**GTICKET SCHEDULING**

**by**

**Yusuf Emirhan Şahin**

**Mehmethan Turan**

**Rıdvan Batuhan Arkan**

**CSE 331 Operating Systems Design**

**Term Project Report**

**Yeditepe University**

**Faculty of Engineering**

**Department of Computer Engineering**

**Spring 2020**

**ABSTRACT**

This paper presents GTicket scheduling, a randomized scheduling algorithm. With multiprogramming and multithreading becoming a norm in the computer world, process scheduling became a very significant part of the modern operating systems. A computers perform mostly relies in its scheduling algorithm. For that we need a fair scheduler. But with GTicket, instead of processes being an important factor of a scheduler, we instead opted for a user group perspective. GTicket tries to be fair to the operating systems user groups, not its processes. We are going to discuss, test and analyze our performance metrics in the report.

**TABLE OF CONTENTS**

[**INTRODUCTION**](#_gjdgxs)4

[**DESIGN and IMPLEMENTATION**](#_30j0zll)6

[**TESTS and RESULTS**](#_2jxsxqh)10

[**CONCLUSION**](#_qsh70q)12

[**REFERENCES**](#_32hioqz)13

# INTRODUCTION

In the world of operating systems there are lots of scheduler options that we can choose. So before we start to talk about GTicket, our scheduler that we implemented within the Linux Kernel 2.4, we want to inform the reader with other scheduler algorithms. First of all we want to talk about Lottery Scheduler and MLFQ.

Lottery scheduler is a scheduling algorithm that tries to fair share the system resources by giving every process a ticket value. Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets doesn’t need to be linear, granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as Shortest job next and Fair-share scheduling.

MLFQ, or multi-level feedback queue is also an another scheduling algorithm that’s been used. It was developed by Fernando J. Corbató in 1962. In MLFQ, the scheduler has multiple queues to put processes. When the first process comes, scheduler will put it in the first queue. If the process is completed within the time quantum of the given queue, it leaves the system. If not, the scheduler will insert it at the end of the lower queue. This continues until the process finishes, or the process hits the last process queue. For scheduling, the scheduler will always start to pick processes from higher to lower level queues.

In this project our group will change our normal scheduling policy in Linux Kernel 2.4.20 to the GTicket scheduling policy. We used Linux Kernel 2.4.20 since its ease of understanding and implementation of our scheduling policy. The scheduler is the part of an operating system that governs the process queue, which determines what to run next. 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.

GTicket scheduling is a type of process scheduling that runs quite differently from native Linux scheduler. In the native scheduler, the scheduler governs processes fairly. In our scheduler we take a different look at our fairness, and instead of running processes fairly, we give our user groups a fair way of schedule policy. For our team, each process starts with 8 tickets. Each process can hold a minimum of 1 ticket and maximum of 15 tickets. When the scheduler selects the next process what it does is first gets the maximum ticket count of the processes, and after taking the maximum value, it selects a random value between 1 to maximum ticket value. After all of that we finally select the next process by selecting a process in line that has a bigger value of tickets than our random value.

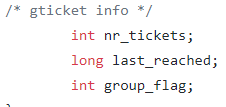
The kernel stores the list of processes in a circular doubly linked list called the task\_list. Each element in the task list is a process descriptor of the struct task\_struct, which is defined in <linux/sched.h>. The task structure contains all the information about a specific process. We want to change the linux scheduling policy into lottery scheduling policy. For doing that we need to create a ticket number integer variable in task\_struct and initialize it 8 when the process instantly created, in fork.c.

We don’t want to delete the native scheduling policy, because we want to compare it to GTicket and get to a conclusion. We created a system call named pteamt and initialized a value named gticket\_policy in it to switch between the scheduler policies.

# DESIGN and IMPLEMENTATION

* 1. **Changes in <sched.h>**

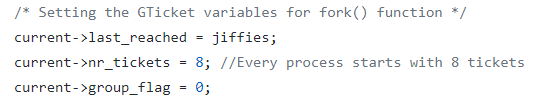
In sched.h we need to declare all the variables that we are going to use in our scheduler. Because in our scheduling policy, we give every process a ticket, unlike the native scheduler. That’s why we initialize our ticket number as nr\_tickets. After that we also declare the last\_reached and group\_flag variables. We’ll talk about group\_flag in detail at 2.3 of our report.



The reason why we declare our last\_reached variable is that we use jiffies in our scheduler, which has the type long.

* 1. **Changes in <fork.c>**

In fork.c we need to add few variables in order to use our scheduler policy. Fork.c initiates all the important variables for us. When a new process is created, we define the number of tickets that it has as 8. We also define the last\_reached variable as jiffies. And at last we declare the group\_flag as 0.



