***Submitted By:***

Name: Ganesh KC

Registration Number: 11807278

Section: K18SJ

Group: A

Student E-mail: [ganesh.timilsina777@gmail.com](mailto:ganesh.timilsina777@gmail.com)

Github links:

Profile : <https://github.com/GaneshTimilsina>

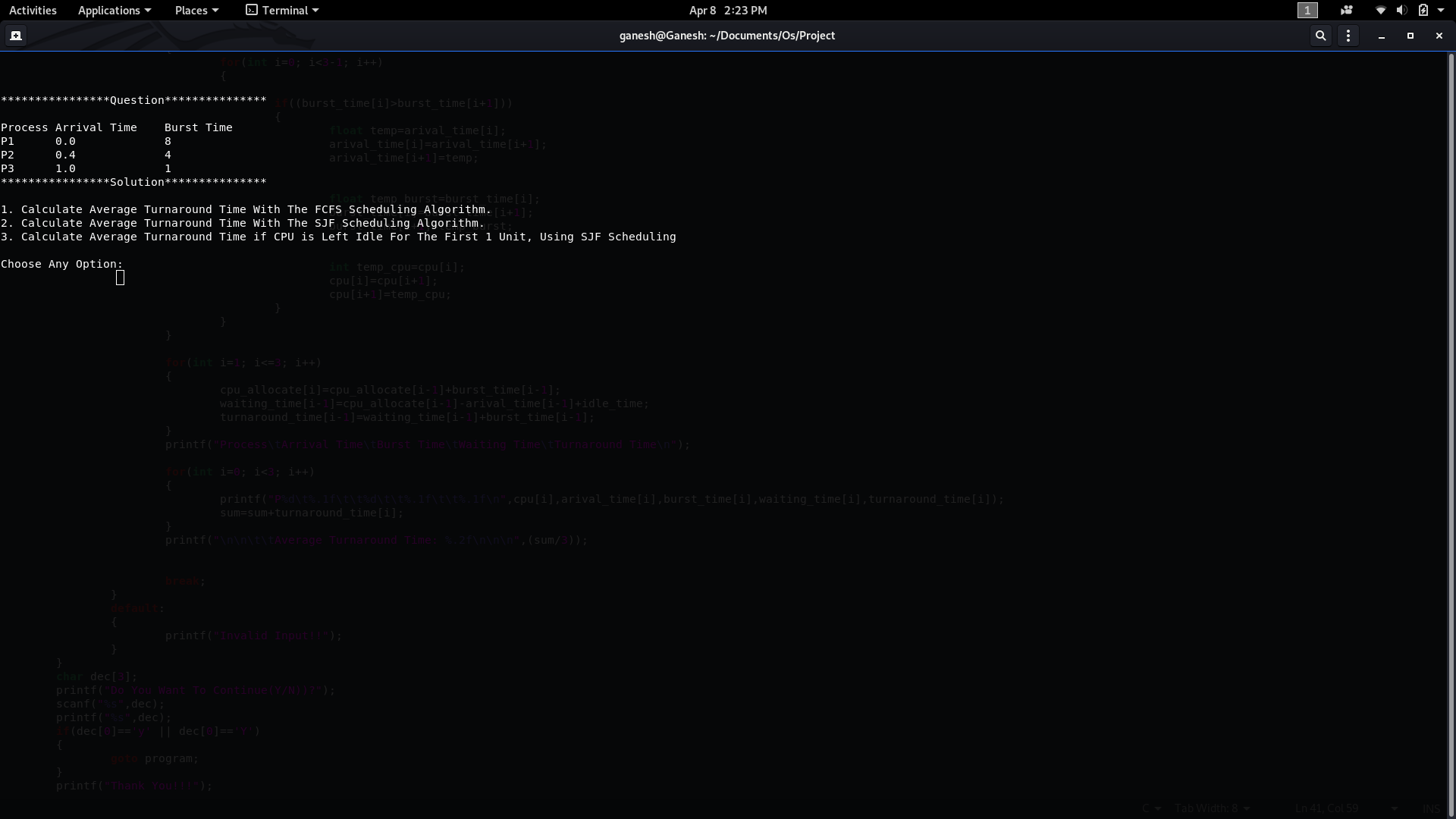
Q.6 : <https://github.com/GaneshTimilsina/Operating-System-Assignments/blob/master/OS_Project_Q6.c>

Q.21 : <https://github.com/GaneshTimilsina/Operating-System-Assignments/blob/master/OS_Project_Q21.c>

Problem 6:

Process name, arrival time and burst time was given so I declared the array and provided the respected value.

