PRIORITY & DEADLINE ALGORITHM

A J-Component Project Report

Submitted by

Aniket Vishwakarma

Under the guidance of

Gopinath M.P.

in partial fulfillment for the award of the degree of

B. Tech.

in

COMPUTER SCIENCE AND ENGINEERING

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

MAY 2020

Priority & Deadline Algorithm for CPU Scheduling

name

Abstract— **In a realistic example of CPU Scheduling, it is often the case that some processes need to be finished before a fixed amount of time. Also, with these deadlines, the processes in themselves carry a weight of importance (priority). To give a simple analogy, while making calculations for a game, the calculation check ammo strictly must be done before the calculations to fire a bullet, this is analogy for the deadline. My Priority & Deadline algorithm is a pre-emptive CPU Scheduling Algorithm which takes into account the priority and deadline of a process. My Algorithm creates a "VIRTUAL TIME CHART" and places the processes in such a way that the deadline and the priorities of the processes available at that time are respected. Obviously for some rare particular arrangement of processes it is impossible to execute all the processes before their deadline, but my algorithm will create an optimum time chart to make sure the best possible outcome is ensured while respecting the priority and the deadline of the process**

Index Terms— CPU Scheduling, Priority, Deadline, Robin Time, Waiting Time, User Experience, Average Turnaround time, Time Chart, Gantt Chart

I. INTRODUCTION

CPU Scheduling Algorithms are away of indexing or organizing the tasks which a CPU has to execute.

Every Second a CPU receive millions of tasks to execute, and it has to organize these tasks in a productive manner in order to facilitate a smooth user experience.

An Algorithm allows the CPU to organize these incoming tasks. Every task has an arrival time, a buffer, and a priority. There are many ways by which different algorithm undergo different approach to try and provide a useful mechanism for process execution which will yield efficient results and a smooth experience for the user prevent lags or data loss.

First Come First Serve – This Algorithm is a very basic one. It categorizes the incoming tasks based on the order which they come in. That's all! Even though its extremely simple, it works very nicely.

Shortest Job First – This Algorithm sorts the already available processes according to their buffer time. The process which requires the shortest amount of time to execute is the process which gets executed first.

Priority Algorithm – This Algorithm pays respect to the priority associated with the process. A process with the higher priority is the process which will get picked first.

All these Algorithms have their own merits and demerits. One thing to notice is that none of these algorithms have an kind of a deadline for their processes. This means that the processes have a chance of being executed indefinitely. But that is not realistic. In a real setting, certain processes need to be finished before a fixed amount of time, this might be because their result needs to be used further in another calculation, or simply that the result acquired by executing the process might be rendered useless past a specific deadline. In a realistic setting, we cannot allow processes to be executed without any sense of urgency. Introducing a deadline, and exercising an algorithm which pays respect to both the Priority and the Deadline associated with the concerned process is the goal of this project.

In This Project we will create an algorithm which will attempt to, at every second, pick the best suited process to execute, keeping in mind the priority and deadline of all the available process

II. LITERATURE SURVEY

Summary/Gaps/Limitations/Future Work identified in the Survey

1. CPU Scheduling is the process of scheduling due tasks in an orderly manner for the CPU to execute to provide the most efficient results. This usually involves an incoming array of processes which the user wants the CPU to execute, each process associated with a buffer, priority and a deadline. The CPU after receiving these tasks must schedule them in such a way so that the most important tasks are looked after and given a higher priority, as if the CPU fails to execute these tasks, the functioning of the entire system might be hampered. Creating a suitable CPU scheduling algorithm is all about developing an algorithm which arranges these incoming tasks in a way which generates maximum profit for the CPU and allows the computer to function with efficiency

2. The issues or "Gaps" so to speak in the previous CPU Scheduling Algorithm, (Mainly speaking Priority Scheduling Algorithm) is that many of the lower priority tasks get lesser to no attention some times if the tasks with a higher priority have an incredibly high buffer time. This results in the starvation of such processes. Lets take an example from real life, if studying and building a carrier is a task of high priority in ones life, and partying and having fun with friends is a task of lower priority, someone might spend her/his entire life chasing a carrier (the higher priority task) and give little to no time to the other smaller things in life. While this is certainly a way of living which will fetch a good high paying carrier, some might argue that a balance must be maintained, and that nothing is good in abundance. This is why introducing a deadline to your tasks would be equivalent to making a stricter schedule for studying, which then leaves time for refreshment activites.