The reason why we set numbers in the pointer current is that when we fork a program, we call the do\_fork function. Calling that, our forked process becomes our current process. We are also going to use task\_struct in sched.c, because all processes do contain task\_struct.

* 1. **Changes in <sched.c>**

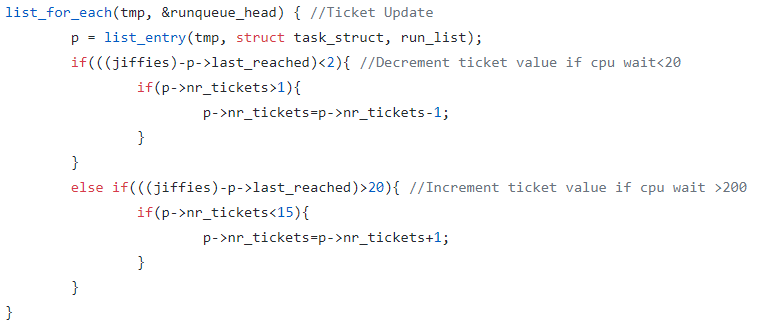
After we design and implement what we did on 2.1 and 2.2 we started to alter our standard scheduler. First of all, we need to add a flag to choose which scheduler to work, so we declare our flag variable, so we declare it, and check it on repeat\_scheduler.



C:\Users\Yusuf\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Ek Açıklama 2020-05-20 041623.png

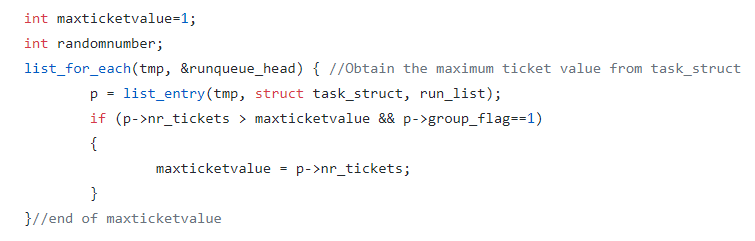
If gticket\_policy is 0, we’ll use the standard schedule policy, and if it’s 1 we’ll use GTicket schedule policy.

After we switch to GTicket scheduler, what we do first is that we update our ticket values.



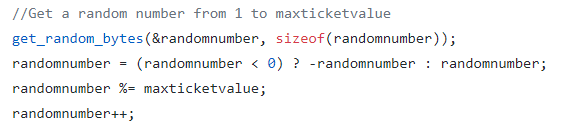
In this part of our code we want to do a loop in our run\_list and look at every processes last\_reached value. If the process were to be wait less than 20 ms and has more than 1 ticket, we decrement a ticket from it. If the process were to be wait more than 200 ms and has less than 15 ticket we increment a ticket from it. We also reset the last\_reached value after setting the ticket values. The reason why we declared min wait time as 2 and max wait time as 20 is because jiffies updates itself in every 10 ms.

In this part of our code we select the maximum number of tickets that we have in our not processed processes.

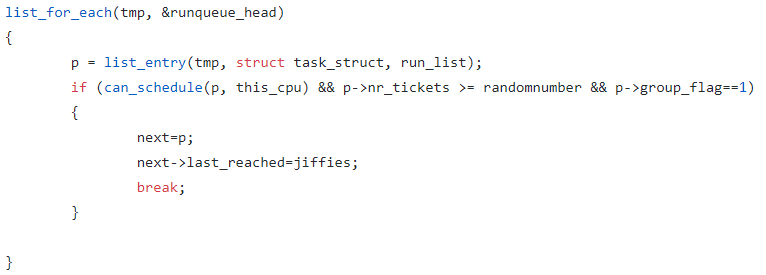


If the number of tickets in a specific process is more than maxticketvalue, we will assign the value of it to maxticketvalue. We check for group\_flag value, because we don’t want to get the ticket value of a processed item.

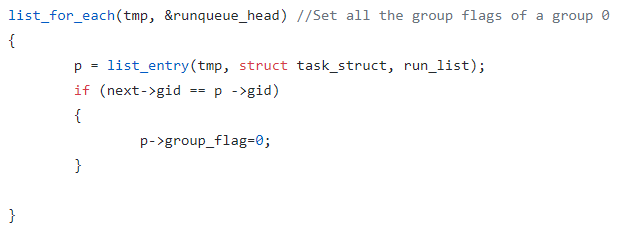
Then we need to get a random value for scheduling.



