UID greater than or equal to 1000, we proceed. We keep track of the UID of the user owning the task. Then, we check if the user already exists in our array. If so, we increment their process count; otherwise, we create a new user entry.

Part 3: Assigning Tickets

We define the number of tickets each user will receive. Then, for each task in the run queue, we allocate tickets to each task based on its user's process count. If a task belongs to a user, it receives a fraction of the user's tickets; otherwise, it gets a default number of tickets.

Part 4: Calculating Total Tickets

We calculate the total number of tickets available in the system by summing up the tickets of all tasks eligible for scheduling.

Part 5: Generating Random Number

We generate a random number to determine the winning ticket.

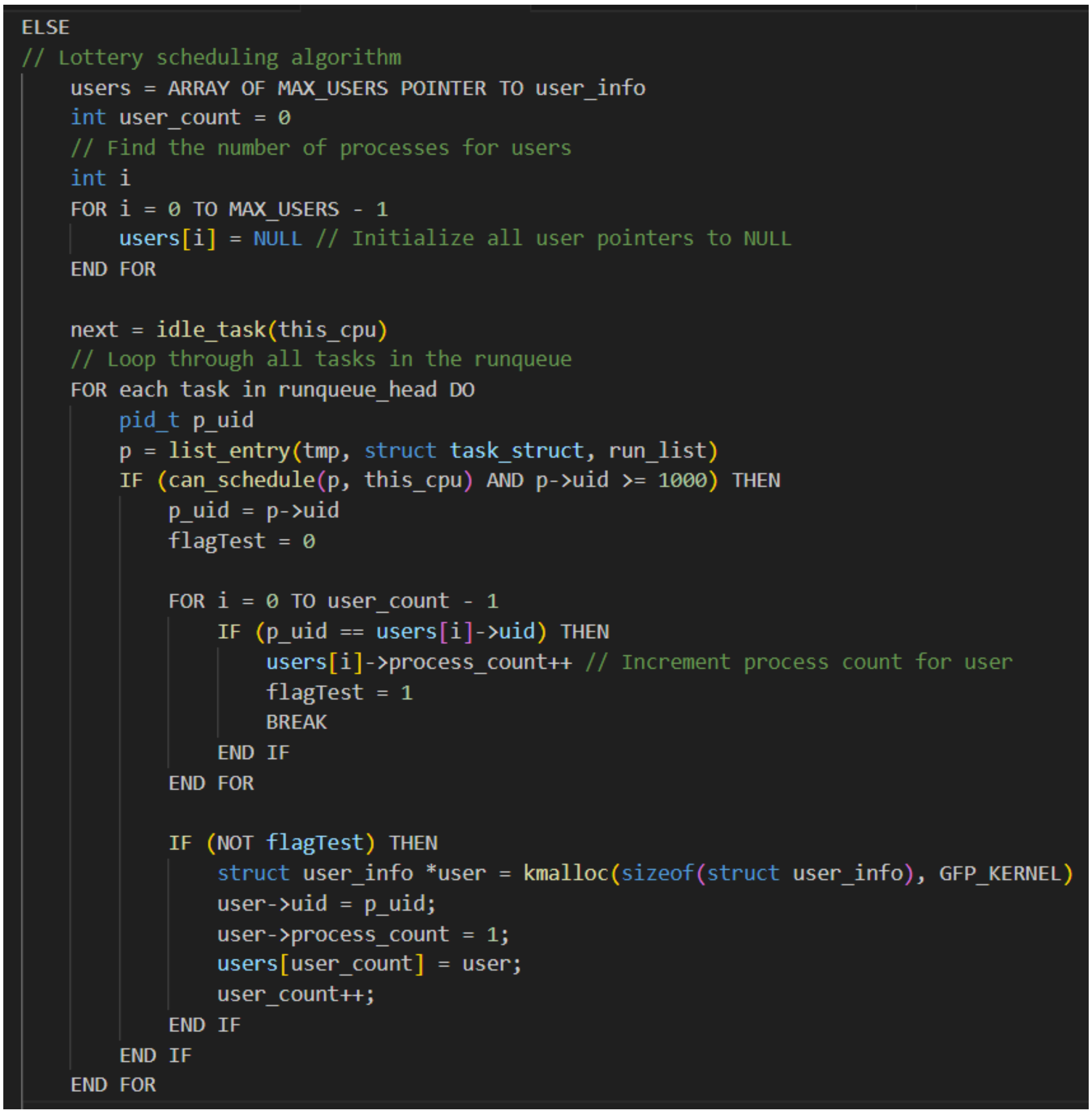
Part 6: Selecting the Next Task

If there are no users, we select the idle task. Otherwise, we calculate the winning ticket's position in the range of total tickets. We then iterate through the tasks again, adding up their tickets until we find the task whose ticket range includes the winning number.

Part 7: Freeing Memory

Finally, we free the memory allocated for storing user information.

So, in essence, the lottery scheduling algorithm works by distributing tickets to tasks based on their user's process count and then selecting the next task to execute based on a random lottery draw. This ensures fairness in task selection while considering the workload of each user.