To give the answer to the question given to me I have made 3 cases using the switch statement.

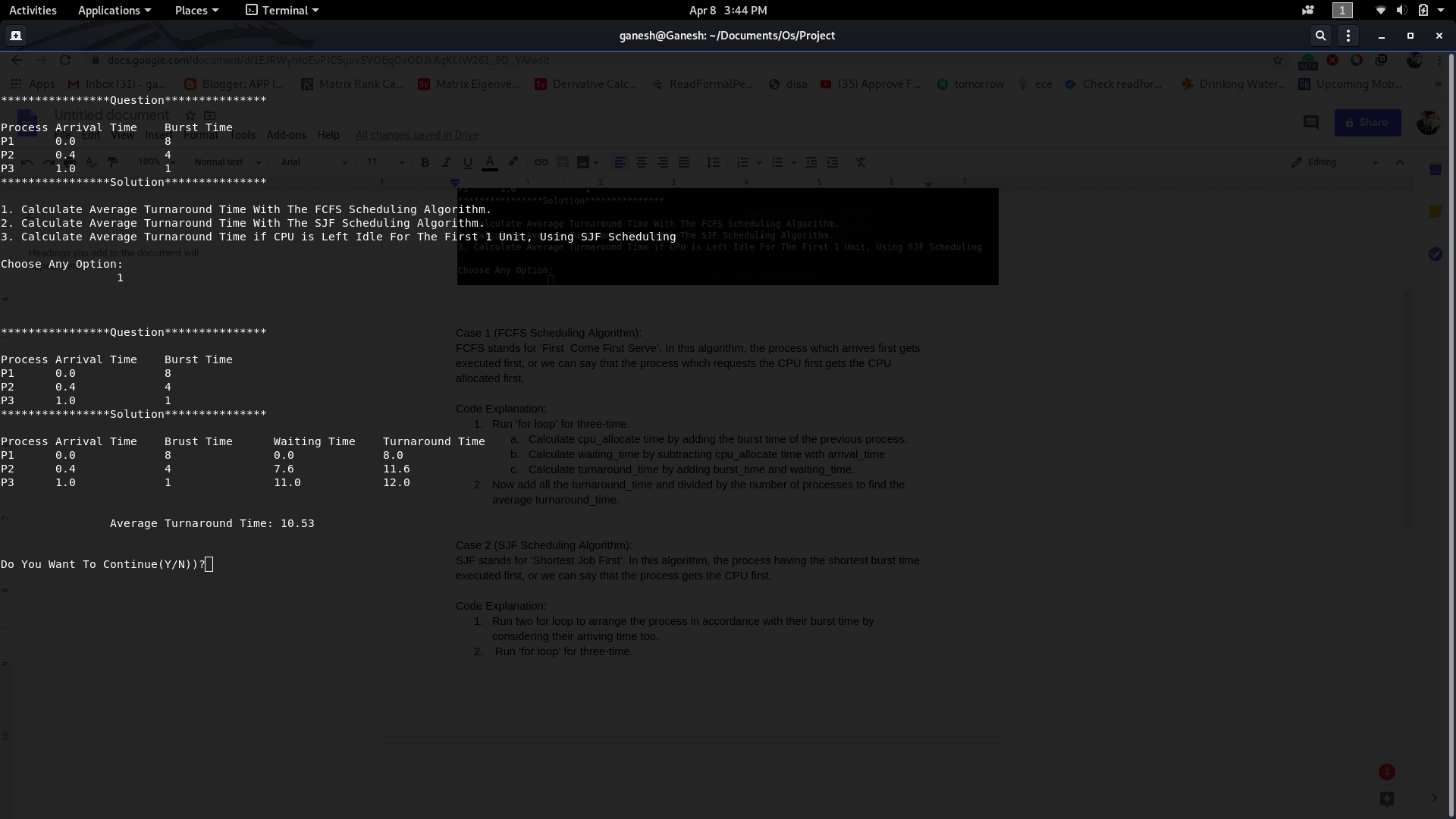


Case 1 (FCFS Scheduling Algorithm):

FCFS stands for ‘First Come First Serve’. In this algorithm, the process which arrives first gets executed first, or we can say that the process which requests the CPU first gets the CPU allocated first.