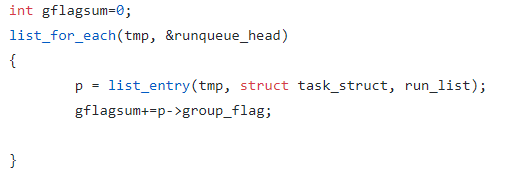
After getting our maximum ticket value that we can obtain, we start to get a random value. Using get\_random\_bytes() function, scheduler assigns a random number in variable randomvariable. Doing that, we check if it’s negative or not, we don’t want a number that’s negative. We also use modulus to get a number from 1 to our maxticketvalue number. Because a random number can be anything. Finishing it, for the reason we can’t get a random number that equals to 0, scheduler increments our random variable by one.



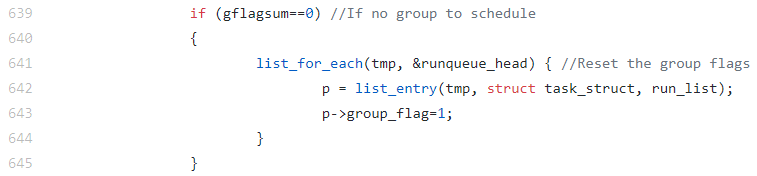
The code part above is our rescheduling part. This is the most important part of our scheduler. We check if a process is in the ready queue, the number of tickets of it are bigger than our random value and also check if the group flag of our contestant process is 1. If a process can meet all of these criteria, we declare the next process as it. We also update the last\_reached value of the newly scheduled process. Finding the next process, scheduler breaks from our loop.



When we finish rescheduling, we need to check the next scheduled processes group id and the waiting ones, because we need to know that the process we use is in the same group. Then, we will assign all of their group\_flag variables 0.



In this part, we check if there are still groups that’s waiting to run their processes. We do that by looping through the task queue and incrementing the value of gflagsum.

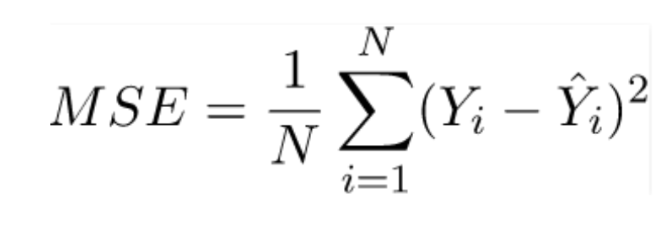


At the end of our scheduler, we look at the gflagsum variable. If it is more than zero, it means that there are still groups that are waiting to be executed. If it equals to zero it means that all of the groups executed their processes, so we switch all of the group\_flag variables of processes as 1.

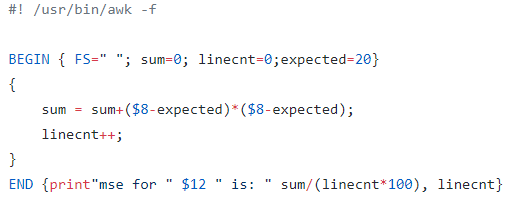
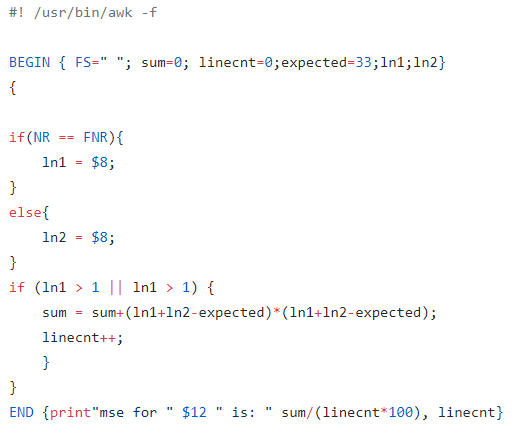
# TESTS and RESULTS

After successfully implementing GTicket scheduling algorithm, we compiled our kernel and starte the system using it. After that we took 1000 samples each by using the default Linux scheduler and our scheduler implementation. We used the Linux top command for it.

Finished getting our sample values, we stripped our lines via using Linux grep command for it. After doing all of this we wrote an AWK code to calculate MSE (Mean Square Error) value.



Expected value is 100/process\_number for default scheduler and 100/group\_number.

