FAIR-SHARE CPU SCHEDULAR

by

Mehmet Şemdin Aktay

Ekrem Çağlayan

Ali Berk Islam

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2023

ABSTRACT

This project investigates the modification of the default scheduler, which is a component of the Linux operating system kernel, to improve fairness based on user-level fairness. The objective of this study is to explore how user-level fairness can be enhanced to achieve fair behavior by the scheduler. Various scenarios and metrics were employed to evaluate the performance of the current default scheduler. The impact of the proposed modifications on performance was analyzed, and the results were presented in a comparative manner.

TABLE OF CONTENTS

1. INTRODUCTION

A scheduler is a tool or system commonly used to organize and plan tasks, jobs, or activities within time intervals.

Linux traditionally utilizes an algorithm called the "O(1) scheduler" as its scheduler. This scheduler is characterized by having a fixed execution time and performs well, particularly when there are numerous processes.

The scheduler we have developed is known as the Fair-Share scheduler. The Fair-Share scheduler is a type of scheduler used to distribute resources fairly in multi-user systems. This type of scheduler manages the time slots and priorities allocated to each user's processes. Resources are shared equitably among users, ensuring that one user cannot dominate all resources or adversely affect the performance of other users in the system.

The second part of our report discusses how we designed these schedulers, providing more detailed information on our improvements. The third section examines the results of our tests and highlights the differences between the default scheduler and our implementation. The report concludes with graphs and tables to support our findings.

2. DESIGN and IMPLEMENTATION

The CPU scheduler is a component within an operating system that is used to manage CPU resources efficiently.

The primary task of the CPU scheduler is to schedule and prioritize CPU usage among different processes or tasks. When an operating system needs to run multiple processes simultaneously, the CPU scheduler arranges these processes in a queue and determines which process to execute on the CPU.

2.1. DEFAULT SCHEDULAR

The Linux Default Scheduler utilizes a time-sharing algorithm. The scheduler in the Linux kernel uses a unit called a quantum to allocate CPU time slices to processes. The quantum time slice for the Linux Default Scheduler is 10 ms.

A quantum is a time interval determined by the operating system, specifying how long a process can run at a time. When a process is assigned a quantum, it gains access to the CPU for that quantum duration. Once the quantum expires, if the process is ready to continue or if it is waiting behind another higher-priority process, it will be put in the queue.

A separate counter is maintained for each process. As a process consumes CPU time, the counter value decreases. When the counter reaches zero, it indicates that the process has exhausted its quantum and is removed from the CPU. Subsequently, other processes are scheduled, and the scheduler switches between them. The Linux kernel

continuously updates the counter value during the execution of the scheduler algorithm. Depending on the states and priorities of the processes, the counter values are adjusted, and transitions between processes occur.

The Linux Default Scheduler arranges processes based on their priority level (nice). Each process is assigned a priority level, and processes with higher priority levels receive more CPU time compared to processes with lower priority levels. The priority level is determined by the nice value assigned to the process, which can be set by the user or the system. The priority is calculated as 20 minus the nice value. For example, a higher nice value corresponds to a lower priority.

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)

Additionally, the Linux kernel considers the relative weights of the processes. The weight assigned to a process determines its share of CPU time allocation. Processes with higher weights receive more CPU time, while those with lower weights receive less CPU time. The weight value is used to adjust the relationship between process priorities. For example, in SCHED_OTHER, the weight value is calculated as "20 - nice + counter," while in RT_PRIORITY, the weight value is calculated as "1000 + rt_priority" (real-time priority).

In this manner, the Linux Default Scheduler prioritizes processes based on their priority levels and weights using the quantum time-sharing algorithm. Higher-priority or higher-weight processes receive more CPU time, allowing them to execute more quickly, while lower-priority or lower-weight processes receive less CPU time and have lower priority. This efficient utilization of system resources is achieved through the scheduler's algorithm.

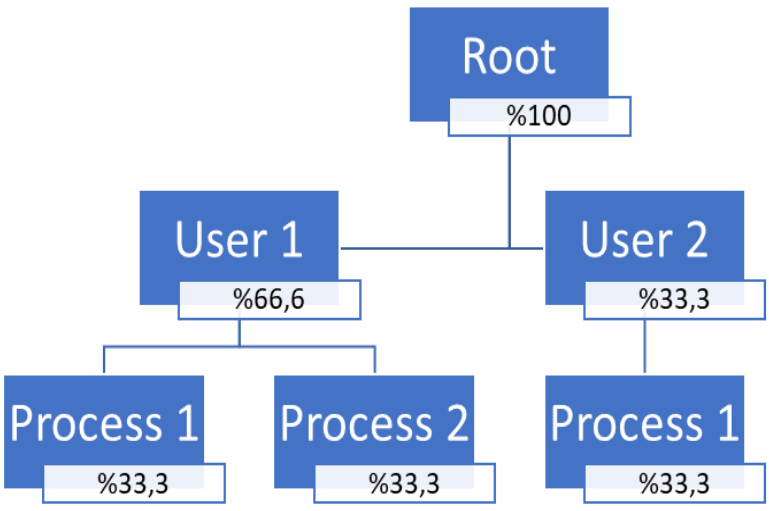


Figure 2.1.1

2.2. FAIR-SHARE SCHEDULAR

The Fair-Share Scheduler is a scheduling algorithm used to distribute resources fairly among users.