Code Explanation:

1. Run ‘for loop’ for three-time.
   1. Calculate cpu\_allocate time by adding the burst time of the previous process.
   2. Calculate waiting\_time by subtracting cpu\_allocate time with arrival\_time
   3. Calculate turnaround\_time by adding burst\_time and waiting\_time.
2. Now add all the turnaround\_time and divided by the number of processes to find the average turnaround\_time.

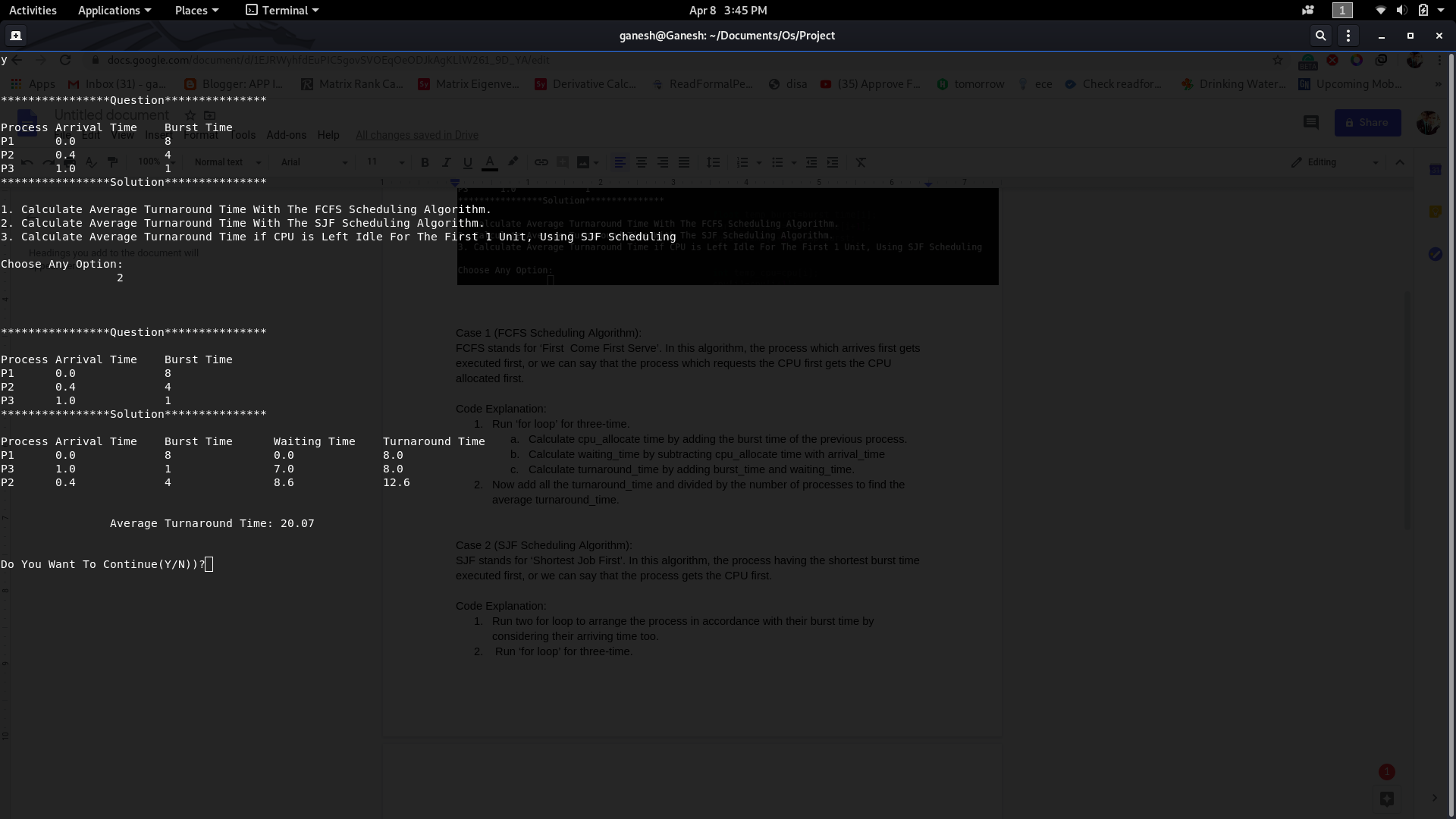


Case 2 (SJF Scheduling Algorithm):

SJF stands for ‘Shortest Job First’. In this algorithm, the process having the shortest burst time executed first, or we can say that the process gets the CPU first.

