CSE 331 – Operating Systems Design
Project Report

System Calls and Fair-share CPU Scheduling

by

Deniz Tuana ERGÖNÜL

Orçun Soner EREN

Yeditepe University

Faculty of Engineering and Architecture

Department of Computer Engineering

2016

TABLE OF CONTENTS

1. INTRODUCTION

The aim of this project is to get started with understanding the Linux kernel. Some of the definitions which were taught during the semester, especially, “user-space/kernel-space”, “system call” and “scheduling” will be practised.

The project is divided in two phases; the first phase is how to add a system call which returns some values (uid, pid, nice, counter and etc.) of the current process, the second will focus on how to implement a fair-share CPU scheduler.

2. PREPARATIONS

2.1. Prior-Knowledge

User-Space/Kernel-Space

An operating system is an interface between the user and the hardware. Operating system(OS) provides an environment where users and programs cannot directly access the hardware. Kernel mode is trusted and user mode is untrusted. A switch between user mode and kernel mode can be made.

Every system call contains a trap. Trap is a software interrupt and it results in a switch to kernel mode. It is caused by division by zero or invalid memory access.

Kernel is the main OS program and it is always in primary memory. Kernel is capable of creating a memory space and creating a process.

Normally, a user-space program cannot reach a kernel-space variable. In our particular case, there is no way to reach the values of the current process from a user-space program. System calls are used to achieve this, to reach the kernel from user-space.

2.2. System Preparation

In this project, Red Hat Linux release 9(2.4.20-8) is going to be used. “/usr/src/linux-2.4.20” contains the kernel codes that will be used during the project work. Every time a change is made in the kernel space, the kernel must be compiled, in other words, the system must be booted to see the changes. Note that the user must be “root” to boot the system. If the kernel cannot be loaded into the memory properly, the system cannot work.

3. PHASE I: ADDING A SYSTEM CALL

3.1. Purpose

Add a system call to the kernel. Return the values of the current process. Learn how to make a system call and read/write to user space from kernel space or to kernel space from user space by adding a new function to the kernel.

3.2. Definitions

3.2.1. System Call

System calls are to reach the functions inside the kernel. System calls execute in kernel mode. Every system call causes a trap. A set of system calls is what programs think the OS is. Trapping the kernel can be achieved by system calls. The user cannot reach the trusted code directly. Processes are also created by making system calls.

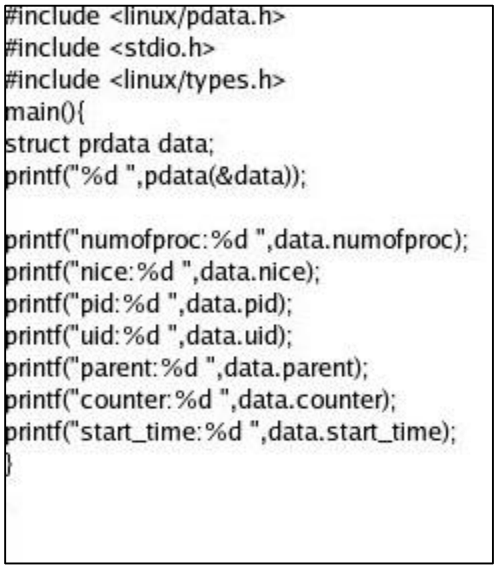
When a system call is called, a software interrupt is done to the kernel. Kernel stops its current work to handle the interrupt. After the interrupt is handled, kernel resumes its own work.

3.2.2. Task Struct

Every process has a “task_struct”. In “task_struct”, the information about the process is kept. Nice, counter, pid, uid, parent and start_time values of the processes can be found in this struct. Every process knows the previous and the next process. Note that the value of “numofproc” can only be read by atomic_read().

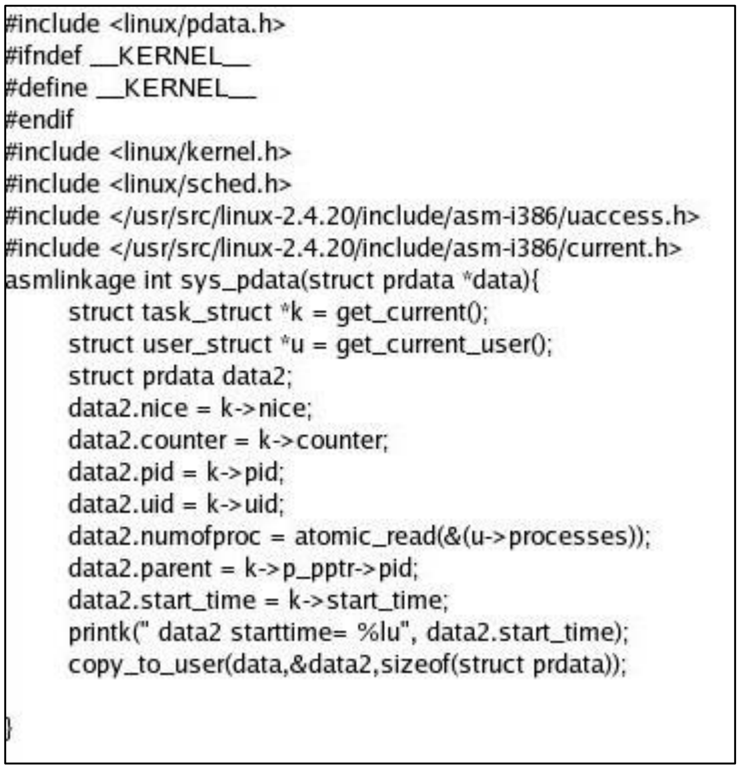
3.3. Procedure

The system call definitions must be added to both kernel-space and user-space. “Copy_to_user” is used to copy the bytes in kernel-space to the user. First step must be to find the current process and then return the values of it. The current process is the process who calls the system call. First, system calls are defined in system call table. By this way, the new system call is added in system call trap table. The reason why is defining a system call into x86 CPU. “unistd.h” is contained in kernel space which keeps system call definitions. The new system call is also added to “unistd.h”. Thus, our system call is defined in kernel space. After that, the system call must be defined in user space and then in kernel space system call’s header is created. It is also created in user space. Thus, the connection between kernel space and user space is created. Lastly, kernel compilation must be done.



```
#include <linux/pdata.h>
#include <stdio.h>
#include <linux/types.h>
main(){
struct prdata data;
printf("%d ",pdata(&data));

printf("numofproc:%d ",data.numofproc);
printf("nice:%d ",data.nice);
printf("pid:%d ",data.pid);
printf("uid:%d ",data.uid);
printf("parent:%d ",data.parent);
printf("counter:%d ",data.counter);
printf("start_time:%d ",data.start_time);
}
```

