**Fair Ruffle Scheduling**

**By**

**Adnan ÇIĞTEKİN  
Ahmet Mete KARAYAKA  
Berkay KOÇAK**

**CSE 331 Operating Systems Design**

**Term Project Report**

**Yeditepe University**

**Faculty of Engineering**

**Department of Computer Engineering**

**Spring 2018**

**ABSTRACT**

In this project our intend is to create a scheduler that shares the CPU fair between all the users and fair between the processes of an user.  
In the first section “introduction”, we briefly explain what scheduling is, what are the criteria that affect scheduling performance and we explain the scheduling algorithms that are used or referenced today. Also in this chapter we state our problem and briefly explain our solution to this problem.  
In the second section “design and implementation”, we explain our solution design and implementation in detail, using some charts and pseudo codes. In the design sub-section we talk about the theory and the behavior of the solution while in the implementation section we are interested with physical structure of the solution.  
In the third section, we focus and testing our solution and commenting about it while comparing with other solutions.  
In the fourth and the last section, we explain argue if we succeed solving the problem, we also state some improvements for this solution.

**TABLE OF CONTENTS**

[**INTRODUCTION**](#_1fob9te) **5**

[**DESIGN and IMPLEMENTATION**](#_3znysh7) **9**

[**TESTS and RESULTS**](#_2et92p0) **12**

[**CONCLUSION**](#_tyjcwt) **20**

[**REFERENCES**](#_3dy6vkm) **21**

# LIST OF FIGURES

**Figure Page**

1.1 Fair-Ruffle Implementation.....…………………………………….…….8

2.1 User Fairness Implementation.……………………….………….………9

2.2 Ticket System Implementation.…………………………………….…...10

3.1 Single User – Single Process (FRS) Chart ……..……………………….13

3.2 Single User – Single Process (DS) Chart ……..………………………...13

3.3 Single User – Two Processes (FRS) Chart ……………………………...14

3.4 Single User – Two Processes (DS) Chart……………………………......14 3.5 Two Users – Two Process (FRS) Chart ……..………………….……....15

3.6 Two Users – Two Process (DS) Chart ……..…………………………....15

3.7 Two Users – Three Processes (FRS) Chart ……………………………...16

3.8 Two Users – Three Processes (DS) Chart …………………………….....16

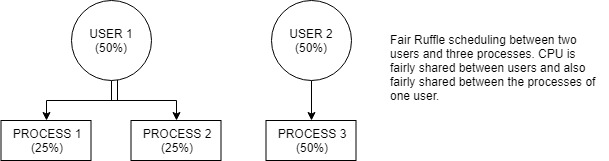
3.9 Two Users – Four Processes (FRS) Chart ………………………….…....17

