Implementation on Linux Kernel 2.4.27

by

Nilhan Süer

Ezgi Kara

Ceren Yazgan

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2023

ABSTRACT

Abstract In this project, we develop a new chance-based scheduling algorithm for the Linux kernel 2.4.27 and evaluate its fairness performance in comparison to the standard Linux scheduler. In a VirtualBox VM, all coding and testing are carried out. operating a 2.4.27 kernel.

TABLE OF CONTENTS

Contents

1. INTRODUCTION

In operating systems, a scheduler is a vital component that oversees the execution of processes or threads on a computer system's CPU (Central Processing Unit). Its main objective is to distribute the CPU's time among various processes or threads, aiming for efficient and equitable utilization of system resources.

The scheduler's role involves determining which processes or threads are allowed to run, when they can run, and for how long. This decision-making process relies on specific scheduling algorithms, which may vary depending on the operating system and its configuration. The chosen scheduling algorithm seeks to optimize system performance, responsiveness, throughput, fairness, or a combination of these factors.

Having the best possible outcome and fairness throughout the systems has always been the priority. For this project we tried to see if we could create a fairer scheduling algorithm than the default Linux 2.4.27 scheduler. We observed our experiments average CPU utilization and MSE calculation for several users with several process'. And then, we compared the test cases between each other. To achieve a fair full system we made the required adjustments to this kernel and explained this whole procedure from to start to finish.

2. DESIGN and IMPLEMENTATION

The scheduling algorithm that we will develop and use is called a Fair-Share Scheduling. We'll talk over the intricacies of the Default and Fair-Share Scheduler designs.

2.1. DESIGN

The order in which processes are scheduled under the "SCHED_OTHER" scheduling policy is determined by the scheduler in the Linux 2.4.20 Kernel using counter and nice values.

The counter value represents the amount of CPU-Time that a process has consumed. Based on its static priority, a process is given a counter value when it is first created. The counter value drops as the operation progresses.

The process is preempted (temporarily paused) and put back in the run queue when the counter value hits zero. The epoch ends and new counter values are generated for each process once all the counter values for processes in the run queue are zero.

A user or a program can alter the nice value to change the priority of a process in relation to other processes. lesser lovely values indicate higher priorities, while higher nice values indicate lesser priorities.

- Higher nice → low priority (nice : 0 – 20)
- Lower nice → high priority (nice : -19 – 0)
- Medium priority → nice=0 (in fork, a process gets 0 value for nice)

The periodic process selection mechanism computes a weight value using the goodness function in "sched.c". A process's priority is represented by its weight value.

- If counter value of a process is 0 then weight is directly = 0
- Processes with SCHED_OTHER → weight = 20-nice+counter
- Processes with RT_PRIORITY → weight = 1000 + rt_priority

When the counter value is reset, it is modified in accordance with the pleasant value of the process. This effectively enables users and programs to change the priority of processes by allowing the Kernel to allocate more CPU time to processes with lower nice values and less CPU time to processes with higher nice values. We were entrusted with implementing a chance-based scheduling system. The "Fair-Share Scheduler" algorithm.

2.2. IMPLEMENTATION

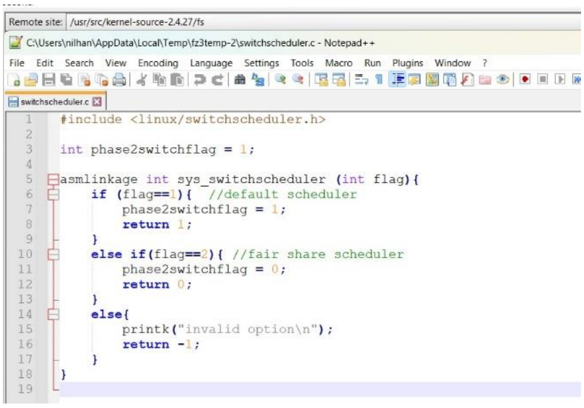


Figure 1 *switchscheduler() system call code in switchscheduler.c*

Figure 1 shows how we introduced a straightforward system call to the kernel that modifies the value of the "phase2switchflag" int variable, which we specified above.

We may divide the functionality of the default scheduler using if else statements and switch between the default scheduler and our scheduler whenever we want by using the "phase2switchflag" flag in the "sched.c" file.

