LOTTERY SCHEDULER REPORT

by

BARIŞ CAN SERTKAYA

BATUHAN AKIN

KIVANÇ ONAT TÜRKER

ORHAN BARIŞ UZEL

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2024

ABSTRACT

We developed Lottery Scheduling algorithm for Linux kernel, in this report we will be exploring implementation and analysis of our Lottery Scheduler. The Lottery Scheduler assigns “ticket” to processes and uses random draws to select the next process to run, aiming for fairer CPU time distribution. Here in the report we done our tests and will present our results.

Contents

1. INTRODUCTION

Process management in OS involves several things can be done on processes which are creating, destroying and managing while the scheduler decides which one to run. This project we are introducing a Lottery Scheduler, an alternative to traditional scheduling method. Lottery Scheduler uses tickets to randomly select processes for execution, aiming to improve fairness and efficiency. We modified the Linux kernel, added new system calls and created a test file to monitour our scheduler's performance. By comparing the Lottery Scheduler with Default Scheduler, we monitored its impact on CPU distribution. This report details the implementation, challenges and performance evaluation of the Lottery Scheduler.

We modified the "Repeat Schedule" part of the Default Scheduler Algorithm and recreated that part for our Lottery Scheduling Algorithm. We also implemented switching system call to change the flag and turn default scheduler into our lottery algorithm. In this report we will also discuss that changes aswell.

2. DESIGN AND IMPLEMENTATION

2.1. Default Scheduler

Default Scheduler is designed to prioritize every process should use CPU time equally and fair. In the Default Scheduler we have nice value which determines the priority values. Each process is given with that nice numbers.

This means that with time-sharing algorithm every process gets close and fair time frames for their work. It uses priority (the “nice” number) and time slices to figure out which process gets to use the CPU next.

2.2. Lottery Scheduler

Our Lottery Scheduler introduces a probalistic approach to CPU scheduling. Each process is assigned a number of “tickets” and the scheduler randomly selects a ticket to determine the next process to run. This method aims to provide a new way to distribute CPU time, which is worse in means of fairness, as process with more tickets have a higher probability of being selected but are not guaranteed to monopolize the CPU.

As Key features of the Lottery Scheduler we have:

Ticket Allocation: Processes receive a certain amount of tickets based on their priority. High-priority processes get more tickets.

Random Selection: A random number generator selects a ticket and the corresponding process is scheduled to run.

2.3. Implementation

Modifying the Scheduler

The Implementation required significant changes to the kernel's "sched.c" file. We introduced a new policy constant for the Lottery Scheduler and we modified the "schedule()" function to work with the ticket-based selection mechanique.

As Key features of modifications:

Random Selection Algorithm: We integrated a random number generator to select tickets and determine the next process to run.

System Call Integration: We integrated a switch-flag system call so that we can allow user-space to active Lottery Scheduler.

Code Implementations

Initially we had to define constants at the beginning of the "sched.c" file:

Outside of The Lottery Algorithm

```
#define MAX_USERS 3
#define BASE_TICKET_PER_USER 12
```

Since we were supposed to work with maximum 3 users MAX_USERS constant is set to 3, it is going to be used to set the size for user array we will use further in the code.

Since 12 is divisible by 3 and 4 we have chosen 12 as the ticket count per user.

```
extern int scheduleFlag;
```

This extern int variable is used to access to scheduleFlag which controls the algorithm switch.

When scheduleFlag is equal to 0 default algorithm runs, when it is 1 system switches to lottery algorithm. Switching mechanism is handled by the system call we have added.

Inside of The Lottery Algorithm

```
int user_process_count[MAX_USERS] = {0};
    int user_tickets[MAX_USERS] = {0};
    int total_tickets = 0;
    unsigned int lottery_ticket;
    int cumulative_tickets = 0;
    int i;
```

At the beginning of the schedule function we have added the above arrays and variables. Newly added arrays are for to hold the current process count per existing users and how many tickets do each process of each user have.

lottery_ticket is straightforward, it is for the ticket number that is generated randomly.

cumulative_tickets is for checking in which process does the random generated lottery ticket falls.