3.10 Two Users – Four Processes (DS) Chart …………………………….......17

3.11 Two Users – Four Processes Alternative (FRS) Chart………...…….…...18

3.12 Two Users – Four Processes Alternative (DS) Chart ……..…………......18

# INTRODUCTION If you are using computers frequently, you might think that a computer does many operations at the same time. You would not be wrong to come up with such inference because if you observe our modern-day computers, they do really look like they are doing many tasks at the same time. However this is not the case. In computers only one process can be run by the CPU at a time while the other processes have to wait to be executed. What you really see when you observe the computers is an illusion of “multi-tasking” by switching between these waiting processes to be executed in CPU, very fast. If you want to skip the basic information behind scheduling, you can skip to chapter 1.4 where you can find our problem definition and scheduler algorithm 1.1) An Overview of Scheduling We can now further discuss this illusion. Scheduling can be explained basically as; a unit which is called “scheduler” chooses the next process to be executed from a process queue and using another unit called “dispatcher” to put this process into CPU and save the state of replaced process. More specifically scheduler runs a decision algorithm on the processes that are on ready queue whenever a process terminates or gets interrupted which is generally an I/O interrupt. After selecting a new process to be executed, scheduler informs the dispatcher about this process by sending the process’ data to the dispatcher. Then dispatcher saves the information of the current process that runs on the CPU and gives the control of the CPU to the incoming process. This “cycle” runs on the computer infinitely. Both scheduler and dispatcher are a part of the operating system. The main idea behind the scheduling is to never let CPU to be idle, CPU is the fundamental resource of the computer thus whenever CPU becomes idle Operating System runs the scheduling algorithm. In multiprogramming systems, scheduling is exceptionally critical for overall system performance since there are many processes in the main memory and the alignment that these processes are selected to be executed is important for system performance. In modern day systems, this replacement generally runs when the process that is being executed completes its CPU-Burst and leaves for an I/O burst. CPU scheduling occurs in four different circumstances; I) When a process switches from the running state to the waiting state (for example, as the result of an I/O request for an invocation wait() for the termination of a child process). II) When a process switches from the running state to the ready state (for example, when an interrupt occurs). III) When a process switches from the waiting state to ready state (for example, at completion of I/O). IV) When a process terminates. When scheduling takes place only under circumstances I and IV, we say that the scheduling scheme is nonpreemptive that is once the CPU has been allocated to a process, the process holds the CPU until the process terminates or switching to waiting state. Otherwise scheduling scheme is preemptive.[[1]](#footnote-2) 1.2) Criteria that Affects CPU Scheduling There are many criteria that we use to compare the performances of scheduling algorithms. Some of these criteria are as follows; *- CPU Utilization:* How long does this algorithm keeps the CPU busy, an algorithm should reduce the idle time as much as possible. This is the main criteria that we are going to use in our Fair-Ruffle Scheduling comparisons. This criteria is heavily affected by CPU-intensive and I/O intensive processes. *- Throughput:* Number of process that are completed per time unit is called throughput. An algorithm should have a relatively large value of throughput comparing to another algorithms. *- Turnaround Time:* The amount of time that a process takes to complete. Depending on the type of the process, turnaround time should be minimized. *- Waiting Time:* The amount of time that a process waits in the ready queue. Waiting time should be minimized in order to avoid starvation.[[2]](#footnote-3) *- Fairness:* Scheduling algorithm should share the CPU resource fairly among multiple users. 1.3) Scheduling Algorithms 1.3.1) *First Come First Serve Scheduling* This scheduling algorithm is the simplest scheduling algorithm. As the name would suggest, the process that arrives at the ready queue first is given to the CPU first to be executed. This algorithm is used in the early computers. First come first serve scheduling is a non-preemptive algorithm. 1.3.2) *Round Robin Scheduling* In this scheduling algorithm, we need to define a new term; time quantum. Time quantum means that the current process that is being executed is going to be interrupted in a frequency that is decided. This interrupts allow the CPU to be shared fairly between the processes and hinder starvation. These interrupts are generated by a module called the timer. In modern-day computers, every scheduling algorithm have this interrupt system. It can be understood from the definition of time quantum, that this algorithm is preemptive. 1.3.3) *Shortest Time to Completion First Scheduling* In this scheduling algorithm, scheduler chooses the process that has the lowest amount of time remaining until termination, from the ready queue. One great problem with this algorithm is that, we need to predict the future. Scheduler does not know how long a process will run but this algorithm requires us to have that knowledge. This algorithm is nonpreemptive. 1.3.4) *Shortest Time Remaining to Completion First Scheduling* In this scheduling algorithm, scheduler chooses the process that has the lowest amount of time remaining until termination, from the ready queue just like the STCF algorithm. But this algorithm has one aspect that makes it preemptive; when a process arrives in the system that has lower amount of time to complete than the current process that being executed, scheduler allocates the CPU to the incoming process. Just like STCF, this algorithm also requires us to predict the future. Also the algorithm is the optimal algorithm, meaning that we need to estimate our scheduler algorithms to this algorithm. 1.3.5) *Priority Scheduling* In this scheduling algorithm, every process is assigned a priority value. Scheduler allocates the CPU to the highest priority process in the ready queue. In case there are multiple processes in the ready queue that have the same priority value, scheduler uses another algorithm to choose between them, this algorithm is generally first come first serve. Priority scheduling can be implemented in both preemptive and nonpreemptive way. 1.3.6) *Multilevel Queue Scheduling* In this scheduling algorithm, the ready queue is divided in to multiple queues. The processes are assigned to one of these queues and they cannot travel between queues (We emphasize this because the next algorithm is the dynamic version of this algorithm). These assignment are generally based on the type of the process. In each queue there is a different scheduling algorithm. Also scheduler needs to schedule the queues in this algorithm. 1.3.7) *Multilevel Feedback Queue Scheduling* This scheduling algorithm is basically same as the multilevel queue scheduling but the exception is that the processes can travel between the queues. Just like the multilevel queue scheduling this algorithm has different scheduling algorithm in queues and another scheduling algorithm between the queues.