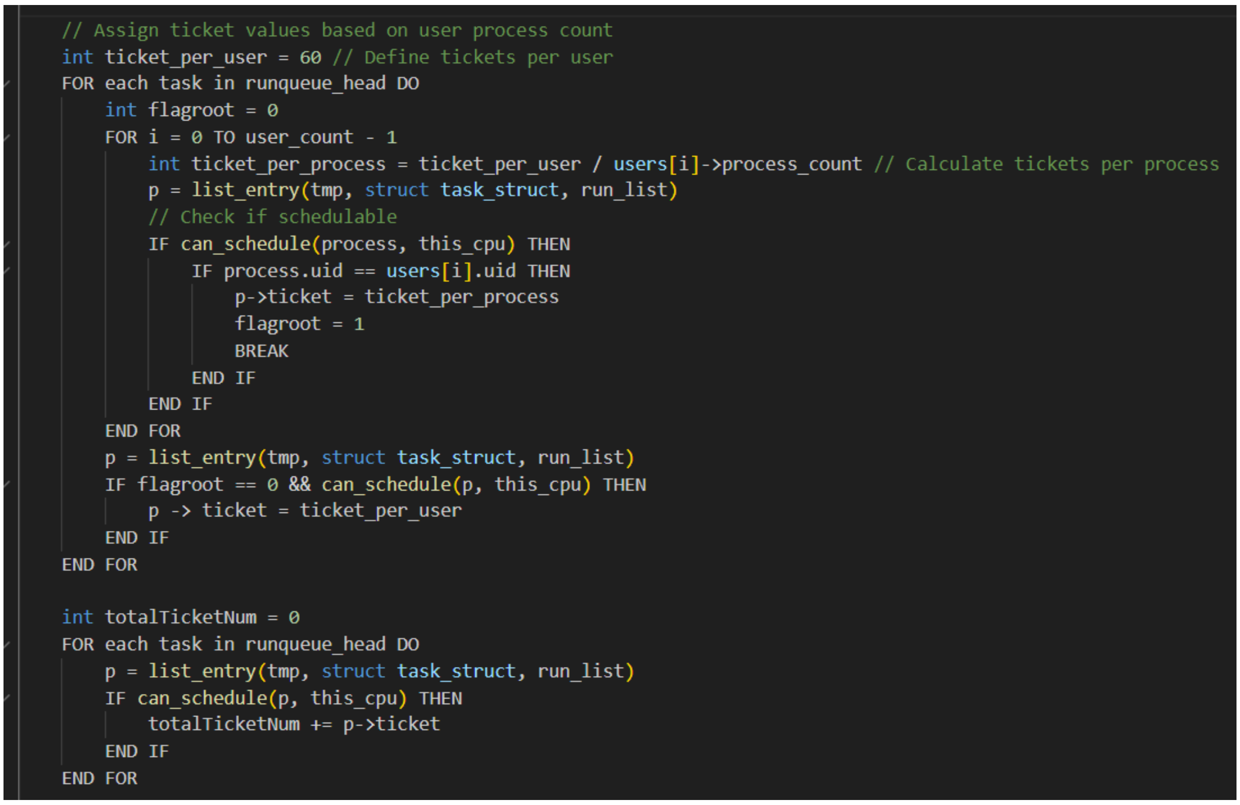
```
ELSE
// Lottery scheduling algorithm
    users = ARRAY OF MAX_USERS POINTER TO user_info
    int user_count = 0
    // Find the number of processes for users
    int i
    FOR i = 0 TO MAX_USERS - 1
        users[i] = NULL // Initialize all user pointers to NULL
    END FOR

    next = idle_task(this_cpu)
    // Loop through all tasks in the runqueue
    FOR each task in runqueue_head DO
        pid_t p_uid
        p = list_entry(tmp, struct task_struct, run_list)
        IF (can_schedule(p, this_cpu) AND p->uid >= 1000) THEN
            p_uid = p->uid
            flagTest = 0

            FOR i = 0 TO user_count - 1
                IF (p_uid == users[i]->uid) THEN
                    users[i]->process_count++ // Increment process count for user
                    flagTest = 1
                    BREAK
                END IF
            END FOR

            IF (NOT flagTest) THEN
                struct user_info *user = kmalloc(sizeof(struct user_info), GFP_KERNEL)
                user->uid = p_uid;
                user->process_count = 1;
                users[user_count] = user;
                user_count++;
            END IF
        END IF
    END FOR
```

