

# **SYSTEM CALLS AND LOTTERY TICKET POLICY ON LINUX KERNEL 2.4.27**

**by**

**Elif Dikmen**

**Gamze Nur Erdem**

**Ali Abbasi Dolatabadi**

**CSE 331 Operating Systems Design  
Term Project Report**

**Yeditepe University  
Faculty of Engineering  
Department of Computer Engineering  
Spring 2024**

## **ABSTRACT**

Operating system schedulers are crucial for optimizing OS performance. Over the years, numerous algorithms have been developed to enhance scheduler efficiency. Generally, schedulers aim to ensure process fairness. However, different scheduling algorithms may prioritize processes based on specific attributes, which can be beneficial or detrimental depending on the system architecture and device needs. In this project, we implemented a lottery ticket scheduler algorithm, designed to provide user-based fairness. This implementation was tested on the Linux Kernel 2.4.27.

## **TABLE OF CONTENTS**

1. Introduction.....	4
2. Design and Implementation.....	5
2.1. DESIGN.....	5
2.2 IMPLEMENTATION.....	6
4. TESTS and RESULTS.....	11
4.1. Test Case 1.....	18
4.2. Test Case 2.....	20
4.3. Test Case 3.....	22
4.4. Test Case 4.....	24
5. Conclusion.....	26
6. References.....	27

# 1. INTRODUCTION

Schedulers are specialized computer programs responsible for managing process scheduling in various ways. They control the order in which processes run on a given CPU and ensure that the operating system (OS) always has at least one process ready to execute when the CPU becomes idle. With multiple processes and programs running simultaneously on a computer, schedulers coordinate and manage these tasks to provide a seamless user experience, effectively hiding task changes between processes. Process-fair schedulers are commonly used in operating systems to prevent indefinite postponement of tasks, maintaining fairness.

Throughout history, various algorithms and methodologies have been developed for scheduling, prioritizing optimal outcomes and fairness. In this project, we explore a unique scenario by focusing on a user-group-based fairness scheduler instead of a process-fair one. This approach is particularly useful in environments where several user groups share a dedicated computer, such as a server. For simplicity, we concentrate on one of the earliest Linux kernels (Linux Kernel 2.4.27) to illustrate the principles involved.

Our goal is to achieve user fairness across the entire system by making the necessary adjustments to this kernel and specifying the execution order of the proposed algorithm. Once we have a stable scheduler, we conduct various tests to evaluate its performance. The project is divided into four parts:

1. Scheduler Principles and Algorithms: We examine and explain the fundamental principles and algorithms used in schedulers.
2. System Calls and Scheduler Switching: We introduce and test special system calls that allow switching between the default and proposed schedulers.
3. Testing and Evaluation: We conduct a series of tests on both the default and proposed schedulers to assess the impact of our changes.
4. Analysis and Conclusion: We analyze the test results obtained in the third part to draw final conclusions about the scheduler's performance.

By following these steps, we aim to demonstrate the feasibility and effectiveness of a user-group-based fairness scheduler in enhancing system fairness and performance.

## **2. DESIGN and IMPLEMENTATION**

The type of scheduling algorithm that we will be designing and implementing is called a Fair Share Scheduler. We will be explaining the details of the design for both Default and Fair Share Scheduler.

### **2.1. DESIGN**

To implement a user-fair scheduler in Linux kernel version 2.4.27, several modifications to the kernel source code are necessary. The goal is to ensure CPU usage is distributed equally among users rather than processes. The following steps outline the required changes.

**User-Based Process Distribution:** We need to distribute CPU time among users by utilizing the uid parameter, which identifies the user. In the Linux kernel code, this is specified as the uid\_t type. The uid value is found in the /usr/src/linux-2.4.27-10/include directory within the sched.h header file. This file contains the task\_struct definition, which holds the process uid information. In the sched.c file, which contains the scheduler algorithm, we will modify the process selection logic to account for the uid value. The scheduler will compare the uid parameter of each process to ensure equitable CPU distribution among users.