**1.4) Fair-Ruffle Scheduling** The problem is to implement an algorithm that is going to fairly share the CPU among the users with high CPU utilization. Not just fair among the users, but fair among the processes as well. We briefly talked about the algorithms that are being used for this purpose and we also came up with an algorithm idea to solve this problem, that is: Fair Ruffle Scheduling.   
 Now that we have discussed what scheduling is, what are the criteria we use to compare and calculate the efficiency of scheduling algorithms and the general scheduling algorithms, we can discuss the scheduling algorithm that we are going to implement.  
 In fair ruffle scheduling the main idea is to share the CPU fairly among users. Then in a user’s ready queue, we try to share the CPU, again, fairly among the processes. This fair distribution among the process is done by a “lottery ticket system” that is further discussed in design and implementation chapter 2. The comparison of the scheduling algorithm to default Linux Scheduler is discussed in chapter 3 using samples in both algorithms and finally in chapter 4 we discuss our results and conclude the fair-ruffle scheduling algorithm usability.   
 An example resource allocation of fair ruffle scheduling would be;  
  
  
 Figure 1.1

# DESIGN and IMPLEMENTATION 2.1) Design

In fair-ruffle scheduling we have two subcategories; fairness among users, fairness among the processes.

*Fairness among users* is done by defining a flag to the all current users of the system. Whenever a process is executed, the process’ user is marked the by the flag so that scheduler cannot choose that user again to execute a process that belongs to that user. When all users have one of their processes executed, these flags are changed so that we can start to execute for any user again. This can be understood easily by an example; assume that we have 2 users with one having 1 process and the other having 2 processes.