(The user-space code to print out the values)



```
#include <linux/pdata.h>
#ifndef __KERNEL__
#define __KERNEL__
#endif
#include <linux/kernel.h>
#include <linux/sched.h>
#include </usr/src/linux-2.4.20/include/asm-i386/uaccess.h>
#include </usr/src/linux-2.4.20/include/asm-i386/current.h>
asmlinkage int sys_pdata(struct prdata *data){
        struct task_struct *k = get_current();
        struct user_struct *u = get_current_user();
        struct prdata data2;
        data2.nice = k->nice;
        data2.counter = k->counter;
        data2.pid = k->pid;
        data2.uid = k->uid;
        data2.numofproc = atomic_read(&(u->processes));
        data2.parent = k->p_pptr->pid;
        data2.start_time = k->start_time;
        printk(" data2 starttime= %lu", data2.start_time);
        copy_to_user(data,&data2,sizeof(struct prdata));

}
```

(Kernel-space code reaching the values which the user cannot reach on its own)

3.4. Conclusion

User space programs cannot reach kernel space values on their own. A system call must be created and defined both in kernel space and user space to achieve this.

4. PHASE II: IMPLEMENTING A CPU SCHEDULER

4.1. Purpose

Learning more about how process management is designed, particularly how the scheduler is implemented. The scheduler is the heart of the multiprogramming system; it is responsible for choosing a new task to run whenever the CPU is available.

A hint is the brute force solution of the problem, which has a high overhead, it is O(nlogn). An answer to the question “How can the overhead can be lowered?” will be the solution.

Mean square error(MSE) must be used for the difference tests between the original scheduler and the fair-share scheduler.

Note that the assumption is the processes are CPU burst, CPU bound processes. Actually, this is rare since a process is usually I/O burst.

4.2. Definitions

4.2.1. CPU Scheduler

One of the kernel’s functions is scheduling the CPU between processes. OS manages resource sharing in general. OS switches one process to another. It ensures that address spaces do not touch each other.

A CPU bound process is for example while(1). Note that the assumption is, there is no I/O bound processes.

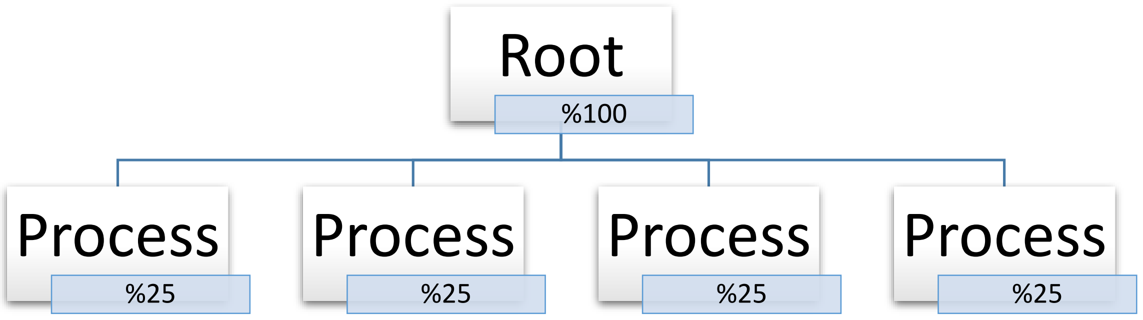
A process can be available for execution(ready), partially in main memory(swapped out), in secondary memory or not started yet. Processes have states(waiting, running, ready) kept in “sched.h”. Ready queue is a linked list of process control blocks. They are the processes ready to run. Still, the priority must be known. Choosing the next process to run is achieved by scheduling. Scheduler is the component of kernel and it decides which process will be run. The scheduler is responsible for utilizing the system and executing multiple processes simultaneously. The reason of using a scheduler is to achieve multitasking.

On a single-processor system, there is no way to run many processes at the same time. OS provides a pseudo-parallelism to achieve this. CPU scheduling is basically changing the state of a process from ready to run. Starvation must be taken care of.

There is no way to know how long a process is going to run. Although, assumptions can be made by some counter values.

Note that kernel processes are kept in memory and this is why their time is short.

4.2.1.1. Original CPU Scheduler



Tree 1.1

The original CPU scheduler shares the CPU among processes using the information how many processes there are. In other words, the original scheduler is not interested in how many users or groups there are in the system, running processes.

The scheduler is implemented with one running queue for all available processors. At every scheduling, this queue is locked and every task on this queue get its time slice update. This scheduling algorithm have three scheduler policies; First-In First-Out (SCHED_FIFO), Round Robin (SCHED_RR), time-shared process (SCHED_OTHER).

4.2.1.2. Fair-share CPU Scheduler

The scheduler which is going to be implemented must care about how many users and groups there are instead of processes.

4.3. Procedure

In terminal, to see the processes who uses the CPU most, “top” command is used. (Note that for the system processes “0” is shown as the time. This means the processes are being done in microseconds or miliseconds.)

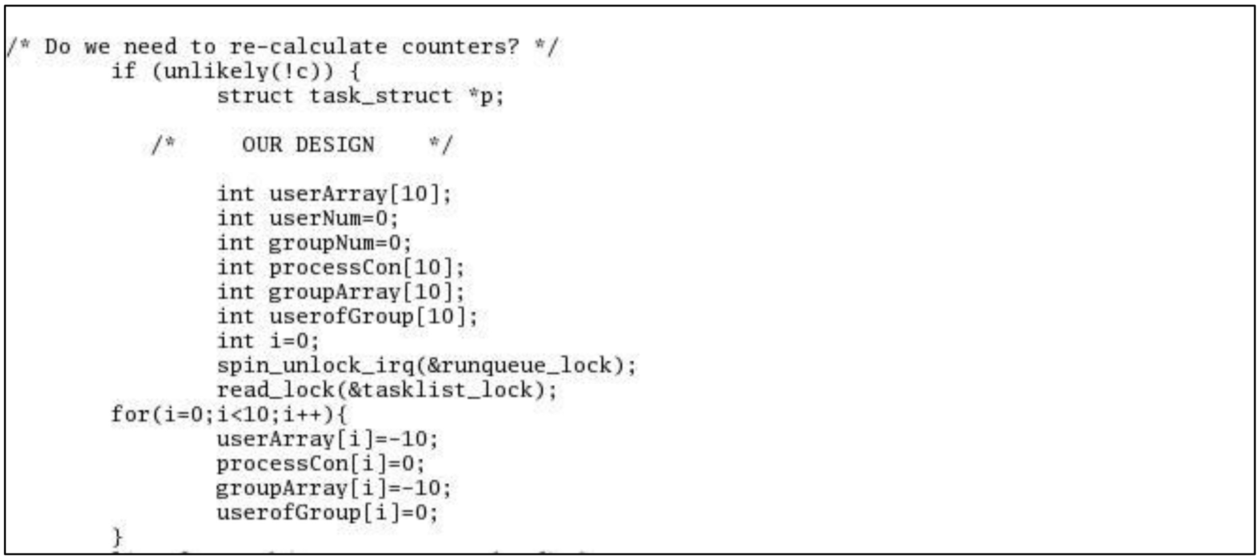
“Sched.c” and “sched.h” files are the most important files in this part of the project. Process states(running,waiting…) are kept in sched header.