3. Speaking in professional terms, introducing a deadline to the tasks will make sure that the higher
   priority tasks are completed in due time, and they don't drag on forever, and if there is free time, the CPU can include some tasks with a lower priority to make sure that a balance is maintained, and starvation is avoided

4. Certain limitations in the CPU Scheduling algorithm which I am about to display would be that .it cannot always execute every task available in the process queue. Sometimes due to deadline constraints and priority issues, some of the tasks reach their deadline without complete and proper execution. For example, Suppose there is a task with a very high priority, high buffer and a deadline which is close, in order to give priority to that task, other lower priority tasks might be set aside. Now some of these lower priority tasks have a deadline which is far away. These tasks can be dealt with later. But some of these tasks will have a deadline which is not that far away. In giving preference to the task with the higher priority and closer deadline, other

lower priority tasks with a close deadline might get ignored.

5. There is not much that can be done about this problem really, as giving preference to tasks with the higher priorirty sometimes means sacrificing other unimportant tasks. This is comparable to someone sacrificing a nice dinner in a hotel with friends and family to stay at home and prepare for an upcoming assessment or examination.

6. Other limitations include the fact that due to the extensively pre-emptive nature of the algorithm, the average turnaround time is extremely high.

7. Future scope for this project would be to eradicate the problem of executing processes that never actually complete execution before their deadline expires. This means the units of time which were spent executing these processes are wasted. Certain mechanisms could be implemented to make sure that time of the CPU is not wasted in executing processes which do not complete execution. One solution would be to increase the time after which the time chart is refreshed. This means that it won't be refreshed at every time unit, and new arrived processes will be given attention not on every time unit, but at a specific robin time. This would make sure that the processes which begin execution are not interrupted by other incoming processes.

III. PROPOSED METHODOLOGY

Introduction

When organizing a bunch of processes into a process queue for due execution, it is very important to make sure the right qualities are checked before and considered before sorting the processes. The various algorithms for CPU Scheduling give importance to different attributes in associated with a process. The Priority & Deadline Algorithm pays importance to, as the name suggests, a Priority and a Deadline. The Priority associated with the process determines the importance of the process (a higher priority process MUST be executed without fail), and the deadline of the process give sort of a schedule to the process execution, and makes sure that certain higher priority processes are dealt with within a certain time frame, so that there is free time remaining for some of the lower priority processes, in order to avoid starvation, which is a huge problem in a regular Priority Algorithm which does not pay respect to the deadline, which causes starvation lower priority processes.

To facilitate the same the following steps are followed of the algorithm designed:

1. Including the Libraries:

   #include<stdio.h>, #include <sys/types.h>

   #include<stdlib.h> , #include<sys/shm.h>

   #include<stdbool.h> , #include<sys/ipc.h>

   #include<iostream>, #include<semaphore.h>, #include<pthread.h>

2. Construct various structures for arrival, process (has an added attribute "mark"), priority and deadline, each with an
    - Id
    - Arrival
    - Buffer
    - Priority
    - Deadline

3. For every structure create an array queue, which has its own modified insert and remove function. This will make further coding extremely convenient.

4. Take Input from the user in the form of ID, ARRIVAL, BUFFER, DEADLINE and PRIORITY. This input is taken into the arrival queue, which sorts them according to arrival so that we can simulate the process of CPU execution.

5. After this, the main function calls the Priority_Deadline function which then executes the priority and deadline algorithm and prints out a simulation of CPU execution and a profit

