**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**

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**

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

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

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

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

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

# INTRODUCTION

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 a process scheduling that runs quite different 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 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 bigger value of tickets than our random value.

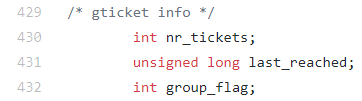
The kernel stores the list of processes in a circular doubly linked list called the task\_list. Each element in the task list is a process descriptor of the struct task\_struct, which is defined in <linux/sched.h>. The task structure contains all the information about a specific process. We want to change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

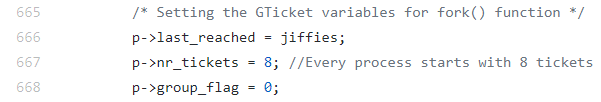
In sched.h we need to declare all the variables that we are going to use in our scheduler. Because in our scheduling policy, we give every process a ticket, unlike the native scheduler. That’s why we initialize our ticket number as nr\_tickets. After that we also declare the last\_reached and group\_flag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

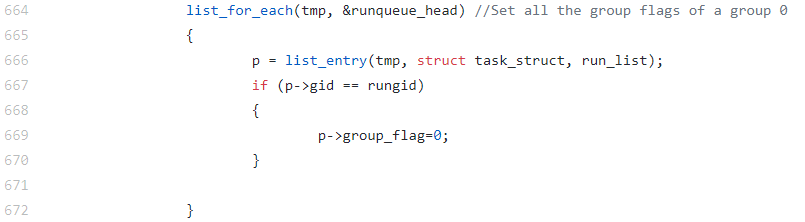
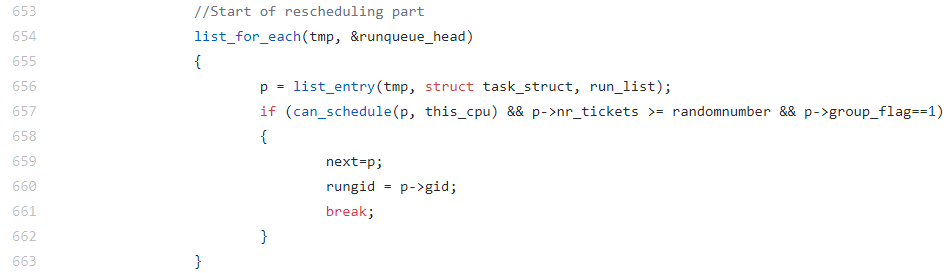
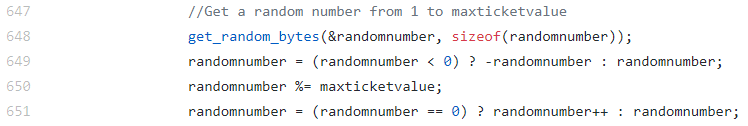
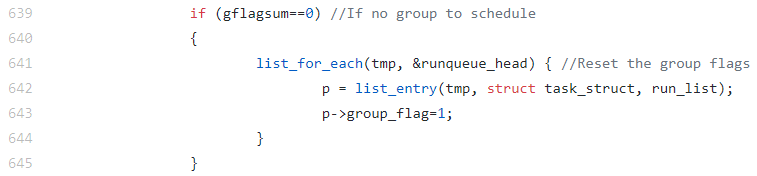
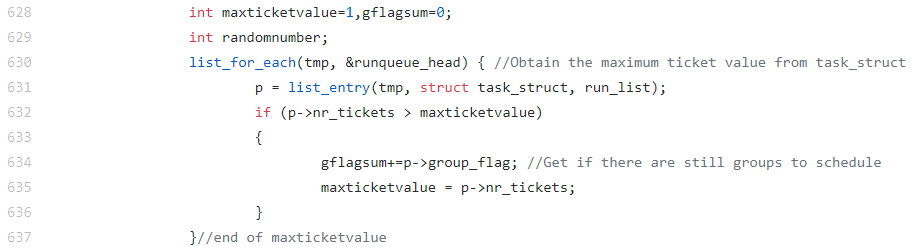
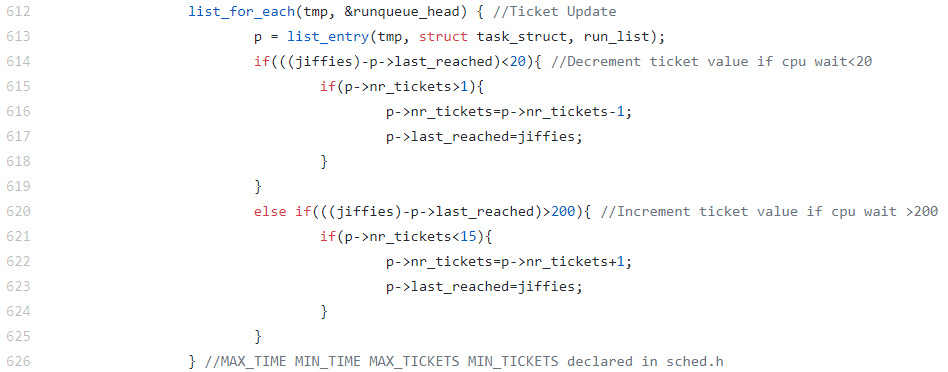


The reason why we set numbers in the pointer p is that the fork.c declares our task\_struct as p in our do\_fork() function. We are also going to use task\_struct in sched.c, because all processes does contain task\_struct.

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



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



# TESTS and RESULTS

# CONCLUSION

# REFERENCES