Code Explanation:

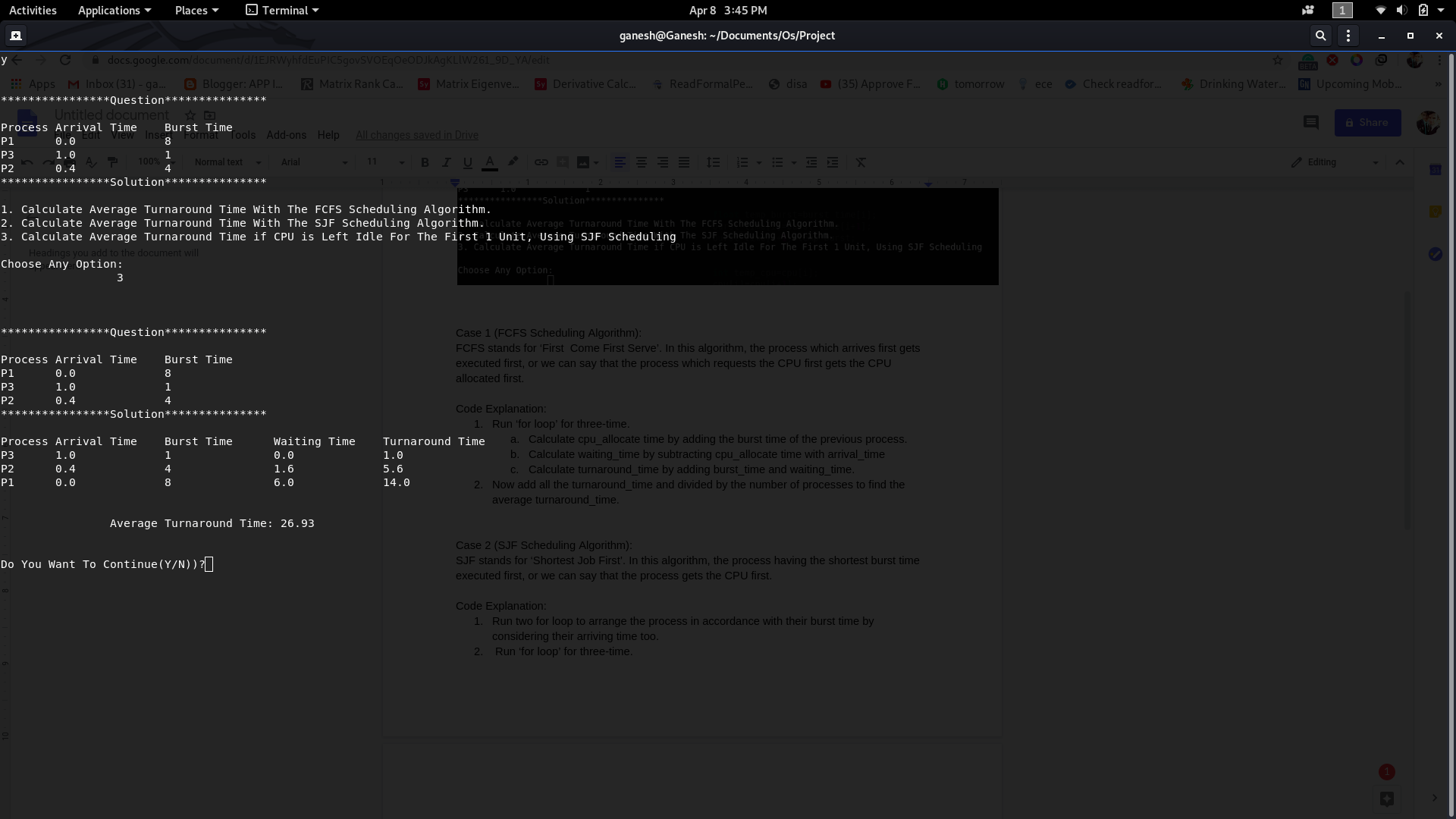
1. Run two for loop to arrange the process in accordance with their burst time by considering their arriving time too.
2. Run ‘for loop’ for three-time.
   1. Calculate cpu\_allocate time by adding the burst time of the previous process.
   2. Calculate waiting\_time by subtracting cpu\_allocate time with arrival\_time
   3. Calculate turnaround\_time by adding burst\_time and waiting\_time.
3. Now add all the turnaround\_time and divided by the number of processes to find the average turnaround\_time.



