# G-TICKET SCHEDULER

**by**

**Ahmet SALTAN**

**Caner KAYA**

**Maruf Emre KARATAY**

**CSE 331 Operating Systems Design**

**Term Project Report**

**Yeditepe University**

**Faculty of Engineering**

**Department of Computer Engineering**

**Spring 2020**

**ABSTRACT**

The fair scheduler allows processes to use CPU evenly based on a criterion. It can be based on processes, users, or groups. Linux scheduler is process-based, it tries to give all processes even time in CPU. Our gticket scheduler is based on groups, it tries to give all groups even time in CPU. Our implementation is not allowing a process to take CPU if a process from the same group has been in CPU while other processes from other groups have not been in CPU yet basically. Also, the g-ticket value of processes is used to prioritize the processes. The higher the g-ticket value is, the more chance it is getting the CPU and vice versa. Tests show that gticket scheduler works properly, but it is not fair based on processes inside groups.

# TABLE OF CONTENTS

# 1. INTRODUCTION

## What is CPU Scheduling?

One of the most important aspects of the operating system design is the efficient use of a processor. If the operating system support multiprocessing, the only thing to do is to keep multiple processes in memory and run them in order. This decision of process running time is CPU scheduling.

## Who makes the CPU Scheduling? Schedulers

The Schedulers in OS are the algorithms that help in the system optimization for maximum performance. Schedulers make the above-mentioned scheduling. That is, schedulers decide which process will run and which process will wait. This decision of process running time that is scheduling is made by scheduler algorithms. CPU scheduler algorithms aim to make the system efficient, fast, and fair.

## Why it is important to have a fair scheduler?

It's important to balance long-running jobs and ensure that the lighter jobs can be run quickly. For example; Just as it isn't fair for someone to bring a loaded shopping cart to the 10-items-or-less checkout, the operating system shouldn't give an unfair advantage to a process that will interfere with the criteria we listed (CPU utilization, wait time, throughput).

## What is the Fair-Share Scheduling?

**Fair-share scheduling** is a scheduling strategy for computer operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes.

**Suppose that we have 2 groups and its users**

**We have a 50% CPU usage per group.**

| Group1 | 50% |
|---|---|
| User1 | 16.6% |
| User2 | 16.6% |
| User3 | 16.6% |

| Group2 | 50% |
|---|---|
| User4 | 25% |
| User5 | 25% |

## What Are the Schedulers Not Inside the Linux Kernel?

**Two scheduling algorithms other than GTicket and Default Scheduler:**

### 1) Lottery Scheduling

**Lottery Scheduling** is a type of process scheduling and it is different from other scheduling algorithms. In lottery scheduling, processes are scheduled randomly. Firstly, this algorithm gives a lottery ticket to each process. That is, every process has some tickets. Then, the scheduler picks a random ticket and process having that ticket is the winner and it is executed for a time interval. After that process running, another ticket is picked by the scheduler. This algorithm solves the problem of starvation.

### 2) Shortest-Job-First Scheduling

**Shortest-Job-First Scheduling** checks the processes one by one. Then, the scheduler chooses the shortest process from the processes and gives a priority to execute. That is Shortest-Job-First algorithm schedule processes in the order in which the shortest job is done first. Thus, it has a minimum average waiting time.

## 2. DESIGN and IMPLEMENTATION

To be able to switch between Linux scheduler and g-ticket scheduler, we added two if statements to our system call. If the system call is called with the second parameter (*option*) equal to 1, scheduler changes to the g-ticket scheduler by changing global variable *GOPTION* in **sched.c** to 1. If the system call is called with *option* equal to 0, scheduler changes to default Linux scheduler by changing *GOPTION* value to 0. By default, *GOPTION* is equal to 0. To be able to change *GOPTION* value which is not inside the c file of our system call, we defined the *GOPTION* variable as an **extern** in the c file of our system call.

```
#include <linux/cprocessinf.h>
extern int GOPTION;
asmlinkage int sys_cprocessinf(struct prcdata *data, int option, int nicev){
    if(option == 1)
    {
        cli();
        GOPTION = 1;
        sti();
        return 0;
    }
    else if(option == 0)
    {
        cli();
        GOPTION = 0;
        sti();
        return 0;
    }
```

