Term Project Phase II: Fair Share Scheduler

by

Mevlüt Akif Şahin

Ulaş Can Demirbağ

Emirhan Tala

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Fall 2024

ACKNOWLEDGMENTS

We sincerely thank Prof. Dr. Şebnem Baydere for her guidance and Research Assistant Osman Kerem Perente for his exceptional support and dedication throughout this project.

ABSTRACT

Operating system schedulers play an important role in managing system performance. Over the years, different scheduling algorithms have been developed to improve efficiency. While some schedulers aim to ensure fairness among processes, some algorithms prioritize certain types of processes based on specific criteria. In this project, we implemented a Fair Share Scheduler as a replacement for the default scheduler in Linux Kernel 2.4.27. Unlike process-based fairness, this approach focuses on ensuring fair CPU usage among user groups. For testing and analysis, we also implemented a new system call that allows switching between the default scheduler and our custom Fair Share Scheduler. We then conducted a series of tests with different configurations to compare the performance of both schedulers and analyze their CPU usage.

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

Schedulers are essential components of operating systems, managing how processes are executed on a CPU. They decide the order of process execution, ensuring that when the CPU becomes idle, there is always a process ready to run. This coordination is crucial since modern systems run multiple applications and processes simultaneously. Users should not notice any delays or interruptions as these switches happen.

To achieve this, process-fair schedulers are often used. They ensure that no process is indefinitely delayed from running. However, in some cases, fairness may be considered in terms of user groups rather than individual processes. This approach is useful in systems like servers where multiple users share the same machine.

In this project, we aim to develop a Fair Share Scheduler for Linux Kernel 2.4.20. Unlike process-fair schedulers, a user-based fairness approach ensures that all users receive equal CPU time, regardless of the number of processes each user has. Our goal is to adjust the scheduler to follow this user-centric approach.

To achieve this, we divide the project into four stages. First, we examine the fundamental concepts of process scheduling and the algorithms involved. Next, we introduce a new system call to enable switching between the default and custom scheduler. Then we performed a series of tests to measure the performance of our and default scheduler. Finally, we analyze the test results and discuss the impact of the changes.

2. SCHEDULING ALGORITHMS

Schedulers use different strategies to determine the execution order of processes. These strategies are defined by scheduling algorithms. Schedulers are typically divided into two main categories: pre-emptive and non-pre-emptive. Both aim to optimize CPU usage, improve throughput, and minimize turnaround, waiting, and response times.

2.1 PRE-EMPTIVE SCHEDULERS

Pre-emptive scheduling allows processes to be interrupted and replaced by higher-priority processes. This means that a running process can be paused to give the CPU to a more important task. Priority is the key factor here, and higher-priority tasks get access to the CPU first. In this type of scheduling, transitions can happen from running to waiting, waiting to ready, or ready to running.

2.2 NON-PRE-EMPTIVE SCHEDULERS

In non-preemptive scheduling, once a process is assigned to the CPU, it retains control until it finishes or voluntarily gives up control. Unlike pre-emptive scheduling, non-preemptive schedulers do not interrupt an active process. A context switch only occurs when a process finishes its execution or moves to a waiting state.

2.3 SCHEDULING ALGORITHMS EXAMPLES

Several algorithms are used to schedule processes. Below are some widely used scheduling techniques.

2.3.1 FIRST COME FIRST SERVE (FCFS)

The FCFS scheduler operates like a queue. The first process to arrive is the first one to be executed. It is simple to implement, often using a FIFO (First In, First Out) queue to manage processes. However, it may cause long wait times for processes arriving later in the queue.

2.3.2 ROUND ROBIN (RR)

In Round Robin scheduling, each process gets an equal time slice, often called a time quantum. After a process uses its time, it goes to the end of the queue, and the next process starts. This method prevents starvation and allows for better responsiveness, especially in real-time systems.

2.3.3 SHORTEST TIME TO COMPLETION FIRST (STCF)

The SJF scheduler selects the process with the shortest execution time. This approach minimizes the average waiting time but requires knowledge of process execution times. It can also be implemented as preemptive. One example is shown in the next subtitle.

2.3.4 SHORTEST REMAINING TIME TO COMPLETION FIRST (SRTCF)

This is a pre-emptive version of STCF. If a new process arrives with a shorter remaining execution time than the current process, the scheduler will pause the current process and switch to the new one. This approach may cause frequent context switches but reduces overall waiting time.

2.3.5 MULTI-LEVEL QUEUE (MLQ)

The MLQ scheduler uses multiple queues to classify processes. Each queue may have different scheduling policies, and processes are assigned to queues based on characteristics like priority, memory size, or CPU usage. This approach ensures that high-priority tasks are handled differently from low-priority tasks.