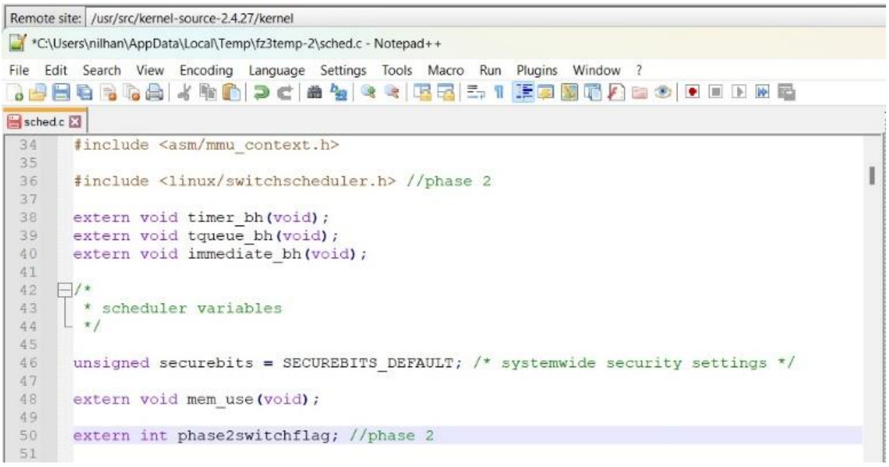


***Figure 2** Functions defined in "sched.c"*

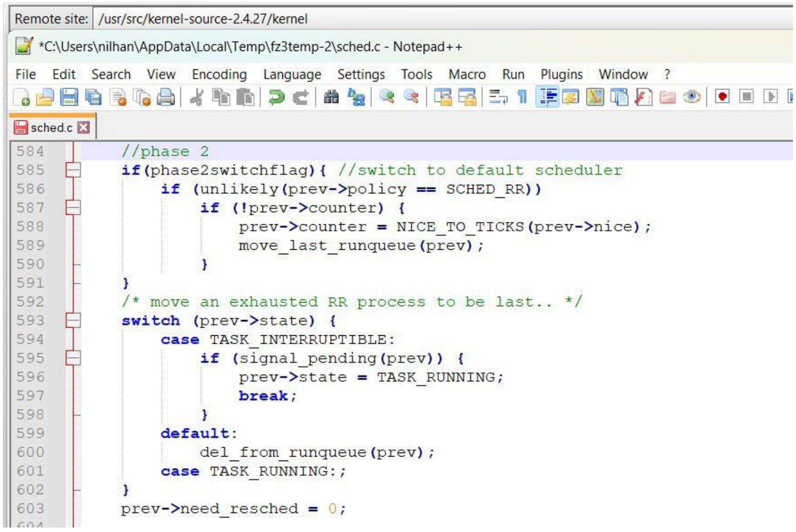
In Figure 2 are the three functions we've implemented in "sched.c". The functions **timer_bh**, **tqueue_bh**, and **immediate_bh** are declared with **extern** keyword, indicating that they are defined elsewhere and their definitions can be found in another file or module.

timer_bh(void): This function is likely related to handling timer events or tasks. It might be responsible for processing or managing timers in the system.

tqueue_bh(void): This function might be associated with a task queue. It could be responsible for processing tasks or events from a queue.

immediate_bh(void): The name suggests that this function is associated with immediate or urgent processing. It may handle time-sensitive or high-priority tasks.

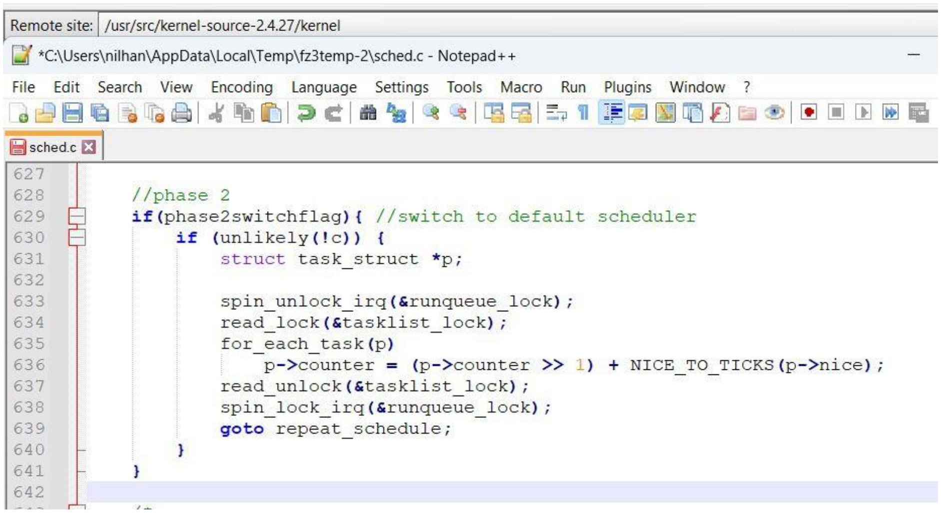
It's important to note that the actual functionality and purpose of these functions can only be determined by examining the implementation or the codebase where they are defined.



```
//phase 2
if(phase2switchflag){ //switch to default scheduler
    if (unlikely(prev->policy == SCHED_RR))
        if (!prev->counter) {
            prev->counter = NICE_TO_TICKS(prev->nice);
            move_last_runqueue(prev);
        }
}
/* move an exhausted RR process to be last.. */
switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:;
}
prev->need_resched = 0;
```