G-ticket scheduler needs processes to have three extra variables which are called *gticket*, *gtime*, and *available*, so we added these variables inside **task struct** structure, which can be found in **sched.h**. Initially, we want the gticket variable to be equal to 6, gtime variable to be equal to current jiffies value and available variable to be equal to 1. So we added necessary parts at **fork.c**, inside **do_fork** function.

```
p -> gtickets = 6;  /* initial gticket assignment */
p -> gtime = jiffies;   /* initial jiffies assignment */
p -> available = 1; /* initial available assignment */
```

The *gticket* variable is used to increase the chance of a process getting in CPU if it has not been in CPU in a long time by increasing the *gticket* value of that process and decrease the chance of a process getting in CPU if it has been in CPU in a short time by decreasing the *gticket* value of that process. As seen, we need to hold the time data to accomplish that so we have a *gtime* variable. The *available* variable is used to implement equal CPU usage time between groups.

We added an if condition inside **sched.c** file under **repeat_schedule:** part. It is for deciding which scheduler to run using the *GOPTION* variable as mentioned above. Before deciding which process to take CPU, we adjusted the *gticket* and *gtime* variables of all processes. We created a variable called *cur_time* and initialized it to current jiffies value to compare time values. If less than 4 ms has elapsed from the last time process was on CPU, the process loses one ticket but it can not be less than 1. If more than 12 ms has elapsed from the last time process was on CPU, the process gains one ticket but it can not be more than 11. We only consider processes that a user running, so we check whether these processes group id is greater than 500 or not. Because all users are in a group that has a group id greater than 500.

After adjusting the *gticket* values, we have to find the maximum number of tickets a process has. We only want to consider the processes that have the *available* variable is equal to 1. Because that means these processes can get the CPU, a process from the group that they belong to has not been in CPU yet and vice versa. We have declared a variable called *max_of_tickets* which is initially equals to 1 and changed it if any process has more tickets than *max_of_tickets* value. Then we used a variable called *grandom* to hold a random value between 1 and *max_of_tickets*.

```
list_for_each(tmp, &runqueue_head)
{
    p = list_entry(tmp, struct task_struct, run_list);
    if(p -> gid > 500)
    {
        if(cur_time - p -> gtime < 4)
        {
            if(p -> gtickets > 1)
            {
                p -> gtickets = p -> gtickets - 1;
            }
        }
        if(cur_time - p -> gtime > 12)
        {
            if(p -> gtickets < 11)
            {
                p -> gtickets = p -> gtickets + 1;
            }
        }
        if(p -> available == 1)
        {
            if(p -> gtickets > max_of_tickets)
            {
                max_of_tickets = p -> gtickets;
            }
        }
    }
}
```

```
get_random_bytes(&grandom, sizeof(grandom));
if(grandom < 0)
{
    grandom = grandom * (-1);
}
grandom = (grandom % max_of_tickets);
grandom = grandom + 1;
```

We have a variable called *selected_gid*. It is used to store the group id of the selected process. It is initialized to -1 because we only want to select one process at an iteration. So we check if it is equal to -1 before we select a process to get the CPU and after that, we change it to the selected processes group id. Since it is not equal to -1 anymore, another process can not be chosen.

To select a process to get the CPU, we check its group id. If it is not greater than 500, it is treated by the default Linux scheduler. Else, we check whether its *available* variable is equal to 1 and its *gtickets* variable is greater than or equal to the *grandom* variable or not. And also we check the *selected_gid* variable as mentioned above. If all requirements are met, we pick this process as the next process and change its *gtime* variable to *cur_time* and *available* variable to 0. Also, we get its group id and store it inside the *selected_gid* variable.

```
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))
    {
        if(p -> gid > 500)
        {
            if(p -> available == 1)
            {
                if(p->gtickets >= grandom)
                {
                    if(selected_gid==-1)
                    {
                        selected_gid = p -> gid;
                        p -> gtime=cur_time;
                        next = p;
                    }
                }
            }
        }
        else
        {
            int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
            c = weight, next = p;
        }
    }
}
```