3. DESIGN AND IMPLEMENTATION

3.1 LINUX DEFAULT SCHEDULER

The **time-sharing algorithm** in Linux ensures that every process gets a fair opportunity to use the CPU. Here's how it works:

Each process is assigned a priority value. Lower numbers indicate higher priority, meaning those processes get to use the CPU sooner. Higher numbers indicate lower priority, meaning those processes have to wait longer. This priority value is called **"nice"**.

- **Higher "nice" value (0 to 20) → Lower priority** (process waits longer).
- **Lower "nice" value (-19 to 0) → Higher priority** (process runs sooner).
- **Default priority → nice = 0** (medium priority).

The nice value could also be explained as being nice to other processes. If a process is nice (positive values), they will let others run more and if a process is not nice (negative values) it will try to run as much as possible.

Each process gets a specific amount of CPU time called a **time slice**. Processes with higher priority (lower "nice" value) get longer time slices, while lower-priority processes get shorter ones.

Each process has a counter that tracks its remaining CPU time. When a process starts using the CPU, the counter begins counting down from its allocated time slice. Once the counter reaches zero, the process stops, and it must wait for its next turn.

The system ensures fairness by giving every process a chance to run. If multiple processes have the same priority, they **take turns** using the CPU in a round-robin fashion. No process can monopolize the CPU for too long, ensuring that no process is left waiting for an extended period.

3.2 FAIR SHARE SCHEDULER

The **Fair Share Scheduler** ensures fairness not only for processes but also for users. Instead of treating each process as an independent unit, it groups processes by the users who own them. The goal is to distribute CPU usage evenly among users rather than processes.

For instance, suppose there are three users (User A, User B, and User C) with different numbers of processes:

- **User A** has 1 process.
- **User B** has 2 processes.
- **User C** has 2 processes.

A standard scheduler would see 5 processes and give each one an equal share of CPU time (20% each). In contrast, the **Fair Share Scheduler** sees 3 users and divides the CPU equally among them. Each user receives 33.3% of the CPU time. Then, within each user's allocation, the available CPU time is distributed to the processes owned by that user. This means:

- **User A's** single process gets 33.3% of the CPU.
- **User B's** two processes split 33.3%, so each of them gets 16.65%.
- **User C's** two processes also split 33.3%, so each of them gets 16.65%.

If a new user logs in or one of the users starts a new process, the scheduler dynamically adjusts the CPU allocation to ensure fairness.

The Fair Share Scheduler is useful in multi-user systems, like servers, where users compete for shared resources. By focusing on user-based fairness rather than process-based fairness, the system prevents any single user from monopolizing CPU time.

3.3 IMPLEMENTATION

The implementation of the **Fair Share Scheduler** algorithm requires us to calculate the counter of every process accordingly. This calculation is done by multiplying a weight with the base counter value. This base counter value is the counter value used in the default scheduler, counter value is 6 when the nice value is 0 by default. So we can calculate the fair share counter by multiplying this base counter value by some weight. The calculation of the weight requires us to know how many processes the user of this process has. Thankfully this value is already stored by the kernel in the **"user_struct"** under **"task_struct"**.

In the implementation, to get isolated and consistent test results, only the counter values of processes with uids greater than 1000 are calculated using the **fair share scheduler** algorithm. Other users, such as root and some other background system users, are always ignored by a simple if statement.

```
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t   groups[NGROUPS];
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;
```

Figure 1 Process Credentials

```
struct user_struct {
        atomic_t __count;       /* reference count */
        atomic_t processes;     /* How many processes does this user have? */
        atomic_t files;         /* How many open files does this user have? */

        /* Hash table maintenance information */
        struct user_struct *next, **pprev;
        uid_t uid;
};
```

Figure 2 User Struct

We can read this **atomic_t processes** variable using the **atomic_read()** function that is included under **asm/atomic.h**. By reading this **p→user→processes** value **to process_count** variable we have successfully acquired the number of processes that the user of a process has.

We can use this **process_count** to calculate a weight. Since we know that the counter value must go down when the number of processes of the user is higher. This means that we have an inverse relationship between the number of processes and the counter value of the process. Thus, we can calculate the weight value as **1/process_count.**

```
process_count = atomic_read(&p->user->processes);
if (p->uid >= 1000 && process_count > 0){
    weight = 1.0f/(float)process_count; // Weight of process for that user
    p->counter = ((p->counter >> 1) + NICE_TO_TICKS(p->nice))*weight;
```

F**igure 3** CPU time sharing

By multiplying this weight value by the default counter value, we have successfully achieved to share the CPU time across processes fairly.

