SYSTEM CALLS AND LOTTERY SCHEDULER

by

İlke Savaşoğlu

Tılsım Elif Usyılmaz

Ulaş Deniz Eraslan

CSE 331 Operating Systems Design

Term Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

Spring 2021

TABLE OF CONTENTS

1. INTRODUCTION

Preemptive Scheduling

When a process changes its state from running to ready or from waiting to ready preemptive scheduling happens.

At some time, when process is happening, CPU is dedicated to it for a limited time, after it stops, if that process still has some CPU burst time, the process gets back in the ready queue in order to execute.

Some algorithms that depend on preemptive scheduling are:

- Round Robin (RR): is a scheduling algorithm that limits process executions in same time slots.
- Shortest Remaining Time First (SRTF): is firstly selecting the process with the shortest amount of time remaining in order to execute until it is done or a new process that has a shorter remaining time adds in queue.

Non-Preemptive Scheduling

When a process changes its state from running to waiting or terminates, non-preemptive scheduling happens. When CPU dedicates its self to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can

allocate the CPU to another process.

Some algorithms that depend on non-preemptive scheduling are:

- Shortest Job First (SJF): is a type of scheduling policy which selects the process which has the shortest execution time to execute next.

Non-Preemptive Scheduling and Preemptive Scheduling

- First Come First Serve (FCFS): is a scheduling algorithm which chooses processes to execute depending on their time of arrival. It is the easiest and the simplest policy among all.

The main aim of this project is to understand the concepts of user-space and kernel-space while getting familiar with Linux and its functions. In addition learning and practicing system calls, process management and lottery scheduler are true purposes of this project.

Kernel space is a computer program which is at the core of a computer's operating system that has all the control of everything in the system. User-space is all of the code of the computer that runs outside of the kernel.

The project is divided into two phases:

Phase 1 : Adding System Call:

The purpose of this phase is adding a new system call in Linux with the given prototype and test it in the user space by taking the information about the current process and changing some of its properties.

Phase 2: Lottery Scheduler:

The purpose of this phase is creating 'Lottery' which is a scheduling mechanism that shares CPU among sets of processes equally according to the users that own those processes and observing the CPU utilization.

2. DESIGN and IMPLEMENTATION

2.1 PHASE I

2.1.1 Purpose

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. When we made a change with pre-defined system call in the Unix operating system, we observed the change with the system call.

2.1.2 Procedure

The system call definitions must be added to both kernel-space and user-space. “Copy_to_user” is used to copy the values of variables 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. We take the information of current process with get_current(). After we take information of current process , we take that values and copy them to struct that we called prcdata in kernel space. This new system call is 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.

```c
#include <linux/mysyscall.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    struct prcdata userdata;
    int check = mysyscall(&userdata);
    if (check == 0)
    {
        printf("printing counter: %d\n",userdata.counter);
        printf("printing nice: %d\n",userdata.nice);
        printf("printing prio: %d\n",userdata.prio);
        printf("printing weight: %d\n",userdata.weight);
        printf("printing pid: %d\n",userdata.pid);
        printf("getpid() : %d\n", getpid());
        printf("printing uid: %d\n",userdata.uid);
        printf("getuid() : %d\n", getuid());
        printf("printing nofprocess: %d\n",userdata.nofprocess);
        printf("Nice is %d\n",userdata.nice);
        int a;
        a = nice(3);
        mysyscall(&userdata);
        printf("New nice is %d\n",userdata.nice);
    }
    else
    {
        printf("Error");
    }
    return 0;
}
```