After selecting the process, if the selected group id is greater than 500, we trace all processes and change their *available* variable to 0 if their group id is equal to *selected_gid*.

```
if(selected_gid > 500)
{
    list_for_each(tmp, &runqueue_head)
    {
        p = list_entry(tmp, struct task_struct, run_list);
        if(p -> gid == selected_gid)
        {
            p -> available = 0;
        }
    }
}
```

Then we calculate the sum of *available* values of all the processes that have group id greater than 500. We store the result in a variable called *sum*. It is initialized to 0. If the *sum* is equal to 0, it means that a process from all groups has been in CPU, so we have to change the *available* values of all processes that have group id greater than 500 back to 1.

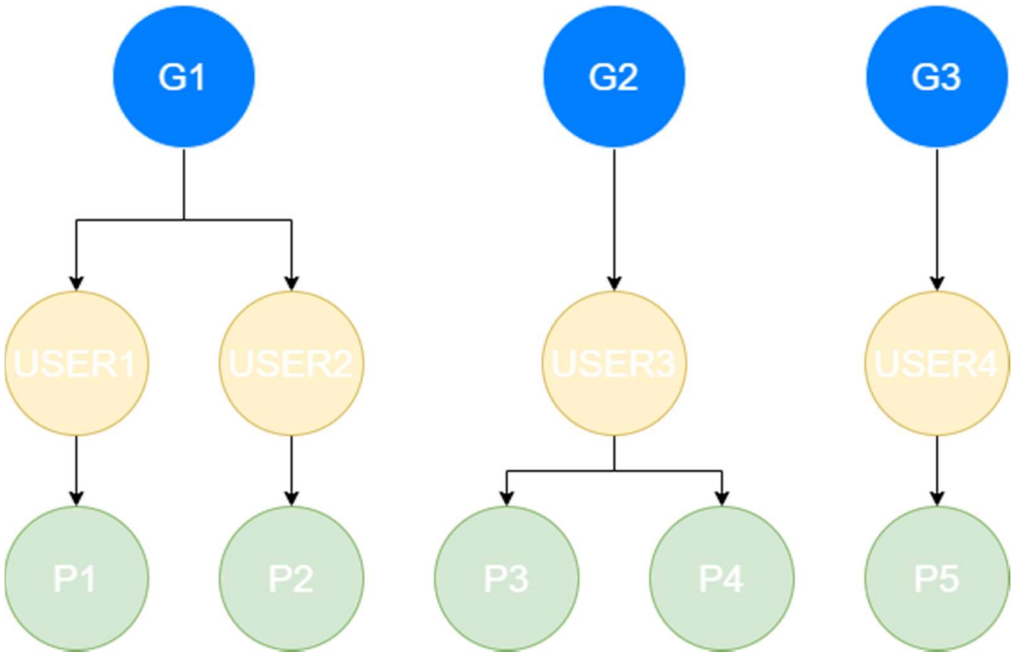
```
sum = 0;
list_for_each(tmp, &runqueue_head)
{
    p = list_entry(tmp, struct task_struct, run_list);
    if(p -> gid > 500)
    {
        sum = sum + p -> available;
    }
}

if(sum == 0)
{
    list_for_each(tmp, &runqueue_head)
    {
        p = list_entry(tmp, struct task_struct, run_list);
        if(p -> gid > 500)
        {
            p -> available = 1;
        }
    }
}
```

Finally, we add the re-calculation of counters part which already exists in default Linux scheduler for the processes that have group id less than or equal to 500.

## 3. TESTS and RESULTS

First of all, to write test results to file we use 2 shell script files. In the first shell script file, we run **5 processors** which are given in the below diagram. Then, we run the second file which writes **"top"** command to txt files.