4. TESTS AND RESULTS

We implemented a Makefile for running test cases and retrieving the data from them. Overall, the Makefile automatizes our test process. Here is a detailed review of how each test is implemented and what it measures.

4.1 COMMON STRUCTURE FOR ALL TESTS

For each test case, specific rules—test1, test2, and test3—were added to the Makefile, along with an 'all' rule to run all tests simultaneously. test1, test2, and test3 each execute numerous necessary rules.

4.1.1 COMPILATION

Each test starts with the compilation necessary to run object files from source code using the utility gcc.

```
1  int main()
2  {
3      int i = 0;
4      while (1)
5      {
6          i = (i + 1) % 1000;
7      }
8  }
9
```

Figure 4 test.c

```
1  test1-exes:
2      gcc -o u1p1.o test.c
3      gcc -o u1p2.o test.c
4      gcc -o u2p1.o test.c
```

Figure 5 Rule For "*.o" Files

4.1.2 EXECUTING

The executable in a controlled environment by running them using different user accounts will simulate multi-user operation to make sure each process gets executed under the owner's respective user.

```
1  test1-u1:
2      su u1 -c './u1p1.o &'
3      su u1 -c './u1p2.o &'
4
5  test1-u2:
6      su u2 -c './u2p1.o &'
```

Figure 6 Rule To Run Processes For Each User

4.1.3 MONITORING

Top is a command used for monitoring the performance of the system, mainly CPU, which can be configured to record the data at certain intervals into log files. This helps in capturing the response of the system continuously under load.

```
1  test1-run:
2      for i in 1 2 3 4 5 6 7 8 9 10; do \
3          top -d 0.2 -n 100 -b > result/test1/${FOLDER}/SchedTop$$i.txt ; \
4      done
5
```

Figure 7 Rule To Get Samples

4.2 TEST CASES

Three test cases have been prepared to compare the default scheduler with the implemented fair scheduler.

4.2.1 TEST CASE 1

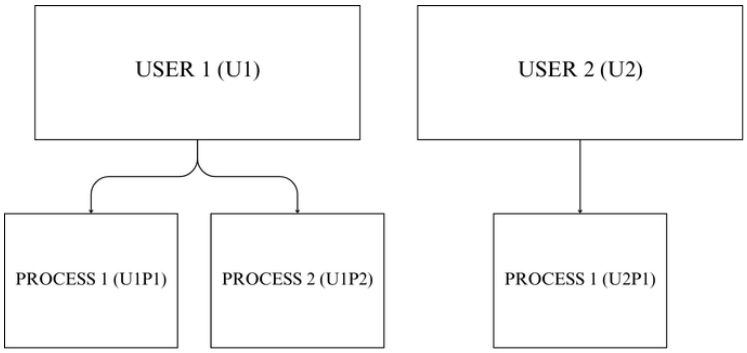

Figure 8 Diagram of Test Case 1

We run three programs, u1p1, u1p2, u2p1. The generated logs are analyzed to calculate the average CPU usage for each program and compute the mean squared error (MSE) against an expected usage value.

4.2.2 TEST CASE 2

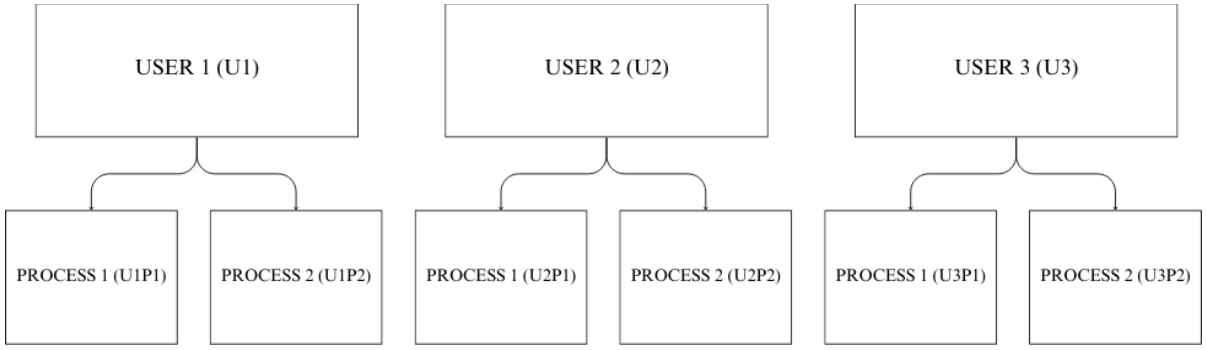

Figure 9 Diagram of Test Case 2

This test adds more complexity by running additional processes, such as u1p1, u1p2, u2p1, u2p2, u3p1, and u3p2. This test gives us an insight on how evenly the CPU resources are distributed among multiple processes.