I variable is a general loop iterator.

```
next = idle_task(this_cpu);
        total_tickets = 0;
        cumulative_tickets = 0;
        for (i = 0; i < MAX_USERS; i++) {
            user_process_count[i] = 0;
            user_tickets[i] = 0;
        }
```

This part resets the array, so that at each run of the algorithm a newly added user or process can be added to the system.

```
list_for_each(tmp, &runqueue_head) {
            p = list_entry(tmp, struct task_struct, run_list);
            if (p->state == TASK_RUNNING) {
                if(p->uid > 1000){
                    int index = p->uid % 10;
                    user_process_count[index]++;
                }
            }
        }
```

Above code is the mechanism to find how many processes do each user have.

```
for (i = 0; i < MAX_USERS; i++) {
            if (user_process_count[i] > 0) {
                user_tickets[i] = BASE_TICKET_PER_USER / user_process_count[i];
            }
        }
```

Here we find how many tickets do processes of each user have.

```
list_for_each(tmp, &runqueue_head) {
```

```
            p = list_entry(tmp, struct task_struct, run_list);
            if (can_schedule(p, this_cpu)) {
                if(p->uid > 1000){
                    int index = p->uid % 10;
                    p->tickets = user_tickets[index];
                } else {
                    p->tickets = BASE_TICKET_PER_USER;
                }
                total_tickets += p->tickets;
            }
        }
```

After finding how many tickets do each processes need to have , now with the above code we set each processes ticket values, which we have defined in task_struct, to the ticket numbers we have calculated.

```
  if (total_tickets == 0) {
            next = idle_task(this_cpu);
       }
```

If the total tickets equal to zero, that means no processes are working, so in order to avoid a divide by zero error we set the next process to run manually.

```
else {
            get_random_bytes(&lottery_ticket, sizeof(lottery_ticket));
            lottery_ticket %= total_tickets;

            list_for_each(tmp, &runqueue_head) {
                p = list_entry(tmp, struct task_struct, run_list);
                if (can_schedule(p, this_cpu)) {
                    cumulative_tickets += p->tickets;
                    if (cumulative_tickets > lottery_ticket) {
                        next = p;
                        break;
                    }
                }
            }
       }
```

If there are processes to run then generate a random lottery ticket number, then start adding the ticket numbers from the first process untill we reach the lottery ticket, then we set the next process the run.

System Call

```
#include <linux/switchFlag.h>

int scheduleFlag = 0;

asmlinkage int sys_switchFlag(int value)
{
     scheduleFlag = value;
     return scheduleFlag;
}
```

The above code is the system call for switching the algorithm.

Adding Tickets in Task_Struct

```
struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch with care
     */
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags;    /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit;    /* thread address space:
                        0-0xBFFFFFFF for user-thead
                        0-0xFFFFFFFF for kernel-thread
                    */
    struct exec_domain *exec_domain;
    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth;     /* Lock depth */

/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
    int tickets;
    long counter;
    long nice;
```

A new ticket field is added inside of the task_struct.

```c
#include <linux/switchFlag.h>

int main(int argc, char ** argv)
{
    int value;
    value = atoi(argv[1]);
    switchFlag(value);

    printf("%d\n",value);
    return 0;
}
```

We use the above code to use system call to switch the algorithm in the home folder.

Issues Encountered

We faced difficulties when declaring and defining schedule flag using extern keyword. Some implementations didn't reflect switched flag to sched.c we had to try different locations.

Improper kernel compilations led to buggy kernel builds which we couldn't see our changes in the scheduling algorithm despite successful compilations.

We encountered kernel freezes and unresponsive virtual machine after switching our schedule flag due to malfunctioning scheduling algorithm.

We faced a bug which made "top" command froze our virtual machine. This led to kernel panic and we had to recompile several times.

Faulty use of "unlikely" part in the schedule function led to unexpected algorithm behaviours and kernel freezes.

After completing implementation of scheduling algorithm, when we switched schedule flag before creating user processes, our modulus operation threw divide by zero error.

3. TESTS and RESULTS

In this part we will share our test results for both Default Scheduler and Lottery Scheduler.