List_for_each travels the process only in the run queue while for_each travels all of the processes.

When sched.c is viewed, the current scheduling algorithms can be seen. With an else statement, a new scheduler can be called with a system call.

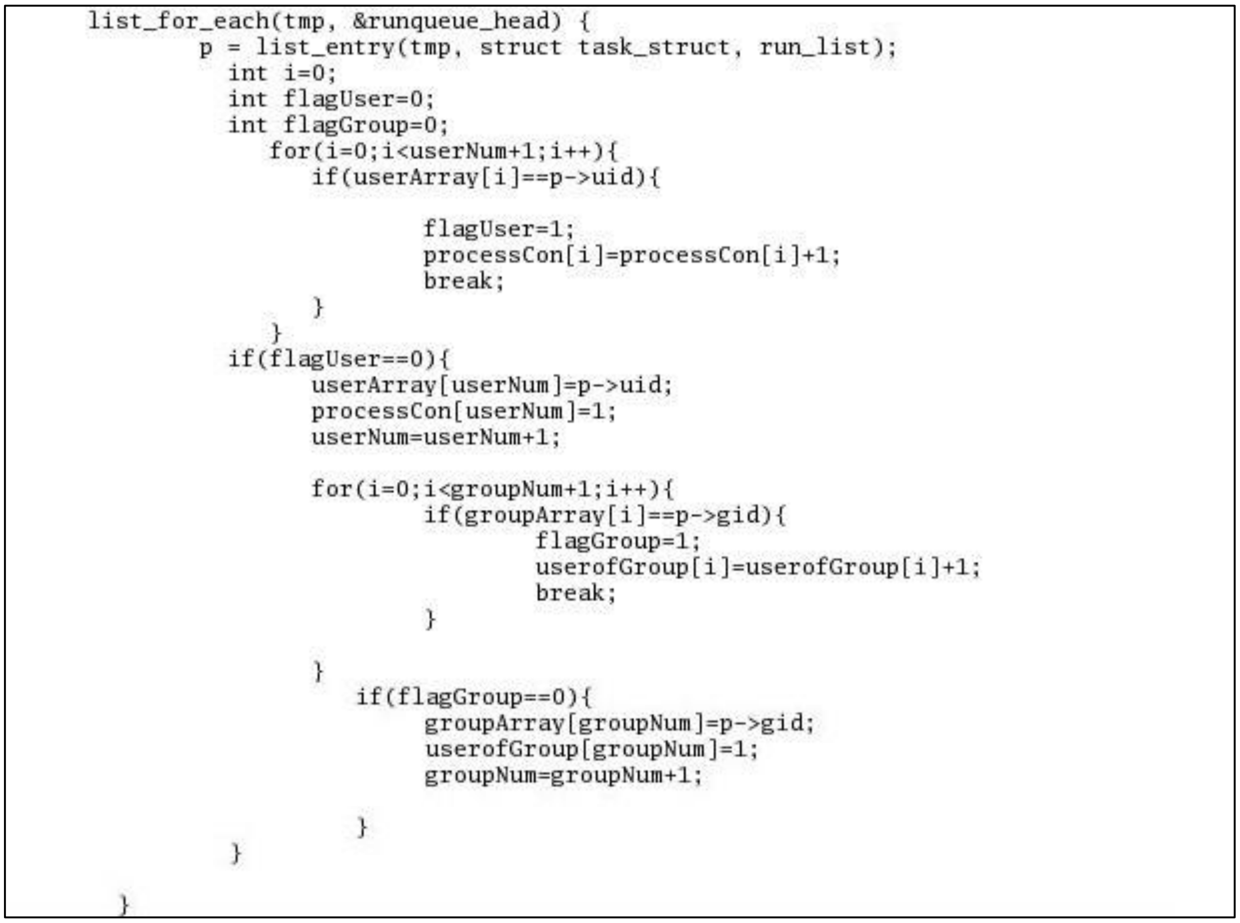
When counters reach 0, they must be reassigned. In this reassignation progress, they must be assigned to the numbers which will ensure the fairness.

All the work must be done thinking in a dynamic way, meaning that, the information that how many groups, users and processes are there must be known. If the number of how many groups and users are there is known, the counter values can be recalculated using this information. An array of users and the groups are needed.



```
/* Do we need to re-calculate counters? */
        if (unlikely(!c)) {
                struct task_struct *p;

           /*     OUR DESIGN    */

                int userArray[10];
                int userNum=0;
                int groupNum=0;
                int processCon[10];
                int groupArray[10];
                int userofGroup[10];
                int i=0;
                spin_unlock_irq(&runqueue_lock);
                read_lock(&tasklist_lock);
        for(i=0;i<10;i++){
                userArray[i]=-10;
                processCon[i]=0;
                groupArray[i]=-10;
                userofGroup[i]=0;
        }
```