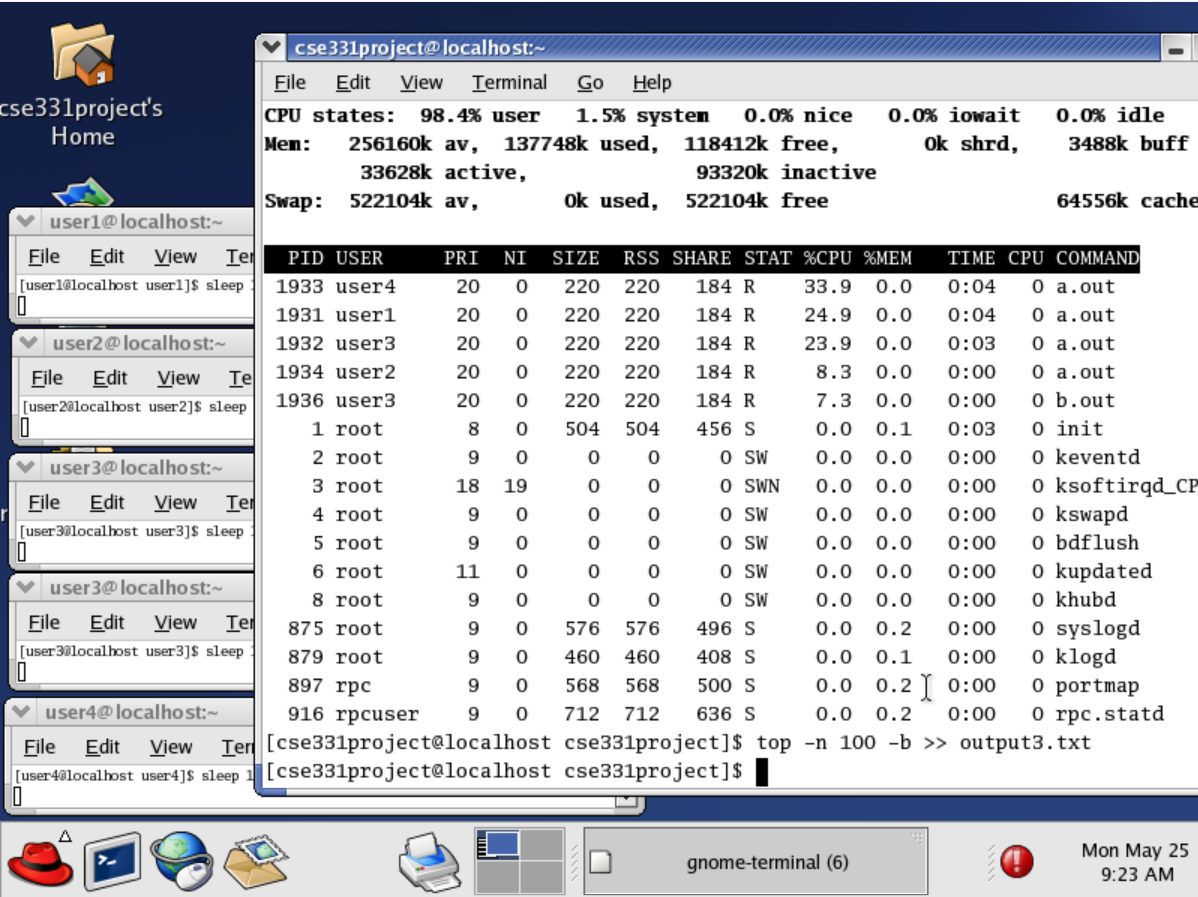


For Default Scheduler

For GTicket Algorithm

Testing the algorithm, we used a shell script for each user to run a CPU intensive C code. By doing that, we gathered a broader data set for our calculations.

We had 4 users, 3 user groups and 5 processes in our tests. User1 and User2 each had 1 process, and were in the first group. User 3 had 2 processes, and were in the second group. And lasty, User 4 had 1 process, and were in the second group.



# CONCLUSION

In conclusion we saw default scheduler executes with less mean square value. We already expect to see less error because of behavior 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 GTicket we had higher mean square value because of how we perceive the job scheduling on it. Instead of being fair to the processes on the system, we try to be fair for the user groups. In most ways, especially with our virtual machine system, it greatly decreases the performance of the operating system. But it could be beneficial lets for say server applications. If there is only one server with multiple users, and if some of them have higher priorities to do a job, GTicket could be more efficient for this kind of application. But still, because of the fact that we also share the cpu usage in pure luck (mind you, we give every single process a ticket and just raffle them) it would also be bad for the performance of our system. It just depends our use case.

# REFERENCES

<https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf>

<http://pages.cs.wisc.edu/~remzi/OSTEP/>

<https://en.wikipedia.org/wiki/Multilevel_feedback_queue>

<https://en.wikipedia.org/wiki/Lottery_scheduling>

<https://www.grymoire.com/Unix/Awk.html>

<https://ryanstutorials.net/bash-scripting-tutorial/bash-loops.php>