We took txt files from Linux with the FileZilla app. We inserted results into an Excel file and used built-in formula features to calculate **total CPU usage** of **groups** and to calculate **Mean Square Error** (MSE). We constructed the excel file with 2 pages. The first one for **group-oriented calculation** and the second one for **process-oriented calculation**. The first page consists of 6 different tables. The first two of them are the results of the "top" command.

| Gticket Scheduler | | | | | Default Scheduler | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Group 1 | Group 2 | Group 2 | Group 3 | | Group 1 | Group 1 | Group 2 | Group 2 | Group 3 |
| User 2 | User 3 | User 3 | User 4 | | User 1 | User 2 | User 3 | User 3 | User 4 |
| P2 | P3 | P4 | P5 | | P1 | P2 | P3 | P4 | P5 |
| 13.8 | 20.7 | 12.8 | 27.7 | | 19.9 | 21.9 | 17.9 | 17.8 | 17.9 |
| 3.5 | 28.9 | 3.1 | 32.9 | | 19.9 | 19.5 | 20.3 | 20.3 | 19.5 |
| 4.1 | 27.3 | 4.7 | 30.9 | | 20.3 | 20.3 | 20.3 | 20.3 | 19.9 |
| 4.3 | 27.3 | 3.5 | 30.9 | | 20.3 | 19.3 | 19.1 | 19.1 | 20.3 |
| 5.3 | 26.7 | 5.1 | 32.1 | | 19.1 | 20.1 | 20.3 | 20.3 | 20.3 |
| 6.3 | 25.9 | 6.7 | 30.1 | | 20.3 | 20.3 | 20.3 | 19.5 | 19.1 |
| 6.5 | 28.1 | 6.3 | 32.7 | | 20.3 | 19.9 | 19.1 | 19.9 | 20.3 |
| 6.5 | 26.5 | 7.3 | 33.5 | | 19.1 | 19.5 | 19.1 | 19.1 | 19.1 |
| 6.9 | 25.9 | 6.9 | 33.3 | | 18.1 | 17.9 | 19.1 | 19.1 | 19.1 |
| 6.5 | 26.9 | 6.3 | 32.3 | | 18.3 | 19.1 | 18.9 | 17.9 | 18.9 |

The second two tables show the CPU usage of groups that were founded with formula excel functions. An example of the feature is given below.


Some part of the tables is given below.

| | Default Scheduler Groups | | | | Gticket Scheduler Groups | | |
|---|---|---|---|---|---|---|---|
| | Group 1 | Group 2 | Group 3 | | Group 1 | Group 2 | Group 3 |
| Predicted | 33.3333 | 33.3333 | 33.3333 | | 33.33333 | 33.33333 | 33.33333 |
| | Actual Values | | | | Actual Values | | |
| | 41.8 | 35.7 | 17.9 | | 36.6 | 33.5 | 27.7 |
| | 39.4 | 40.6 | 19.5 | | 32.4 | 32 | 32.9 |
| | 40.6 | 40.6 | 19.9 | | 33 | 32 | 30.9 |
| | 39.6 | 38.2 | 20.3 | | 35.4 | 30.8 | 30.9 |
| | 39.2 | 40.6 | 20.3 | | 32.6 | 31.8 | 32.1 |
| | 40.6 | 39.8 | 19.1 | | 33.4 | 32.6 | 30.1 |
| | 40.2 | 39 | 20.3 | | 32.8 | 34.4 | 32.7 |
| | 38.6 | 38.2 | 19.1 | | 32.6 | 33.8 | 33.5 |
| | 36 | 38.2 | 19.1 | | 33.2 | 32.8 | 33.3 |

The third two tables demonstrate squared errors and MSE which are calculated with formulas again. The formulas and some parts of the table are given below.

$$MSE = \frac{1}{n}\sum\underbrace{\left(y - \hat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

To get squared errors:


To get MSE of each group:

To calculate the average of MSE:


| | Default Scheduler Errors | | | |
|---|---|---|---|---|
| | Group 1 | Group 2 | Group 3 | AVG |
| MSE: | 38.5541 | 38.432 | 185.368 | 87.45122 |
| | Squared Error | | | |
| | 71.6844 | 5.60111 | 238.188 | |
| | 36.8044 | 52.8044 | 191.361 | |
| | 52.8044 | 52.8044 | 180.454 | |
| | 39.2711 | 23.6844 | 169.868 | |
| | 34.4178 | 52.8044 | 169.868 | |
| | 52.8044 | 41.8178 | 202.588 | |
| | 47.1511 | 32.1111 | 169.868 | |
| | 27.7378 | 23.6844 | 202.588 | |
| | 7.11111 | 23.6844 | 202.588 | |
| | 16.5378 | 12.0178 | 208.321 | |