Case 3:

Calculated the average turnaround time where CPU left idle for the first 1 unit of time by using SJF scheduling algorithm

1. Here for the first unit of time, the CPU remains idle so all the Process comes in the queue and further process by using SJF algorithm.
2. Run two for loop to arrange the process in accordance with their burst time.
3. Run ‘for loop’ for three-time.
   1. Calculate cpu\_allocate time by adding the burst time of the previous process.
   2. Calculate waiting\_time by subtracting cpu\_allocate time with arrival\_time and add idle time i.e. 1.
   3. Calculate turnaround\_time by adding burst\_time and waiting\_time.
4. Now add all the turnaround\_time and divided by the number of processes to find the average turnaround\_time.



.

**Q.6 Code:**

#include<fcntl.h>

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

void main()

{

float arival\_time[3]={0.0,0.4,1.0};

int burst\_time[3]={8,4,1};

float waiting\_time[3];

int option;

float sum, avg\_turnaround\_time;

int cpu[3]={1,2,3};

float turnaround\_time[3];

int cpu\_allocate[4];

cpu\_allocate[0]=0;

sum=0;

program:

system("clear");

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Question\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("Process\tArrival Time\tBurst Time\n");

for(int i=1; i<=3; i++)

{

printf("P%d\t%.1f\t\t%d\n",cpu[i-1],arival\_time[i-1],burst\_time[i-1]);

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Solution\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("1. Calculate Average Turnaround Time With The FCFS Scheduling Algorithm.");

printf("\n2. Calculate Average Turnaround Time With The SJF Scheduling Algorithm.");

printf("\n3. Calculate Average Turnaround Time if CPU is Left Idle For The First 1 Unit, Using SJF Scheduling");

printf("\n\nChoose Any Option:\n\t\t ");

scanf("%d",&option);

switch(option)

{

case 1:

{

//FCFS

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Question\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("Process\tArrival Time\tBurst Time\n");

for(int i=1; i<=3; i++)

{

printf("P%d\t%.1f\t\t%d\n",cpu[i-1],arival\_time[i-1],burst\_time[i-1]);

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Solution\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

for(int i=1; i<=3; i++)

{

cpu\_allocate[i]=cpu\_allocate[i-1]+burst\_time[i-1];

waiting\_time[i-1]=cpu\_allocate[i-1]-arival\_time[i-1];

turnaround\_time[i-1]=waiting\_time[i-1]+burst\_time[i-1];

}

printf("Process\tArrival Time\tBrust Time\tWaiting Time\tTurnaround Time\n");

for(int i=0; i<3; i++)

{

printf("P%d\t%.1f\t\t%d\t\t%.1f\t\t%.1f\n",cpu[i],arival\_time[i],burst\_time[i],waiting\_time[i],turnaround\_time[i]);

sum=sum+turnaround\_time[i];

}

printf("\n\n\t\tAverage Turnaround Time: %.2f\n\n\n",(sum/3));

break;