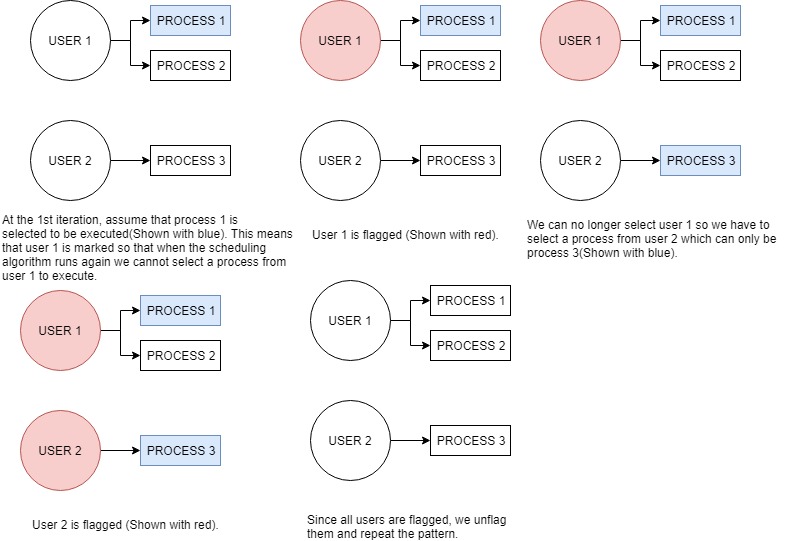


Figure 2.1

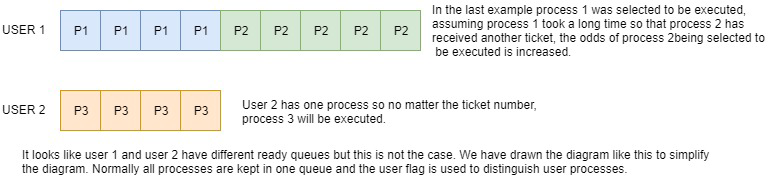
*Fairness among processes* are done by a “lottery-ticket” system. Every process is given a number of tickets when they are initialized. These tickets dynamically incremented or decremented depending on how much time has passed since the process last taken the CPU. If a process was given control of the CPU longer than a specified amount of time, these tickets are incremented. This basically means the longer a process waits for the CPU the larger number of tickets it is going to have. However if a process was given control of the CPU shorter than the specified amount of time, these tickets are decremented. This basically means that the shorter a process waits for the CPU, the smaller number of tickets it is going to have. Then we randomly select a winning ticket from these tickets which means the more a process has tickets the more likely it is going to win and get the CPU. Let us visualize this ticket system for the previous example;  


Figure 2.2

This is the design philosophy we followed while implementing our scheduling algorithm.   
  
**2.2) Implementation** We have implemented the Fair Ruffle scheduling algorithm in RedHat Linux with Linux version 2.4.20. To implement fair ruffle scheduling we need to follow these steps (Pseudo-code implementation):   
  
I) We need to define a flag in “user\_struct” in the file “sched.h”.

user\_struct{  
…  
var flag := 0;  
}  
II) We need to define ticket number variable and time variable that holds the jiffies in “task\_struct” in the file “sched.h”.  
task\_struct{

…  
var numberOfTicket;   
var lastJiffies;  
}  
III) We need to initialize the values of the ticket number variable and the time variable that holds jiffies in the file “fork.c”. This allows us the increment of decrement number of tickets by checking the time variable.  
*…*  
process->numberOfTicket := 4;  
process->lastJiffies := jiffies;  
…  
  
IV) Lastly we need to implement the scheduler in the file “fork.c”.   
*…*repeat\_schedule:int sumOfTickets;  
int selectedFlag;  
var userID;  
 selectedFlag := -1;  
 for (all processes in ready queue) begin  
 if (process->user->flag == 0) begin  
 userID = process->user->uid;  
 process->user->flag = 1;  
 selectedFlag = -1;  
 break;  
 end  
 end  
 if (selectedFlag == -1) begin  
 for (all processes in ready queue) begin  
 process->user->flag = 0;  
 end  
 goto repeat\_schedule;  
 end  
 sumOfTickets = 0;  
 for (all processes in ready queue) begin  
 if (process->numberOfTickets < 7) and (jiffies - process->lastJiffies > 12) begin

process->numberOfTickets = p->numberOfTickets +1;

end

else if (process->numberOfTickets > 1) and (jiffies - process->lastJiffies) < 4) begin

process->numberOfTickets = process->numberOfTickets -1;

end

if (process->user->uid == userID) begin

sumOfTickets = sumOfTickets + process->numberOfTickets;

end  
 int rand = 0;

int prizeTicket = random();

if (prizeTicket < 0) begin

prizeTicket = prizeTicket \* -1;

end

for (all processes in ready queue) begin

if(process->user->uid == userID) begin

prizeTicket = prizeTicket - process->numberOfTickets;

if (prizeTicket <= 0) begin

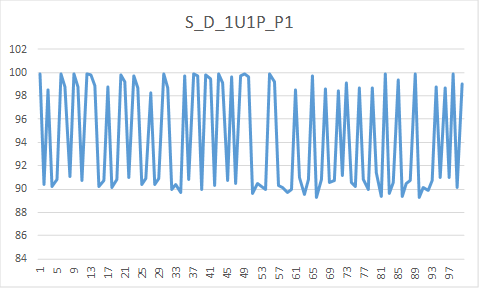
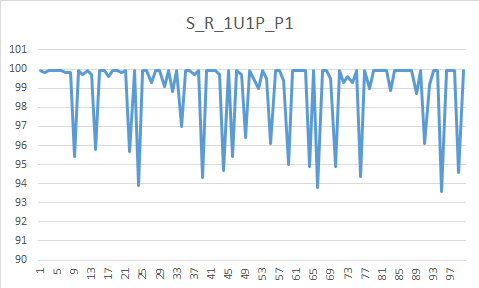
nextToBeRun= process;