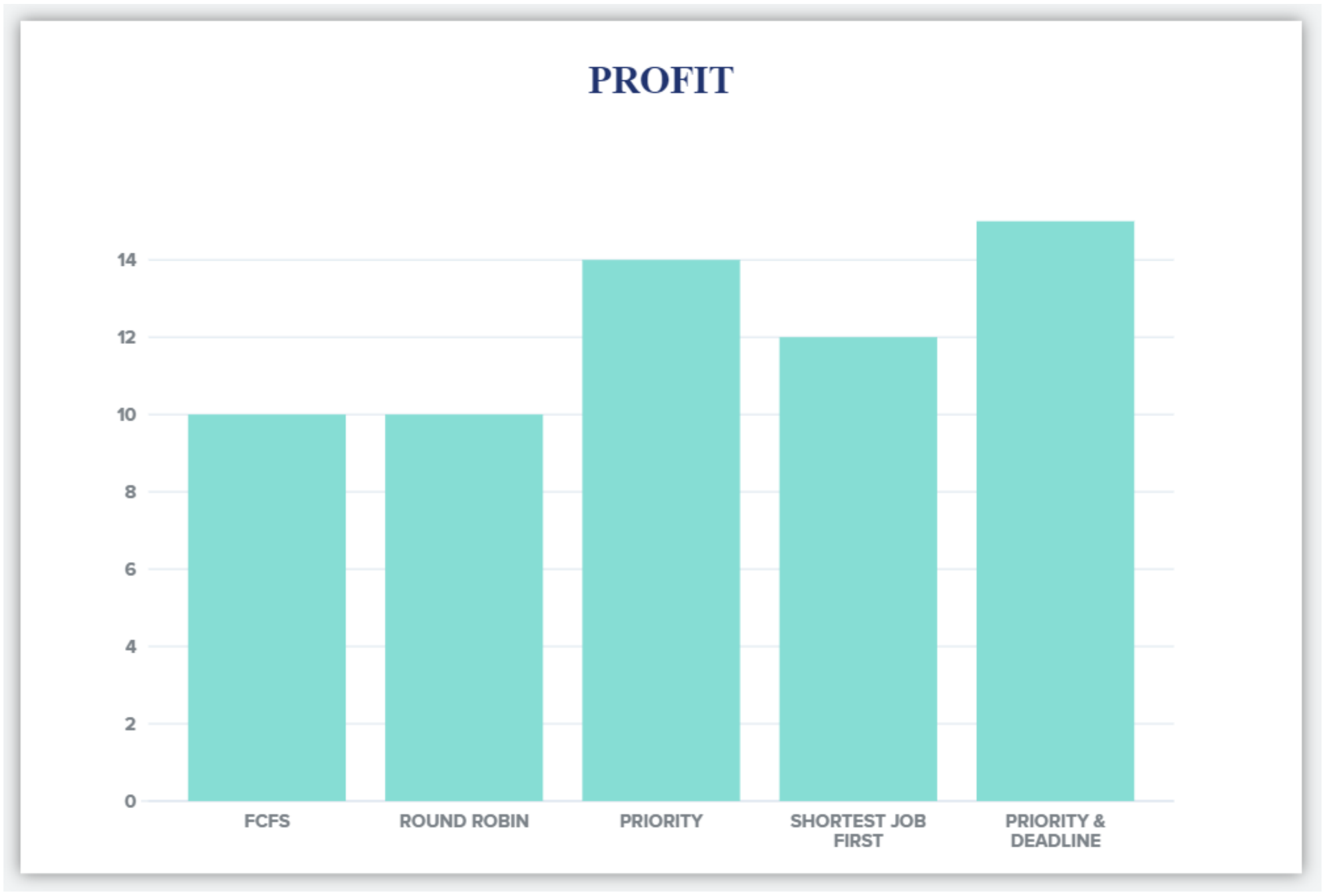
Framework, Architecture or Modules of the Proposed System:

Requirements: - C++
- C++ compiler (Xcode or g++ or gcc)
- Stdio.h
- Stdlib.h
- Stdbool.h
- Iostream
- Terminal