4.2.3 TEST CASE 3

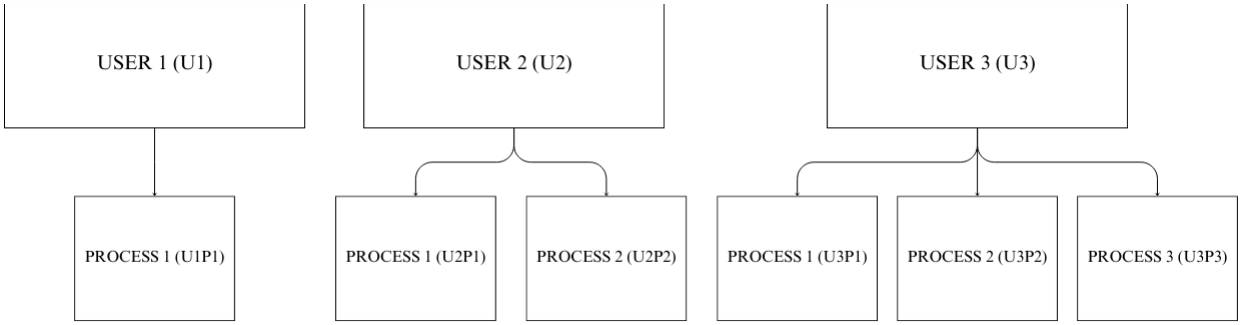

Figure 10 Diagram of Test Case 3

Further scales the test environment by adding more processes: u1p1, u2p1, u2p2, u3p1, u3p2, u3p3, with a view to maximizing the test coverage.

4.3 RESULT COMPILATION

Every test compiles the results into log files, even with the smallest details such as averages of individuals and calculations of MSE. In this way, the entire test process goes smoothly and in coordination.

Besides that, the Makefile was created to receive an argument called FOLDER from the outside so that all the tests can be run independently first under the default scheduler and then under the fair share scheduler. It allows, in such a setup, the logging of results of each configuration to be separated, ensuring that the results are reliable and all the tests are automated.

```
test1-avg:
    for i in 1 2 3 4 5 6 7 8 9 10; do \
        cat result/test1/${FOLDER}/SchedTop$$i.txt | grep 'u1p1.o' | \
         awk 'BEGIN{total=0;num=0}{num=num+1;total=total+$$9;print $$9}\
         END{avg=total/num;print "total=", total;print "average=", avg}' >> \
         result/test1/${FOLDER}/u1p1avg.txt ; \
        cat result/test1/${FOLDER}/SchedTop$$i.txt | grep 'u1p2.o' | \
         awk 'BEGIN{total=0;num=0}{num=num+1;total=total+$$9;print $$9}\
         END{avg=total/num;print "total=", total;print "average=", avg}' >> \
         result/test1/${FOLDER}/u1p2avg.txt ; \
        cat result/test1/${FOLDER}/SchedTop$$i.txt | grep 'u2p1.o' | \
         awk 'BEGIN{total=0;num=0}{num=num+1;total=total+$$9;print $$9}\
         END{avg=total/num;print "total=", total;print "average=", avg}' >> \
         result/test1/${FOLDER}/u2p1avg.txt ; \
    done

test1-mse:
    grep "average" result/test1/${FOLDER}/u1p1avg.txt | \
    awk 'BEGIN{sum=0; n=0; if ( "${FOLDER}" == "fair" ) {x=25;} \
    else {x=33.33;}}{sum=sum+(($$2-x)*($$2-x)); n=n+1}\
    END{print "mse", sum/n}' >> result/test1/${FOLDER}/test1MSE.txt
    grep "average" result/test1/${FOLDER}/u1p2avg.txt | \
    awk 'BEGIN{sum=0; n=0; if ( "${FOLDER}" == "fair" ) {x=25;} \
    else {x=33.33;}}{sum=sum+(($$2-x)*($$2-x)); n=n+1}\
    END{print "mse", sum/n}' >> result/test1/${FOLDER}/test1MSE.txt
    grep "average" result/test1/${FOLDER}/u2p1avg.txt | \
    awk 'BEGIN{sum=0; n=0; if ( "${FOLDER}" == "fair" ) {x=50;} \
    else {x=33.33;}}{sum=sum+(($$2-x)*($$2-x)); n=n+1}\
    END{print "mse", sum/n}' >> result/test1/${FOLDER}/test1MSE.txt

test1-mse-avg:
    grep "mse" result/test1/${FOLDER}/test1MSE.txt | \
    awk 'BEGIN{sum=0; n=0}{sum=sum+$$2; n=n+1}\
    END{print "avg-mse",sum/n}' \
    >> result/test1/${FOLDER}/test1MSE.txt