Figure 3 *Time Slice defined in "sched.c"*

This code fragment deals with a particular situation in the default scheduler where, in phase 2, if the preceding process was scheduled with the RR policy and it has used up all of its time slice, it is granted a fresh time slice depending on its "nice" value and is moved to the tail of the run queue. This conduct gives other processes a chance to run and aids in ensuring fairness in process scheduling.



```
//phase 2
if(phase2switchflag){ //switch to default scheduler
    if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }
}
```

Figure 4 *Reacquiring the run queue lock and jumps to a label to repeat the scheduling process in "sched.c"*

When phase2switchflag is true and c is zero, the activities in this code fragment are carried out. After releasing the run queue lock and updating all task counts according to their "nice" values, it reacquires the run queue lock and jumps to a label to resume scheduling. The overall context of the code and the scheduling method being used determine the precise function and setting of these operations.

```
sched.c

int user_process_count[10]; //phase 2

asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;


```

Figure 5 *Array Initialaziton in "sched.c" in "schedule" function*

We initialized an array called "user_process_count" out of the "schedule" function which has 10 elements and keep the number of processes of users.

```
sched.c
    }
    //phase 2 fair-share scheduler
    else {
        if (unlikely(!c)) {
            struct task_struct *p;

            memset(user_process_count, 0, sizeof(user_process_count));

            spin_unlock_irq(&runqueue_lock);
            read_lock(&tasklist_lock);

            for_each_task(p) {
                if(p->uid>=1000 && p->uid<=1009) {
                    user_process_count[p->uid%10]++;
                }
            }

            for_each_task(p) {
                int process_multiplication = 1;

                int i = 0;
                while(i < 10){
                    if(user_process_count[i] != 0){
                        process_multiplication *= user_process_count[i];
                    }
                    i++;
                }

                process_multiplication *= 6;

                if(p->uid>=1000 && p->uid<=1009) {
                    p->counter = process_multiplication / user_process_count[p->uid%10];
                }
                else {
                    p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
                }
            }
            read_unlock(&tasklist_lock);
            spin_lock_irq(&runqueue_lock);
            goto repeat_schedule;
        }
```

Figure 6 *Fair-Share Scheduler in "sched.c"*

In Figure 6, the else part is active when the scheduler is switched to the fair-share scheduler. In the first “for_each_task” block, number of processes of the users is initialized to the “user_process_count” array. We created the indexes of the array by taking the mode of the userIDs. Then in the second “for_each_task” block, we checked if the related indexes of the array is equal to 0. If it is not, then number of processes is mutliplied and initialized to a variable called “process_multiplication”. Then we multiplied it by 6 to make the values bigger without altering the results. At the end, we distributed the “p->counter” values by dividing “process_multiplication” by the number of processes of the related user to make the distribution fair.

main.c

```c
#include <linux/switchscheduler.h>
#include <stdio.h>

main(){
     int scheduler_type;
     printf("Enter 1 or 2 for the scheduler type: ");
     scanf("%d",&scheduler_type);
     switchscheduler(scheduler_type);
}
```

Figure 7 *Main Function*

The program switches between default and fair-share scheduler with respect to the input values. It switches to the fair-share scheduler if the user types "2" and it switches to the default schedular if the user types "1".

3. TESTS and RESULTS

3.1 TESTS

In the testing part, we collected CPU Utilizition values for the processes in each case. There were 5 tests that consist of different number of users and processes.

With the use of “ top –n 100 –d 1 –b > d1t1.txt” command, we received data. Each “top” command took 1 second to collect the data. We have totally 5000 samples. There were 5 testcases and each of them had 10 tests.

All the tests were repeated for both the default and our Fair-Share scheduler. After all our data had been collected, we calculated the Average CPU Utilization and Mean Square Error (MSE) for each process of each test

```c
#include <stdio.h>

main(){
    int i=5;
    while(1){
        i=(i+1) % 1000;
    }
}
```

Figure 8 *C program*

To create the infinite loop we used an CPU intensive C code which makes the program repeated continuously as shown in the Figure 8.

```
CSE331:~# cat d1t1.txt d1t2.txt d1t3.txt d1t4.txt d1t5.txt d1t6.txt d1t7.txt d1t8.txt d1t9.txt d1t10.txt | grep 'u1p1' | awk 'BEGIN{total=0;num=0}{num=num+1;total=total+$9;}END{avg=total/num;print "total=", total;print "average=", avg;}'
```

Figure 9 *Linux Command*

In figure 9, we wrote a command for system to read the testcase files and sum up CPU Utilization values for each case. And with division opearation by the total number of testcases, we reached the average CPU Utilization.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Figure 10 *Mean Square Error Formula*

We used the formula in Figure 10 to calculate the mean square error. ^Y is the predicted value for each processes. And Y is the CPU Utilization value obtained from test.

3.1.1. TEST 1

In TEST1; user1 has 2 processes, user2 has 1 processes.