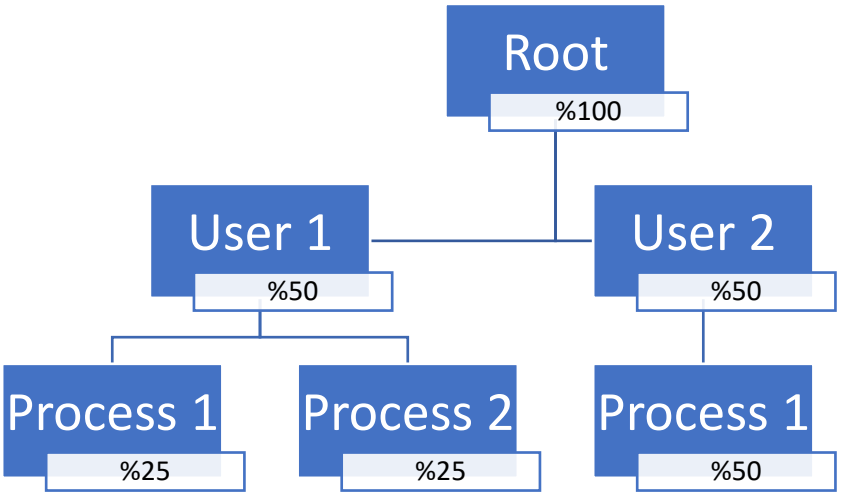


Figure 2.2.1

To create the Fair-Share Scheduler, modifications were made to the existing scheduler by editing the sched.c file in the kernel. This is because the time-sharing component of the scheduler algorithm is managed by the schedule function within sched.c. First, within sched.c, a boolean variable and an integer array were created.

```
bool schFlag = true;

int uid_array[2000];
```

Figure 2.2.2

To control whether we want to run the system with the Default Scheduler or the Fair-Share Scheduler, we use a flag value called **schFlag**. In order to change the flag value while the computer is running, we utilize our own created systemcall called **changeflag**. To create this system all, we first add the line that defines our systemcall in arch/i386/kernel/entry.s within the kernel space.

```
.long SYMBOL_NAME(sys_changeflag) /* 260 */
```

Figure 2.2.3

Next, we define our **systemcall** in include/asm-i386/unistd.h, which is located in the kernel space.

```
#define __NR_changeflag     260
```

Figure 2.2.4

Then, we also define our **systemcall** in asm/unistd.h within the user space.

```
#define __NR_changeflag 260
```

Figure 2.2.5

After that, we create our header file called **changeflag.h** in the include/linux directory within the kernel space.

```
#ifndef __LINUX_CHANGEFLAG_H
#define __LINUX_CHANGEFLAG_H
#include <linux/linkage.h>
#endif
```

Figure 2.2.6

Then, we create a file called **changeflag.c** within the kernel space, which contains the code for our function.

```
#include <linux/changeflag.h>
#include <stdbool.h>
#include <linux/kernel.h>

asmlinkage int sys_changeflag(){
    extern bool schFlag;
    schFlag = false;
    return 0;
}
```

Figure 2.2.7

- Declare a bool variable named **schFlag** externally. This indicates that the flag variable can be accessed from another location and is defined.
- Set the **schFlag** variable to false. This indicates that the flag is false and does not represent a specific condition.

In the same file, add the **"changeflag.o"** part, which is the result of compilation, to the Makefile in order to include it.

```
obj-y :=    open.o read_write.o devices.o file_table.o buffer.o \
            super.o block_dev.o char_dev.o stat.o exec.o pipe.o namei.o \
            fcntl.o ioctl.o readdir.o select.o fifo.o locks.o \
            dcache.o inode.o attr.o bad_inode.o file.o iobuf.o dnotify.o \
            filesystems.o namespace.o seq_file.o xattr.o quota.o \
            processinfoS23.o changeflag.o
```

Figure 2.2.8

In the user space, the "changeflag.h" header file is created within the "linux" directory.

```
#include <linux/unistd.h>
#include <errno.h>
extern int errno;
_syscall0(int, changeflag);
```

Figure 2.2.9

With these steps, we have created our system call. As a result, in user space programs, we can use the changeflag() function to set our flag value to 0.