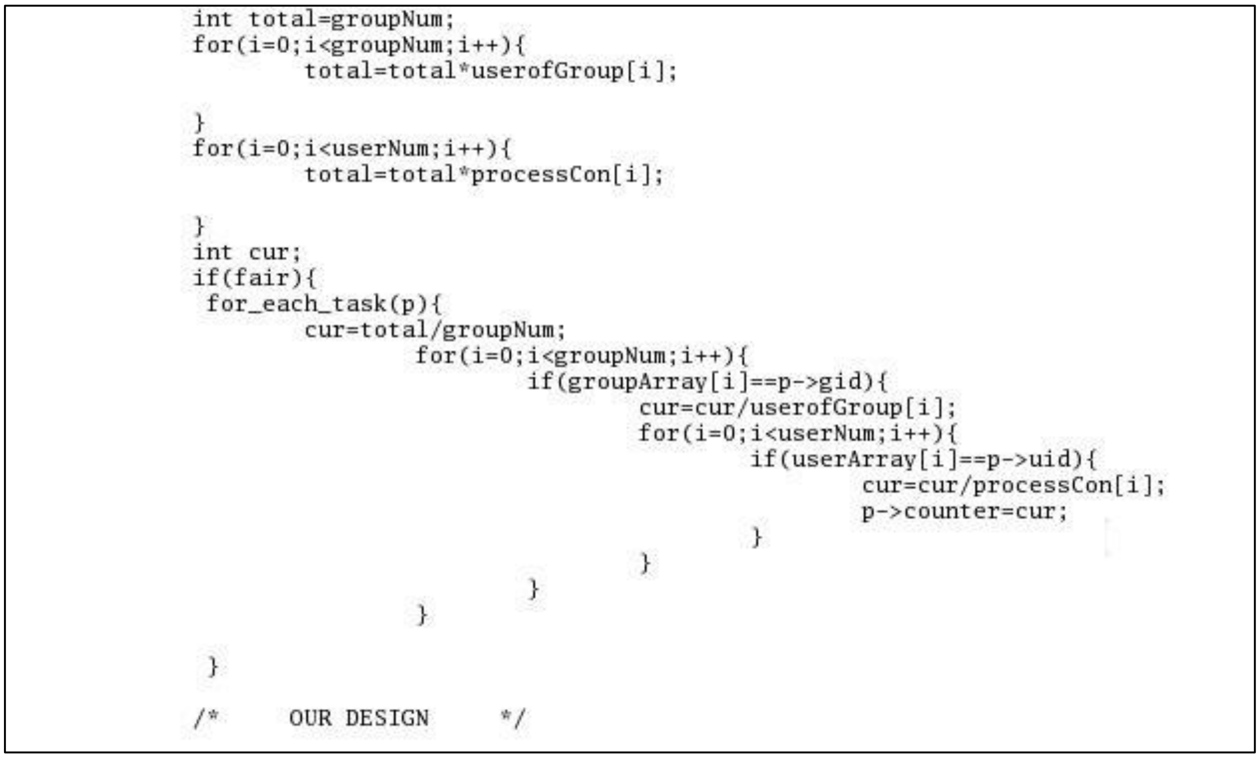
- “processCon” keeps the number of processes that the user have.
- “userofGroup” keeps the number of users that the group have.
- All are initialized.



```
list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
          int i=0;
          int flagUser=0;
          int flagGroup=0;
             for(i=0;i<userNum+1;i++){
                if(userArray[i]==p->uid){

                        flagUser=1;
                        processCon[i]=processCon[i]+1;
                        break;
                }
             }
          if(flagUser==0){
                userArray[userNum]=p->uid;
                processCon[userNum]=1;
                userNum=userNum+1;

                for(i=0;i<groupNum+1;i++){
                        if(groupArray[i]==p->gid){
                                flagGroup=1;
                                userofGroup[i]=userofGroup[i]+1;
                                break;
                        }

                }
                   if(flagGroup==0){
                        groupArray[groupNum]=p->gid;
                        userofGroup[groupNum]=1;
                        groupNum=groupNum+1;

                   }
          }

  }
```

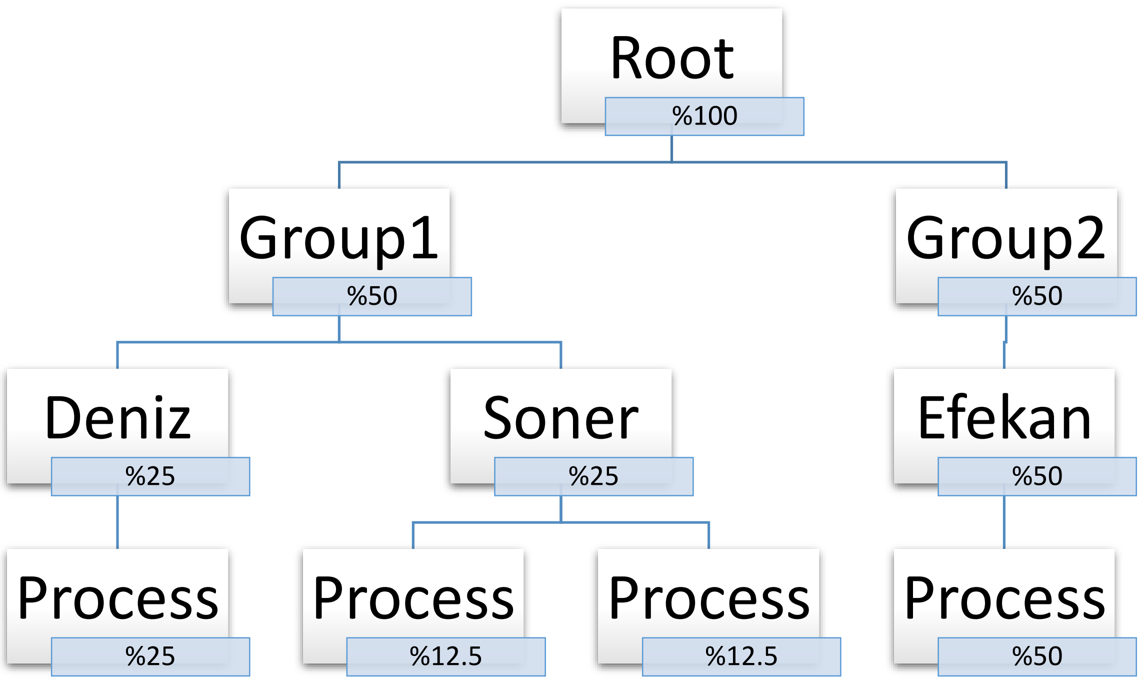
- The first process arrived will go into the if statement since there is no user yet.
- Its “uid” will be assigned as the first member of the userArray and the corresponding “processCon” will now be 1.
- “userNum” will be increased by 1.
- Since there is no group yet, “gid” of the process will be assigned as the first member of the groupArray and the corresponding “userofGroup” will now be 1.
- “groupNum” will be increased by 1.
- If the second process arrived belongs to the same user, it will be detected in the first for loop. “flagUser” will be set to 1. Corresponding processCon will be increased by 1.
- If not, its uid will be assigned as the second member of the userArray(“userNum” is now 2) and the corresponding “processCon” will now be 1.
- “userNum” will be increased by 1.