```

Figure 11 Rules to Calculate MSE

In our testing, we configured the top command to sample data every 200 milliseconds. However, for the second test case, the epoch time of 360 milliseconds caused the process counters to reset, resulting in a perceived CPU usage of 0%. To accelerate the process and obtain accurate results, we calculated the necessary duration for the top command for each test case. For instance, a 0.2-second interval was sufficient for test case one, whereas other test cases required a 0.4-second interval. Without these adjustments, the epoch ends would coincide with these intervals, leading to incorrect 0% CPU usage readings.

4.4 TEST RESULTS

4.4.1 TEST 1 RESULT

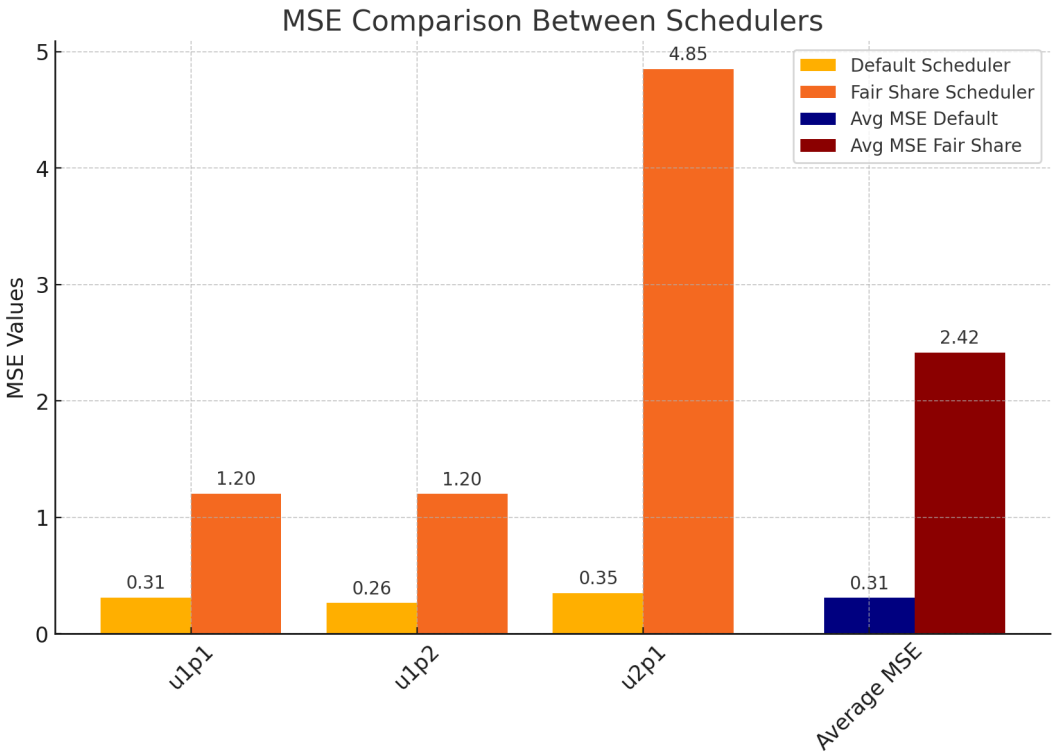

Figure 12 Graph of Test 1 MSE Results

The graph shows the Mean Squared Error (MSE) values for processes belonging to different users under the Default Scheduler and the Fair Share Scheduler, based on their expected CPU usage. The labels `u1p1`, `u1p2`, and `u2p1` represent User 1's Process 1, User 1's Process 2, and User 2's Process 1, respectively. In the Default Scheduler, CPU time is divided equally among all processes, with each expected to receive about 33%.