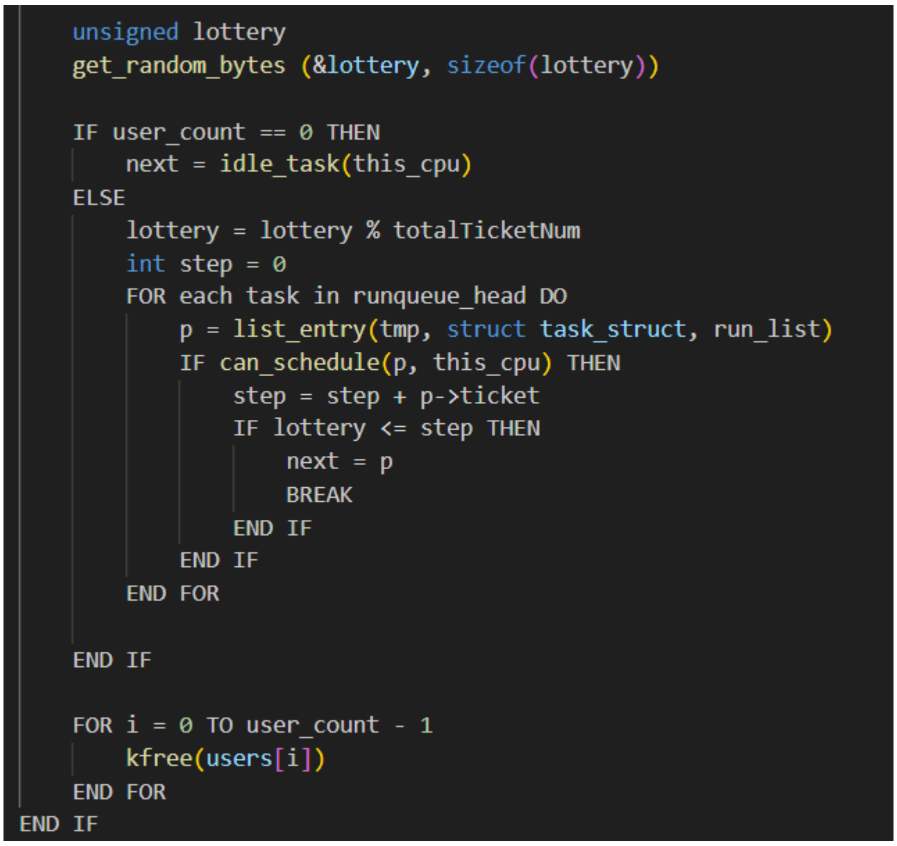
Fig. 2.1. Lottery scheduler’s pseudo code – part 1



```
    // Assign ticket values based on user process count
    int ticket_per_user = 60 // Define tickets per user
    FOR each task in runqueue_head DO
        int flagroot = 0
        FOR i = 0 TO user_count - 1
            int ticket_per_process = ticket_per_user / users[i]->process_count // Calculate tickets per process
            p = list_entry(tmp, struct task_struct, run_list)
            // Check if schedulable
            IF can_schedule(process, this_cpu) THEN
                IF process.uid == users[i].uid THEN
                    p->ticket = ticket_per_process
                    flagroot = 1
                    BREAK
                END IF
            END IF
        END FOR
        p = list_entry(tmp, struct task_struct, run_list)
        IF flagroot == 0 && can_schedule(p, this_cpu) THEN
            p -> ticket = ticket_per_user
        END IF
    END FOR

    int totalTicketNum = 0
    FOR each task in runqueue_head DO
        p = list_entry(tmp, struct task_struct, run_list)
        IF can_schedule(p, this_cpu) THEN
            totalTicketNum += p->ticket
        END IF
    END FOR
```

Fig. 2.2. Lottery scheduler's pseudo code – part 2



```
    unsigned lottery
    get_random_bytes (&lottery, sizeof(lottery))

    IF user_count == 0 THEN
        next = idle_task(this_cpu)
    ELSE
        lottery = lottery % totalTicketNum
        int step = 0
        FOR each task in runqueue_head DO
            p = list_entry(tmp, struct task_struct, run_list)
            IF can_schedule(p, this_cpu) THEN
                step = step + p->ticket
                IF lottery <= step THEN
                    next = p
                    BREAK
                END IF
            END IF
        END FOR

    END IF

    FOR i = 0 TO user_count - 1
        kfree(users[i])
    END FOR
END IF
```

Fig. 2.3. Lottery scheduler's pseudo code – part 3

2.4. SCHEDULER COMPARISON

The default scheduler and the Lottery Scheduler act similarly with only one user by providing processes with an equal amount of CPU usage. The main difference is that while the Default Scheduler achieves this through a counter, offering a more precise approach, the Lottery Scheduler does so by randomly selecting a ticket. This randomness introduces a vulnerability, resulting in less precise outcomes compared to default scheduler. Additionally, the Lottery Scheduler's algorithm is more complex and costly.

As the number of users increases, the default scheduler only considers fairness among processes, not users, which can lead to one user monopolizing the CPU and causing starvation for others. The Lottery Scheduler addresses this issue by ensuring each user receives equal CPU usage, regardless of the number of processes they have. [3]

Therefore, the Lottery Scheduler is beneficial in systems with many users and processes, ensuring fair CPU usage per user. On the other hand, the default scheduler is preferable in smaller and less complex systems due to its simplicity and precision.