- Here, if the process belongs to the same group, it will be detected in this second for loop. “flagGroup” will be set to 1. Corresponding “userofGroup” will be increased by 1.
- If not, “gid” of the process will be assigned as the second member of the groupArray(groupNum is now 2) and the corresponding “processCon” will now be 1.
- “groupNum” will be increased by 1.



```
int total=groupNum;
for(i=0;i<groupNum;i++){
        total=total*userofGroup[i];

}
for(i=0;i<userNum;i++){
        total=total*processCon[i];

}
int cur;
if(fair){
 for_each_task(p){
        cur=total/groupNum;
                for(i=0;i<groupNum;i++){
                        if(groupArray[i]==p->gid){
                                cur=cur/userofGroup[i];
                                for(i=0;i<userNum;i++){
                                        if(userArray[i]==p->uid){
                                                cur=cur/processCon[i];
                                                p->counter=cur;
                                        }
                                }
                        }
                }

 }
/*     OUR DESIGN     */
```

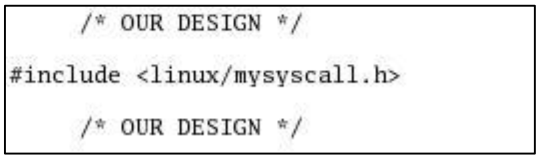
- “total” is initially set to groupNum.
- The logic behind the first for loop is to multiply total by the number of users that the current group(i) has.



Tree 2.1

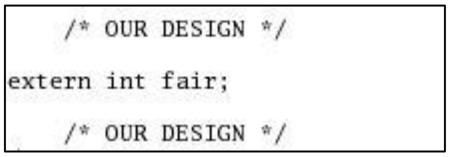
- When this tree is considered, total=2 initally. Then it will be multiplied by userOfGroup[0] which is 2.(group0 has 2 users) Then, new total will be multiplied by 1.(group1 has 1 user) So, the total will be 2*2*1=4 in the end.
- In the second for loop, since the "userNum" is 3, first, total will be equal to 4*the number of processes that the current user(deniz) has.
- 4*1 (deniz has 1 process)
- Then, 4*1*processCon[1] which is 2. (soner has 2 processes)
- Then, 4*1*2*processCon[2] which is 1. (efekan has 1 process)
- total=8
- By the system call, "fair" will be equal to 1 and the if statement will be entered.
- Next statements will be done for each process.
- For p1, cur will be equal to 8/2.
- Then, groupArray will be traced. p1 belongs to group0, so, cur will be equal to 4/userofGroup[0] which is equal to 2. (deniz and soner)
- Then, userArray will be traced. p1 belongs to deniz, so, cur will be equal to 2/processCon[0] which is equal to 1. (deniz has 1 process)

- Likewise, soner's p1 will have cur as 1 and p2 will have cur as 1.
- Efekan will have cur as 4.
- Simply, what is being done here is, for each process, first, divide the CPU by the number of groups.
- Then, find which group the current process belongs to and divide the CPU by the number of users that that group has.
- Then, find which user the current process belongs to and divide the CPU by the number of processes that that user has.
- Make the current process' counter value equal to cur.

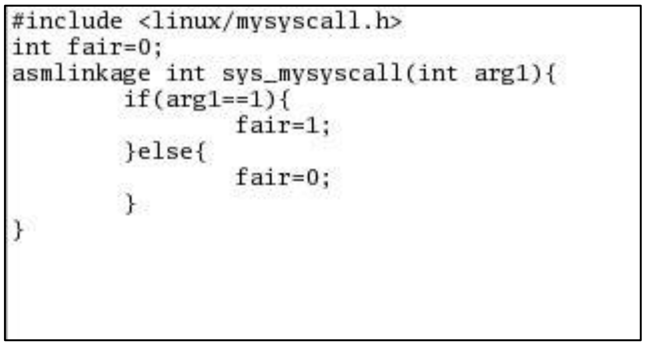


```
     /* OUR DESIGN */
#include <linux/mysyscall.h>
     /* OUR DESIGN */
```

- The header of the system call must be added



```
    /* OUR DESIGN */
extern int fair;
    /* OUR DESIGN */
```



```
#include <linux/mysyscall.h>
int fair=0;
asmlinkage int sys_mysyscall(int arg1){
        if(arg1==1){
                fair=1;
        }else{
                fair=0;
        }
}
```

- This is the system call written to change the current scheduler to fair scheduler.

Add a user:

```
/usr/sbin/useradd deniz
passwd deniz
```

Add a group:

```
/usr/sbin/groupadd group1
```

Add a user to a group:

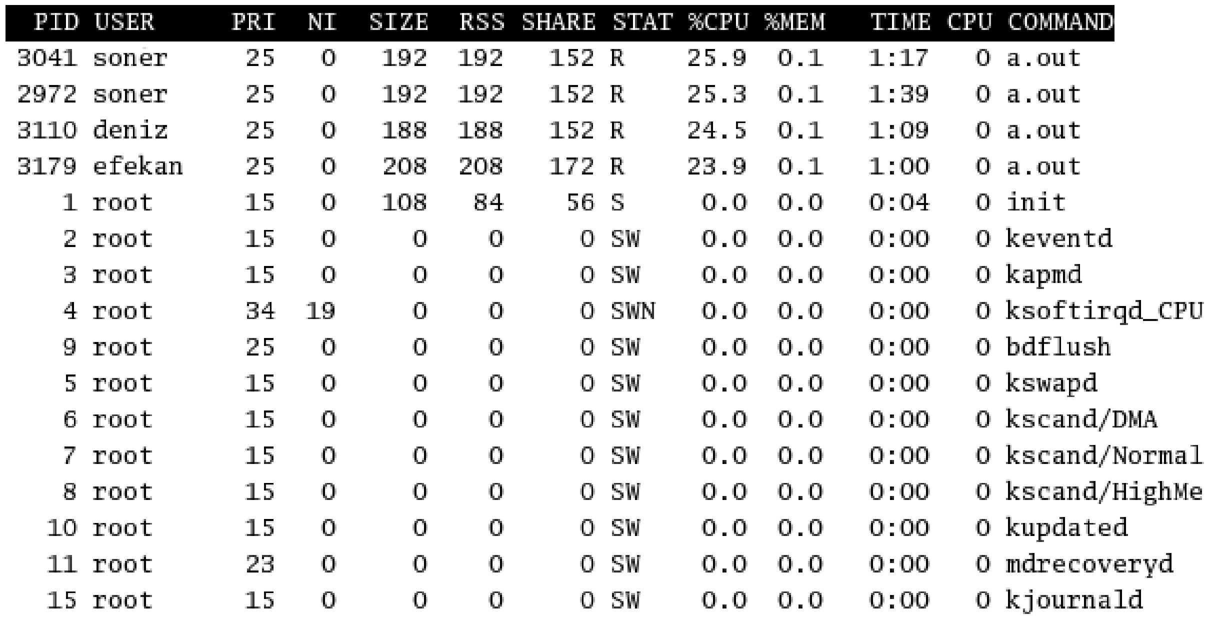
```
/usr/sbin/usermod -G group1 deniz
```