Default Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	33.2185	33.182
USER2	33.4381	

Fair-share Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	25.036	25.0652
USER2	49.8735	

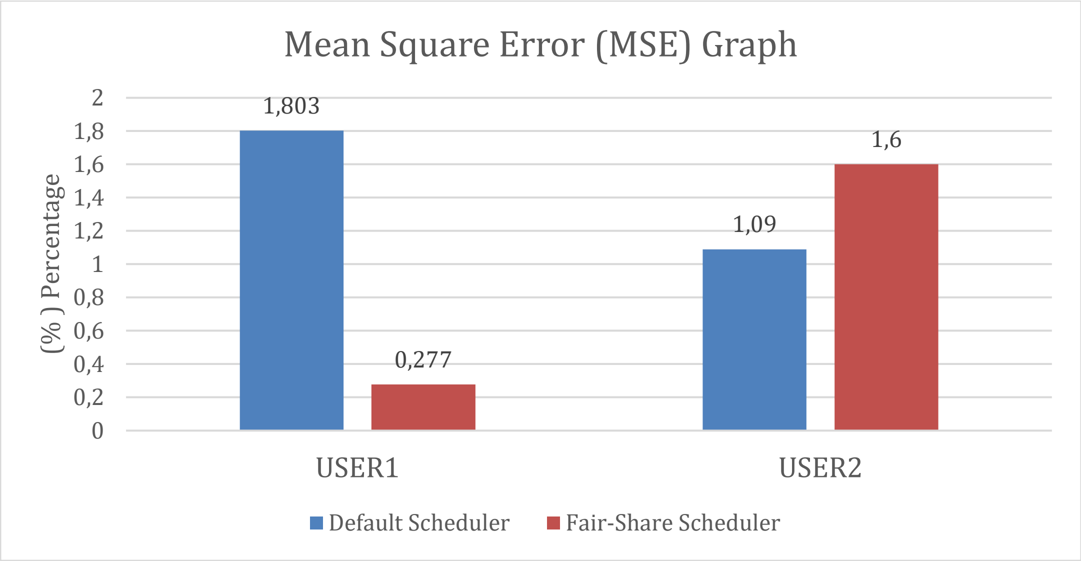


Figure 11 *MSE Graph for Test1*

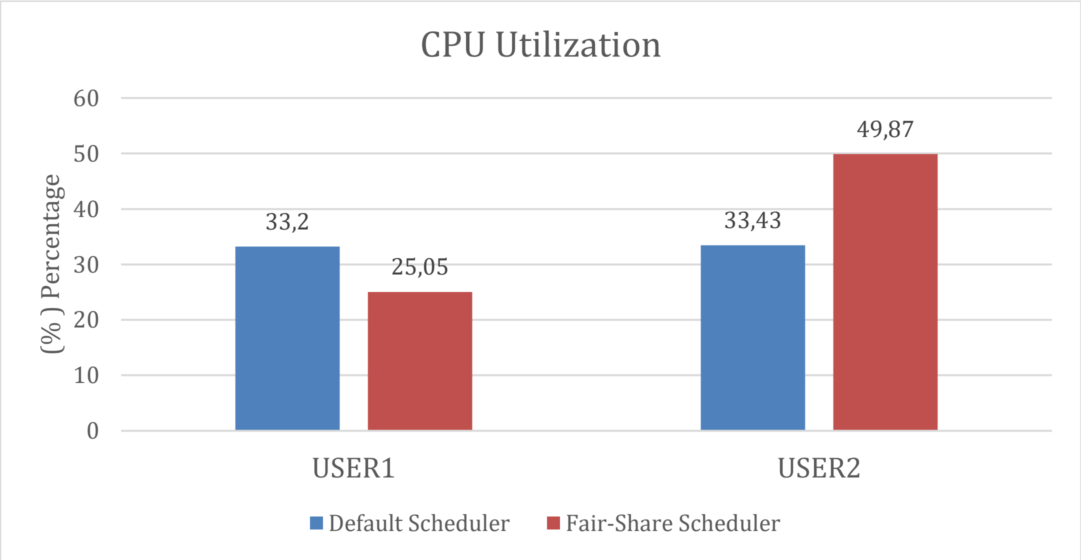


Figure 12 *CPU Utilization Graph for Test1*

3.1.2. TEST 2

In TEST2; user1 has 2 processes, user2 has 2 processes, user3 has 1 processes.