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

```c
#include <linux/mysyscall.h>
#include <linux/sched.h>
#include <asm/current.h>
#include <asm/uaccess.h>
asmlinkage int sys_mysyscall (struct prcdata *data){
    cli();
    struct prcdata kernelspacestructure;
    struct task_struct *x = get_current();
    copy_from_user(&kernelspacestructure,data,sizeof(struct prcdata));
    kernelspacestructure.counter = x->counter;
    kernelspacestructure.nice = x->nice;
    kernelspacestructure.prio = 20-kernelspacestructure.nice;
    kernelspacestructure.weight = 20-kernelspacestructure.nice+kernelspacestructure.counter;
    kernelspacestructure.pid = x->pid;
    kernelspacestructure.uid = x->uid;
    atomic_t a =x->user->processes;
    kernelspacestructure.nofprocess = atomic_read(&a);
    copy_to_user(data,&kernelspacestructure,sizeof(struct prcdata));
    sti();
    if (kernelspacestructure.counter >= 0 && kernelspacestructure.nice >= 0 && kernelspacestructure.prio >= 0 &&
    kernelspacestructure.weight >= 0 && kernelspacestructure.pid >= 0 && kernelspacestructure.uid >= 0 && kernelspacestructure.nofprocess >= 0
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```

(Kernel-space system call)

2.2 PHASE II

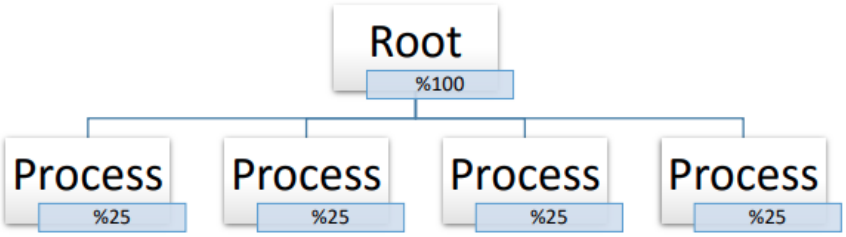
2.2.1 Purpose

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. Our goal is design a new scheduler called Lottery Scheduler and then implement that we designed scheduler.

2.1.2 Procedure

The original CPU scheduler shares the CPU among processes using the information how many processes there are. 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).



(Original CPU Scheduler)

Most important files for our scheduler are ‘sched.h’ , ‘sched.c’ , ‘fork.c’ and our new system call black . In sched.h , there is a struct called task_struct . This struct contains information about process such as current user id (uid) . We defined new variables that called flag and ticket in this struct . Ticket is used for choosing the next process . Flag is used for control check algorithm for users, in sched.c . In fork.c , we assign initial values for ticket and flag , ticket’s initial value is 4 and flag’s initial values is 0. In our new system call black , we define a global variable for changing scheduler behaviour . In sched.c , First thing that code does is checking variable globalvar , if globalvar is 0 default scheduler works . If globalvar is 1 , our new scheduler works . After if else statement , first of all Scheduler travels the runqueue from start to finish in order to arrange the tickets and calculate the total of tickets . In line 671 we get a random number between 1 to total of tickets. With that random number , in line 679 we subtract current process’ ticket until end of the runqueue . If we find a negative value we assign our next process is current process.After we choose the current process , we assign all process’ flag that has same next process’ uid to 1.With this algorithm we achieved user-based scheduler.

```
repeat_schedule:
    next = idle_task(this_cpu);
    c = -1000;
    if(globalvar==0){

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

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
    }
    }//ifbitis

```

```
635         else if(globalvar == 1 ){
636 
637         unsigned long random_number;
638         int x;
639         int totalTicket=0;
640 
641         int i;
642 
643         list_for_each(tmp, &runqueue_head) {
644             p = list_entry(tmp, struct task_struct, run_list);
645             if (can_schedule(p, this_cpu)) {
646 
647         unsigned long sleeep_time = (jiffies-(p->sleep_time))/HZ;
648 
649 
650         if(sleeep_time <= 40){
651             if(p->ticket==1){
652                 p->ticket=p->ticket;
653             }
654             else{
655                 p->ticket=p->ticket-1;
656             }
657          }
658         else if(sleeep_time >= 120){
659             if(p->ticket == 7){
660                 p->ticket=p->ticket;
661             }
662             else{
663                 p->ticket = p->ticket+1;
664             }
665          }
666          totalTicket+= p->ticket;
667         }//if(canschedule)bitis
668     }//list for each bitis
669 
```