Go inside the user to run the process:

```
su - deniz
```

Note that process creation step cannot be made in the user-space, it must be made by root. Then, when running the process, go inside the user and run the process.
Deniz, Efekan and Soner users were made first. Then, two groups named Group1 and Group2 were made. Deniz and Soner were added to Group1 and Efekan was added to Group2. For user Deniz, a process was created by root. For user Soner, two processes were created. Lastly, for user Efekan, a process was created. Note that these processes are CPU-bound. (achieved by while(1) loop)

4.4. Tests



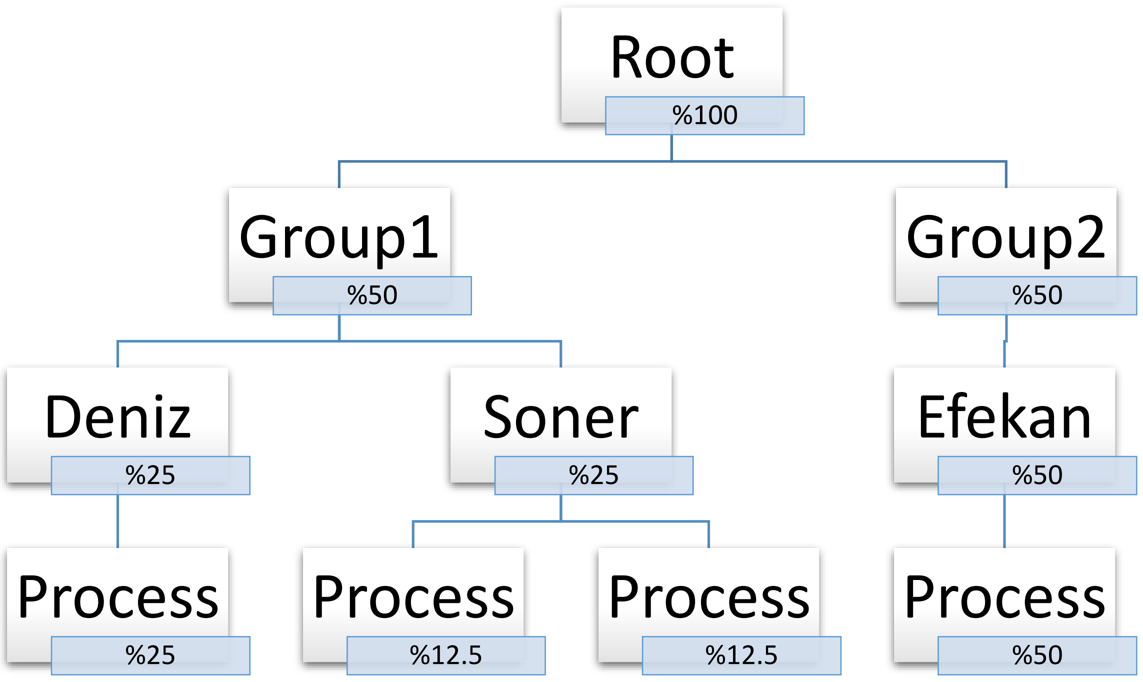
```
  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 3041 soner      25   0   192  192   152 R    25.9  0.1   1:17   0 a.out
 2972 soner      25   0   192  192   152 R    25.3  0.1   1:39   0 a.out
 3110 deniz      25   0   188  188   152 R    24.5  0.1   1:09   0 a.out
 3179 efekan     25   0   208  208   172 R    23.9  0.1   1:00   0 a.out
    1 root       15   0   108   84    56 S     0.0  0.0   0:04   0 init
    2 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kapmd
    4 root       34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd_CPU
    9 root       25   0     0    0     0 SW    0.0  0.0   0:00   0 bdflush
    5 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kswapd
    6 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kscand/DMA
    7 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kscand/Normal
    8 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kscand/HighMe
   10 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kupdated
   11 root       23   0     0    0     0 SW    0.0  0.0   0:00   0 mdrecoveryd
   15 root       15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
```

As it was mentioned before, the original scheduler only cares about the total number of processes, in this particular case it is 4.
It can be seen that CPU utilization for each process is %25 as it was expected.

Fair-share scheduler aims to share the CPU between processes due to the number of groups and users.