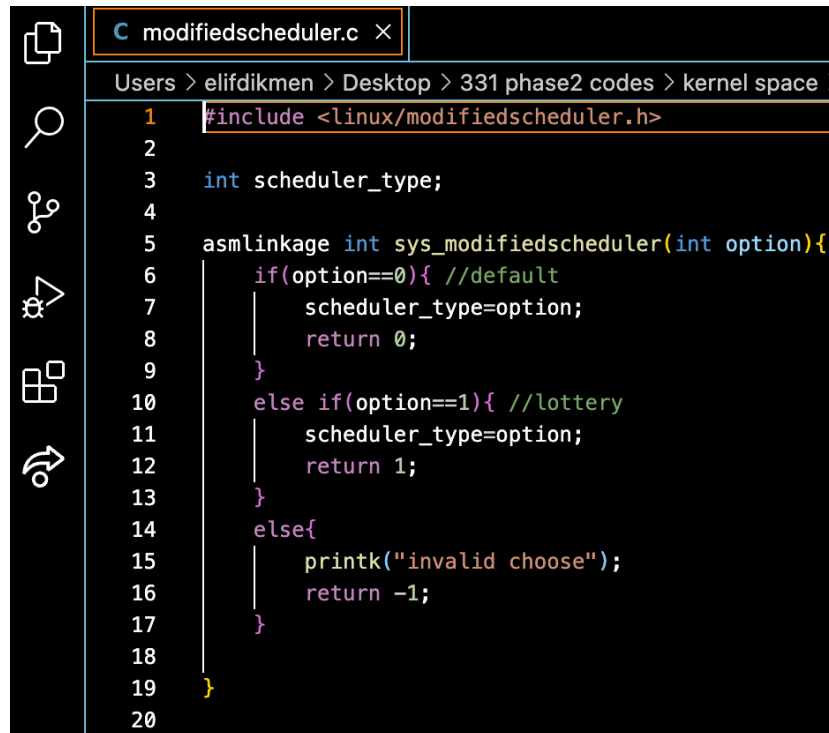
**Ticket-Based Scheduling:** Introduce a ticket\_number parameter to the task\_struct to implement a ticket-based scheduling algorithm. This can be achieved by adding the ticket\_number parameter at the end of the task\_struct in the sched.h file. It is crucial to add new parameters at the end of the task\_struct to avoid disrupting the structure's bit order, which could lead to system-wide failures. The ticket\_number parameter of each process must be updated based on CPU time allocation. The jiffies value, which measures processor time and changes at the end of each CPU cycle, will be used for this purpose.

**Initialization and Adjustment:** Initialize the last\_time values in the fork.c file, where processes are created. After CPU allocation, update these variables for each process phase.

**User-Fair Scheduling Control:** To achieve user-fair scheduling, control the scheduling logic to ensure that once a process is selected, all other processes belonging to the same user are temporarily disabled from running until all processes from other users have been

selected. Rescheduling should occur only after all users have had their processes selected, ensuring equitable CPU distribution across users.

## 2.2. IMPLEMENTATION



```
C modifiedscheduler.c x
Users > elifdikmen > Desktop > 331 phase2 codes > kernel space >
1 #include <linux/modifiedscheduler.h>
2
3 int scheduler_type;
4
5 asmlinkage int sys_modifiedscheduler(int option){
6     if(option==0){ //default
7         scheduler_type=option;
8         return 0;
9     }
10    else if(option==1){ //lottery
11        scheduler_type=option;
12        return 1;
13    }
14    else{
15        printk("invalid choose");
16        return -1;
17    }
18
19 }
20
```

*Figure 1 modifiedscheduler() system call code*

As you can see in Figure 1 we added a simple system call to the kernel that changes the “scheduler\_type” depending on whether the given input is 0 or 1. In “sched.c”, the “sched\_type” variable is described as “extern int sched\_type.” So we can switch from default scheduler to lottery scheduler with this system call.

```

418
419     /*phase 2 */
420     int ticket_number;
421     unsigned long volatile last_time;
422

```

**Figure 2** Lottery Scheduling Algorithm Variables

In "sched.h," we defined three variables to facilitate user-fair scheduling. The first variable, "ticket\_number," stores the ticket number for each process, which is essential for determining the next process to run. The second variable, "last\_time," is used to calculate the elapsed CPU time. The third variable, "selected\_user," holds the user ID of the owner of the selected process.

```

49
50     //phase 2
51     extern int scheduler_type;
52

```

**Figure 3** Scheduler Type Definition Variable

We use this variable in "sched.c" to change scheduler mode. First, we define this variable in "modifiedscheduler.c" in order to use that variable in "sched.c" we use the "extern" keyword.

```

740     /*phase 2*/
741     p->last_time = jiffies;
742     p->ticket_number = 5;
743