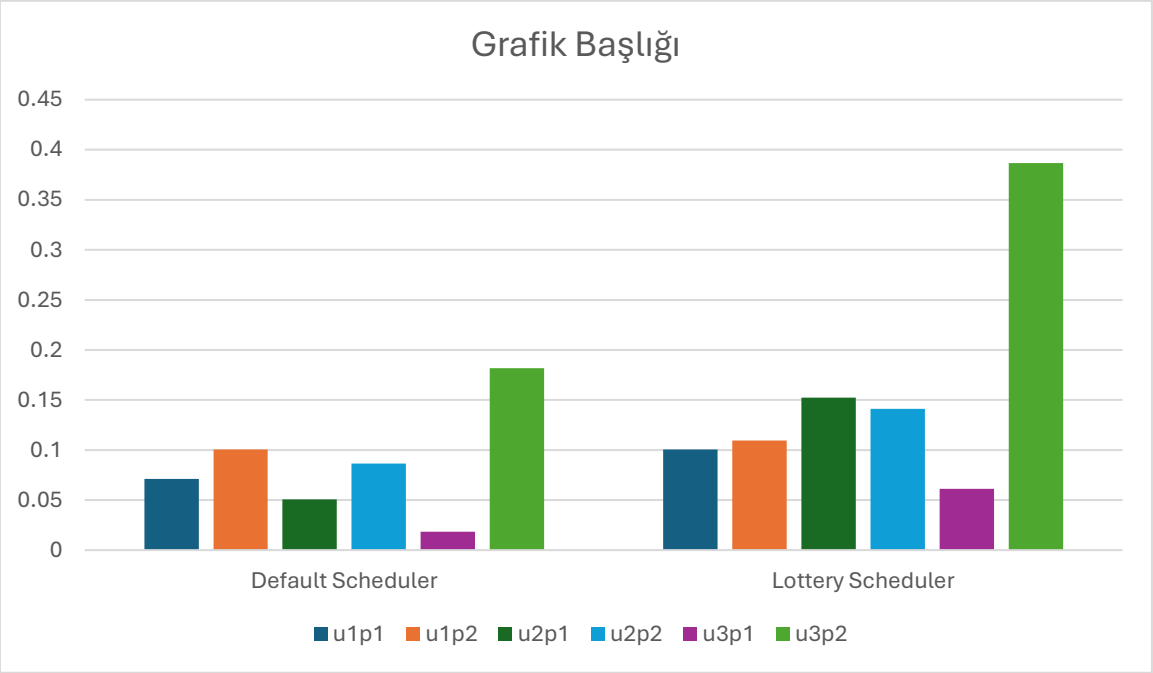
For the testing part, we have tested in 6 different forms via changing the user and process counts. For each testing style we gathered 100 process data 10 times for both default and lottery schedulers. After that we calculated mean cpu usages and calculated MSE. In the next part, we will share our MSE comparison data.

We wrote an sh file to automatize the process, it gave us the CPU usage averages. Then ran it in the OS to get the averages of the data. Here is an example of extraction of Average CPU Usage:

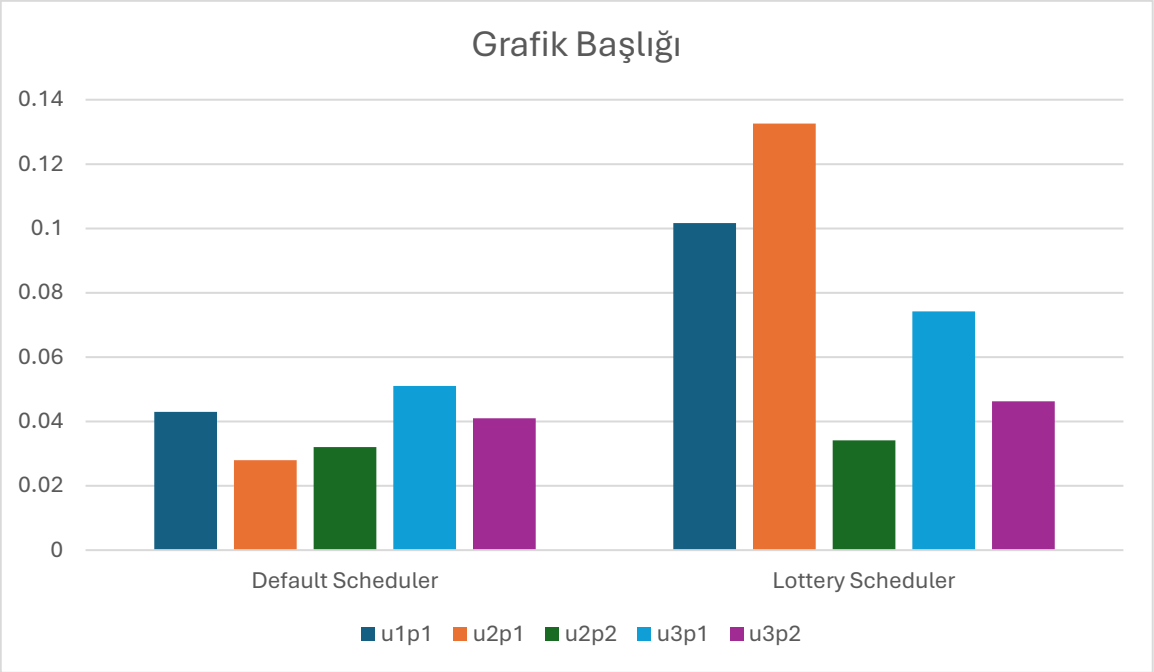
```
CSE331:~# ./extractor.sh 747 lotterylsched100test1.txt
```