3. TESTS and RESULTS

3.1. Test Methodology

We compared the default scheduler with the Lottery Scheduler through a series of tests. Firstly, we created three users using the "adduser" command and generated processes for each user using the "yes >/dev/null &" command, adjusting process count based on test cases. By changing a flag value in the "main.c" file, switching between the default scheduler (flag set to 0) and the lottery algorithm (flag set to 1) achieved. CPU usage was measured using the "top" command with options "-n 100 -d 1 –b" to capture 100 samples at 1-second intervals. Results were saved to separate files (defaultschedNtestM.txt). After terminating the processes, average CPU usage per process was calculated using the "awk" command. The process was repeated with the flag value set to 1 for the Lottery Scheduler. CPU usage data were obtained using the same "top" command (lotterychedNtestM.txt), ten times for each test case, resulting in 1000 samples per case. Because of the Lottery Scheduler's random nature, it requires a higher sample size to ensure accurate CPU usage measurements.

3.2. Test Cases

3.2.1 Test Case 1 – 1 user 2 processes

User1 has 2 processes.

CPU Utilization (%)	**User 1 Process 1**	**User 1 Process 2**
DEFAULT	49.974	49.683
LOTTERY	49.071	49.838

Table 1. CPU utilization for 1 user 2 processes

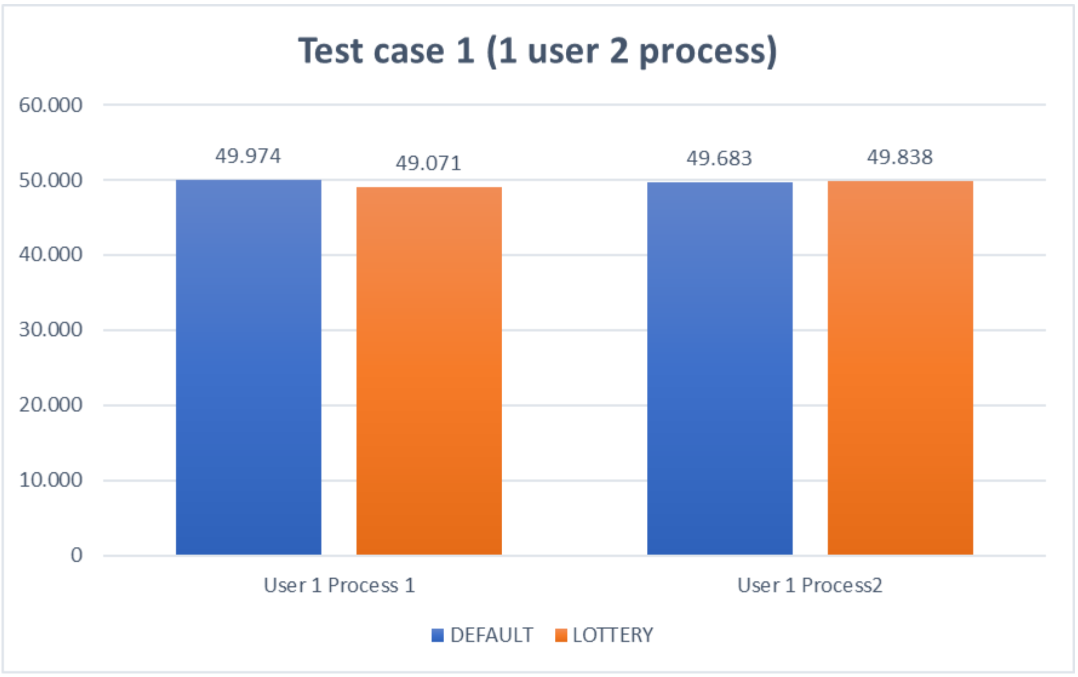


Fig. 3. CPU utilization for 1 user 2 processes

3.2.2 Test Case 2 – 2 users 2 processes

User1 has 1 process, User2 has 1 process.

CPU Utilization (%)	**User 1 Process 1**	**User 2 Process 1**
DEFAULT	49.842	49.979
LOTTERY	50.812	49.263