| | Gticket Scheduler Errors | | | |
|---|---|---|---|---|
| | Group 1 | Group 2 | Group 3 | AVG |
| MSE: | 4.605071 | 4.305444 | 5.095018 | 4.668511 |
| | Squared Error | | | |
| | 10.67111 | 0.027778 | 31.73444 | |
| | 0.871111 | 1.777778 | 0.187778 | |
| | 0.111111 | 1.777778 | 5.921111 | |
| | 4.271111 | 6.417778 | 5.921111 | |
| | 0.537778 | 2.351111 | 1.521111 | |
| | 0.004444 | 0.537778 | 10.45444 | |
| | 0.284444 | 1.137778 | 0.401111 | |
| | 0.537778 | 0.217778 | 0.027778 | |
| | 0.017778 | 0.284444 | 0.001111 | |
| | 1.137778 | 0.017778 | 1.067778 | |

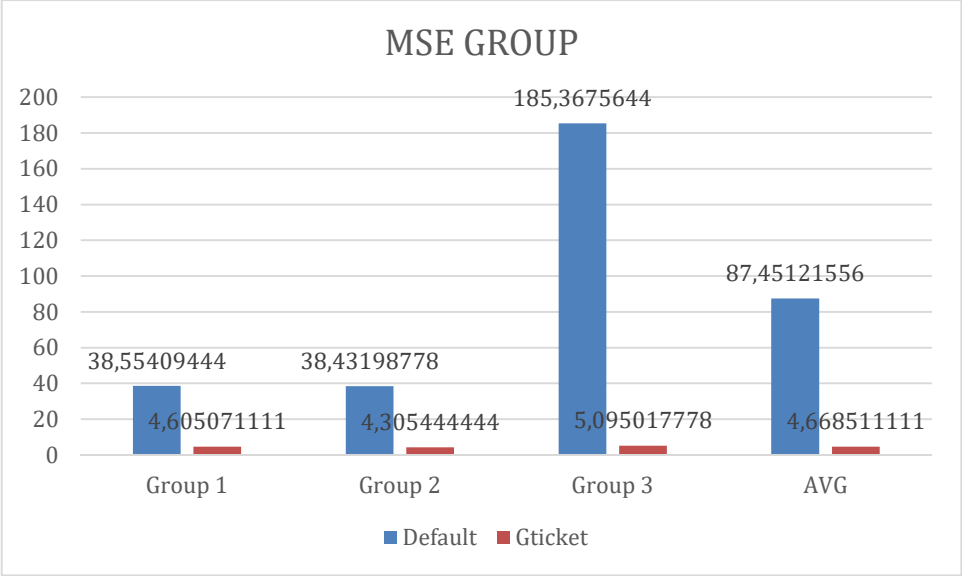
The second page consists of 4 tables for process-oriented calculation. The first two are the same as the first two on page 1 which are results of "top". The second two tables are there for squared errors and MSE founded with the same formulas.