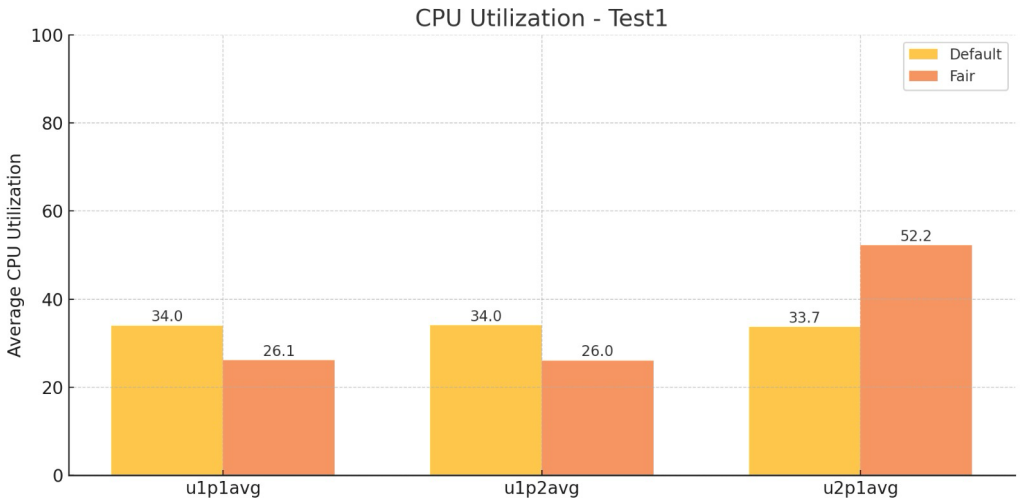

Figure 13 CPU Utilization Percentages of Test 1

In the Fair Share Scheduler, CPU time is shared equally among users. Since User 1 has two processes, each gets 25% of the CPU, while User 2's single process receives 50%. Our calculations show that the CPU utilization matches the expected values with a small variation. Since the default scheduler also shows this variation, we believe the issue comes from our testing method. We collect 100 samples 10 times for each test, which may cause slight differences in the utilization percentages.

The results show that the Fair Share Scheduler produces higher MSE values than the Default Scheduler. For `u1p1`, `u1p2`, and `u2p1`, the Fair Share Scheduler records MSE values of 1.20, 1.20, and 4.85, while the Default Scheduler shows much lower MSE values of 0.31, 0.26, and 0.35. The average MSE is also higher for the Fair Share Scheduler at 2.42, compared to 0.31 for the Default Scheduler. These results indicate that while the Fair Share Scheduler successfully allocates CPU time based on user groups, it leads to larger deviations from the expected values compared to the Default Scheduler, which focuses on process-based fairness.