Default Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	19.8308	19.9571
USER2	19.9362	20.1091
USER3	19.9327	

Fair-Share Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	16.548	16.6883
USER2	16.8473	16.6812
USER3	33.1229	

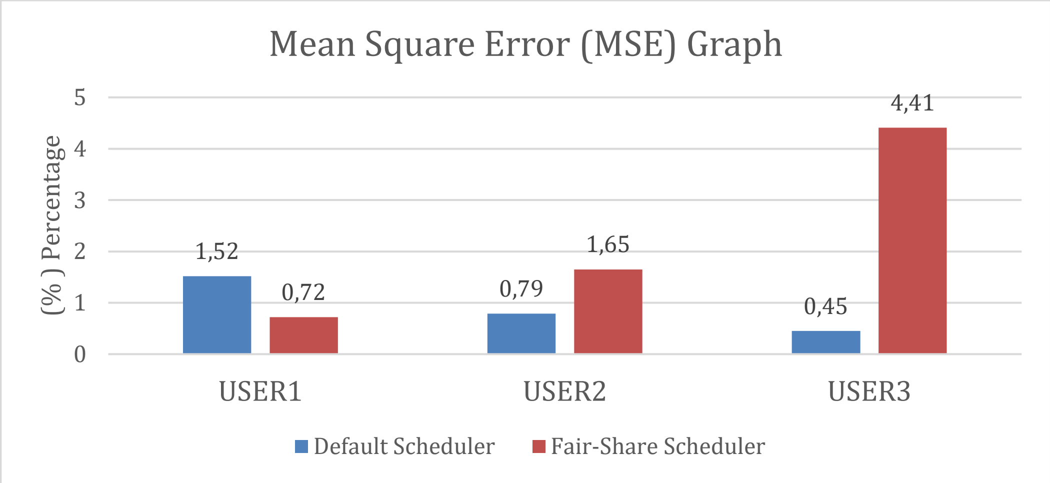


Figure 13 *MSE Graph for Test2*

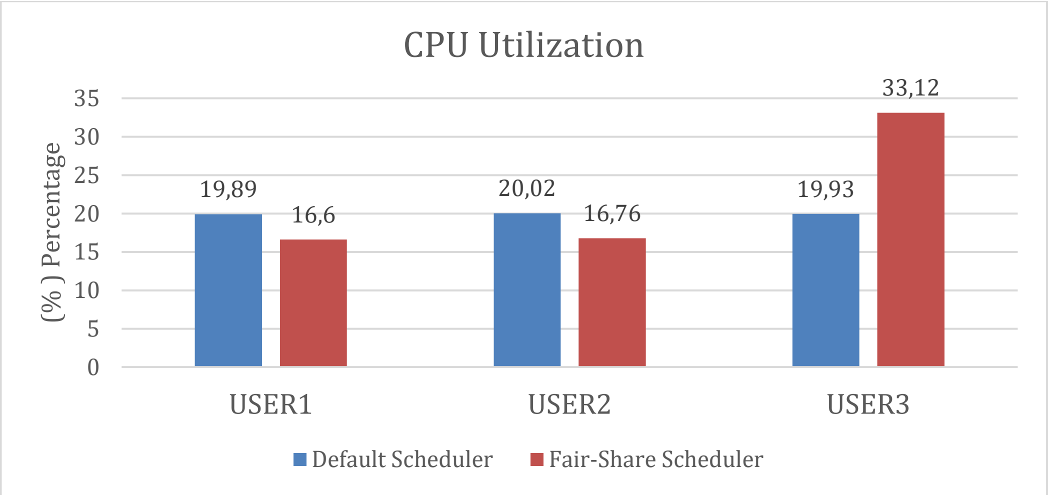


Figure 14 *CPU Utilization Graph for Test2*

3.1.3. TEST 3

In TEST3; user1 has 2 processes, user2 has 2 processes, user3 has 2 processes, user4 has 2 processes.

Default Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	12.4645	12.5497
USER2	12.4914	12.4778
USER3	12.5332	12.3706
USER4	12.3846	12.4857

Fair-Share Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2
USER1	12.4785	12.4962
USER2	12.4932	12.5169
USER3	12.5213	12.4823
USER4	12.5113	12.5257