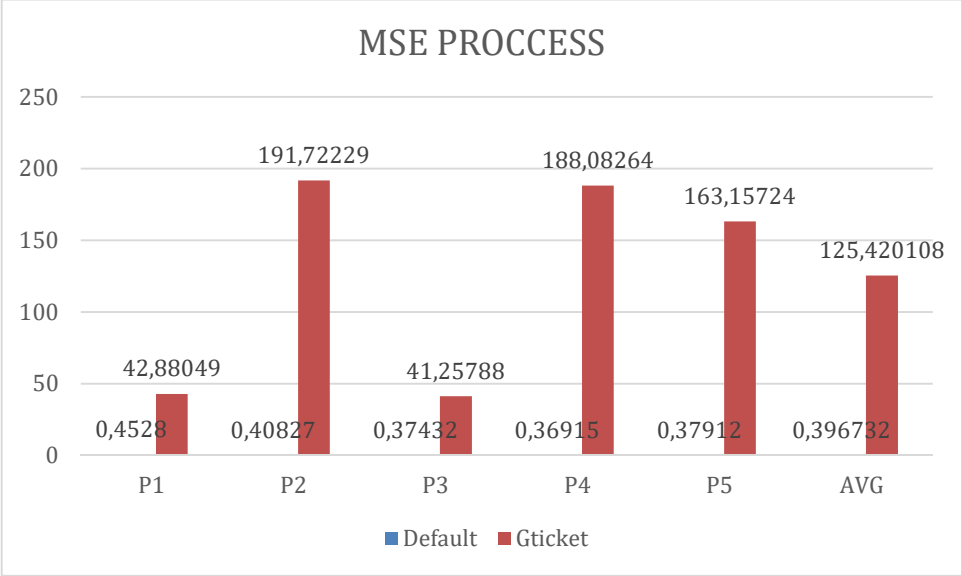
| | Gticket Scheduler ERROR | | | | | |
|---|---|---|---|---|---|---|
| | P1 | P2 | P3 | P4 | P5 | |
| PREDICTED | 20 | 20 | 20 | 20 | 20 | AVG |
| MSE: | 42.88049 | 191.7223 | 41.25788 | 188.0826 | 163.1572 | 125.4201 |
| | 7.84 | 38.44 | 0.49 | 51.84 | 59.29 | |
| | 79.21 | 272.25 | 79.21 | 285.61 | 166.41 | |
| | 79.21 | 252.81 | 53.29 | 234.09 | 118.81 | |
| | 123.21 | 246.49 | 53.29 | 272.25 | 118.81 | |
| | 53.29 | 216.09 | 44.89 | 222.01 | 146.41 | |
| | 50.41 | 187.69 | 34.81 | 176.89 | 102.01 | |
| | 39.69 | 182.25 | 65.61 | 187.69 | 161.29 | |
| | 37.21 | 182.25 | 42.25 | 161.29 | 182.25 | |
| | 39.69 | 171.61 | 34.81 | 171.61 | 176.89 | |
| | 62.41 | 182.25 | 47.61 | 187.69 | 151.29 | |

| | Default Scheduler ERROR | | | | | |
|---|---|---|---|---|---|---|
| | P1 | P2 | P3 | P4 | P5 | |
| | 20 | 20 | 20 | 20 | 20 | AVG |
| MSE: | 0.4528 | 0.40827 | 0.37432 | 0.36915 | 0.37912 | 0.396732 |
| | 0.01 | 3.61 | 4.41 | 4.84 | 4.41 | |
| | 0.01 | 0.25 | 0.09 | 0.09 | 0.25 | |
| | 0.09 | 0.09 | 0.09 | 0.09 | 0.01 | |
| | 0.09 | 0.49 | 0.81 | 0.81 | 0.09 | |
| | 0.81 | 0.01 | 0.09 | 0.09 | 0.09 | |
| | 0.09 | 0.09 | 0.09 | 0.25 | 0.81 | |
| | 0.09 | 0.01 | 0.81 | 0.01 | 0.09 | |
| | 0.81 | 0.25 | 0.81 | 0.81 | 0.81 | |
| | 3.61 | 4.41 | 0.81 | 0.81 | 0.81 | |
| | 2.89 | 0.81 | 1.21 | 4.41 | 1.21 | |

To calculate MSE, we have to determine fair predicted values. We determined them as 100/(group count) which is **33.33** for group-oriented calculation and 100/(process count) which is **20** for process-oriented calculation.