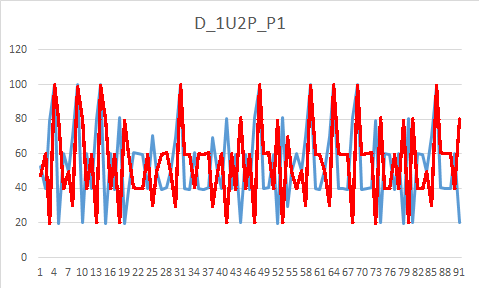
p->lastJiffies = jiffies;

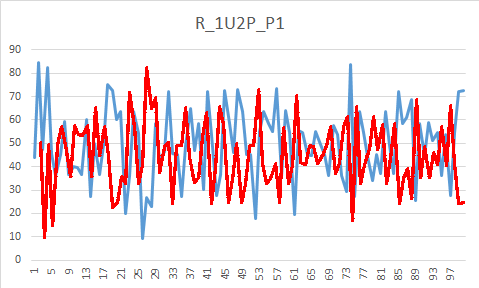
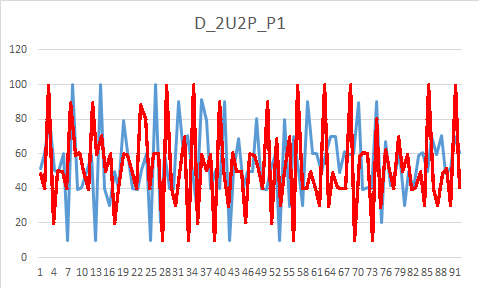
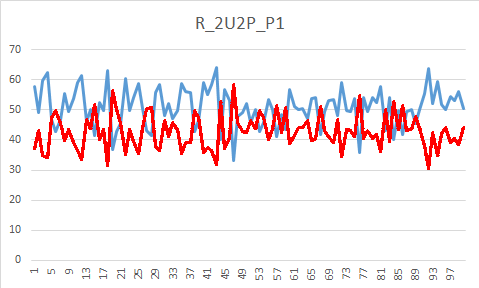
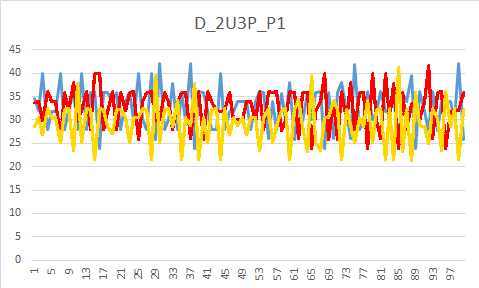
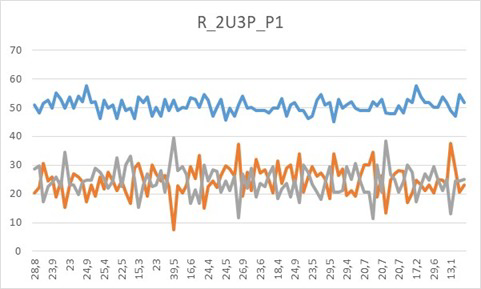
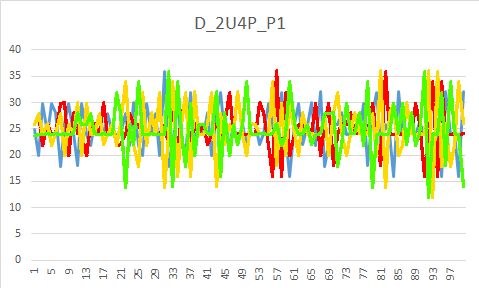
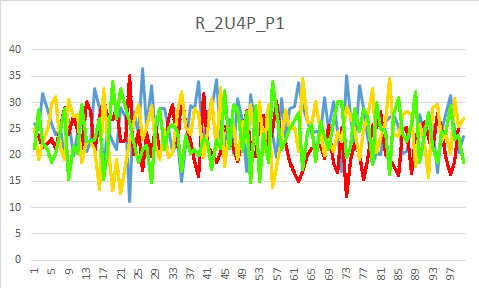
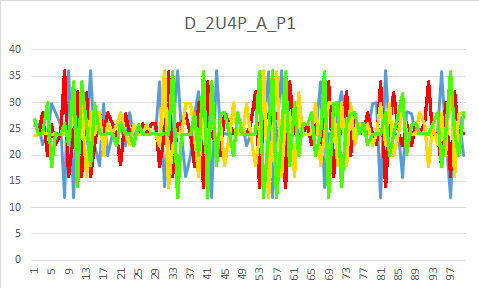
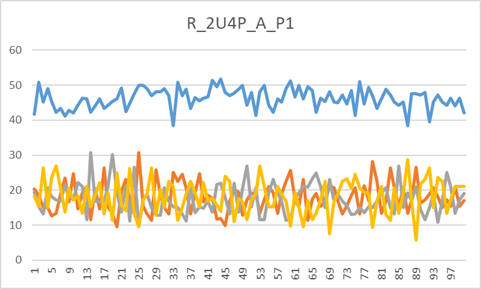
break;