- `Average CPU Usage: 16.237`

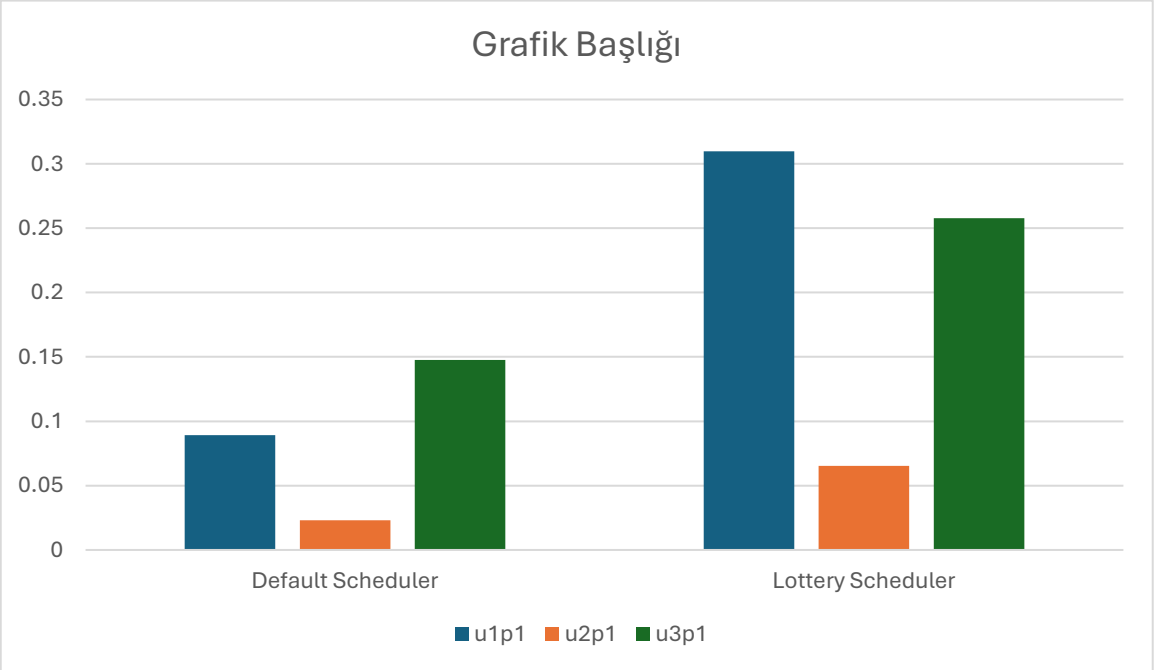
3.1. TEST 1: (3 USER 6 PROCESS)