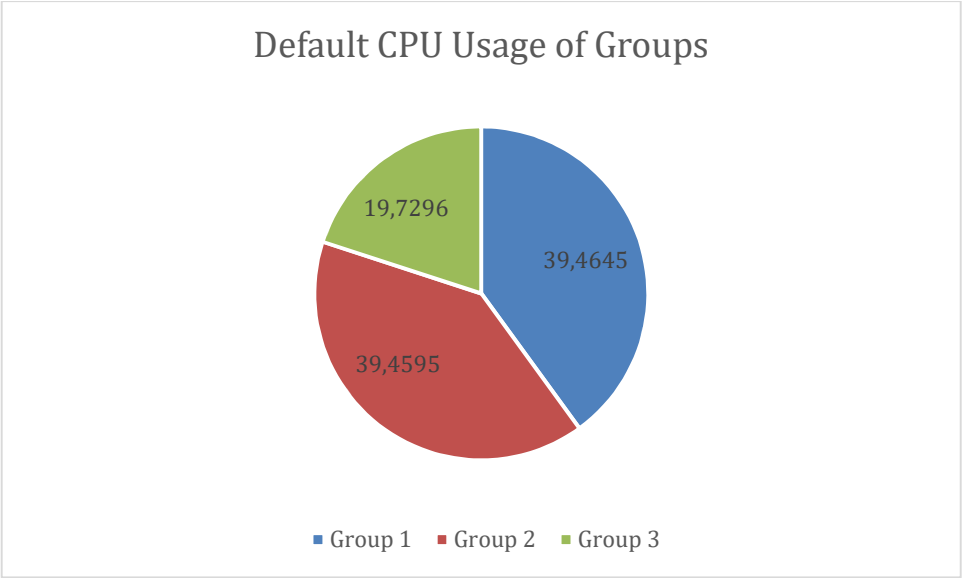
MSE bar charts:

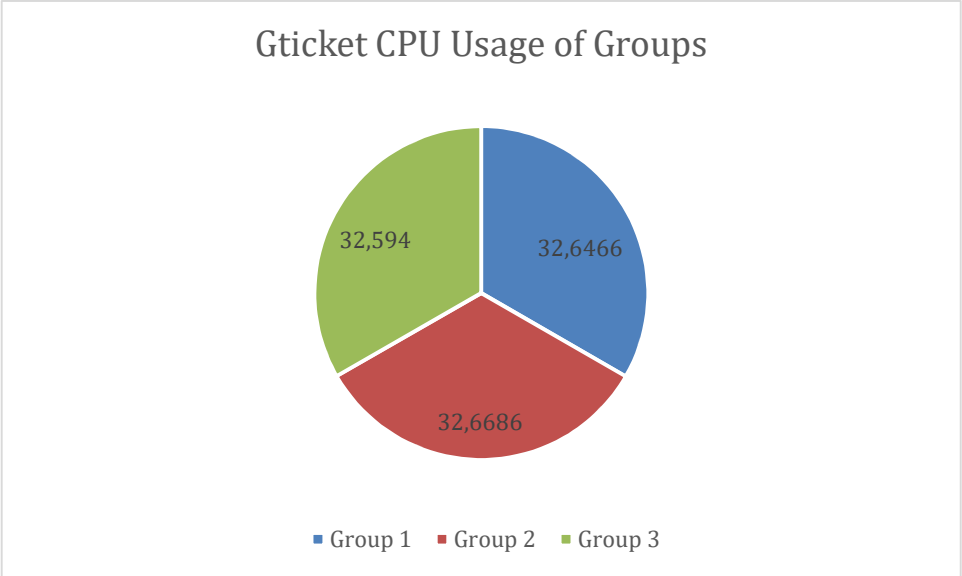



According to tables, our Gticket algorithm is better than the default Linux scheduling algorithm for group-oriented fairness. On the other hand, the default scheduler is far better than the Gticket scheduler for process-oriented fairness so we can not even see default error without the numbers on it.