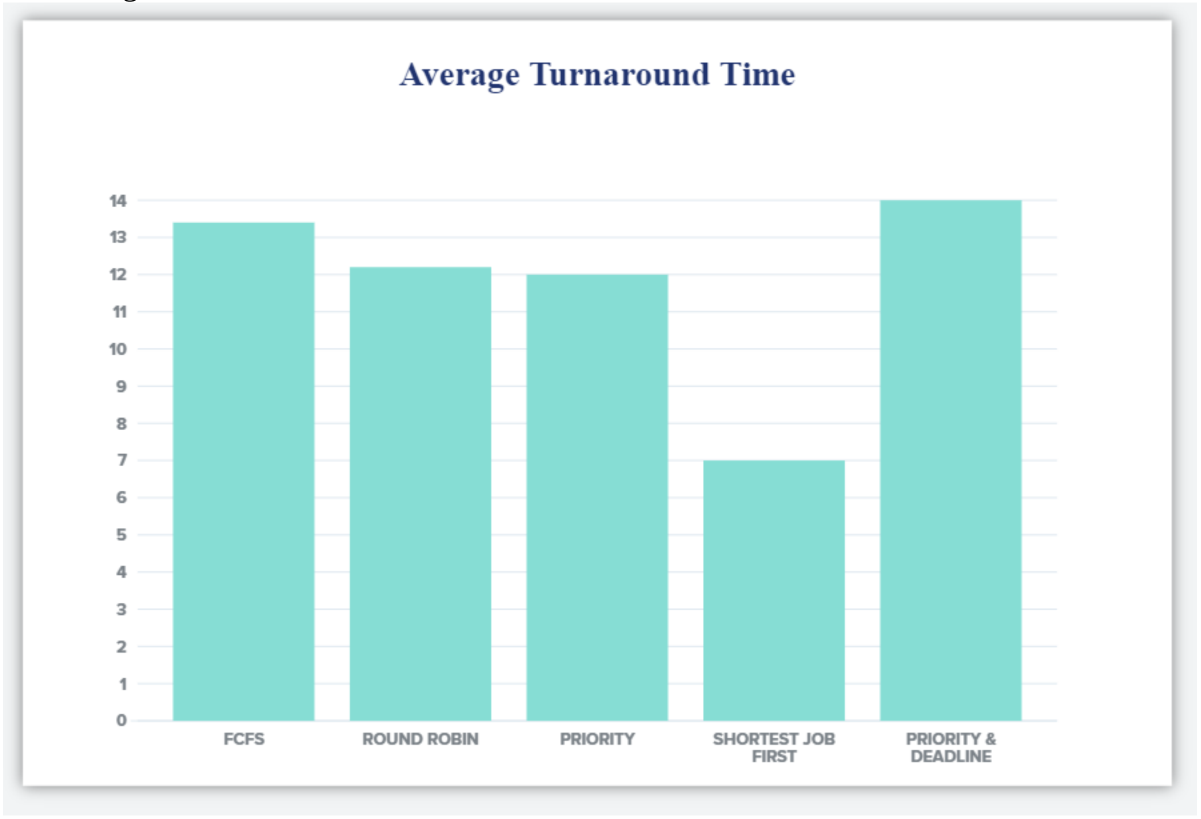
Proposed System Model :

After Analysation the following models are obtained:

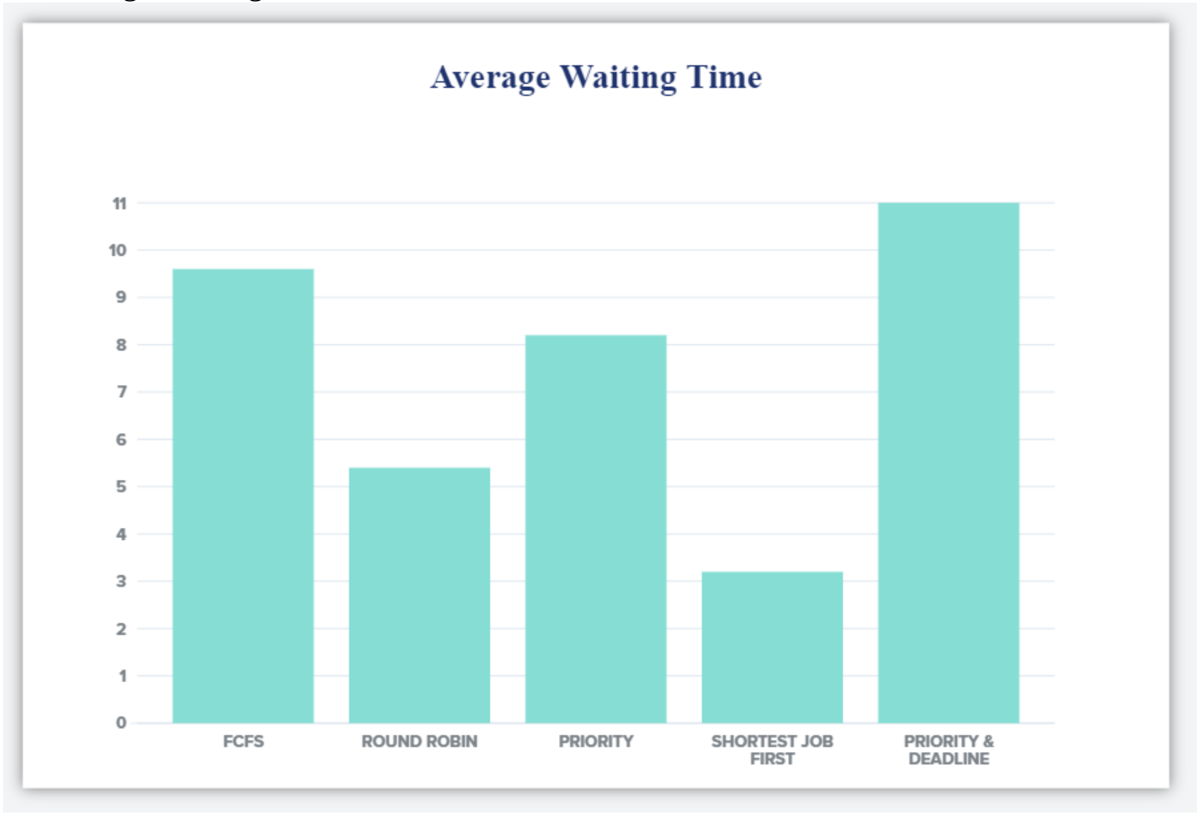
1. Profit earned by each algorithm (Profit is the priorities of the processes added up):