end  
 end  
 end  
When a process is created it is initially given 4 tickets. These tickets are incremented every time the scheduler algorithm is run. If a process did not take the CPU for 120ms it's ticket value is incremented to a maximum of 7 tickets. If a process did take the CPU for 40ms is decremented to a minumum 1 ticket.Now that we have talked about design and implementation of fair ruffle scheduling, we can move into testing the efficiency of the algorithm and compare it to other algorithms.

# TESTS and RESULTS

In this section, we will be comparing our Fair-Ruffle Scheduler with the Red Hat Linux’s default scheduler. With that being said let us denote that for the following graphs ***Y-Axis stands for CPU usage and X-Axis stands for number of samples.***   
**3.1)** For the first test let us compare *one user one process* environment in *default scheduler*;  
 Figure 3.1  
As it can be seen from the chart, default scheduler’s CPU usage varies between 100% and 90% for a single user program.  
- Now, let us compare *one user one process* environment in *Fair-Ruffle scheduler*;  
 Figure 3.2  
As it can be seen from the chart, Fair-Ruffle scheduler’s CPU usage varies between 100% and 94% for a single user program.  
This variation in CPU usage is logical since system processes require CPU usage to complete too.  
Other than Fair-Raffle getting more CPU usage in average schedulers do not have any big differences.

**3.2)** For the second test let us compare *one user two process* environment in *default scheduler;* (P1 is denoted by red, P2 is denoted by blue) Figure 3.3

As it can be seen from the chart, processes get more CPU cycles than the other processes with oscillation. There are some 100% recorded samples in the graph but these are anomalies that was caused by the top command.  
- Now, let us compare *one user two process* environment in *Fair-Ruffle scheduler;* (P1 is denoted by red, P2 is denoted by blue) Figure 3.4  
  
As it can be seen from the chart, processes get more CPU cycles than the other processes with oscillation. This again looks the default scheduler’s behavior. Only difference between both schedulers is CPU usage difference in Fair-Ruffle scheduler is greater than the default scheduler.  
  
 **3.3)** For the third test let us compare *two users two process* environment where user\_1 has one process P1 and user\_2 has one process P2 in *default scheduler;* (P1 is denoted by red, P2 is denoted by blue) ** Figure 3.5As it can be seen from the chart, again, processes get more CPU cycles than the other processes with oscillation.  
- Now, let us compare *two user two process* environment in *Fair-Ruffle scheduler;* (P1 is denoted by red, P2 is denoted by blue)  
 Figure 3.6  
As it can be seen from the chart, again, processes get more CPU cycles than the other processes with oscillation.  
  