```

**Figure 4** Lottery Scheduler Variables on Initialization

In the "sched.h" and "task\_struct" files, as shown in Figure 4, we defined two additional variables. These variables are initialized in the "fork.c" file within the "do\_fork" function, since all processes begin with this function. The first variable, "last\_time," is initialized using the system default variable "jiffies," which tracks the current CPU time. We set "last\_time" to "jiffies" because "jiffies" is constantly updated, and we need to capture the current CPU time

when our process starts. The second variable, "ticket\_number," is initialized to "5" for each process. In this project, all processes begin with a ticket number of "5." The number of tickets will vary based on the CPU time (jiffies) and the system's maximum or minimum ticket number.

```

612 //phase 2
613 if(scheduler_type==0)
614 {
615     list_for_each(tmp, &runqueue_head) {
616         p = list_entry(tmp, struct task_struct, run_list);
617         if (can_schedule(p, this_cpu)) {
618             int weight = goodness(p, this_cpu, prev->active_mm);
619             if (weight > c)
620                 c = weight, next = p;
621         }
622     }
623 }

```

*Figure 5 Iterating through the tasks in a runqueue sequentially*

In figure 4 for each task, it checks if it can be executed on the current CPU. If it is executable, it calculates a weight value, and if this value exceeds a certain threshold, it updates the variables c and next.

```

626 int total_ticket_number=0;
627 int random_ticket;

```

*Figure 6 Variables used for Lottery Scheduling Algorithm*

In the block of the lottery scheduling set of rules where in the "scheduler\_type" variable equals 1, we defined 2 variables as visible in Figure 6. The overall total ticket number is calculated by adding each process's ticket number to the "total\_ticket\_number" variable. We choose methods the use of a random integer so the "random\_ticket" variable is used to save the generated random integer.



```

628 | list_for_each(tmp, &runqueue_head) {
629 |     p = list_entry(tmp, struct task_struct, run_list);
630 |     if(((10*jiffies)-(10*p->last_time))>100)&&(p->ticket_number!=10))
631 |     {
632 |         p->ticket_number = p->ticket_number+1;
633 |     }
634 |     else if(((10*jiffies)-(10*p->last_time))<10)&&(p->ticket_number!=1))
635 |     {
636 |         p->ticket_number = p->ticket_number-1;
637 |     }
638 |     total_ticket_number=total_ticket_number+(p->ticket_number);

```

**Figure 7** Updating and finding sum of the ticket numbers

As illustrated in Figure 7, we iterate through all the processes in the run queue using `list\_for\_each`. During each iteration, we first check the CPU time consumed by the process. This is calculated by subtracting the process's "last\_time" variable from "jiffies".

If the consumed time is greater than 100 and the ticket number is not equal to 10 the process is considered a large process, and we increment its ticket number. Conversely, if the consumed time is less than 10 and the ticket number is not equal to 1, the process is considered small, and we decrement its ticket number. After updating the ticket numbers, we check if any processes in the run queue belong to users who have not been selected before. If the "selected\_user" value of a process is 0, it indicates that the user of that process has not been selected. In this case, we set the `check\_all\_users` variable to 0, indicating the existence of an unselected user, and add the ticket numbers of these processes to the `unselected\_total\_ticket\_number` variable. Finally, we add the ticket value of all processes to the `total\_ticket\_number` variable, regardless of whether the processes are selected or not.

```

640     get_random_bytes(&random_ticket, sizeof(random_ticket));
641     if(random_ticket<0)
642     {
643         random_ticket=random_ticket*(-1);
644     }
645     if(total_ticket_number!=0){
646         random_ticket=(random_ticket%total_ticket_number)+1;
647     }

```

**Figure 8** Generate random ticket

As shown in Figure 8, the function `get\_random\_bytes` generates a random integer, which can be either negative or positive. To select the "lucky" process, we need a positive number; if the generated number is negative, we convert it to positive by multiplying it by -1. Once we generate our random ticket, we need to take its modulus. We take the modulus of the random ticket variable with `total\_ticket\_number`.

```

648     list_for_each(tmp, &runqueue_head){
649         p = list_entry(tmp, struct task_struct, run_list);
650         if (can_schedule(p, this_cpu)){
651             random_ticket=random_ticket-(p->ticket_number);
652             if(random_ticket<=0){
653                 next=p;
654                 break;
655             }
656         }
657     }
658