Table 2. CPU utilization for 2 users 2 processes

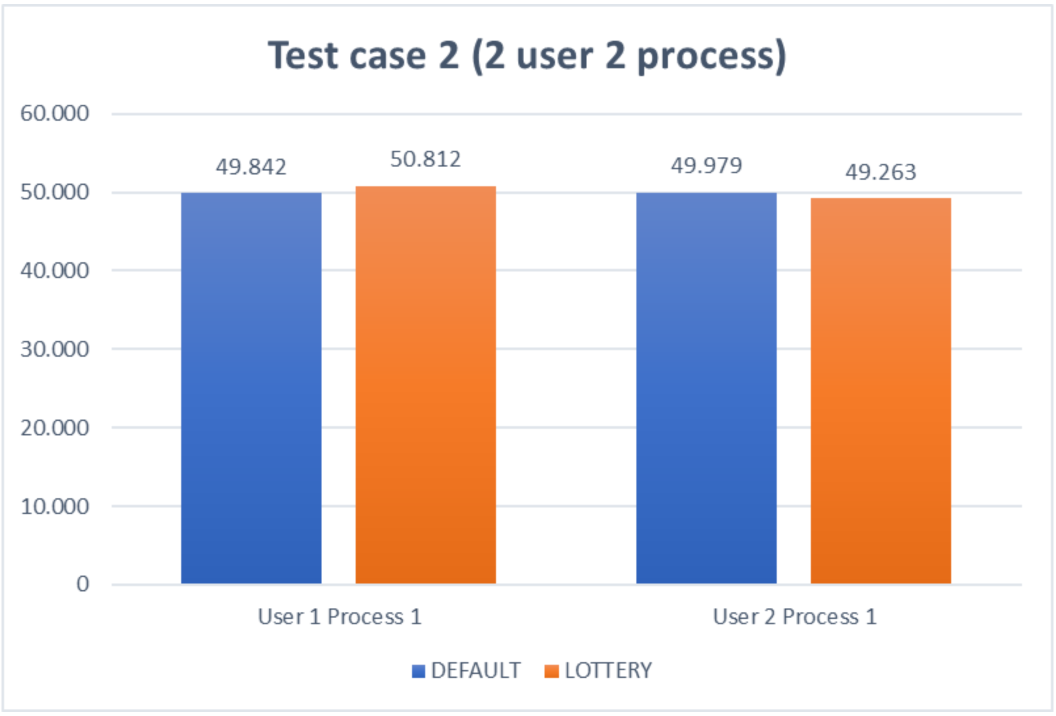


Fig. 4. CPU utilization for 2 users 2 processes

3.2.3 Test Case 2 – 2 users 3 processes

User1 has 1 process, User2 has 2 processes.

CPU Utilization (%)	**User 1 Process 1**	**User 2 Process 1**	**User 2 Process 2**
Default Scheduler	33.435	33.410	33.212
Lottery	50.901	24.835	24.024

Table 3. CPU utilization for 2 users 3 processes

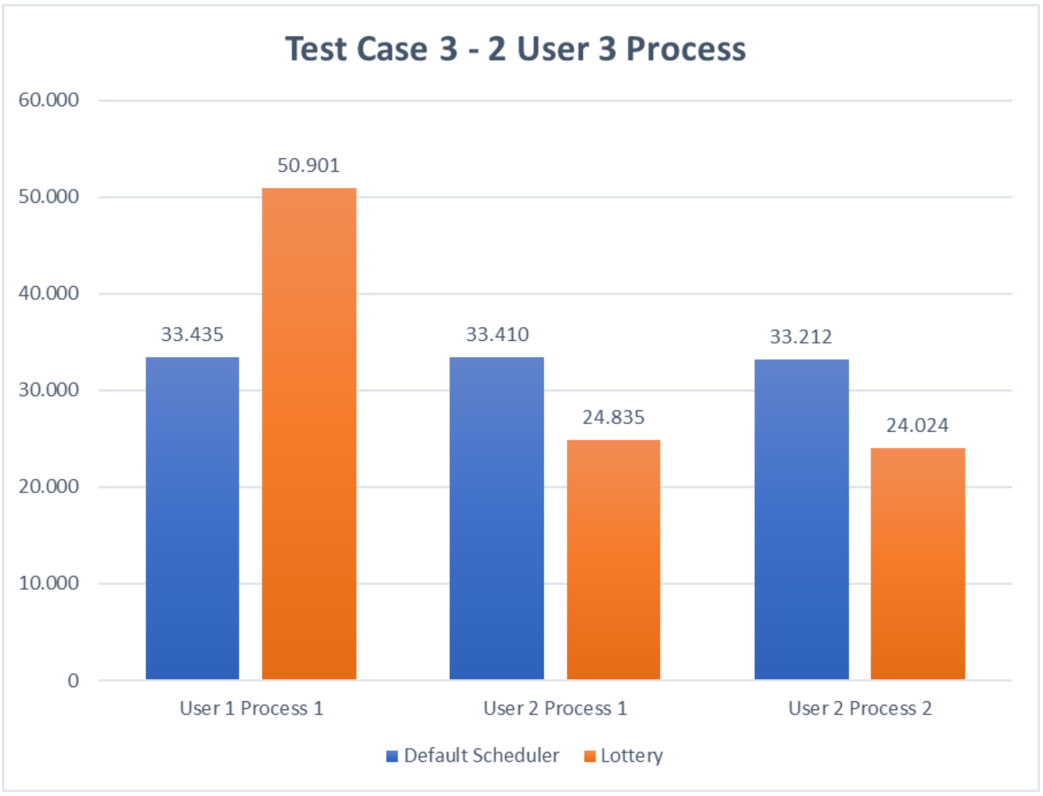


Fig. 5. CPU utilization for 2 users 3 processes

3.2.4 Test Case 3 – 2 users 4 processes

User1 has 1 process, User2 has 3 processes.

