LOTTERY SCHEDULING

by
Necip Fazıl MERCİMEK
Mert UZGÜL
Murat Arda ÖNSÜ

CSE 331 Operating Systems Design
Term Project Report

Yeditepe University
Faculty of Engineering
Department of Computer Engineering
Fall 2019

ABSTRACT

With multiprogramming and multithreading come to computer world scheduling become very significant part of operating system. Computer's perform rely on mostly scheduling algorithm that decide which process or thread execute or wait. This paper we are going to analyze computer performance with different scheduling algorithm and compare their performance.

TABLE OF CONTENTS

1.INTRODUCTION

In this project we are going to change our standard scheduler policy into lottery scheduling policy. we used linux 2.4.20 kernel old version since it is easy to implement and change compare to new linux versions. The scheduler is the component of the kernel that selects which process to run next. The scheduler is the basis of a multitasking operating system. By deciding what process can run, the scheduler is responsible for best utilizing the system and giving the impression that multiple processes are simultaneously executing.

The policy is the behaviour of the scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time. Therefore, it is very important. we have two different scheduling policies: standard linux and lotter scheduling. we are going to compare their performance.
Fork system call is used for creating a new process and values are given for each process. if we want to test linux scheduling policy we need to create a process and observe their performance.linux kernel fork is done by <linux/fork.c>

Lottery Scheduling is a type of process scheduling, somewhat different from other schedulings. Each process initializes with ticket number value 5. Each process may hold a maximum of 9 tickets and minimum of 1 ticket. when the scheduler must choose the process, it draws a random number and selects a lucky process that has the ticket with appropriate this number. Larger tickets increase the likelihood of selection of process.
The kernel stores the list of processes in a circular doubly linked list called the *task list3*. Each element in the task list is a *process descriptor* of the type struct task_struct, which is defined in <linux/sched.h>. The task structure contains all the information about a specific process. We want to change to linux scheduling policy into lottery scheduling policy we need to create ticket number integer variable in task_struct and initialize it 5 when process instantly created, in fork.c.

We don’t want to erase standard linux scheduling policy, we just want to compare its performance with lottery scheduling. That’s why we create system call with policy variables to determine which policy will be used. Then we make a change on <sched.c>. the policy function is written in repeat_scheduler. we use condition between standard scheduling policy and lottery scheduling policy that we wrote under it, by using policy variables which we determine in a system call.

To test their performance first we create 10 .c files. inside this c file, we write infinite while loop. then run them separately by returning their process address (a.out &). then use the ‘top’ command to take their performance and write it into the txt file. After that, we call system call and change scheduling policy to lottery scheduling then do the same procedure for this.

2.DESIGN and IMPLEMENTATION

We need to implement another scheduling policy named lottery scheduling. In this scheduling policy, every process has its own ticket number, unlike standard scheduling policy. That's why we initialize ticket_number as **tn** integer variable and initialize **cpustart** unsigned long variable which shows us to process start time inside <sched.h> file. We initialize cpustart because ticket number is rely on time spent on cpu.

```
421 int tn;
422 unsigned long cpustart;
```

When a process is created it is started with a default ticket number which is 5 and default cpustart which is jiffies*10.We multiplied jiffies by 10 because jiffies is 10 ms and we want our calculation by 0.1 second. Every process is created and take their value by fork function that's why we give their ticket_number value inside of <fork.c> file.

```
665 p->tn=5;
666 p->cpustart=jiffies*10;
```

After we create a ticket number and give its initial value, we go to change <sched.c> file in where all scheduler functions are defined. Scheduling policy is defined under **repeat_scheduler** in line 599**.** We didn't erase standard policy instead we use condition variable name **opsyspolisi** with an integer. We created system call which opsyspolisi is defined and can be changed. Then we called opsyspolisi in sched.c in line 38.

```
38 extern int opsyspolisi;
```

If opsyspolisi is 1 it means we are going to use standard policy function, if it is 2 it means we turn our scheduling policy to lottery policy.