```

**Figure 9** Choosing next process

In Figure 9, we iterate through all the processes in the run queue using `list\_for\_each`. We first check if a process can be scheduled using the `can\_schedule` function. If the process is schedulable, so we subtract its ticket value from the `random\_ticket` variable. If the result of this subtraction is negative or zero, we assign this process as the next one to run and exit the loop. If the result is positive, the iteration continues.

```

660 //phase 2
661 if(scheduler_type==0)
662 {
663     if (unlikely(!c)) {
664         struct task_struct *p;
665
666         spin_unlock_irq(&runqueue_lock);
667         read_lock(&tasklist_lock);
668         for_each_task(p)
669             p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
670         read_unlock(&tasklist_lock);
671         spin_lock_irq(&runqueue_lock);
672         goto repeat_schedule;
673     }
674 }

```

**Figure 10** Reacquiring the run queue lock and jumps to a label to repeat the scheduling process

When scheduler type 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..

### 3. TESTS and RESULTS

#### 3.1 TESTS

In the testing part, we collected CPU Utilization values for the processes in each case. With the use of “ top -n 100 -d 1 -b > d1t1.txt” command, we received data. We conducted these tests for both the default Linux scheduler and our newly developed fair-share scheduler. Once all the necessary data was collected, we utilized an AWK command within a shell script

to calculate the average CPU utilization and Mean Square Error (MSE) for each process in every test. Essentially, we repetitively executed the tests, recorded the CPU utilization data, and then analyzed it using a shell script with an embedded AWK command to determine the average CPU utilization and MSE for each process in both the default and lottery scheduler algorithms.

```
Users > elifdikmen > Desktop > 331 phase2 codes > u
1  #include <linux/modifedscheduler.h>
2  #include <stdio.h>
3
4  main(){
5      int scheduler_type;
6      printf("Enter scheduler type: ");
7      scanf("%d",&scheduler_type);
8      modifedscheduler(scheduler_type);
9  }
10
```

*Figure 11 test.c*

We used the “test.c” file to call our system call with the input parameter 1 and switch to Lottery Scheduler. If we want to return back to the default scheduler, we have to call the “modifedscheduler()” function with 0.

```

1  #include <stdio.h>
2
3  int main(){
4      while(1){
5          }
6      }
7
8

```

**Figure 12** Infinite While Loop

Our “loop.c” file is shown in Figure 12. This program is run by each process in a user for testing purposes. Each process, as shown in Figure, runs in an infinite loop, enabling us to easily observe it

```

CSE331:~# cat lottery3_test4.txt | grep 'u1p5' | awk 'BEGIN{total=0;num=0;}{num=num+1; total=total+$9; print $9}END{ avg =total/ num; print "total=", total; print"average=", avg;}'

```

**Figure 13** Awk Command

In figure 13, we wrote a command for system to read the test case files and sum up CPU Utilization values for each case. And with division operation by the total number of testcases, we reached the average CPU Utilization.

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

**Figure 14** Mean Square Error Formula

We used the formula in Figure 14 to calculate the mean square error (MSE). In this context,  $\hat{Y}$  represents the predicted CPU utilization value for each process, while  $Y$  denotes the actual CPU utilization value obtained from testing.