Within the schedule function, we utilize this flag in the **repeat_schedule** part. If the flag value is 1, the system operates with the **Default Scheduler**. If the flag value is not 1, it operates with the **Fair-Share Scheduler** that we have created.

```
repeat_schedule:
    if( schFlag == 1 )
    {
```

Figure 2.2.10

After that, we need to modify the repeat_schedule part within the schedule function. The following section is **common** to both schedulers:

```
next = idle_task(this_cpu);
c = -1000;
list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Figure 2.2.11

- A task in idle state is assigned as the default task for the processor. This occurs when there are **no active tasks** running on the processor.
- The variable 'c' is assigned a **priority** value of -1000, which represents the lowest priority on the processor.
- A loop is created on the linked list with the starting node runqueue_head. This list contains all the tasks running on the processor.
- Using the temporary node (**tmp**) in the loop, access is made to the actual task structure (struct task_struct) and assigned to variable p.
- It checks if the task is runnable on this processor. The can_schedule function evaluates whether the task meets the conditions (e.g., time quota, priority, etc).

- It calculates the priority of the task. The goodness function relies on specific criteria to calculate the task's priority. These criteria may consider various factors to determine the task's priority, such as waiting time, previous processor state, active memory management, etc.
- If the calculated weight exceeds the current highest **priority** value (c), the c value is updated, and the next variable is set as p. This is done to determine the task with the highest priority.

```
if (unlikely(!c)) {
    struct task_struct *p;

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

    memset(uid_array, 0, sizeof(uid_array));

    for_each_task(p) {
        if (p->state == 0) {
            uid_array[p->uid]++;
        }
    }

    for_each_task(p) {
        if (uid_array[p->uid]) {
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice) /  uid_array[p->uid];
        }
        else {
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
        }
    }