}

case 2:

{

//SJF

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Question\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("Process\tArrival Time\tBurst Time\n");

for(int i=1; i<=3; i++)

{

printf("P%d\t%.1f\t\t%d\n",cpu[i-1],arival\_time[i-1],burst\_time[i-1]);

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Solution\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

for(int i=0; i<3-1; i++)

{

if((burst\_time[i]>burst\_time[i+1]) && arival\_time[i]!=0.0)

{

float temp=arival\_time[i];

arival\_time[i]=arival\_time[i+1];

arival\_time[i+1]=temp;

float temp\_burst=burst\_time[i];

burst\_time[i]=burst\_time[i+1];

burst\_time[i+1]=temp\_burst;

int temp\_cpu=cpu[i];

cpu[i]=cpu[i+1];

cpu[i+1]=temp\_cpu;

}

}

for(int i=1; i<=3; i++)

{

cpu\_allocate[i]=cpu\_allocate[i-1]+burst\_time[i-1];

waiting\_time[i-1]=cpu\_allocate[i-1]-arival\_time[i-1];

turnaround\_time[i-1]=waiting\_time[i-1]+burst\_time[i-1];

}

printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

for(int i=0; i<3; i++)

{

printf("P%d\t%.1f\t\t%d\t\t%.1f\t\t%.1f\n",cpu[i],arival\_time[i],burst\_time[i],waiting\_time[i],turnaround\_time[i]);

sum=sum+turnaround\_time[i];

}

printf("\n\n\t\tAverage Turnaround Time: %.2f\n\n\n",(sum/3));

break;

}

case 3:

{

int idle\_time=1;

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Question\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("Process\tArrival Time\tBurst Time\n");

for(int i=1; i<=3; i++)

{

printf("P%d\t%.1f\t\t%d\n",cpu[i-1],arival\_time[i-1],burst\_time[i-1]);

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Solution\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

for(int j=0; j<=3-2; j++)

{

for(int i=0; i<3-1; i++)

{

if((burst\_time[i]>burst\_time[i+1]))

{

float temp=arival\_time[i];

arival\_time[i]=arival\_time[i+1];

arival\_time[i+1]=temp;

float temp\_burst=burst\_time[i];

burst\_time[i]=burst\_time[i+1];

burst\_time[i+1]=temp\_burst;

int temp\_cpu=cpu[i];

cpu[i]=cpu[i+1];

cpu[i+1]=temp\_cpu;

}

}

}

for(int i=1; i<=3; i++)

{

cpu\_allocate[i]=cpu\_allocate[i-1]+burst\_time[i-1];

waiting\_time[i-1]=cpu\_allocate[i-1]-arival\_time[i-1]+idle\_time;

turnaround\_time[i-1]=waiting\_time[i-1]+burst\_time[i-1];

}

printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

for(int i=0; i<3; i++)

{

printf("P%d\t%.1f\t\t%d\t\t%.1f\t\t%.1f\n",cpu[i],arival\_time[i],burst\_time[i],waiting\_time[i],turnaround\_time[i]);

sum=sum+turnaround\_time[i];

}

printf("\n\n\t\tAverage Turnaround Time: %.2f\n\n\n",(sum/3));

break;

}

default:

{

printf("Invalid Input!!");

}

}

char dec[3];

printf("Do You Want To Continue(Y/N))?");

scanf("%s",dec);

printf("%s",dec);

if(dec[0]=='y' || dec[0]=='Y')

{

goto program;

}

printf("Thank You!!!");

}

**Q.21**

Explanation:

**The Cat and Mice Problems:**

The mouse and cat simulation works by creating a thread for cats and mouse as th1, th2, etc. Each cat thread repeatedly sleeps after the execution and then tries to eat from one of the bowls. Each mouse thread behaves similarly. Each cat and mouse iterates a fixed number of times before terminating.

I have synchronized the cat and mice so that it satisfies the following requirement:

1. Cats and mice should never be eating at the same time.
2. Only one mouse or one cat may eat from a given bowl at any one time.
3. Neither cats nor mice should starve.