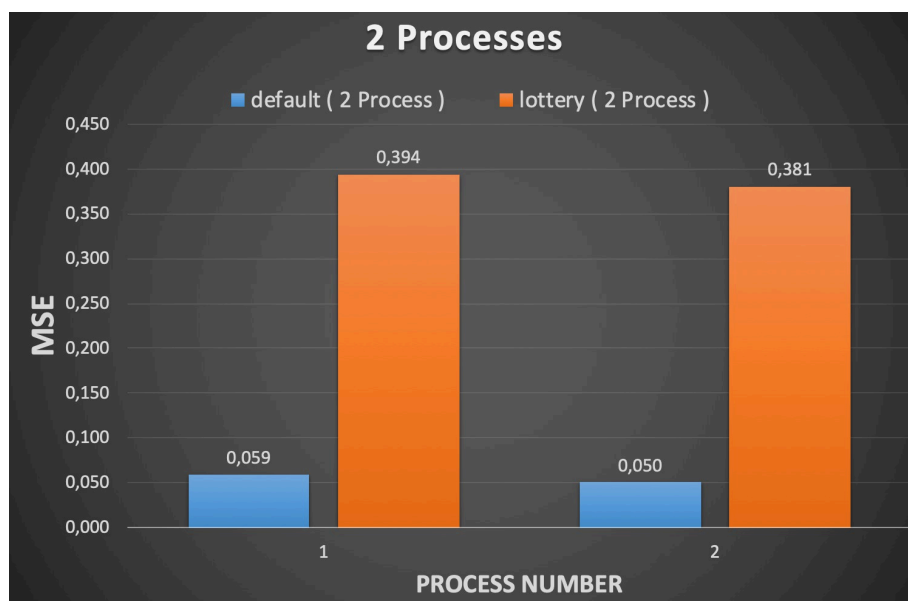
## 4. RESULTS

### 4.1.1 TEST 1

In Test 1, default and lottery scheduler both have 1 user and 2 processes.

default ( 2 Process )							
	u1p1				u1p2		
	%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2
tset1	50,291	50	0,084681		50,222	50	0,049284
test2	49,897	50	0,010609		49,916	50	0,007056
test3	50,042	50	0,001764		49,962	50	0,001444
test4	49,984	50	0,000256		50,046	50	0,002116
test5	50,001	50	1E-06		50,011	50	0,000121
test6	50,037	50	0,001369		50,141	50	0,019881
test7	50,183	50	0,033489		49,951	50	0,002401
test8	49,934	50	0,004356		50,017	50	0,000289
test9	49,529	50	0,221841		50,599	50	0,358801
test10	50,482	50	0,232324		49,748	50	0,063504
Sum:			5,90690E-01			0,504897	
		MSE :	0,059		MSE :	0,050	

lottery ( 2 Process )							
	u1p1				u1p2		
	%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2
tset1	48,68	50	1,7424		51,322	50	1,747684
test2	49,88	50	0,0144		50,118	50	0,013924
test3	49,959	50	0,001681		50,052	50	0,002704
test4	50,154	50	0,023716		49,845	50	0,024025
test5	49,45	50	0,3025		50,548	50	0,300304
test6	50,686	50	0,470596		49,521	50	0,229441
test7	49,427	50	0,328329		50,77	50	0,5929
test8	49,554	50	0,198916		50,486	50	0,236196
test9	50,903	50	0,815409		49,189	50	0,657721
test10	50,208	50	0,043264		49,957	50	0,001849
Sum:			3,941211			3,806748	
		MSE :	0,394		MSE :	0,381	



*Figure 15 Mean Square Error Values for Test 1*

#### 4.1.2 TEST 2

In Test 2, default and lottery scheduler both have 1 user and 3 processes.

default ( 3 Process )													
	u1p1				u1p2				u1p3				
	%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2		
tset1	33,362	33,333	0,000841		33,087	33,333	0,060516		33,594	33,333	0,068121		
test2	33,225	33,333	0,011664		33,007	33,333	0,106276		32,794	33,333	0,290521		
test3	33,627	33,333	0,086436		33,237	33,333	0,009216		33,165	33,333	0,028224		
test4	33,632	33,333	0,089401		33,011	33,333	0,103684		33,433	33,333	0,01		
test5	34,009	33,333	0,456976		33,752	33,333	0,175561		33,322	33,333	0,000121		
test6	32,918	33,333	0,172225		33,735	33,333	0,161604		33,458	33,333	0,015625		
test7	33,377	33,333	0,001936		33,27	33,333	0,003969		33,394	33,333	0,003721		
test8	33,3	33,333	0,001089		33,738	33,333	0,164025		33,012	33,333	0,103041		
test9	32,846	33,333	0,237169		33,707	33,333	0,139876		33,43	33,333	0,009409		
test10	33,089	33,333	0,059536		32,839	33,333	0,244036		34,184	33,333	0,724201		
Sum:			1,11727E+00				1,168763				1,252984		
MSE:			0,112		MSE:			0,117		MSE:			0,125