```
670             //RANDOM NUMBER GENERATE
671             get_random_bytes(&random_number,sizeof(unsigned long));
672             random_number=(random_number%totalTicket)+1;
673             x=(int)random_number;
674
675
676             list_for_each(tmp, &runqueue_head) {
677             p = list_entry(tmp, struct task_struct, run_list);
678             if (can_schedule(p, this_cpu)) {
679             x = x - p->ticket;
680             if(x <= 0 && p->flag == 0)
681             {
682                 p->flag = 1;
683                 int currentuid = p->uid;
684                 list_for_each(tmp, &runqueue_head) {
685                 xa = list_entry(tmp, struct task_struct, run_list);
686                 if (can_schedule(xa, this_cpu)) {
687                     if (currentuid == xa->uid)
688                     {
689                         xa->flag = 1;
690                     }
691                     }
692                 }
693                 next=p;
694                 break;
695             }
696             globalcheck = 1;
697
698         }//if(canschedule)bitis
699     }//list for each bitis
700     if (globalcheck == 1)
701     {
702         list_for_each(tmp, &runqueue_head) {
703                         p = list_entry(tmp, struct task_struct, run_list);
704                         if (can_schedule(p, this_cpu)) {
705                             p->flag = 0;
706                         }
707             }
708             globalcheck = 0;
709     }
710     }
711     //else if (globalvar==1) bitis
```

3. TESTS and RESULTS

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	CPU	COMMAND
1598	secondus	25	0	220	220	184	R	33.9	0.0	0:27	0	a.out
1604	cse331pr	25	0	220	220	184	R	33.7	0.0	0:10	0	a.out
1633	cse331pr	25	0	224	224	184	R	32.1	0.0	0:05	0	a.out

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	CPU	COMMAND
1661	secondus	25	0	224	224	184	R	19.9	0.0	0:15	0	a.out
1662	secondus	25	0	220	220	184	R	19.9	0.0	0:12	0	a.out
1663	cse331pr	25	0	220	220	184	R	19.9	0.0	0:09	0	a.out
1666	cse331pr	25	0	220	220	184	R	19.9	0.0	0:07	0	a.out
1664	cse331pr	25	0	220	220	184	R	19.5	0.0	0:08	0	a.out

The original scheduler only cares about the total number of processes

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	CPU	COMMAND