4.4.2 TEST 2 RESULT

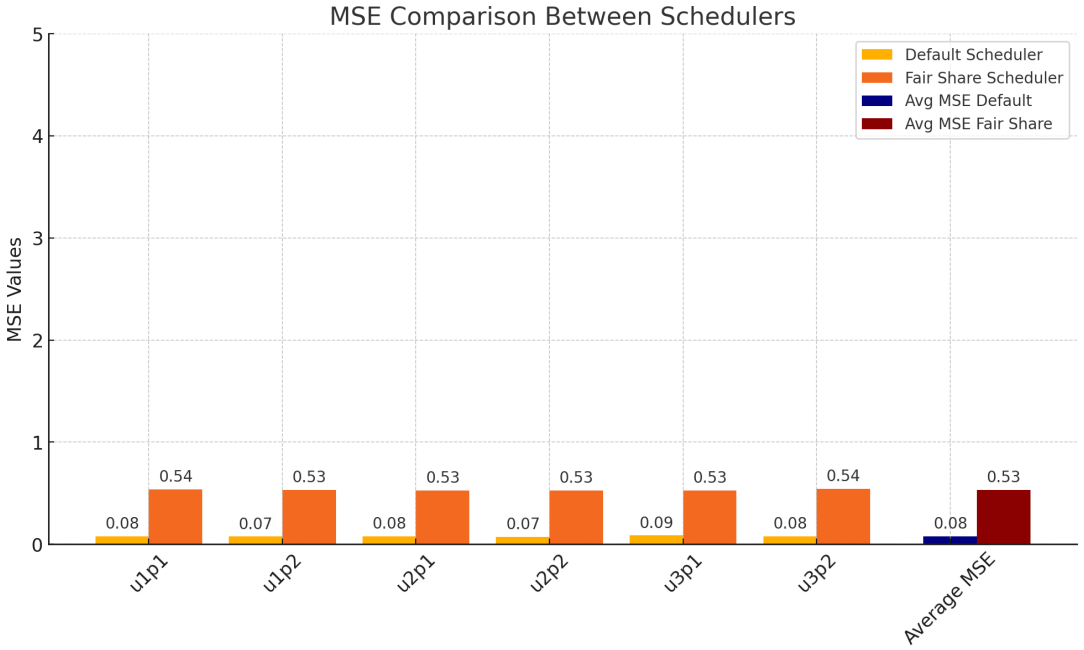

Figure 14 Graph of Test 2 MSE Results

The graph represents the Mean Squared Error (MSE) values for processes belonging to multiple users under the Default Scheduler and the Fair Share Scheduler in Test Case 2. Here, `u1p1`, `u1p2`, `u2p1`, `u2p2`, `u3p1`, and `u3p2` correspond to processes belonging to Users 1, 2, and 3. In this scenario, both schedulers are expected to distribute CPU time evenly among all processes, as each user has an equal number of processes. Each of the six processes is expected to receive approximately 16.67% of the CPU time.

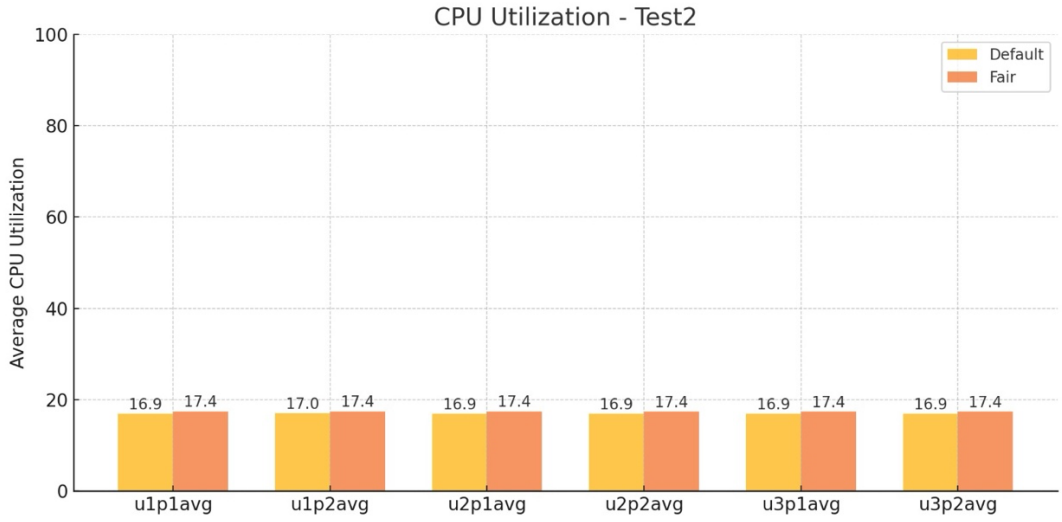

Figure 15 CPU Utilization Percentages of Test 2

As the CPU utilization table shows, the results are close to our expectations, with a slight deflection.

The results show that both schedulers perform similarly, with no significant difference in their CPU allocation. The Default Scheduler achieves an average MSE of 0.08 across all processes, while the Fair Share Scheduler records an average MSE of 0.53. The slightly higher MSE for the Fair Share Scheduler can be attributed to its algorithm's user-centric design, but since all users have the same number of processes, the schedulers effectively behave the same in terms of CPU distribution. This demonstrates that the Fair Share Scheduler aligns closely with the Default Scheduler when the number of processes per user is balanced.