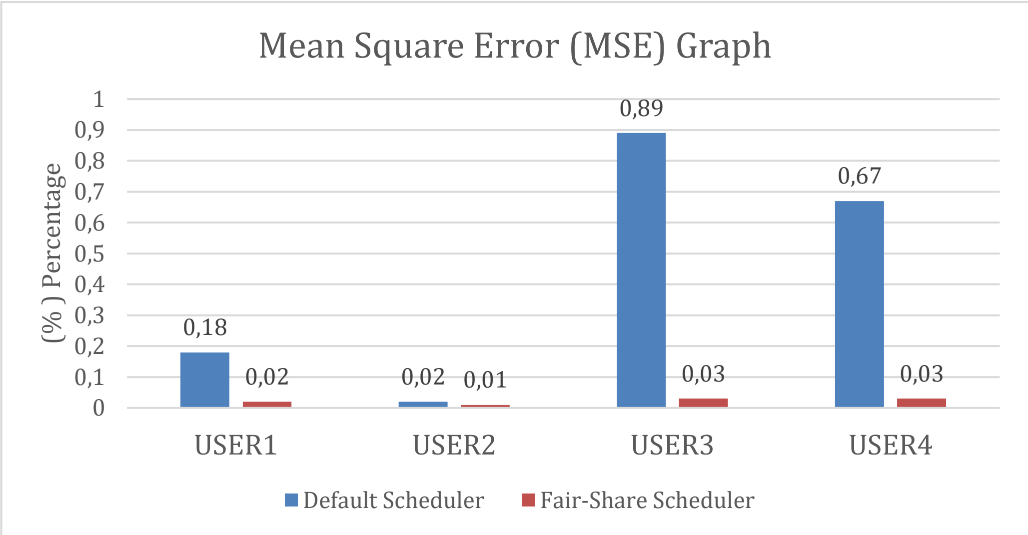


Figure 15 *MSE Graph for Test3*

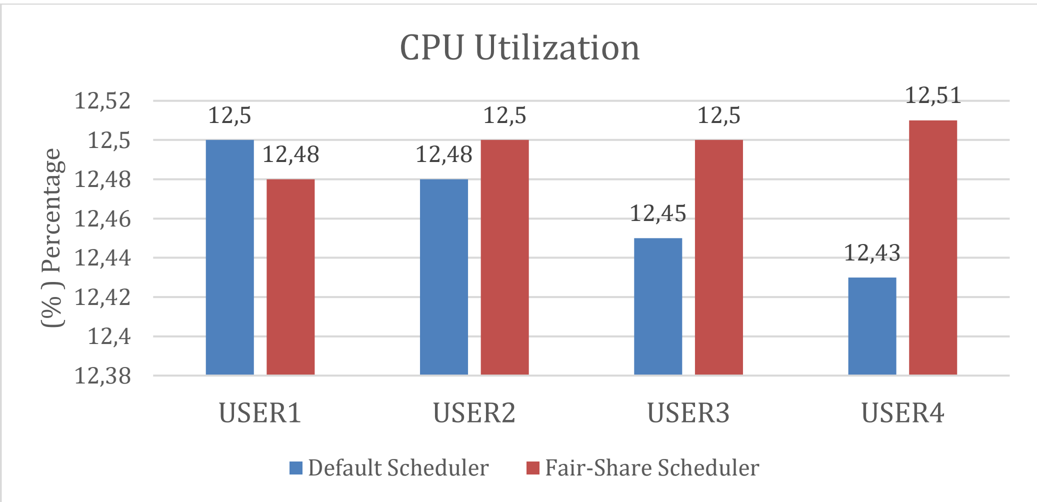


Figure 16 *CPU Utilization Graph for Test3*

3.1.4. TEST 4

In TEST4; user1 has 4 processes, user2 has 2 processes, user3 has 2 processes, user4 has 2 processes , user5 has 1 processes.

Default Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2	PROCESS3	PROCESS4
USER1	9.091	9.0854	8.9616	8.97
USER2	9.0739	9.0634		
USER3	9.136	9.15143		
USER4	9.1253	9.1105		
USER5	9.0005			

Fair-Share Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2	PROCESS3	PROCESS4
USER1	5.0002	5.0001	5.0001	5.0001
USER2	9.9997	9.9999		
USER3	9.9997	9.9996		
USER4	9.9996	9.9996		
USER5	19.9987			