CPU Utilization (%)	**User 1 Process 1**	**User 2 Process 1**	**User 2 Process 2**	**User 2 Process 3**
Default Scheduler	24.960	24.898	25.106	25.075
Lottery	50.970	16.814	15.787	16.709

Table 4. CPU utilization for 2 users 4 processes

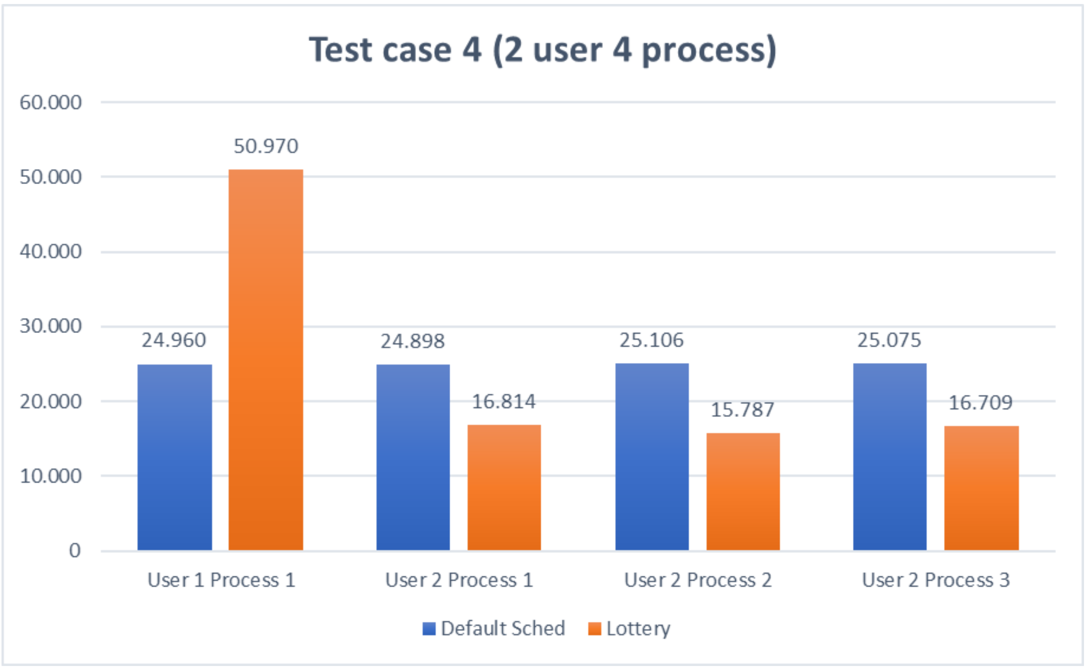


Fig. 6. CPU utilization for 2 users 4 processes

3.2.5 Test Case 5 – 3 users 4 processes

User1 has 2 processes, User2 has 1 process, User3 has 1 process.

CPU Utilization (%)	**User 1 Process 1**	**User 1 Process 2**	**User 2 Process 1**	**User 3 Process 1**
Default Scheduler	24.879	24.788	25.459	24.889
Lottery	17.285	16.791	32.638	33.284

Table 5. CPU utilization for 3 users 4 processes

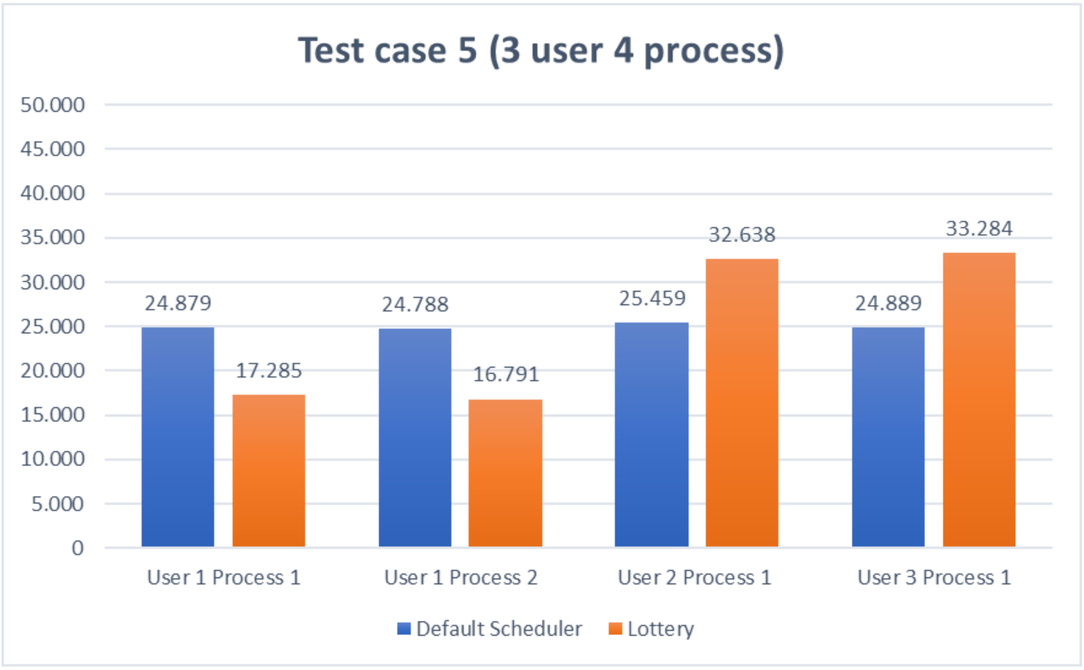


Fig. 7. CPU utilization for 3 users 4 processes

3.2.6 Test Case 6 – 3 users 7 processes

User1 has 1 process, User2 has 2 processes, User3 has 4 processes.