4.4.1. Case 1

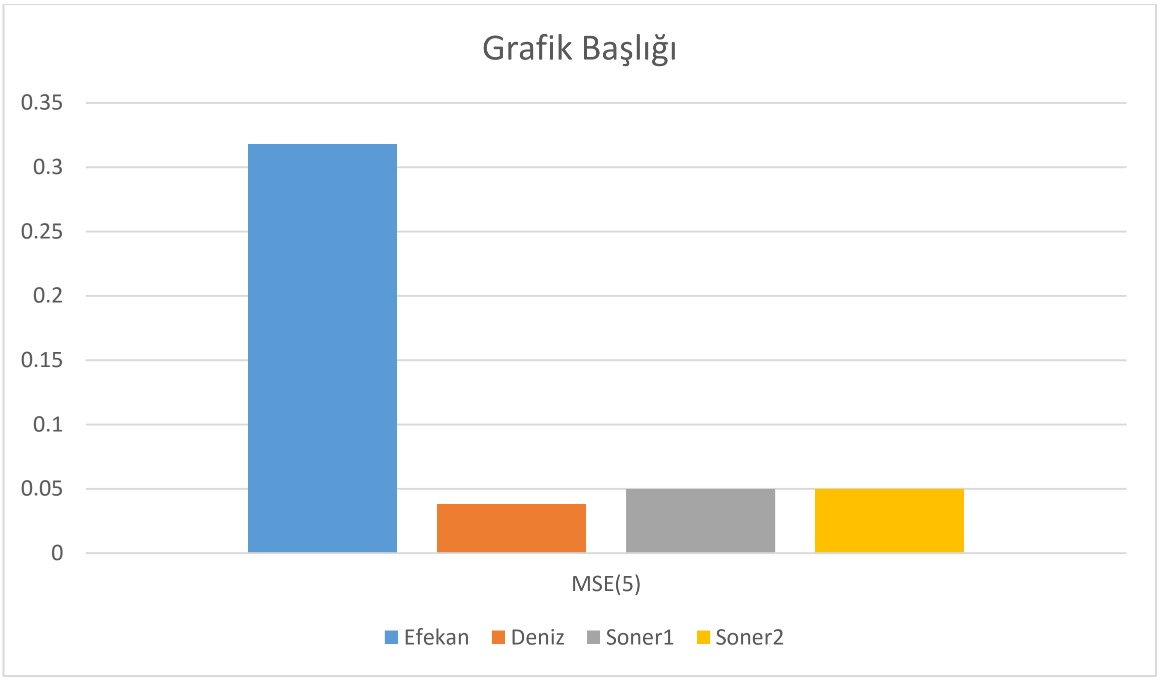
In this case there are 2 groups, “group1”(including “deniz” and “soner”) and “group2”(including “efekan”).
It was expected that each group will get ~%50 of the CPU,
in group1, there are 2 users so each will get ~%25 of the CPU,
user “soner” has 2 processes so each will get ~%12,5 of the CPU,
user “deniz” has only 1 process so it will get ~%25 of the CPU,
in group2, there is only 1 user so it will get ~%50 of the CPU,
user “efekan” has only 1 process so it will get ~%50 of the CPU.



Tree 3.1

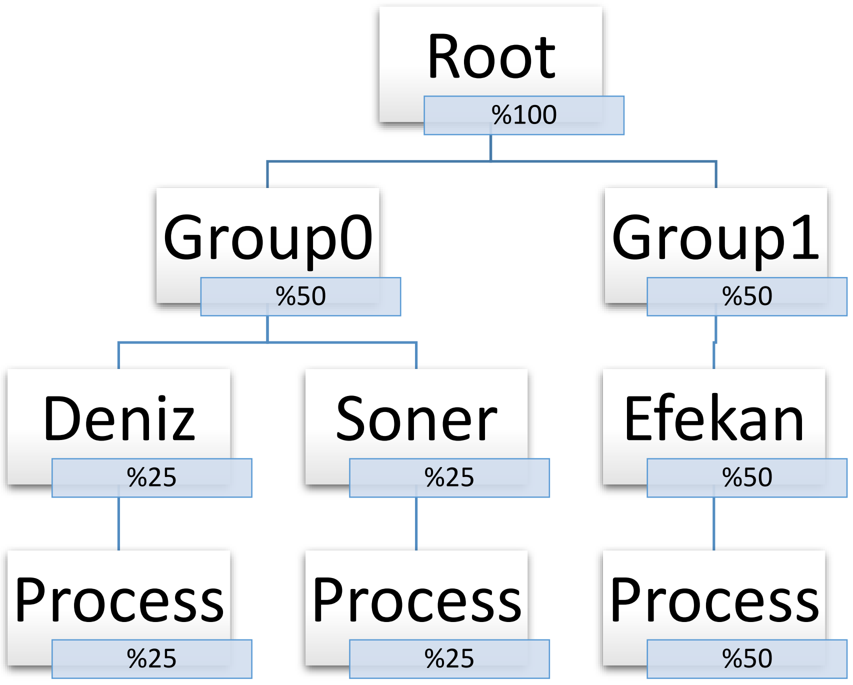
	Efekan	Deniz	Soner1	Soner2
t = 0	48.8	24.8	12.9	12.9
t = 2	50.2	24.8	12.4	12.4
t = 5	49.7	25.1	12.3	12.3
t = 7	49.9	24.7	12.5	12.5
t = 10	49.9	25.1	12.3	12.3
MSE(5)	0.318	0.038	0.05	0.05