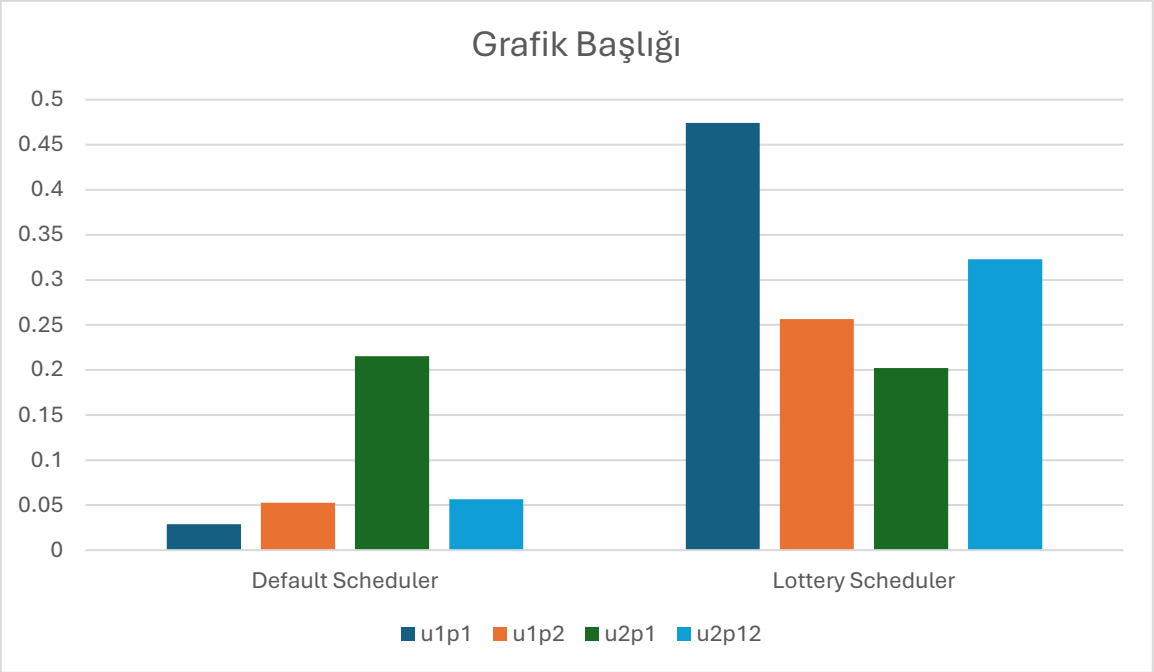
3.2. TEST 2: (3 USER 5 PROCESS)



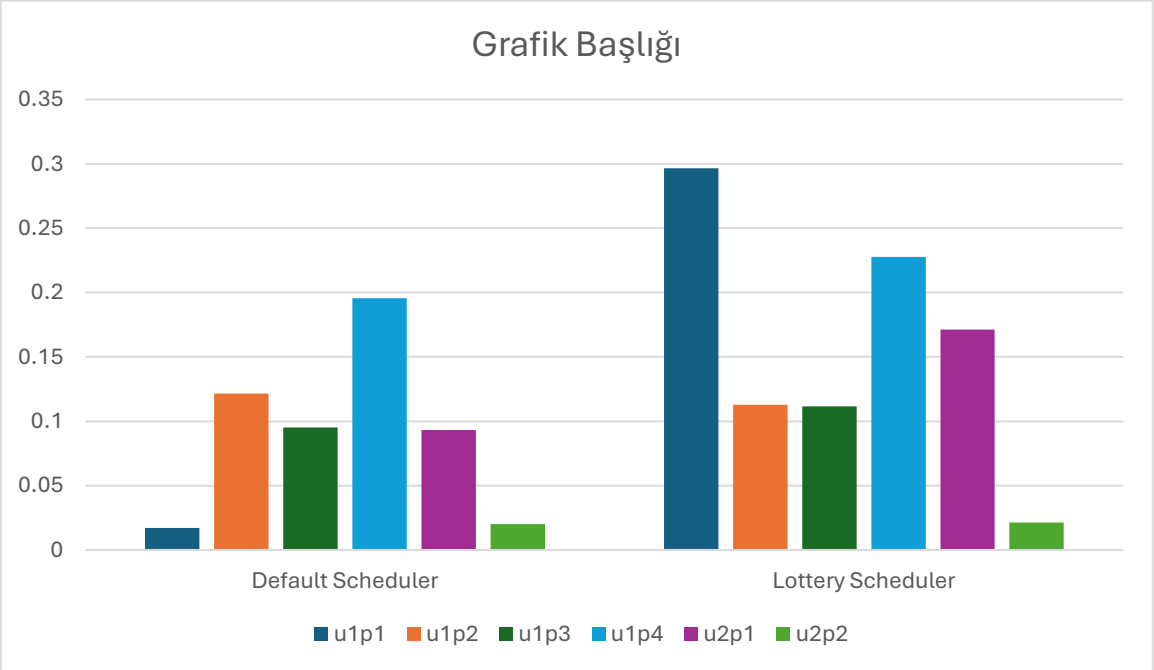
3.3. TEST 3: (3 USER 3 PROCESS)