<u>DEFAULT SCHEDULER</u>:

CPU Utilization (%)	**Process 1**	**Process 2**	**Process 3**	**Process 4**
User 1	14.320			
User 2	14.242	14.261		
User 3	14.298	14.209	14.191	14.191

Table 6.1. CPU utilization of default scheduler for 3 users 7 processes

LOTTERY SCHEDULER:

CPU Utilization (%)	**Process 1**	**Process 2**	**Process 3**	**Process 4**
User 1	33.9133			
User 2	16.7156	16.5746		
User 3	8.1953	7.7972	8.3984	8.4052

Table 6.2. CPU utilization of Lottery Scheduler for 3 users 7 processes

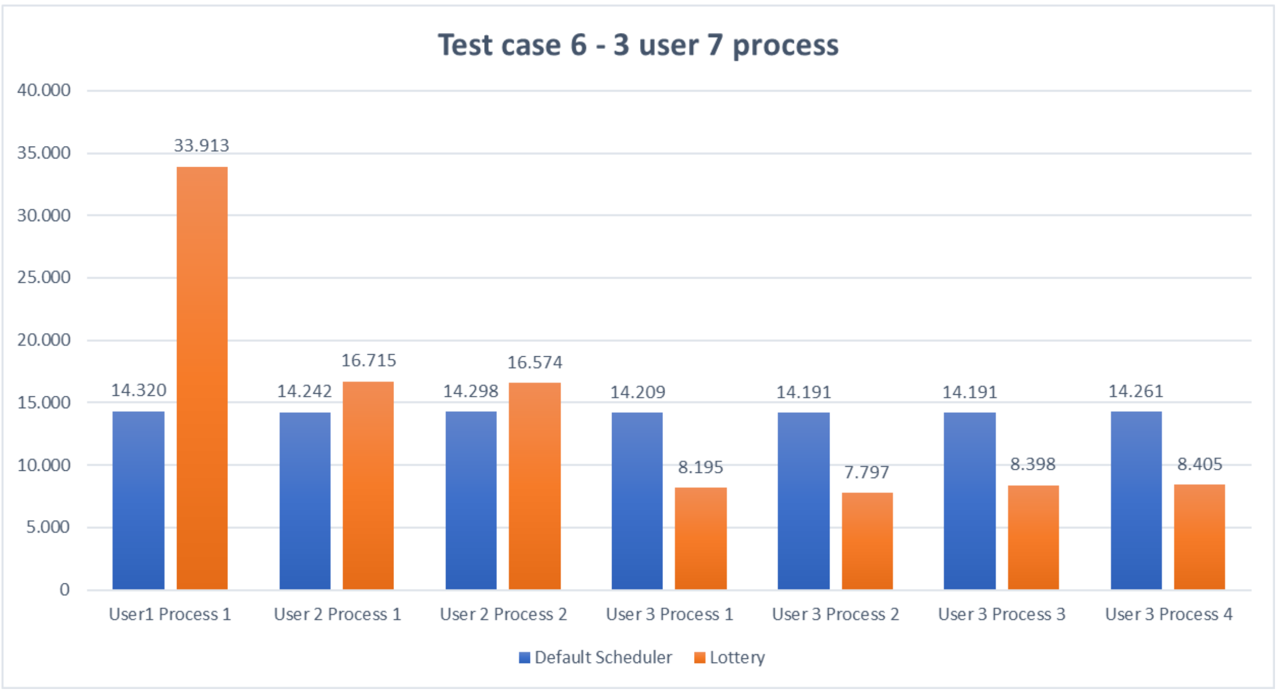


Fig. 8. CPU utilization for 3 users 7 processes

3.2.7 Test Case 7 – 3 users 9 processes

User1 has 2 processes, User2 has 3 processes, User3 has 4 processes.

DEFAULT SCHEDULER:

CPU Utilization (%)	**Process 1**	**Process 2**	**Process 3**	**Process 4**
User 1	11.071	11.204		
User 2	11.207	11.451	11.042	
User 3	10.964	11.146	11.212	10.800

Table 7.1. CPU utilization of default scheduler for 3 users 9 processes

LOTTERY SCHEDULER:

CPU Utilization (%)	**Process 1**	**Process 2**	**Process 3**	**Process 4**
User 1	17.1033	16.5916		
User 2	11.1065	10.9433	11.1666	
User 3	7.7833	8.5030	8.2453	8.2526