    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
    goto repeat_schedule;
}
```

Figure 2.2.12

The main section we modified to create the Fair-Share Scheduler is described above. If the counter value is 0, the code enters this block. Here are the steps:

- A variable p of type **task_struct** is created.
- An array named **uid_array** is created and cleared using the memset function. This array is used to track the number of tasks based on their user identifiers (UIDs).
- A loop is created to iterate over all the tasks. **for_each_task** function allows access to all tasks in the system.
- If the task's state is 0 (i.e., it is in the running state), it increments the corresponding counter in the uid_array based on its user identifier (**uid)**.
- Again, a loop is created to iterate over all the tasks.
- If there is a counter for the corresponding user identifier in the **uid_array**, it calculates the task's counter value. The counter value is updated as the sum of half of the previous value **(p->counter >> 1)** and the **division** of NICE_TO_TICKS(p->nice) by the number of user identifiers **(uid_array[p->uid])**.
- Otherwise, if there is no counter for the corresponding user identifier in the uid_array, it calculates the **task's counter value**. The counter value is updated as the sum of half of the previous value (p->counter >> 1) and NICE_TO_TICKS(p->nice).

3. TEST and RESULTS

To measure the efficiency of the Default Scheduler compared to our own Fair-Share Scheduler algorithm, test cases were created. There are two test cases. In the first test case, there are two users. One user has 2 processes, and the other user has 1 process. In the second test case, there are 3 users. One user has three processes, and the other two users have one process each. The tests are evaluated based on 1000 samples. For the Fair-Share Scheduler algorithm, the tests were performed by taking 100 samples in each of the 10 text files. The CPU values for each user were recorded, and the average values were calculated. For the Default Scheduler algorithm, the values were recorded in a single file with 1000 samples. The average CPU values were calculated based on these results. From these average values, along with the expected values, the Mean Square Error (MSE) values were calculated.

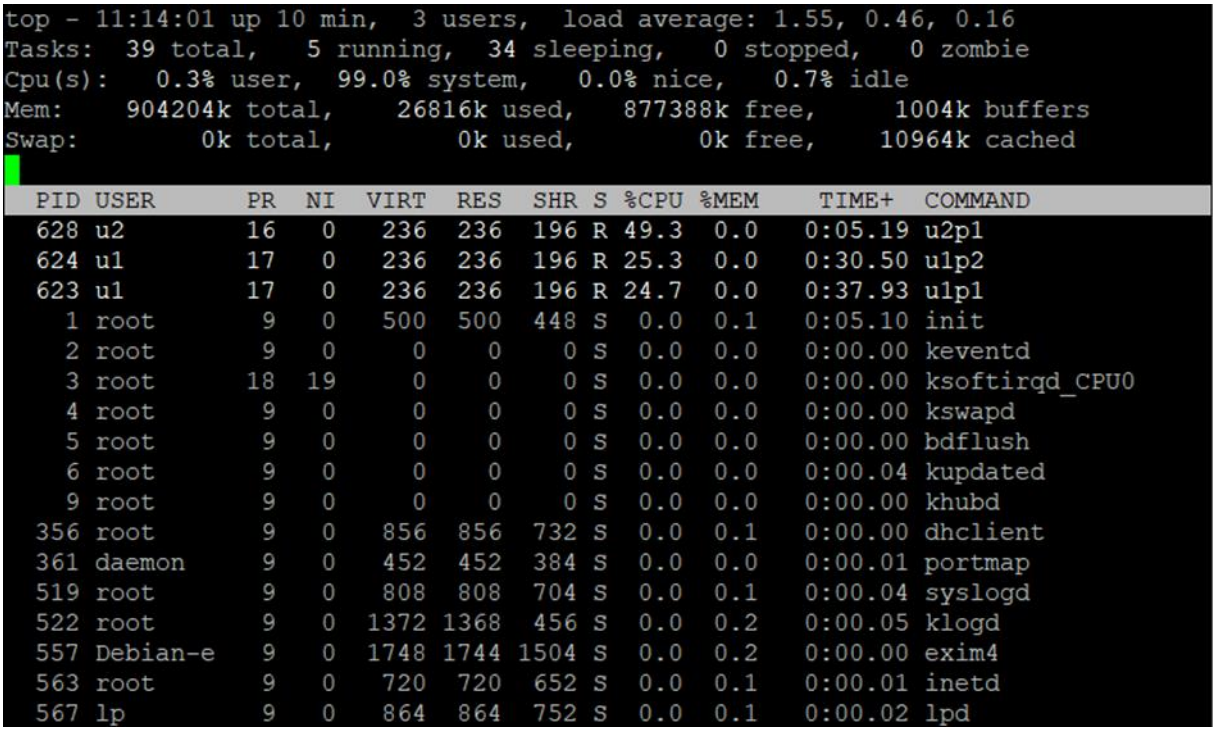


Figure 3.1

The CPU utilization values were obtained from the table shown above (accessed with the 'top' function within the root).

3.1. TEST CASE 1

The first test case consists of 2 users. User 1 has two processes, while User 2 has one process.

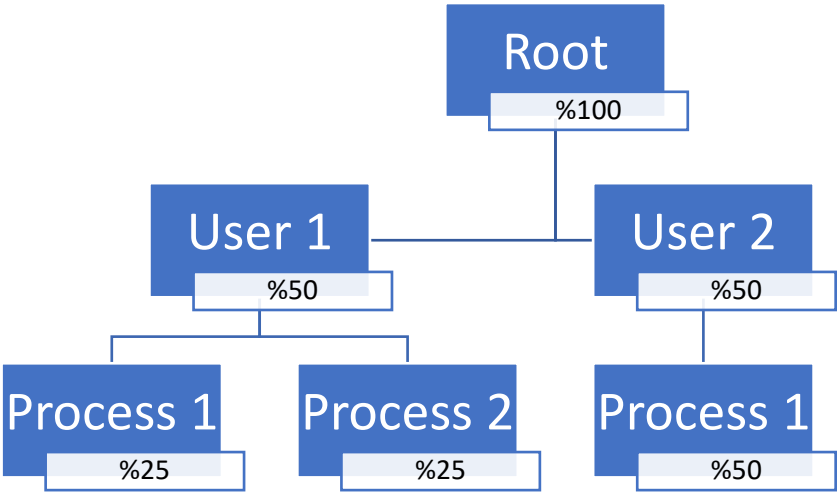
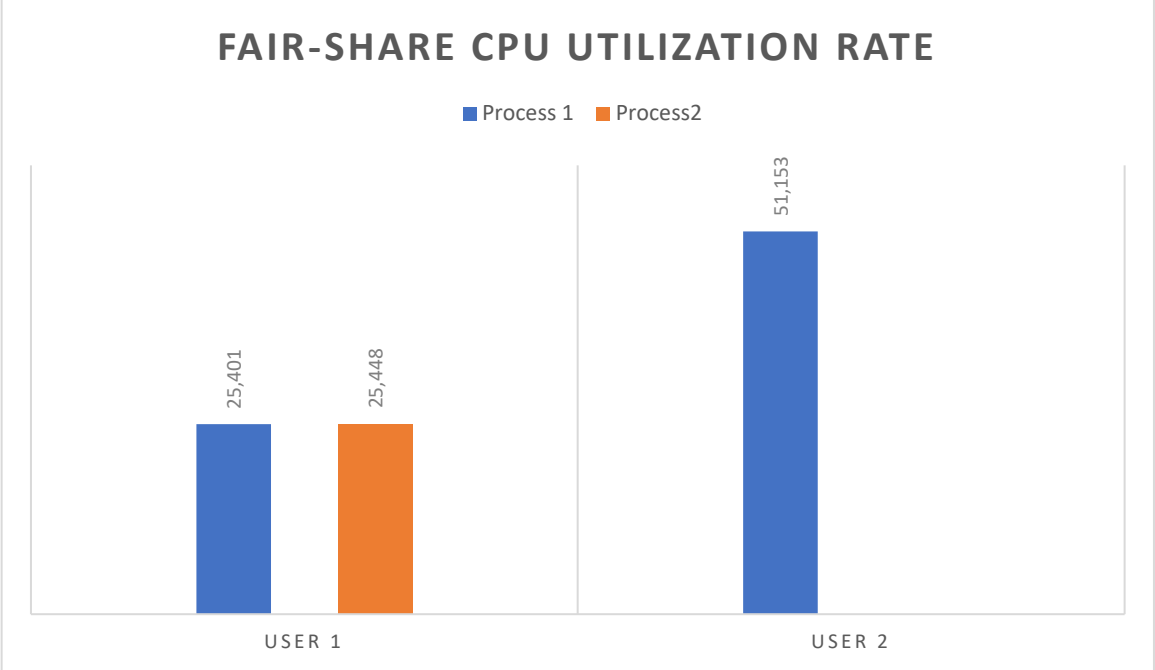


Figure 3.1.1

The figure above shows the expected CPU utilization for processes in a kernel using the **Fair-Share** algorithm.



Graph 3.1.1

The graph above shows the average of the values obtained from our 1000 samples.

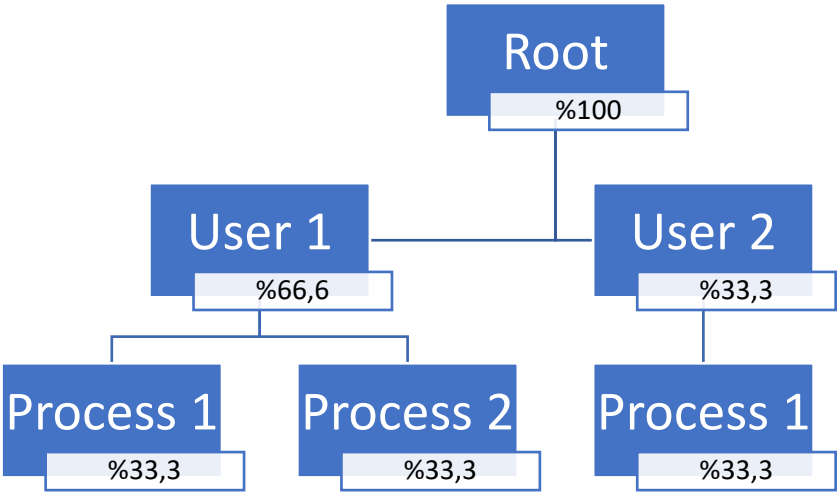
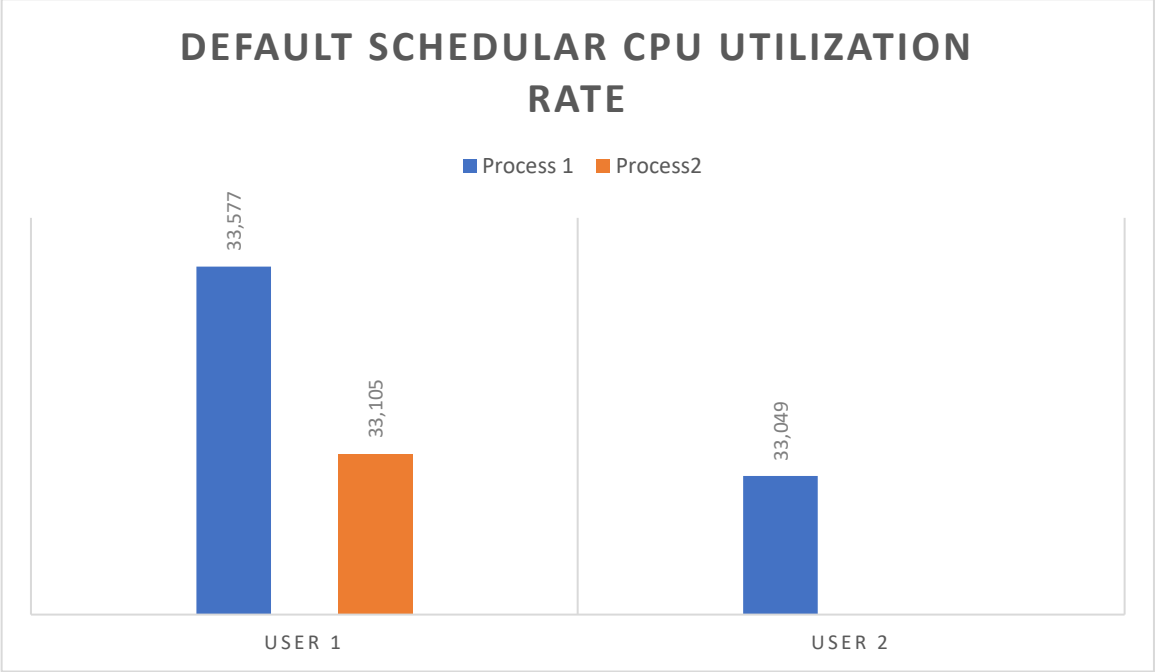


Figure 3.1.2

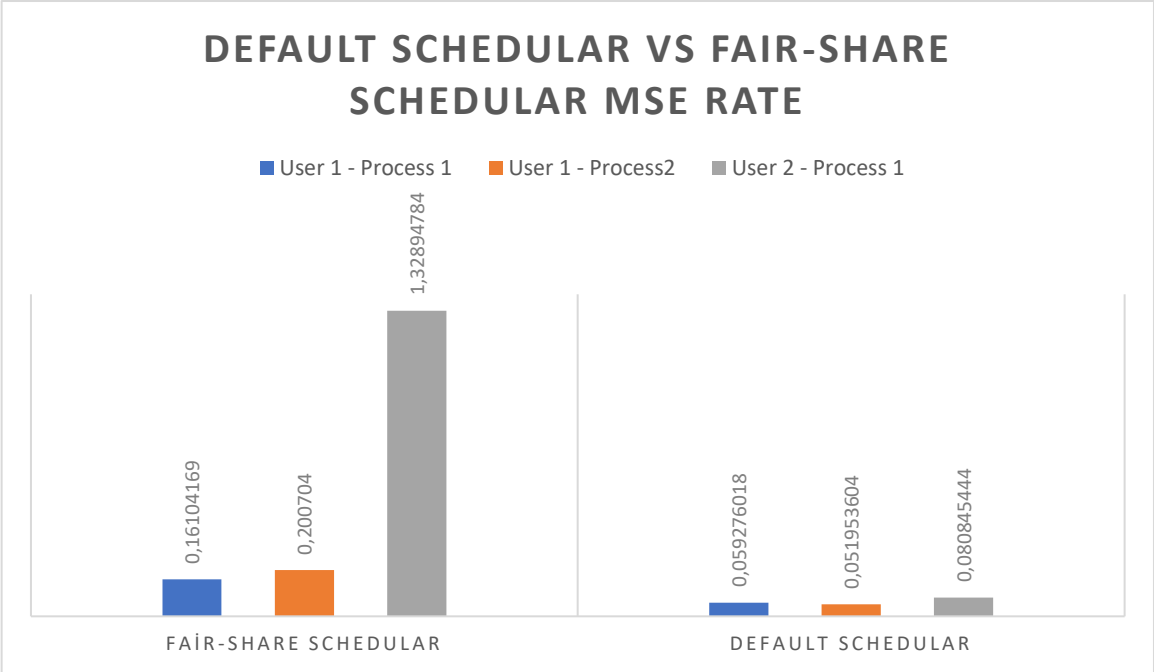