Implementation:

I have created two function cat\_fun and mouse\_fun to solve this problem. I have initialised two semaphores for cats as well as mice. similarly, I have declared 5 threads as th1, th2, th3, th4, th5 and join them respectively.

th1, th2, th3, th4 are used to call cat\_fun() function and th5 is used to call mouse\_fun() function.

**mouse\_fun():**

There is nothing complex in mouse\_function(). At first, the number of mice is 0 so it is increased by 1. The mouse woke up, eat and sleep. when the number of mice becomes it will lock the thread.

**cat\_fun():**

In cat\_fun() function the cat woke up, execute, and sleep. again woke up, execute and sleep. At the time when the number of mice is greater than 0 and the cat is not sleeping yet then the mouse will die. and number of mice is reduced by 1.

Code for Q.21:

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

void \* cat\_fun();

void \* mouse\_fun();

int NumBowls[15],num=0 ,arr[15];

int NumCats=0,NumMice=0;

sem\_t numberOfCats,numberOfMice;

pthread\_t th1,th2,th3,th4,th5;

pthread\_mutex\_t mutex, cat\_utex, mouse\_mutex;

int main()

{ int num=5,x;

int cat\_sem = sem\_init(&numberOfCats,0,5); //Value 0 indicates that the semaphore is to be shared between the threads of process.

int mouse\_sem = sem\_init(&numberOfMice,0,5); //Value 0 indicates that the semaphore is to be shared between the threads of process.

if(cat\_sem!=0)//returns 0 (Zero) on initilizing semaphore successfully

{

printf("Error in initilizing Cat Semaphore");

}

if(mouse\_sem!=0)

{

printf("Error in initilizing Mouse Semaphore");

}

printf("\n\n==================================================================");

printf("\nRules:\n");

printf("1. Cats and mice should never be eating at the same time.\n");

printf("2. Neither cats nor mice should starve.");

printf("\n==================================================================");

pthread\_create(&th1,NULL,cat\_fun,NULL);// pthread\_create is using for Creating a thread

sleep(7);

pthread\_create(&th2,NULL,cat\_fun,NULL);

sleep(3);

pthread\_create(&th3,NULL,cat\_fun,NULL);

sleep(6);

pthread\_create(&th4,NULL,cat\_fun,NULL);

pthread\_create(&th5,NULL,mouse\_fun,NULL);

pthread\_join(th1,NULL); // pthread\_join used to join with a terminated thread!!

pthread\_join(th2,NULL);

pthread\_join(th3,NULL);

pthread\_join(th5,NULL);

pthread\_join(th4,NULL);

printf(“Thank You!!”);

}

void \* mouse\_fun()

{

NumMice=NumMice+1;

arr[NumMice]=NumMice;

int count=NumMice;

sem\_wait(&numberOfMice);

if(NumMice==1)

{

pthread\_mutex\_lock(&mouse\_mutex);

}

printf("\n\nMouse %d Is Eating \n",NumMice);

printf("Mouse %d Is Sleeping \n",NumMice);

sleep(5);

if(count!=arr[count])

{

return NULL;

}

printf("Mouse %d Woke Up And Starts Eating \n",NumMice);

sleep(5);

printf("Mouse %d Has Executed\n",NumMice);

pthread\_mutex\_unlock(&mouse\_mutex);

}

void \* cat\_fun()

{

pthread\_mutex\_lock(&mutex);

NumCats=NumCats+1;

num=num+1;

printf("\n\nCat%d Has Started It's Execution \n",NumCats);

printf("Cat %d Is Now Sleeping \n",NumCats);

sleep(5);

printf("Cat %d Woke Up \n",NumCats);

while(NumMice>0)

{

sem\_destroy(&numberOfMice);

printf("Mouse %d Is Dead \n",NumMice);

arr[NumMice]=-1;

NumMice=NumMice-1;

}

printf("Cat %d Is Now Sleeping Again\n",NumCats);

sleep(5);

printf("\n\nCat %d Woke Up And Starts Eating\n",NumCats);

NumBowls[num]=num;

printf("CAT %d Has Finished It's Execution \n",NumCats);

pthread\_mutex\_unlock(&mutex);

}

**Output:**