Table 3.1



Graph 3.1

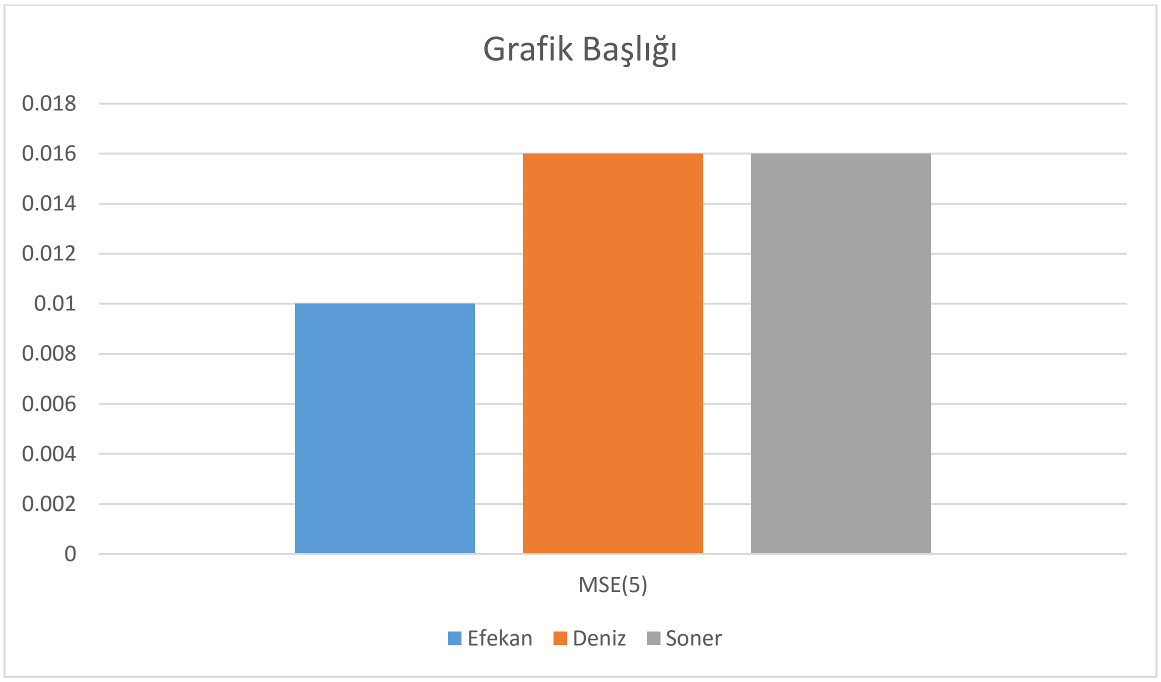
4.4.2. Case 2



Tree 3.2

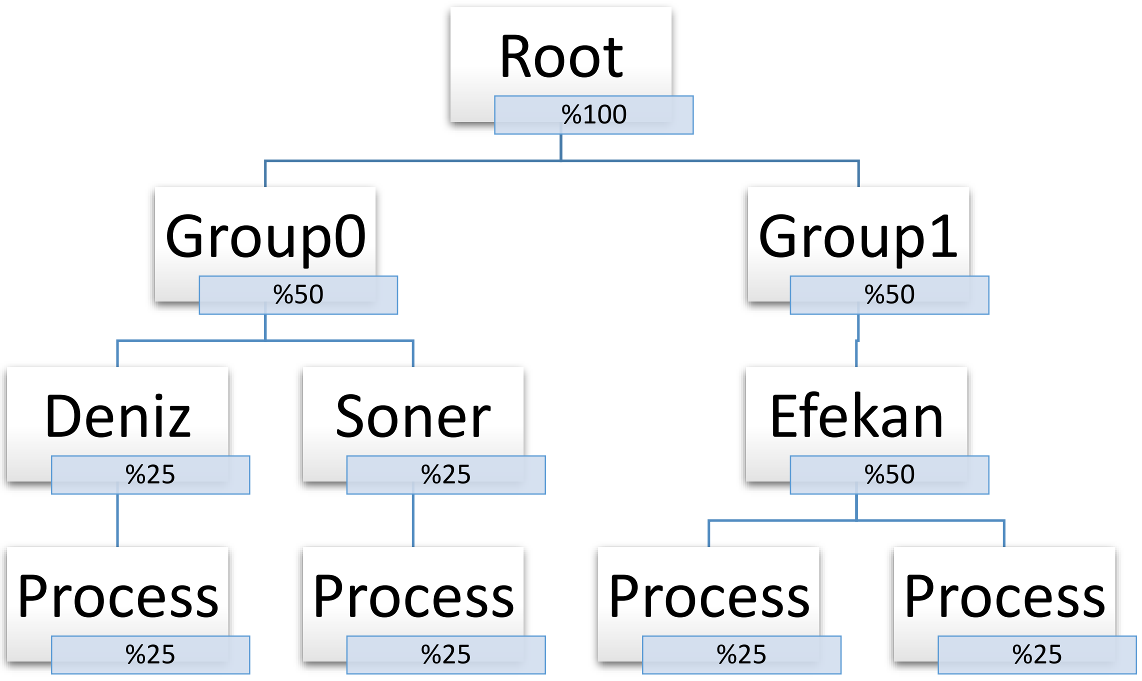
	Efekan	Deniz	Soner
t = 0	49.9	24.9	24.9
t = 2	49.9	24.9	24.9
t = 5	49.9	24.9	24.9
t = 7	49.9	24.9	24.9
t = 10	50.1	24.8	24.8
MSE(5)	0.01	0.016	0.016

Table 3.2



Graph 3.2

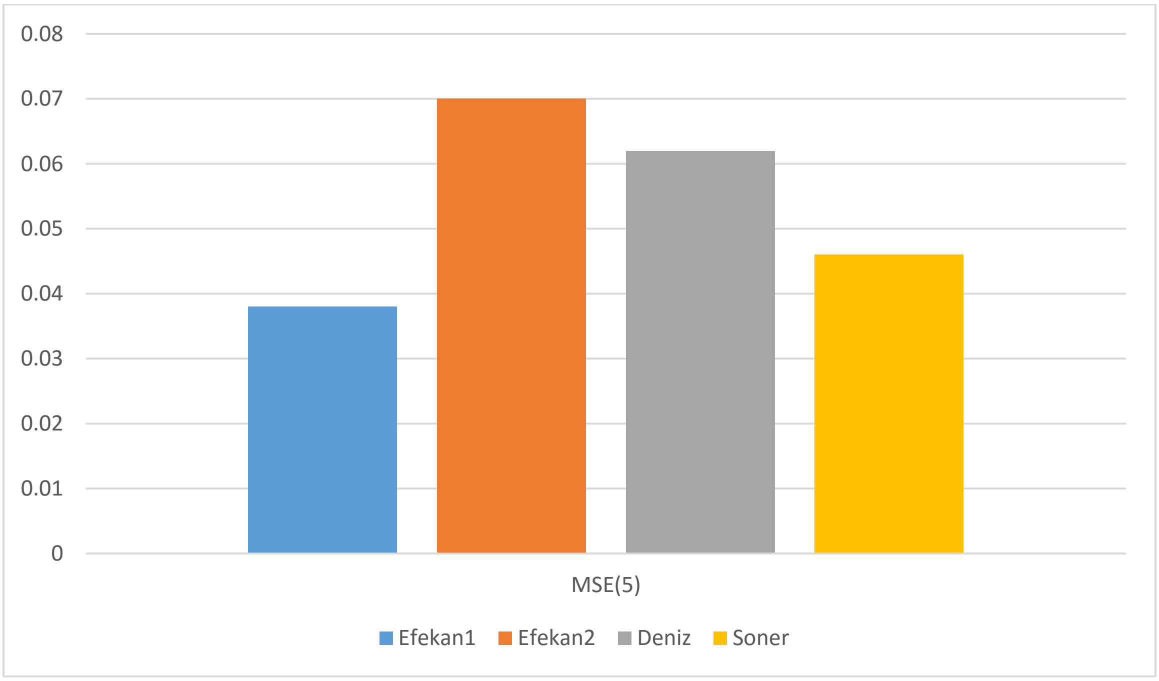
4.4.3. Case 3



Tree 3.3

	Efekan1	Efekan2	Deniz	Soner
t = 0	25.1	24.9	24.7	24.9
t = 1	24.7	24.5	24.7	24.7
t = 2	24.8	25.2	25.0	24.8
t = 10	25.1	25.1	24.7	24.7
t = 11	24.8	24.8	25.2	25.0
MSE(5)	0.038	0.07	0.062	0.046

Table 3.3



Graph 3.3

4.5. Conclusion

Implemented fair-share scheduler is better than the current scheduler considering users and groups in the system.