Table 7.2. CPU utilization of Lottery Scheduler for 3 users 9 processes

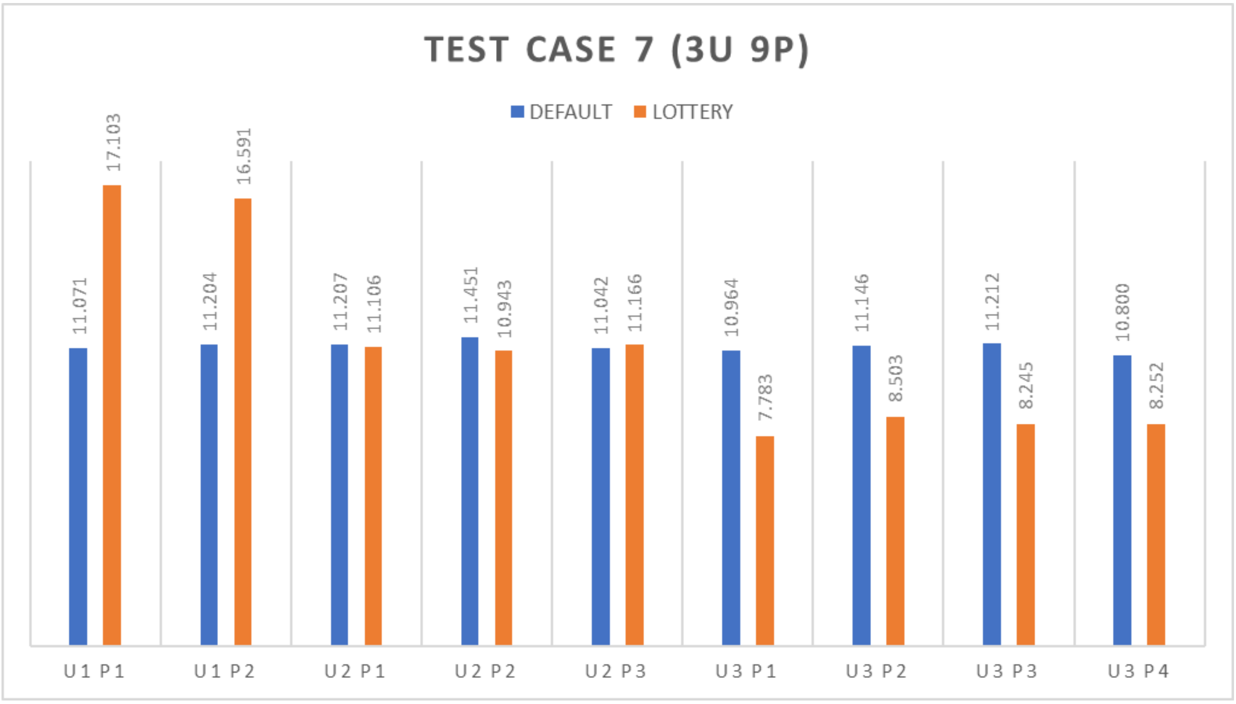


Fig. 9. CPU utilization for 3 users 9 processes

3.3. Results

Our tests revealed that the default scheduler and the Lottery Scheduler have similar values in terms of CPU usage in single-user cases. The default scheduler gives precise results because of the counter values assigned to processes. However, the Lottery Scheduler is dependent on probability, through ticket allocation and random selection. Therefore, it is possible to see minor value differences in the results of the Lottery Scheduler compared to the default scheduler.

In multi-user scenarios, the Lottery Scheduler operates different from the default scheduler. While the default scheduler algorithm only divides the CPU according to processes, the Lottery Scheduler divides the CPU according to the number of users in the system. In the Lottery Scheduler, each user is given an equal number of tickets, which are then divided among their processes. Therefore, as the number of processes a user has increased, the number of tickets assigned to that process decreases. This decreases the likelihood of the process being selected.

As a result, the default scheduler ensures equal CPU usage for each process in multi-user systems. In contrast, the Lottery Scheduler prioritizes users with fewer processes by increasing the probability of running the process of the user with the fewest processes.

4. CONCLUSION

In this project, we implemented a Lottery Scheduler in the Linux 2.4.27 kernel and compared its performance with the default scheduler (SCHED_OTHER). We created a controlled test environment with test users and processes, conducted a series of tests to measure CPU usage under both schedulers and analyzed the results.

As our test results show, while both schedulers provide similar CPU usage for a single user, the default scheduler offers results that are more precise by using a counter. However, in multi-user scenarios, the Lottery Scheduler ensures a fairer distribution of CPU resources among users, preventing any single user from monopolizing the CPU.

By implementing the Lottery Scheduler, we achieved a less precise, and more complex scheduling system. However, it is fairer in multi-user environments. This suggests that the Lottery Scheduler is better suited for systems with many users, while the Default Scheduler is more useful for simpler, less complex systems.

REFERENCES

[1] Baydere, Sebnem., Perente, Osman Kerem. (2024). CSE331: Operating Systems Design [Lecture & Lab Notes]. Yeditepe University.

[2] Petrou, David et al. (1999). "Implementing Lottery Scheduling - Matching the Specialization in Traditional Schedulers." Proceedings of the 1999 USENIX Annual Technical Conference (USENIX 1999).

[3] Waldspurger, Carl A., Weihl, William E. (1994). "Lottery Scheduling: Flexible Proportional-Share Resource Management." Proceedings of the 1994 Operating Systems Design and Implementation Conference (OSDI '94).

[4] Palix, Nicholas et al. (2011). "Fauts in Linux: Ten Years Later." ACM SIGPLAN Notices.

[5] Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/2.4.27/source/kernel/sched.c