Under repeat_scheduler we create 5 integer value named searchbit, rndmbit, randomtn, ttn, and flag. **ttn** called as 'total ticket numbers' and it first initializes value 0. To find ttn we sum all running processes ticket number by the function **list_for_each**. This function helps us to search for every running process. Inside list_for_each function we used **list_entry** function to take a process and use **can_schedule** function to determine if it can be scheduled. If yes add this process ticket number by total ticket number variable ttn and repeat this procedure for every process inside list_for_each.

```
640     list_for_each(tmp, &runqueue_head)
641     {
642             p = list_entry(tmp, struct task_struct, run_list);
643             if (can_schedule(p, this_cpu))
644             {
645                     ttn+=p->tn;
646             }
647     }
```

rndmbit called as 'random bit', we used this to take random value for ticket. **get_random_bytes** function helps us to take random value inside kernel.

We use two parameters inside get_random_bytes:variable which random value is to write inside and size of the random variable. our variable is & rndmbit and our size is sizeof(unsigned int) since rndmbit is initialize as type of integer.

```
650     get_random_bytes(&rndmbit,sizeof(unsigned int));
```

randomtn called as 'random_ticket_number' to find randomtn we created do-while loop with condition 'randomtn<0'. We did this because we wanted to avoid negative number for ticket number. Inside loop we created condition if ttn is equal to zero then return null. After condition we get mod randombit by ttn and replace result to randontn. We did because we wanted to prevent our ticket number exceed our total ticket number.

```
648     do
649             {
650             get_random_bytes(&rndmbit,sizeof(unsigned int));
651             if(ttn==0)
652             {
653                     return NULL;
654             }
655             randomtn=rndmbit % ttn;
656     }while(randomtn<0);
```

Our algorithm to find the process that by randomtn is first we initialize **searchbit** with value 1. Then we start to traverse all running process by list_for_each function. We take every process ticket number and sum it by searchbit. When searchbit exceed or equal randomtn we understand that process to be selected is found and loop must be stopped. Then we use break command to exit the loop.

```
657     list_for_each(tmp, &runqueue_head)
658             {
659                     p = list_entry(tmp, struct task_struct, run_list);
660                     if (can_schedule(p, this_cpu) && flag==0)
661                     {
662                             searchbit+=p->tn;
663                             if(searchbit>=randomtn)
664                             {
665                                     next=p;
666                                     break;
667                             }
668                     }
669             }
```

As we mentioned that at the beginning process ticket number relies on time that process spent on time inside cpu. If the process elapses 20 ms it loses 1 ticket. And if process elapses 200 ms it gains 1 ticket. each process can have a maximum of 9 tickets and a minimum of 1

ticket. Time can be determined in kernel by **jiffies** variable. We use prev process jiffies and subtract it's cpu start time to decide if process will have gain or loss a ticket.

```
671     if((jiffies*10)-prev->cpustart<20)
672     {
673             if(prev->tn>1)
674             {
675                     prev->tn=prev->tn-1;
676                     prev->cpustart=0;
677             }
678     }
679     else if((jiffies*10)-prev->cpustart>200)
680     {
681             if(prev->tn<9)
682             {
683                     prev->tn=prev->tn+1;
684                     prev->cpustart=0;
685             }
686     }
```

3.TESTS and RESULTS

After we succesfully implemented lottery scheduler we compiled kernel and reboot the system and executed **My Opsys Kernel** which we created. Then we started creating infinite loop processes to compare default scheduler and lottery scheduler. We took 100 samples for each process #2 - #10 to compare schedulers by using this command below

top -n 100 >> name_of_file.txt

After taking samples we write an awk code to calculate Mean Square error(MSE) which subtract every real value by expected value and take square every one of them , then divided their summation by number of sample.

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

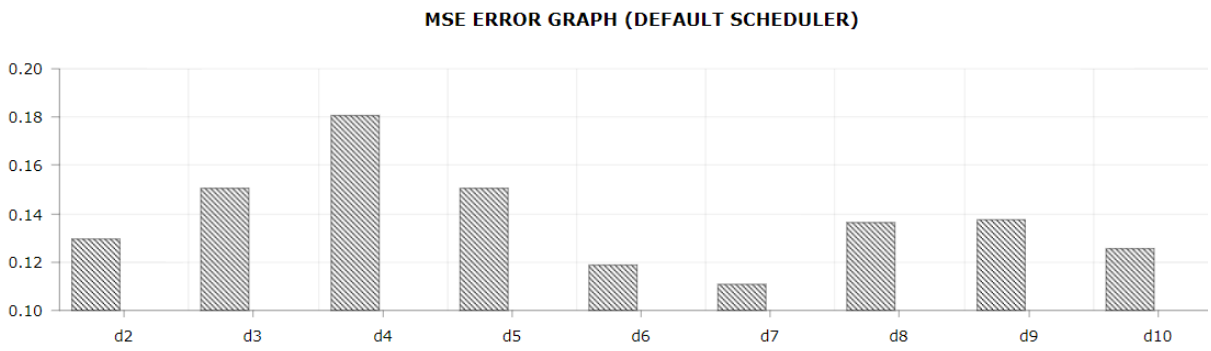
Expected Value for CPU Utilization of each process is calculated with: (values of ^Y).