The figure above shows the expected CPU utilization for processes in a kernel using the **Default Schedular** algorithm.



Graph 3.1.2

The graph above shows the average of the values obtained from our 1000 samples.

When looking at CPU utilization, it can be observed that User 1 has an advantage when the Default Scheduler algorithm is used. This is because User 1 has almost twice the CPU utilization compared to the other user. On the other hand, when the Fair-Share Scheduler is used, User 2 has an advantage. Since the scheduler works on a user basis, User 2, despite having only one process, has CPU utilization equal to the combined CPU utilization of User 1's two processes.



Graph 3.1.3

When looking at the MSE values, it can be observed that the MSE values for User 1 Process 1 and User 1 Process 2 are 10 times higher than the MSE values achieved by the Default Scheduler. The MSE value for User 2 Process is almost 20 times higher than the MSE values obtained by the Default Scheduler. As a result, our Fair-Share Scheduler, although reaching the expected values, has achieved them with less accuracy compared to the Default Scheduler, as indicated by the higher MSE values.

3.2. TEST CASE 2

Our second test case consists of 3 users. User 1 has three processes, User 2 has one process, and User 3 also has one process.

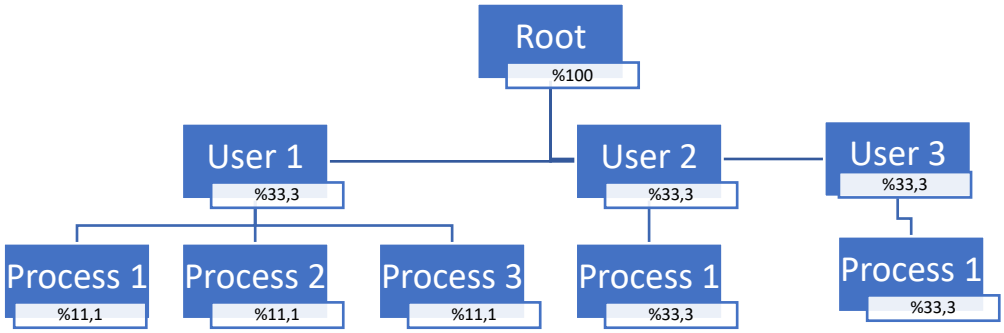
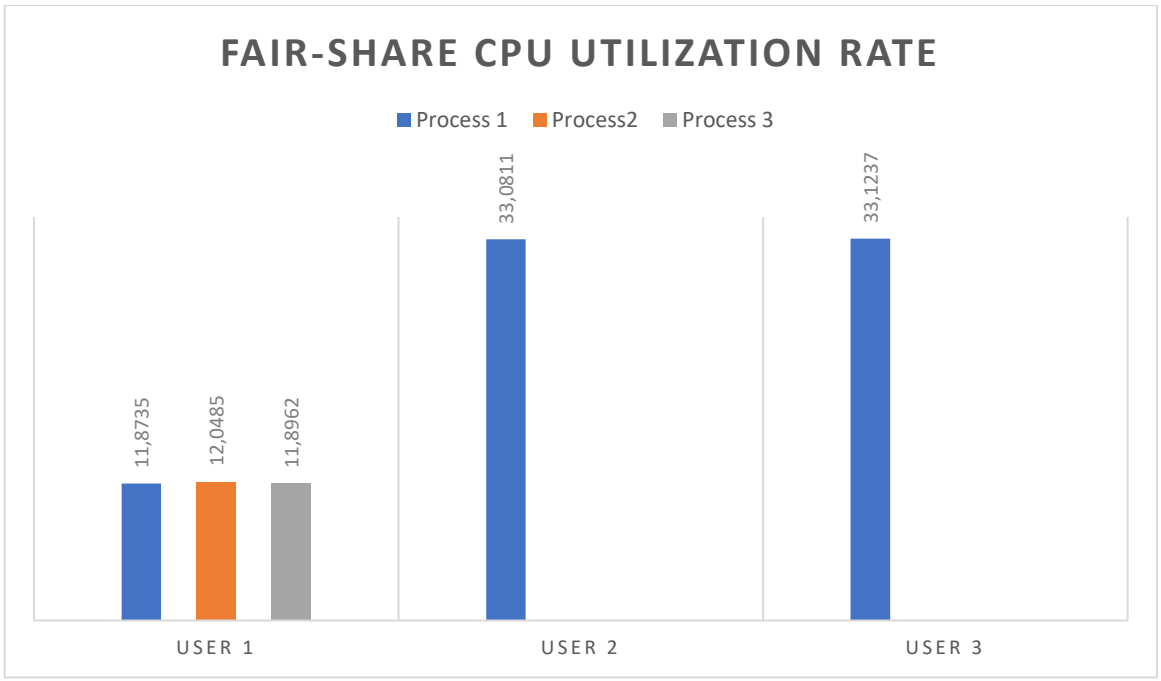


Figure 3.2.1

The figure above shows the expected CPU utilization for processes in a kernel using the **Fair-Share** algorithm.



Graph 3.2.1