4.4.3 TEST 2 RESULT

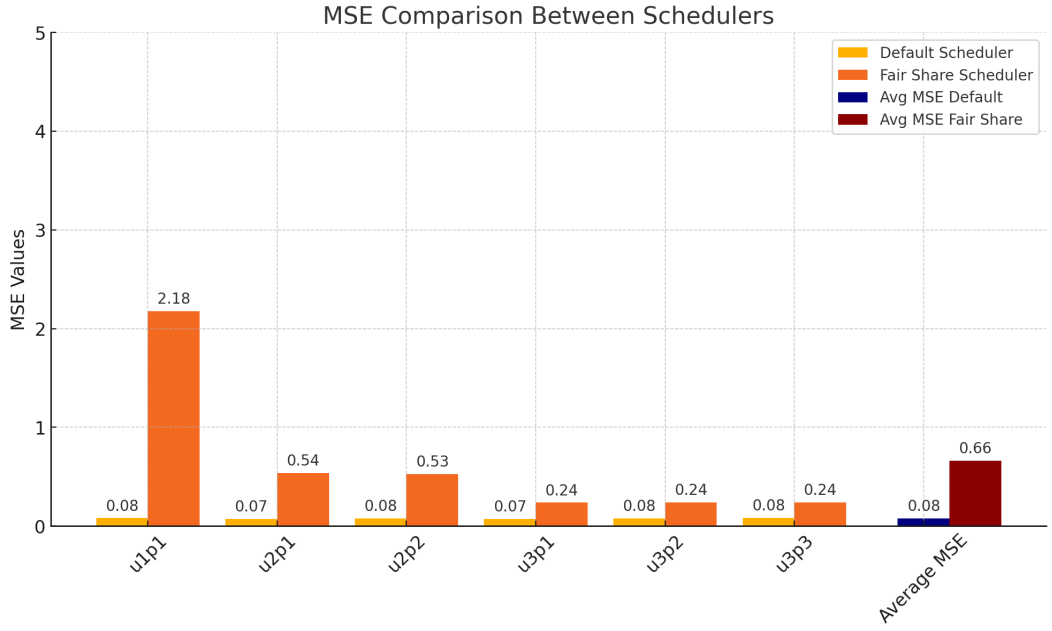

Figure 16 Graph of Test 3 MSE Results

The graph represents the Mean Squared Error (MSE) values for processes belonging to multiple users under the Default Scheduler and the Fair Share Scheduler in Test Case 3. The labels `u1p1`, `u2p1`, `u2p2`, `u3p1`, `u3p2`, and `u3p3` represent processes belonging to Users 1, 2, and 3. The Default Scheduler is expected to allocate CPU time equally among all processes, with each receiving approximately 16.67%. On the other hand, the Fair Share Scheduler distributes CPU time based on user groups, providing 33.33% to User 1 (for its single process) and 33.33% each to Users 2 and 3, which is then shared equally among their processes.

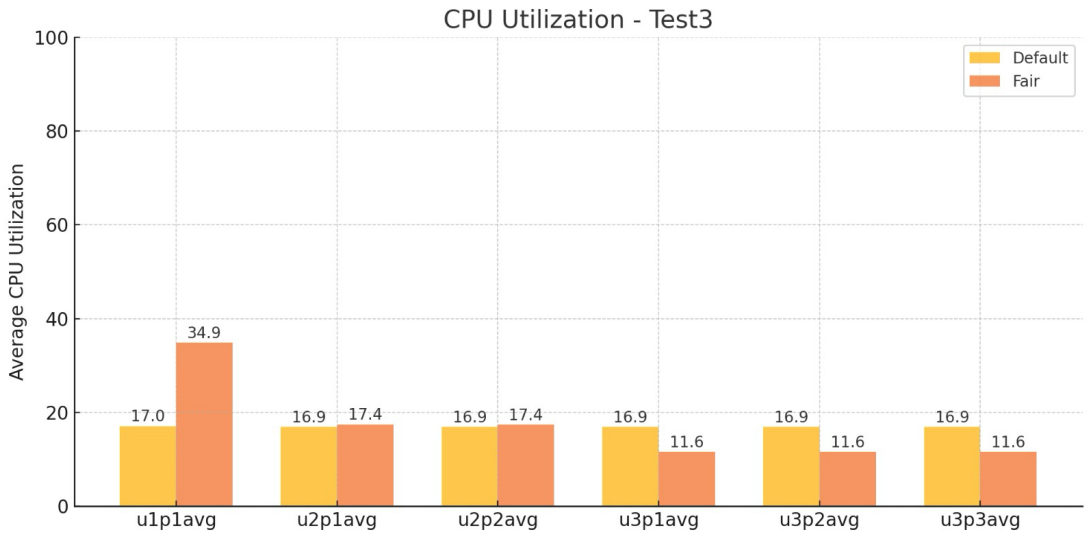

Figure 17 CPU Utilization Percentages of Test 3

Like the other two test and their results, the CPU utilizations are close to our expectations with a tiny deviation.

The results highlight the differences in performance between the two schedulers. The Default Scheduler achieves consistently low MSE values, averaging 0.08 across all processes, indicating accurate adherence to process-centric fairness. In contrast, the Fair Share Scheduler shows varied MSE values, with the highest deviation recorded for `u1p1` at 2.18.

5. CONCLUSION

This project successfully implemented and analyzed a Fair Share Scheduler in comparison to the Default Scheduler in the Linux kernel. The results demonstrate that the Fair Share Scheduler achieves its goal of user-centric fairness by distributing CPU time equally among users, regardless of the number of processes they own.

The trade-offs highlighted in this study indicate that the Fair Share Scheduler is well-suited for multi-user systems where user fairness outweighs individual process fairness, while the Default Scheduler is more appropriate for scenarios where fairness at the process level is crucial.

6. REFERENCES

https://elixir.bootlin.com/linux/2.4.27/source