1483	secondus	20	0	220	220	184	R	51.1	0.0	1:28	0	a.out
1355	cse331pr	20	0	280	280	236	R	25.3	0.1	1:29	0	a.out
1440	cse331pr	20	0	280	280	236	R	23.4	0.1	0:57	0	a.out

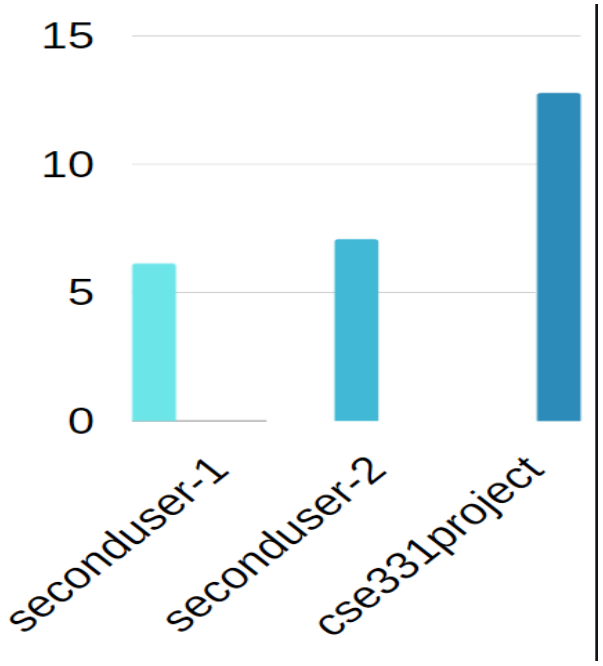
PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	CPU	COMMAND
1483	secondus	20	0	220	220	184	R	50.2	0.0	2:24	0	a.out
1355	cse331pr	20	0	280	280	236	R	24.7	0.1	1:59	0	a.out
1440	cse331pr	20	0	280	280	236	R	24.7	0.1	1:28	0	a.out

1584	secondus	20	0	220	220	184	R	30.8	0.0	6:16	0	a.out
1585	secondus	20	0	220	220	184	R	19.6	0.0	4:15	0	a.out
1582	cse331pr	20	0	280	280	236	R	18.6	0.1	3:44	0	a.out
1581	cse331pr	20	0	280	280	236	R	17.7	0.1	3:40	0	a.out
1583	cse331pr	20	0	280	280	236	R	13.0	0.1	3:43	0	a.out

Lottery scheduler aims to share the CPU equally to every user .

CASE 1 – Default scheduler with 2 user 3 processes:

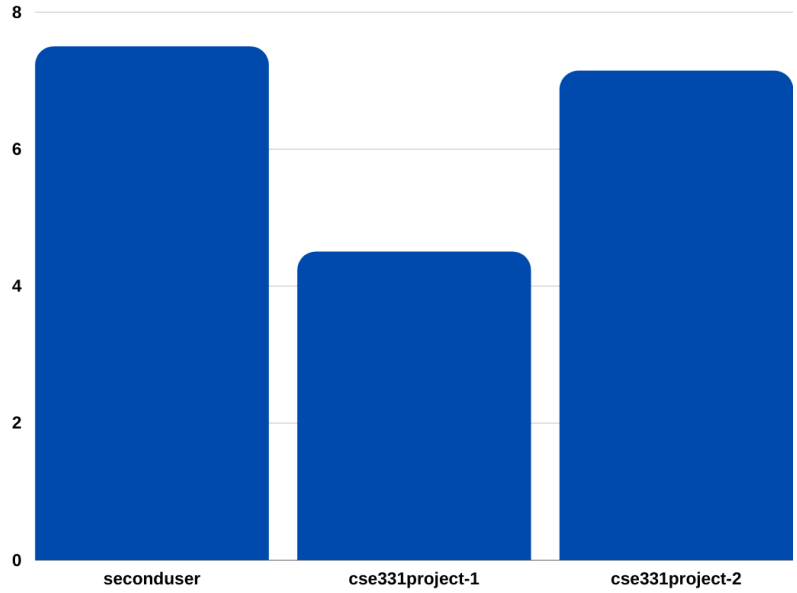
	Seconduser-1	Seconduser-2	Cse331project
t=4:21	34.8	34.8	29.8
t=4:33	29.7	33.7	35.7
t=14:44	33.5	36.5	29.6
t=15:12	36.4	29.5	33.5
t=18:56	29.7	30.7	38.7
t=20:11	36.5	33.5	29.6
t=21:31	29.6	33.6	36.5
t=23:29	32.5	34.5	29.6
t=24:55	32.4	29.5	37.4
t=26:25	32.6	37.5	29.6



(MSE(10) for each process)

CASE 2 – Lottery scheduler with 2 user 3 processes:

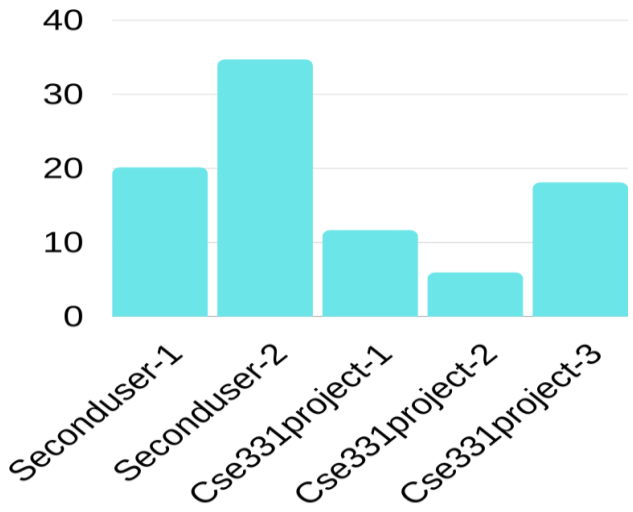
	Seconduser	Cse331project-1	Cse331project-2
t = 2.12	49.5	26.6	22.9
t = 2.15	49.1	31.8	18.3
t = 2.18	47.3	27.3	24.5
t = 2.22	51.4	28.6	19.1
t = 3.40	46.4	29.3	22.7
t = 30.46	48.5	28.9	21.7
t = 30.50	50.2	27.9	18.8
t = 31.04	49.6	27	20.7
t = 60.13	45.4	27.2	26.4
t = 60.16	51.9	26.3	20.9



(MSE(10) for each process)

CASE 3 – Lotter scheduler with 2 user 5 processes:

	Seconduser-1	Seconduser-2	Cse331project-1	Cse331project-2	Cse331project-3
t=0:25	25.7	19.8	12.8	12.8	27.7
t=0:27	32.3	18.6	21.5	13.7	13.7
t=4:50	30.1	15.3	13	19.3	22.1
t=4:53	31.3	19.8	18.5	14.6	16.6
t=14:09	27	18.6	16.5	16.5	17.2
t=21:04	24.5	23.7	16.6	15.8	19.3
t=23:30	29	17.8	18.8	18.8	15.3
t=29:06	30.9	16.6	21.6	13.6	17.3
t=60:26	30.1	21.4	14.3	19.8	14.3
t=60:40	24.3	19.2	21.7	16	18.5



(MSE(10) for each process)

4. CONCLUSION

Implemented lottery scheduler is better than the current scheduler considering users in the system.

REFERENCES

- The Linux Process Manager: The internals of scheduling, interrupts and signals.
- Understanding the Linux Kernel.
- Lecture notes.