**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)5

[**TESTS and RESULTS**](#_2jxsxqh)9

[**CONCLUSION**](#_qsh70q)10

[**REFERENCES**](#_32hioqz)11

# 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.

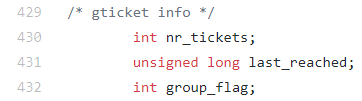
The kernel stores the list of processes in a circular doubly linked list called the task\_list. Each element in the task list is a process descriptor of the struct task\_struct, which is defined in <linux/sched.h>. The task structure contains all the information about a specific process. We want to change 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>**

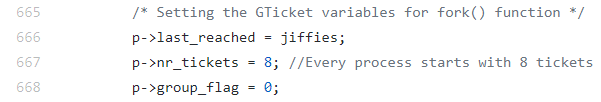
In sched.h we need to declare all the variables that we are going to use in our scheduler. Because in our scheduling policy, we give every process a ticket, unlike the native scheduler. That’s why we initialize our ticket number as nr\_tickets. After that we also declare the last\_reached and group\_flag 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 unsigned 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 p is that the fork.c declares our task\_struct as p in our do\_fork() function. We are also going to use task\_struct in sched.c, because all processes does contain task\_struct.

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

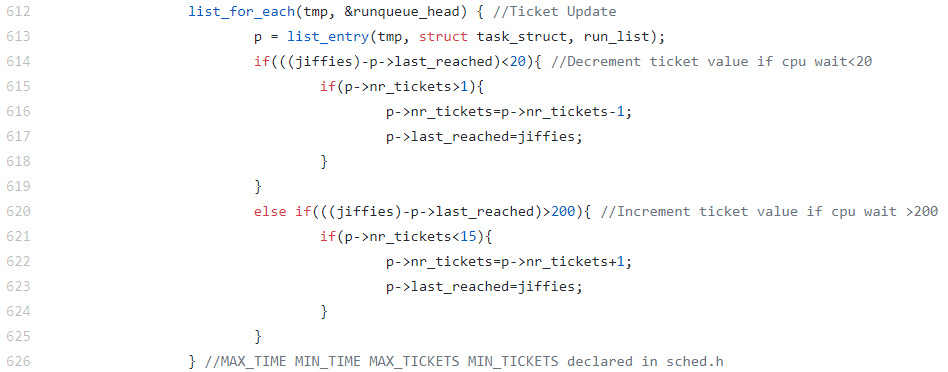
After we design and implement what we did on 2.1 and 2.2 we started to alter our standard scheduler. First of all, we need to add a flag to choose which scheduler to work, so we declare our flag variable, so we declare it, and check it on repeat\_scheduler.



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

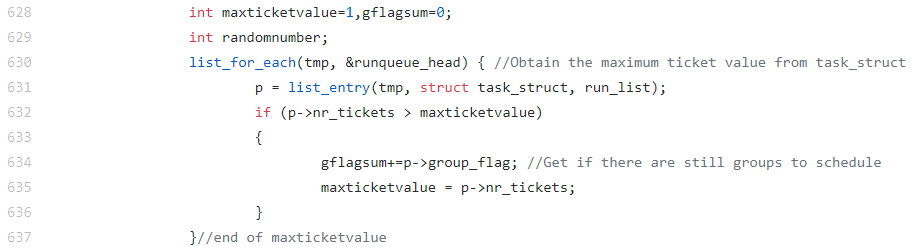
If gticket\_policy is 0, we’ll use the standard schedule policy, and if it’s 1 we’ll use GTicket schedule policy.

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



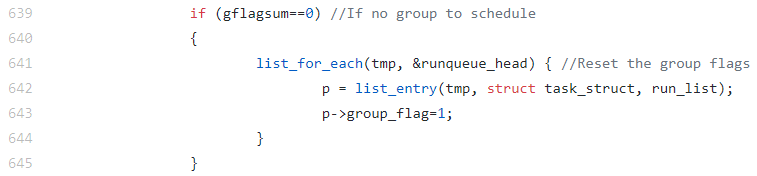
In this part of our code we want to do a loop in our run\_list and look at every processes last\_reached value. If the process were to be wait less than 20 ms and has more than 1 ticket, we decrement a ticket from it. If the process were to be wait more than 200 ms and has less than 15 ticket we increment a ticket from it. We also reset the last\_reached value after setting the ticket values.