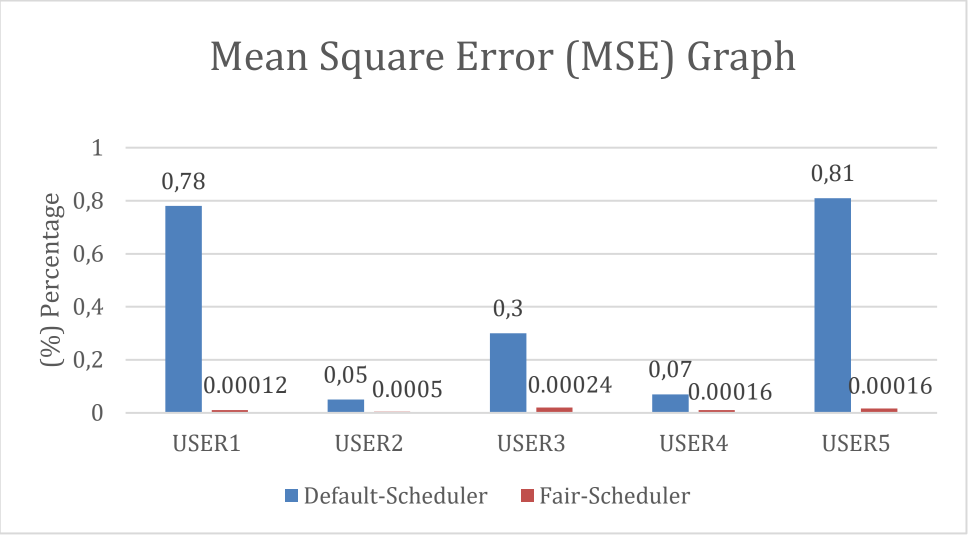


Figure 17 *MSE Graph for Test4*

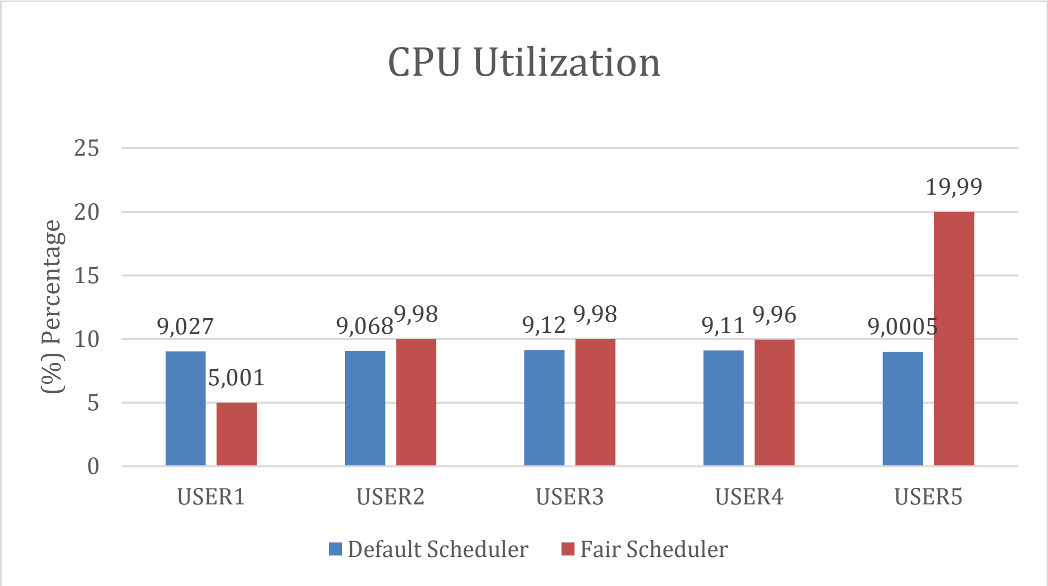


Figure 18 *CPU Utilization Graph for Test4*

3.1.5. TEST 5

In TEST5; user1 has 4 processes, user2 has 2 processes, user3 has 2 processes, user4 has 2 processes , user5 has 1 processes, user6 has 1 processes.

Default Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2	PROCESS3	PROCESS4
USER1	8.2742	8.2117	8.2126	8.2344
USER2	8.2763	8.2817		
USER3	8.3381	8.3111		
USER4	8.3299	8.2453		
USER5	8.1728			
USER6	8.2385			

Fair-Share Scheduler

CPU UTILIZATION (%)	PROCESS1	PROCESS2	PROCESS3	PROCESS4
USER1	4.2	4.1998	4.1998	4.1998
USER2	8.3002	8.3002		
USER3	8.3002	8.3002		
USER4	8.3002	8.3002		
USER5	16.6936			
USER6	16.6936			