The graph above shows the average of the values obtained from our 1000 samples.

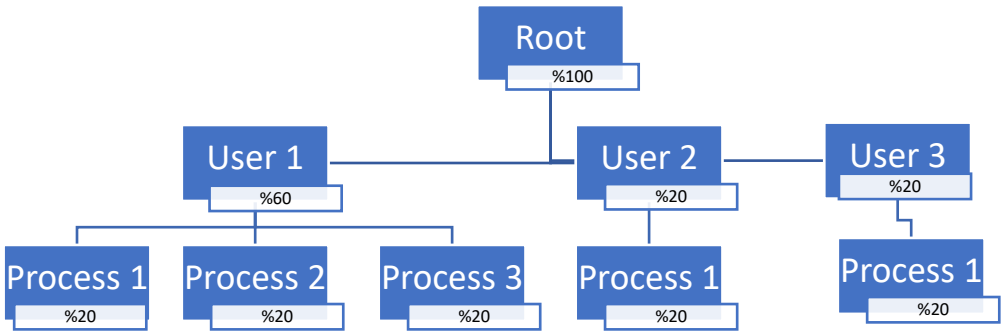
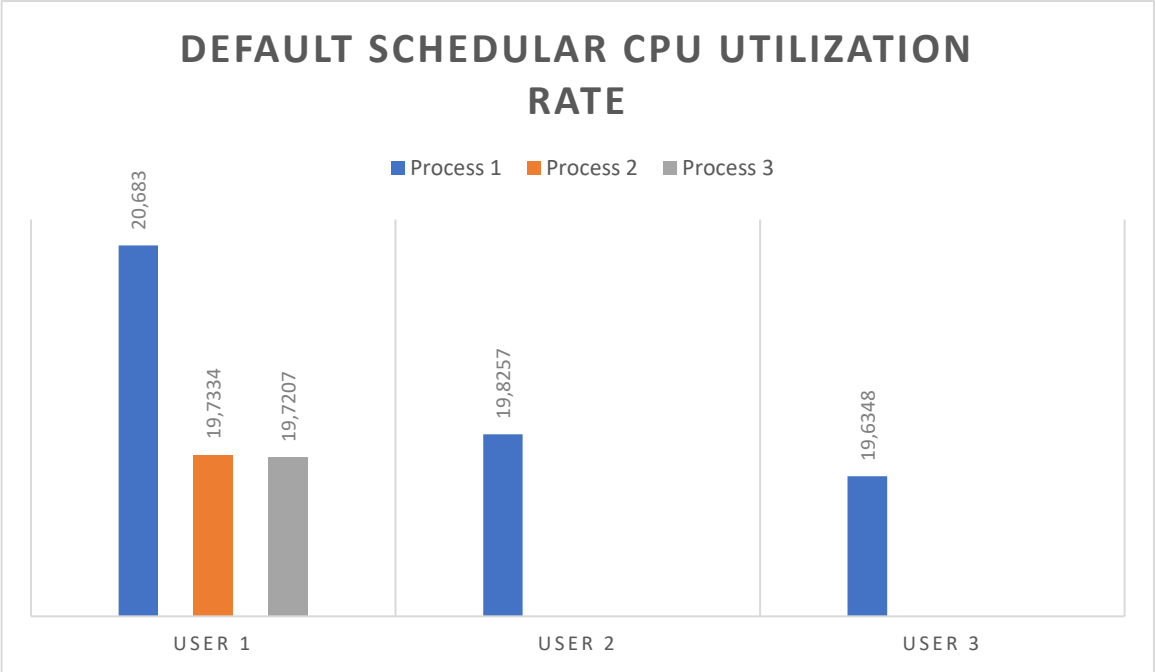


Figure 3.2.2

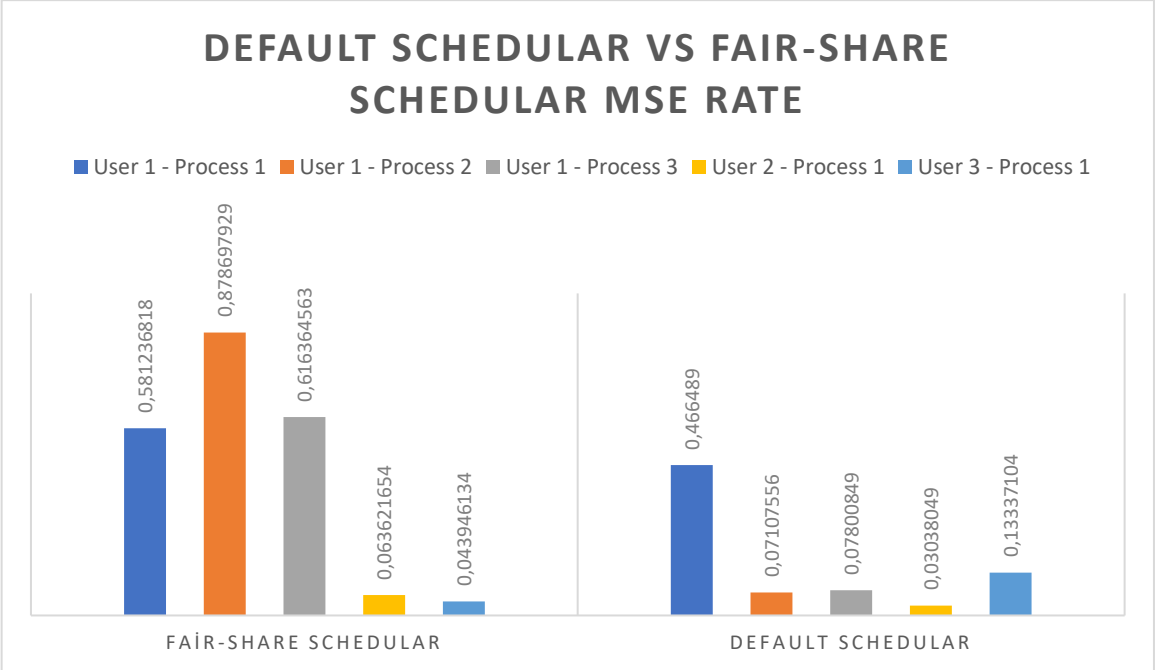
The figure above shows the expected CPU utilization for processes in a kernel using the Default Schedular algorithm.



Graph 3.2.2

In the above graph, the average values of the results obtained from our 1000 samples are shown.

When looking at CPU utilization, User 1 is more advantageous when Default Scheduler algorithm is used. This is because it operates on a per-process basis, and in terms of total utilization, User 1 has nearly three times the CPU usage compared to the other users. On the other hand, when Fair-Share Scheduler is used, User 2 and User 3 are more advantageous. Since the scheduler operates on a per-user basis, even though they have only one process each, their CPU usage is nearly three times that of User 1's single process.



Graph 3.2.3

When looking at the MSE (Mean Square Error) values, User 1 Process 1, User 2, and User 3 have reached almost the same error rate as the Default Scheduler. However, when we look at the MSE values for User 1 Process 2 and User 1 Process 3, it is apparent that they deviate significantly from the calculated values. This indicates a larger error in predicting the CPU utilization for those processes in the Fair-Share Scheduler.

4. CONCLUSION

In conclusion, the choice between using the default scheduler or our Fair-Share scheduler depends on the specific circumstances and requirements of the system.

The default scheduler, with its time-sharing algorithm, is suitable in situations where there are numerous processes running simultaneously. It effectively allocates CPU time based on process priorities and weights, ensuring efficient resource utilization. It is advantageous for scenarios where individual process performance is prioritized, as higher-priority processes receive more CPU time.

On the other hand, the Fair-Share scheduler is designed to distribute resources fairly among users in multi-user systems. It operates on a per-user basis, ensuring that no single user dominates all resources or negatively impacts the performance of others. This scheduler is beneficial when the goal is to provide equal opportunities to all users and prevent resource monopolization.

However, our Fair-Share scheduler implementation exhibited higher Mean Square Error (MSE) values compared to the default scheduler, indicating less accurate prediction of CPU utilization for certain processes. Therefore, in situations where precise prediction and control over individual process performance are crucial, the default scheduler may be preferred.

In summary, the default scheduler is recommended when prioritizing individual process performance, especially in systems with numerous processes. On the other hand, our Fair-Share scheduler is more suitable for multi-user environments, where fair resource distribution among users is a priority, even though it may have slightly less accurate predictions of CPU utilization.