In 628 we declare our maxticketvalue as 1 and gflagsum as 0. It’ll be useful on the next parts of our scheduler design.

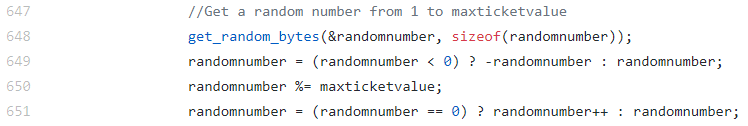


If the number of tickets in a specific process is more than maxticketvalue, we will assign the value of it to maxticketvalue.

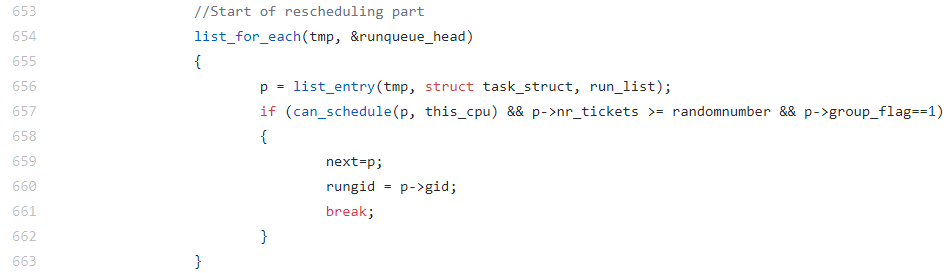
Whilst we are looping in the run\_list, we also sum all of the group\_flag variables in gflagsum because of the next part of our code:



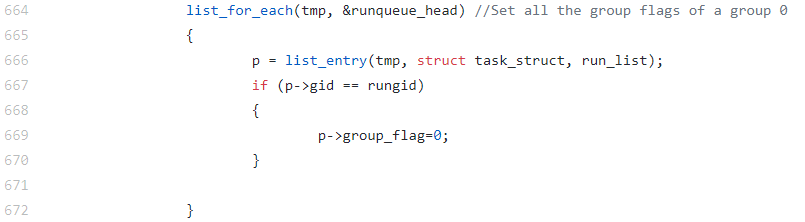
After we get the summation of all the group flags we’ll check if it’s equal to zero. We do this because if there is no group to schedule next, we won’t be able to run our processes. If all of the group flags are 0, we’ll set them 1 on all of them in lines 641 to 644.



In line 647 we start to get a random value. Using get\_random\_bytes() function in line 648 we assign a random number in our variable randomvariable. In line 649, 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 0 to our maxticketvalue number. Because a random number can be anything. And at last in line 651 we check if our randomnumber is 0 or not, we also don’t want a random variable that equals to 0.



At line 654 we come to our rescheduling part of the code. 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 at line 657. If a process can meet all of these criteria, we declare the next process as it in line 659. We also get the user group id of this process in line 660, and break from our loop.



In this last part we just set all of the processes that has the same user group id with the process that we selected to 0. The line 664 to 672 shows the process

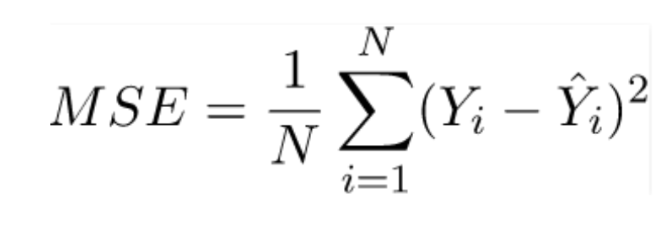


Our variable rungid is declared in line 558 of sched.c. It has the type gid\_t because in the task\_struct of every process we contain them with the same variable type.

# 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.

# CONCLUSION

# 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>