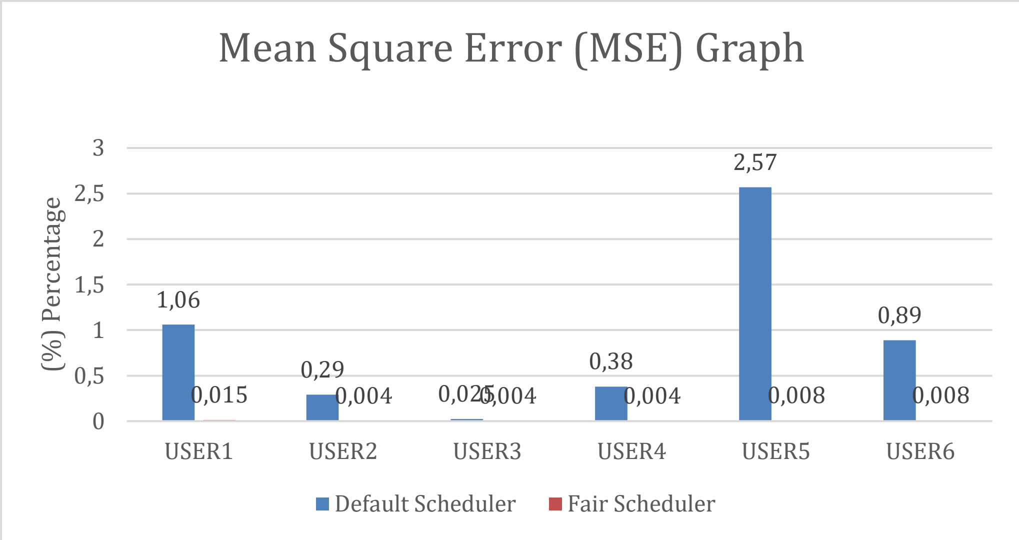


Figure 19 *MSE Graph for Test5*

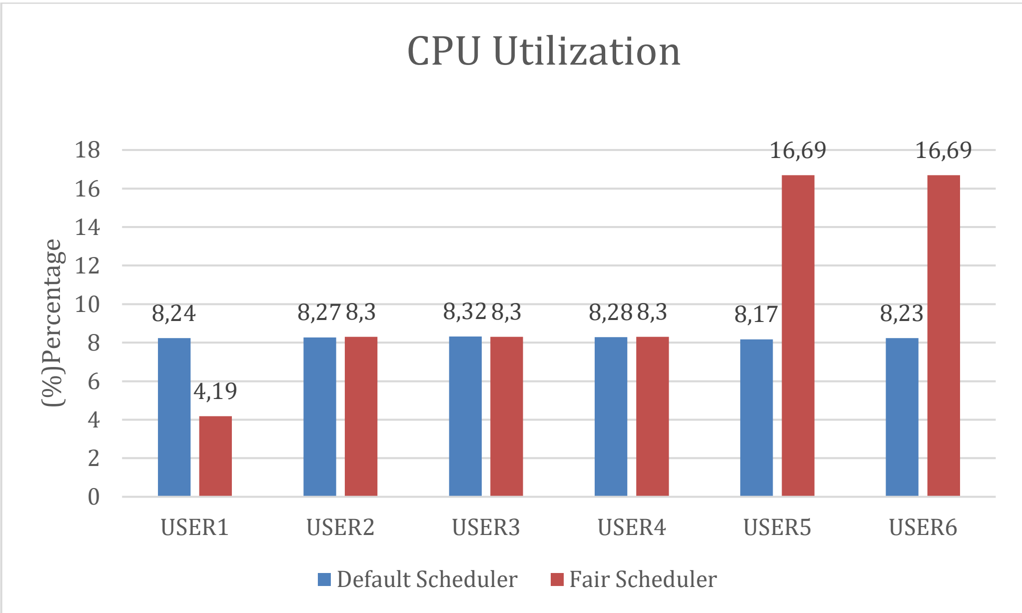


Figure 20 *CPU Utilization Graph for Test5*

3.2. RESULT

3.2.1 TEST1

In Figure 11, we can see from the MSE values that we have obtained from the processes that the default scheduler has much higher values compared to our Fair scheduler for User 1 and the processes have shown us that the fair scheduler for User 2 has substantially higher values than Default scheduler. In Figure 12 **“**CPU Utilization Graph for Test1” we can be clearly seen from the graph, the Average CPU Utilization values for the processes in both schedulers.

3.2.2 TEST2

In Figure 13, we can see from the MSE values that we have obtained from the processes that the default scheduler has much higher values compared to our Fair scheduler for User 1 and the processes have shown us that the fair scheduler for User 2 and User3 has substantially higher values than Default scheduler. In Figure 14 **“**CPU Utilization Graph for Test2” we can be clearly seen from the graph, the Average CPU Utilization values for the processes in both schedulers.

3.2.3 TEST3

In Figure 15, we can see from the MSE values that we have obtained from the processes that the default scheduler has much higher values compared to our Fair scheduler for all Users.. In Figure 16 **“**CPU Utilization Graph for Test3” we can be clearly seen from the graph, the Average CPU Utilization values for the processes in both schedulers.

3.2.4 TEST4

In Figure 17, we can see from the MSE values that we have obtained from the processes that the default scheduler has much higher values compared to our Fair scheduler for all Users.. In

Figure 18 **"**CPU Utilization Graph for Test4" we can be clearly seen from the graph, the Average CPU Utilization values for the processes in both schedulers.

3.2.5 TEST5

In Figure 19, we can see from the MSE values that we have obtained from the processes that the default scheduler has much higher values compared to our Fair scheduler for all Users.. In Figure 20 **"**CPU Utilization Graph for Test5" we can be clearly seen from the graph, the Average CPU Utilization values for the processes in both schedulers.

4. CONCLUSION

In conclusion, our Fair scheduler and the default scheduler in Linux 2.4.27 take two different tacks when it comes to scheduling computer system tasks. A dynamic algorithm is used by the default scheduler to evenly distribute resources among all processes.

Which process gets to be scheduled as the next on the CPU is determined by process values like "counter," "nice," and the goodness() function. On the other hand, our fair scheduler takes a probabilistic method.

Both schedulers have advantages and disadvantages of their own. The default scheduler is renowned for its ease of use, effectiveness, and complete fairness of the process, but it is not customizable for alterations in resource allocation. However, it may be less effective and need more computer resources to implement than our fair scheduler, which is more adaptable and can be tweaked to give a more equitable distribution of resources.

In the end, the system in question's unique demands and requirements will determine the scheduler to use. Both the Linux 2.4.27 default scheduler and the fair scheduler have advantages and disadvantages, and the best option will depend on the particular situation and the system's objectives.

REFERENCES

- Operating System Concepts (Peter Baer Galvin , Greg Gagne , Avi Silberschatz)
- Understanding the Linux Kernel, First Edition (Daniel P. Bovet , Marco Cesatti)
- The Linux Process Manager: The internals of scheduling, interrupts and signals (John O’Gorman)