  
  
  
  
  
**3.4)** For the fourth test let us compare *two users three process* environment where user\_1 has one process P1 and user\_2 has two processes P2, P3 in *default scheduler;* (P1 is denoted by red, P2 is denoted by blue, P3 is denoted by yellow)  
 Figure 3.7  
As it can be seen from the chart, processes converge on 33% as we would expect from the default scheduler.  
- Now, let us compare *two user three process* environment in *Fair-Ruffle scheduler;* (P1 is denoted by blue, P2 is denoted by yellow, P3 is denoted by grey)  
  Figure 3.8  
Our first major difference between the schedulers can be observed in this chart. While default scheduler tries to converge every process to 33%, Fair Ruffle scheduler tries a fair sharing between the users by giving user\_1’s process a total of 50% and giving the user\_2’s processes a 50% which is shared by the processes by 25% each. (P2 and P3 were located higher than the location they need to be due to graph creating software so we had to use another software that caused color difference).  
  
  
**3.5)** For the fifth test let us compare *two users four process* environment where user\_1 has two process P1, P2 and user\_2 has two processes P3, P4 in *default scheduler;* (P1 is denoted by red, P2 is denoted by blue, P3 is denoted by yellow, P4 denoted by green)  
 Figure 3.9 As expected, default scheduler shares the CPU between the processes fairly.  
- Now, let us compare *two user four process* environment in *Fair-Ruffle scheduler;* (P1 is denoted by green, P2 is denoted by blue, P3 is denoted by red, P4 denoted by green)  Figure 3.10  
  
Since there are two users with two separate processes, Fair-Raffle scheduler share the CPU equally among all with every process receiving 25% CPU.  
  
  
  
  
**3.6)** For the sixth and the final test let us compare *two users four process* with a twist; user\_1 has three process P1, P2, P3 and user\_2 has one process P4 in *default scheduler;* (P1 is denoted by red, P2 is denoted by blue, P3 is denoted by yellow, P4 denoted by green)  Figure 3.11  
As expected, default scheduler shares the CPU between the processes fairly.  
- Now, let us compare *two user four process* environment in *Fair-Ruffle scheduler;* (P1 is denoted by blue, P2 is denoted by yellow, P3 is denoted by grey, P4 denoted by orange)  
 Figure 3.12  
Again, major difference between the schedulers can be observed in this chart. While default scheduler tries to converge every process to 33%, Fair Ruffle scheduler tries a fair sharing between the users by giving user\_1’s process a total of 50% and giving the user\_2’s processes a 50% which is shared by the processes by 16% each. (P2, P3 and P4 were located higher than the location they need to be due to graph creating software, so we had to use another software that caused color difference).

Notes about tests;   
I) While testing, we had experienced a bug where, we needed a process that always runs on the operating system. Otherwise, when we switched the scheduler algorithm system would crash.

II) In the tests 3.2 and 3.3 and 3.4, there are some samples which has 100% CPU usage. This is caused by the top commands “-d 0.1” option, so they should be ignored.

III) When switched to Fair-Ruffle scheduler, we experienced system slow down even though the CPU was 0% idle.

# CONCLUSION Our problem was to create a scheduler algorithm that would share the CPU fair among users and fair between the processes of a user. By using Fair-Ruffle scheduling algorithm that we defined in the previous chapters we tried to solve this problem. In order to solve the user share problem, we defined a user-flag to the all of the user in the system. In order to solve the processing schedule within a user, we implemented a ticket system that dynamically changes the odds of the processes to be scheduled next. Further inspecting the tests especially 3.4 and 3.6, we can clearly see that Fair-Ruffle scheduling obeys our constraints with some minor deviation. If we compare the uses of both schedulers, default scheduler can be used for everyday computers while Fair-Ruffle scheduler looks fit for server computers since it shares fairly among the users. One improvement on this algorithm can be to finding a more systematic way to pick new processes that is getting rid of the ticket system since it relies too much on randomness.

# REFERENCES 1 Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2012). *Operating System Concepts* (9 Ed.). Wiley Publishing. p. 203-204 2 Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2012). *Operating System Concepts* (9 Ed.). Wiley Publishing. p. 205

1. ### Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2012). *Operating System Concepts* (9 Ed.). Wiley Publishing. p. 203-204

   [↑](#footnote-ref-2)
2. Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2012). *Operating System Concepts* (9 Ed.). Wiley Publishing. p. 205 [↑](#footnote-ref-3)