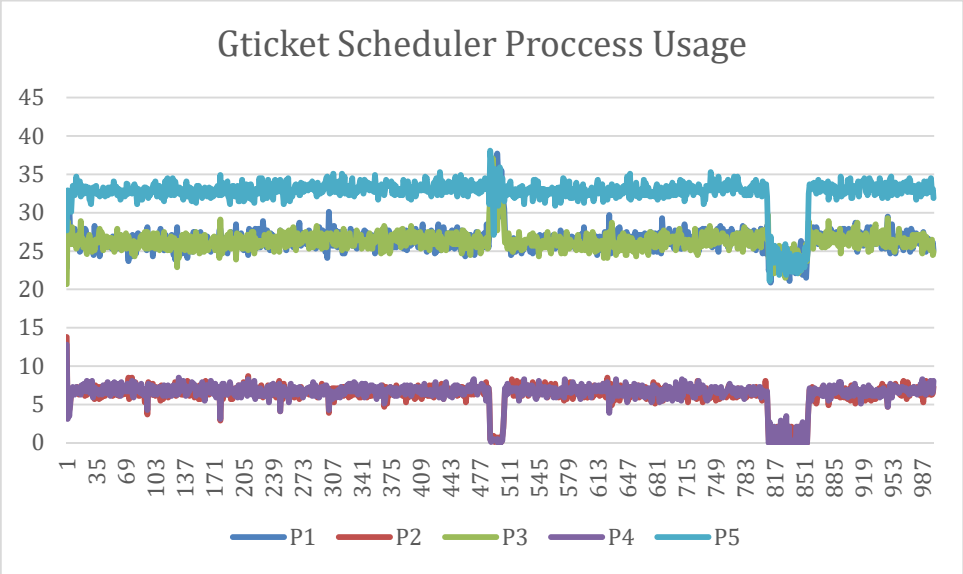
Pie Charts of Usage of Groups:

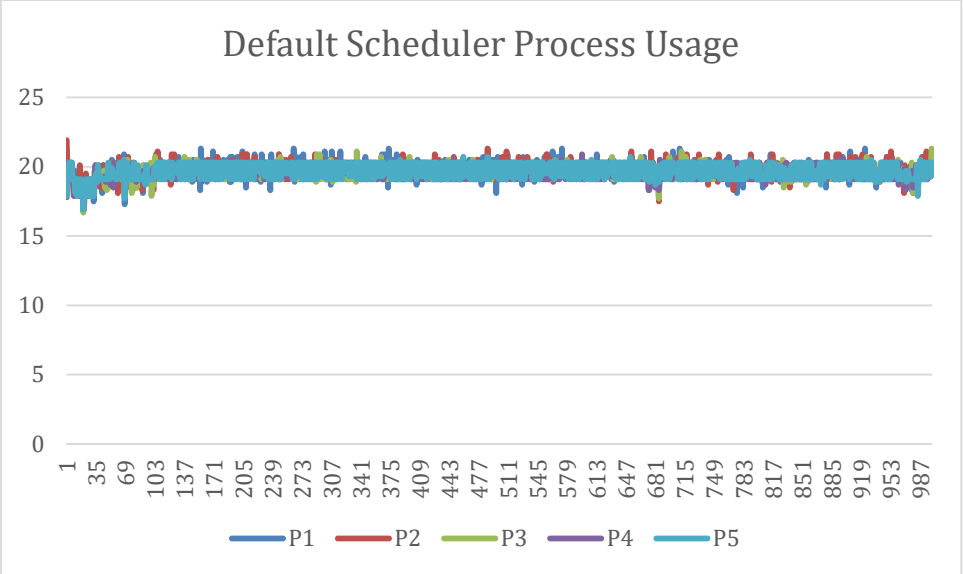



According to the pie charts, we can see that our Gticket algorithm works like expected. Every group has nearly a percentage of 100/3 in our algorithm. In the Linux default scheduler, groups are ignored.

CPU Usage of Processes Chart:




In default scheduler, processes have nearly the same CPU usage. However, in Gticket scheduler processes have different CPU usage but have 33.33% total in the groups. Surprisingly, in Gticket one of the processes in the same group always uses more than another. This simply shows that Gticket is not fair for processes-oriented fairness but fair for group-oriented fairness.

# 4. CONCLUSION

In conclusion, Our Gticket scheduler works like a charm when we just want to focus on group fairness. However, It is not fair when we want to focus on process-oriented fairness. That's why We should keep using the Linux default scheduler when we want fairness between processes. For future work, the algorithm can be improved by implementing a fair CPU usage inside every group, between processes or users.

As a group, We learned system calls, scheduling, and customizing the Linux kernel. To be **fair** ☺ , this project was good and thought us a lot. We struggled a lot at the beginning and we had to arrange meetings with dear assistants. They helped a lot but we could not even need that. Our advice for further projects, show some examples rather than saying do this and that.

## REFERENCES

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html

https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html

https://www.studytonight.com/operating-system/first-come-first-serve

https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1/

CPU Scheduling PDF's which in https://coadsys.yeditepe.edu.tr/