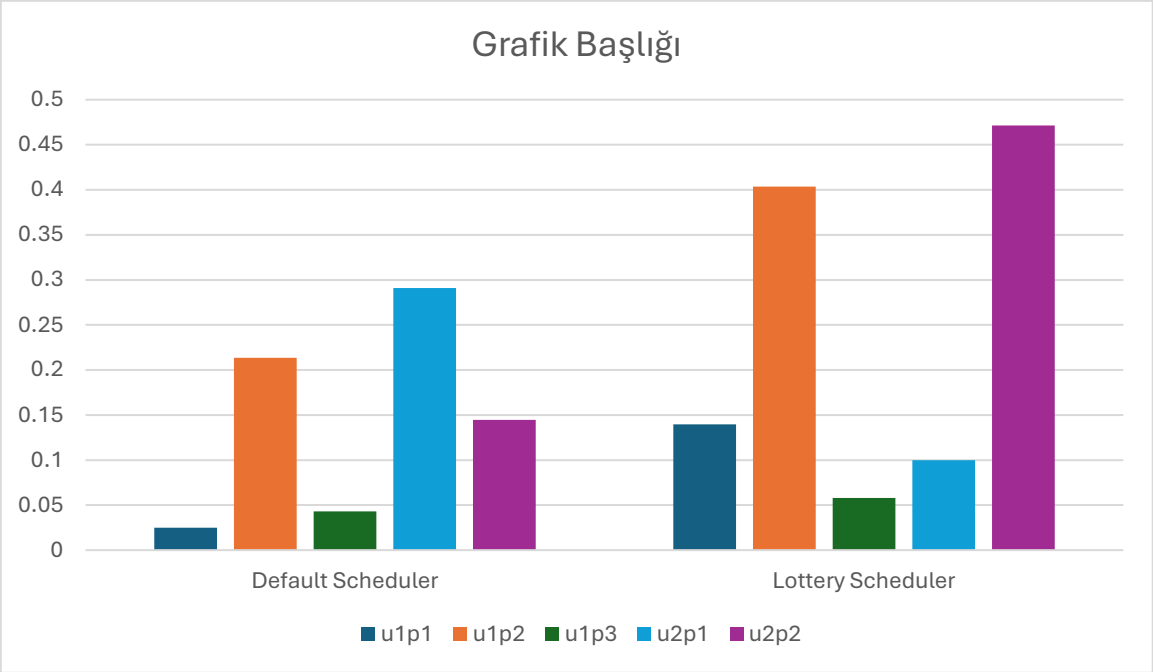
3.4. TEST 4: (2 USER 4 PROCESS)



3.5. TEST 5: (2 USER 6 PROCESS)



3.6. TEST 6: (2 USER 5 PROCESS)



3.7. DISCUSSING THE TEST RESULTS

The tests conducted compare the performance of the Default Scheduler and the Lottery Scheduler across various user and process configurations. Here are the key findings:

CPU Utilization: The Lottery Scheduler generally shows higher variability in CPU utilization. This variability is expected due to the probalistic nature of the Lottery Scheduling.

Mean Square Error(MSE): Lottery Scheduler has higher error margins. This indicates that while the Lottery Scheduler aims for fairness, the randomness introduces inconsistencies.

Process Fairness: Lottery Scheduler achieves a fairer distribution of CPU time among processes over multiple runs. However, in specific instances, process with fewer tickets may still receive less CPU time.

Performance Impact: In some cases, the Lottery Scheduler resulted in more balanced CPU usage across processes, but this came at the cost of increased system overhead and complexity

4. CONCLUSION

Our main goal was to use a randomized strategy to improve CPU time distribution efficiency and fairness. The Lottery Scheduler provides an alternative method to handling process scheduling over the default scheduler by assigning tickets to processes and applying a random selection mechanism. At the end of the tests and significant changes. We finally managed to apply a random selection mechanism which is Lottery Scheduling into our system. But it turned out that because of the random selection, error margins are higher than the default scheduler. So in these terms, using Lottery Scheduler for data with stability needs is not suggested. Lottery Scheduler should only be used in the cases for randomness is acceptable.

Overall, the Lottery Scheduler provides an interesting alternative to the Default Scheduler, offering benefits in fairness at the cost of increased variability and system complexity. Further fine-tuning could mitigate some of the higher error margins observed.

REFERENCES

Operating System Concepts 10th Edition

by Abraham Silberschatz (Author), Peter B. Galvin (Author), Greg Gagne (Author)

Modern Operating Systems 4th Edition

by Andrew Tanenbaum (Author), Herbert Bos (Author)

Class Material Project Videos & Examples

by Prof. Şebnem Baydere (Professor), O. Kerem Perente (Assistant), Gülşah Gökhan Gökçek (Assistant)