lottery ( 3 Process )													
	u1p1				u1p2				u1p3				
	%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2		%CPU	Y	(Y' - Y)^2		
tset1	33,959	33,333	0,391876		33,213	33,333	0,0144		32,22	33,333	1,238769		
test2	33,771	33,333	0,191844		32,806	33,333	0,277729		33,65	33,333	0,100489		
test3	34,625	33,333	1,669264		32,353	33,333	0,9604		33,239	33,333	0,008836		
test4	33,156	33,333	0,031329		34,012	33,333	0,461041		33,057	33,333	0,076176		
test5	33,012	33,333	0,103041		33,828	33,333	0,245025		33,392	33,333	0,003481		
test6	33,361	33,333	0,000784		33,313	33,333	0,0004		33,545	33,333	0,044944		
test7	33,308	33,333	0,000625		34,561	33,333	1,507984		32,346	33,333	0,974169		
test8	33,037	33,333	0,087616		33,563	33,333	0,0529		33,614	33,333	0,078961		
test9	33,197	33,333	0,018496		33,819	33,333	0,236196		33,193	33,333	0,0196		
test10	33,332	33,333	1E-06		32,767	33,333	0,320356		34,115	33,333	0,611524		
Sum:			2,494876				4,076431				3,156949		
MSE:			0,249		MSE:			0,408		MSE:			0,316



Figure 16 Mean Square Error Values for Test 2



### 4.1.3 TEST 3

In Test 3, default and lottery scheduler both have 1 user and 5 processes.

default ( 5 Process )															
	u1p1			u1p2			u1p3			u1p4			u1p5		
	%CPU	Y	(Y'-Y)^2	%CPU	Y	(Y'-Y)^2	%CPU	Y	(Y'-Y)^2	%CPU	Y	(Y'-Y)^2	%CPU	Y	(Y'-Y)^2
tset1	19,472	20	0,278784	19,927	20	0,005329	20,076	20	0,005776	19,82	20	0,0324	19,498	20	0,252004
test2	19,802	20	0,039204	20,043	20	0,001849	20,385	20	0,148225	19,909	20	0,008281	19,795	20	0,042025
test3	19,483	20	0,267289	19,904	20	0,009216	20,053	20	0,002809	19,858	20	0,020164	20,012	20	0,000144
test4	20,085	20	0,007225	20,267	20	0,071289	20,024	20	0,000576	19,298	20	0,492804	19,69	20	0,0961
test5	20,113	20	0,012769	19,32	20	0,4624	20,267	20	0,071289	20,009	20	8,1E-05	20,127	20	0,016129
test6	20,422	20	0,178084	19,715	20	0,081225	19,857	20	0,020449	20,156	20	0,024336	19,727	20	0,074529
test7	19,325	20	0,455625	19,823	20	0,031329	20,73	20	0,5329	20,162	20	0,026244	19,911	20	0,007921
test8	20,12	20	0,0144	20,571	20	0,326041	19,608	20	0,153664	19,889	20	0,012321	19,724	20	0,076176
test9	19,931	20	0,004761	20,225	20	0,050625	19,582	20	0,174724	20,039	20	0,001521	20,037	20	0,001369
test10	19,214	20	0,617796	20,728	20	0,529984	19,557	20	0,196249	20,537	20	0,288369	19,852	20	0,021904
Sum:			1,87594E+00			1,569287			1,306661			0,906521			0,588301
	MSE:		0,188	MSE:		0,157	MSE:		0,131	MSE:		0,091	MSE:		0,059

lottery (5 Process)																			
	u1p1				u1p2				u1p3				u1p4				u1p5		
	%CPU	Y	(Y'-Y)^2		%CPU	Y	(Y'-Y)^2		%CPU	Y	(Y'-Y)^2		%CPU	Y	(Y'-Y)^2		%CPU	Y	(Y'-Y)^2
tset1	20,309	20	0,095481	20,378	20	0,142884	19,604	20	0,156816	19,919	20	0,006561	20,009	20	8,1E-05				
test2	19,846	20	0,023716	20,067	20	0,004489	19,626	20	0,139876	20,472	20	0,222784	20,224	20	0,050176				
test3	19,615	20	0,148225	19,78	20	0,0484	20,099	20	0,009801	20,5	20	0,25	20,225	20	0,050625				
test4	20,023	20	0,000529	20,811	20	0,657721	19,993	20	4,9E-05	20,034	20	0,001156	19,358	20	0,412164				
test5	20,048	20	0,002304	20,816	20	0,665856	19,613	20	0,149769	20,274	20	0,075076	19,293	20	0,499849				
test6	20,533	20	0,284089	19,142	20	0,736164	20,522	20	0,272484	20,159	20	0,025281	19,416	20	0,341056				
test7	20,423	20	0,178929	19,5	20	0,25	19,624	20	0,141376	20,465	20	0,216225	19,76	20	0,0576				
test8	20,438	20	0,191844	20,728	20	0,529984	19,516	20	0,234256	19,267	20	0,537289	20,103	20	0,010609				
test9	20,234	20	0,054756	19,452	20	0,300304	19,504	20	0,246016	21,029	20	1,058841	19,843	20	0,024649				
test10	21,269	20	1,610361	19,956	20	0,001936	19,626	20	0,139876	20,062	20	0,003844	19,156	20	0,712336				
Sum:			2,59023E+00			3,337738			1,490319			2,397057			2,159145				
	MSE:	0,259		MSE:	0,334		MSE:	0,149		MSE:	0,240		MSE:	0,216					