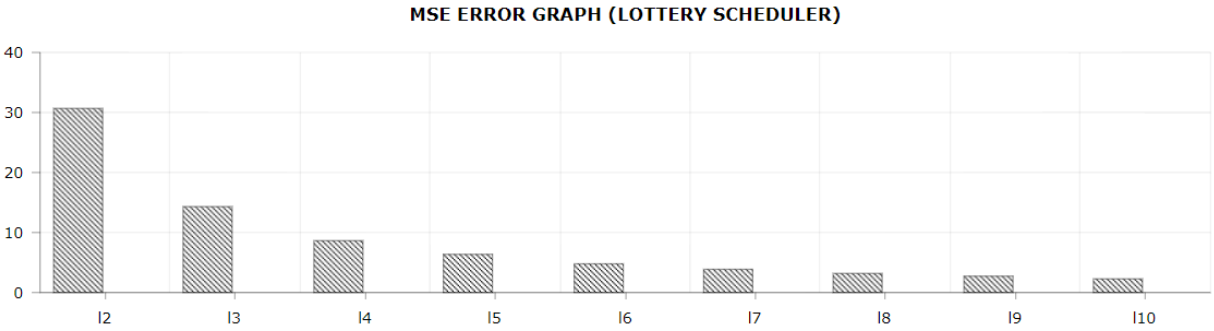
100 / NUMBER_OF_PROCESSES (%)

Because we wanted to expect every processes share same amount of cpu resource. Awk code which we use to calculate MSE is shown below.

```
BEGIN{count=0; expected=100/n; sum=0;flag=0}
{

if (substr($13,1,1) != "P") {flag=0}
if(flag) {
        sum = sum+($9-expected)*($9-expected)
        count = count+1

        #print "******************************"
        #print "$9:",$9
        #print "sum:",sum
        #print "count",count
        #print "******************************"
}
if($9=="STAT") {flag=1}

}
END{print("mse:",sum/count)}
```

After we found MSE for default and lottery scheduler we chart graph for both of them. The graphs are shown below;





MSE result

(for default scheduler)	(for lottery scheduler)
d2:0.1293	l2:30.7785
d3:0.150489	l3:14.3643
d4:0.1807	l4:8.65123
d5:0.15034	l5:6.44216
d6:0.118789	l6:4.70716
d7:0.110976	l7:3.84205
d8:0.13645	l8:3.09194
d9:0.137257	l9:2.63841
d10:0.12532	l10:2.27676

4. CONCLUSION

In conclusion we saw default scheduler executes with less mean squared error.
We already expect to see less error because of behaviour of default scheduler it almost shares cpu utilization same amount for each processes.

Changing number of processes in default scheduler does not change MSE significantly. However in lottery scheduling we encountered with more mean squared error because of the lottery scheduler's behaviour. It shares cpu utilization with a lottery it means resource sharing of each process rely on mostly luck so that it is normal to see bigger error rates. Additionaly when number of process are increasing we encountered with less error rate and continuously decreasing graph because if number of processes increases, the rate of being lucky decreases.

We can also expected to see one process uses significantly large amount of resource while others share very poor in lottery scheduling because as we mention before sharing resource for each process is rely on luck unlike default scheduler.

REFERENCES

https://www2.cs.arizona.edu/~tpatki/lottery.pdf

https://www.geeksforgeeks.org/lottery-process-scheduling-in-operating-system/

https://www.wikizeroo.org/wiki/en/Lottery_scheduling

https://live.amcharts.com/new/edit/

https://www.kernel.org/doc/html/latest/scheduler/index.html

http://fxr.watson.org/fxr/source/?v=linux-2.4.22

https://www.geeksforgeeks.org/introduction-of-system-call/

https://www.kernel.org/doc/html/v4.10/process/adding-syscalls.html