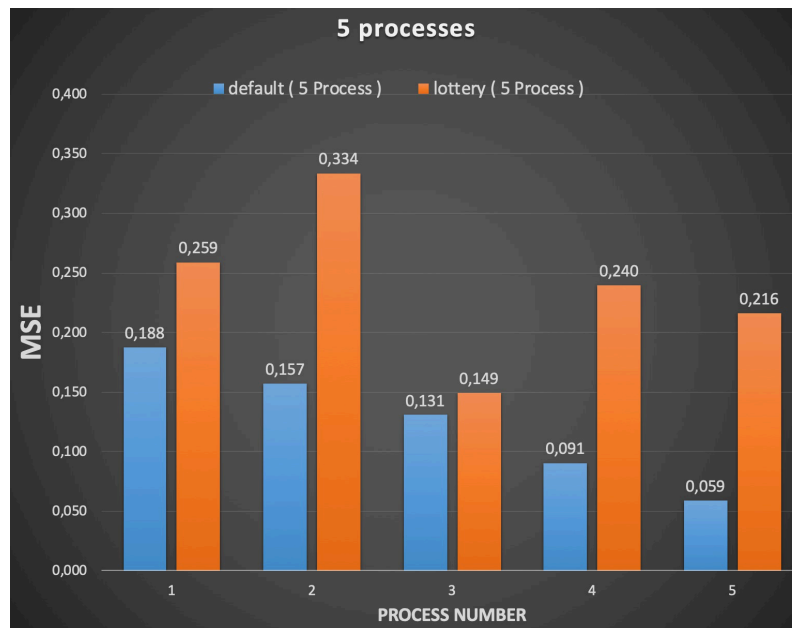


Figure 17 Mean Square Error Values for Test 2



## 5. CONCLUSION

This project demonstrates the implementation of a user-based scheduling algorithm within Linux systems. The structured scheduler aims for user fairness, with our testing revealing a decrease in mean-square-error (MSE) of fairness as the number of users increases. Moreover, when an equal number of processes exists within each user, the MSE also diminishes. The Default Linux Scheduler is anticipated to prioritize process fairness, while the Lottery Scheduler developed in this project prioritizes user fairness. Our testing utilized two distinct methods to assess fairness for both schedulers. In process-oriented tests, it becomes evident that the MSE value of the Default Scheduler is notably lower than that of the Lottery Scheduler.

Both schedulers have their own advantages and disadvantages. The default scheduler is known for its fairness among processes, but it lacks flexibility for adapting to changes in resource allocation. On the contrary, our lottery scheduler offers greater customization options to achieve a more balanced distribution of resources, but it may be less efficient and require additional computational resources to implement. Ultimately, the choice of scheduler depends on the specific requirements and objectives of the system at hand. The Linux 2.4.27 default scheduler and lottery scheduler each possess their own strengths and weaknesses. The optimal decision will be influenced by the specific context and goals of the system.

In summary, user-based scheduling is viable, but it demands precision and a deep comprehension of the operating system when crafting scheduling algorithms and modifying kernels. Additionally, debugging kernel code is intricate, underscoring the importance of thoroughly testing programs and algorithms for any potential logical flaws.

## REFERENCES

- A.Silberschatz et al, “Operating System Concepts”, Addison Wesley , Ed. 5+, P.129-153
- Tanenbaum, “Modern Operating Systems”, Prentice Hall, P.143-161
- R. Arpaci-Durseau, "Three Easy Pieces: Online OS book", P.63-72