2. Average Turn Around Time :



3. Average Waiting Time



IV. PROPOSED SYSTEM ANALYSIS AND DESIGN

The Priority and Deadline Algorithm attempts to provide a realistic deadline to the processes to avoid starvation of processes with smaller priorities

Firstly, the user enters the data for all the processes, this data is then sorted according to the arrival time in the arrival_queue to simulate a cpu time cycle. Once we begin the cpu time cycle simulation we load the processes into the priority queue according to the arrival time.

Once these processes are loaded in the priority queue, they are sorted according to the priority, the priority queue is traversed and the rank and eligibility of every process in the priority queue is decided by the algorithm by using their deadlines.

This is done by imagining a time chart (a process array) with 100 blank spaces (cpu runs for 100 time units), and the processes are picked in descending priority and placed right before there deadline, if a process is already occupying that space, we try to accommodate the concerned process in any time frame before the deadline. If a suitable spot is found it is placed there, else the process is discarded for this time unit (it will be re-considered on the next time unit if with fortune it’s deadline doesn’t expire by that time)

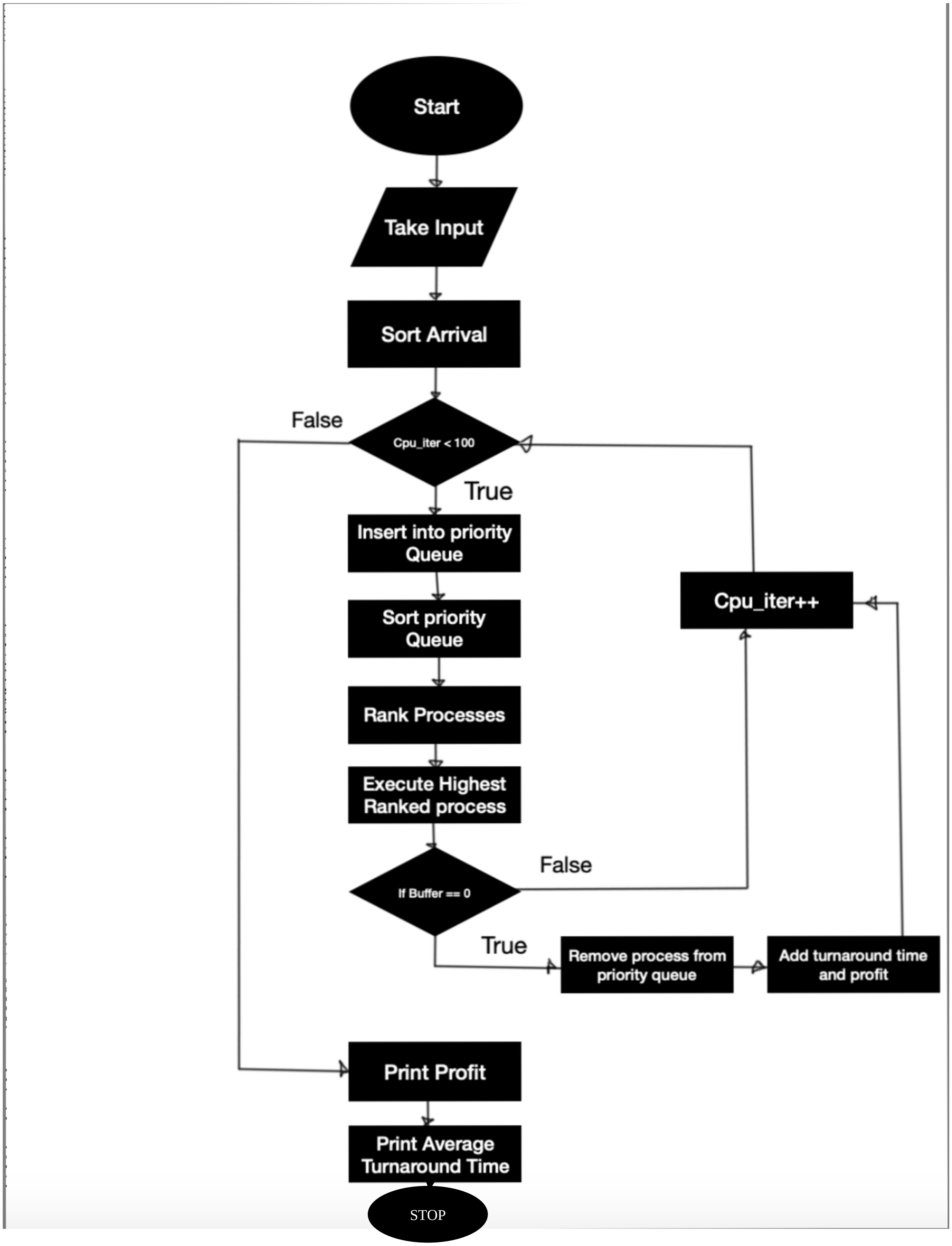
After deciding the appropriate process which is ranked the highest (most suitable for execution at this unit of time), that process is executed.

If the entire buffer of the process is finished, it is removed from the priority queue using the remove() function, and it’s turn around time is added to the total turnaround time, and the total number of process executed is incremented by 1. All this for later calculation of Average TurnAround time.

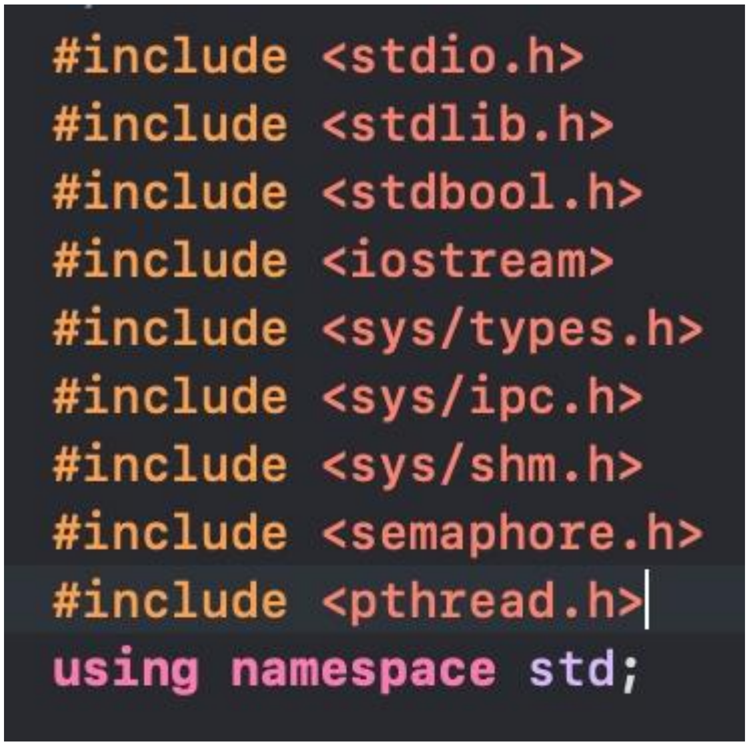
After this, the loop iterates again for the next unit of time, if any new processes arrive, they are added to the priority queue, which is then sorted again.

The imaginary time chart is refreshed entirely after every unit of time to accommodate new processes, and if no new processes join, it just continues in the next unit of time as it should.

V. DESIGN

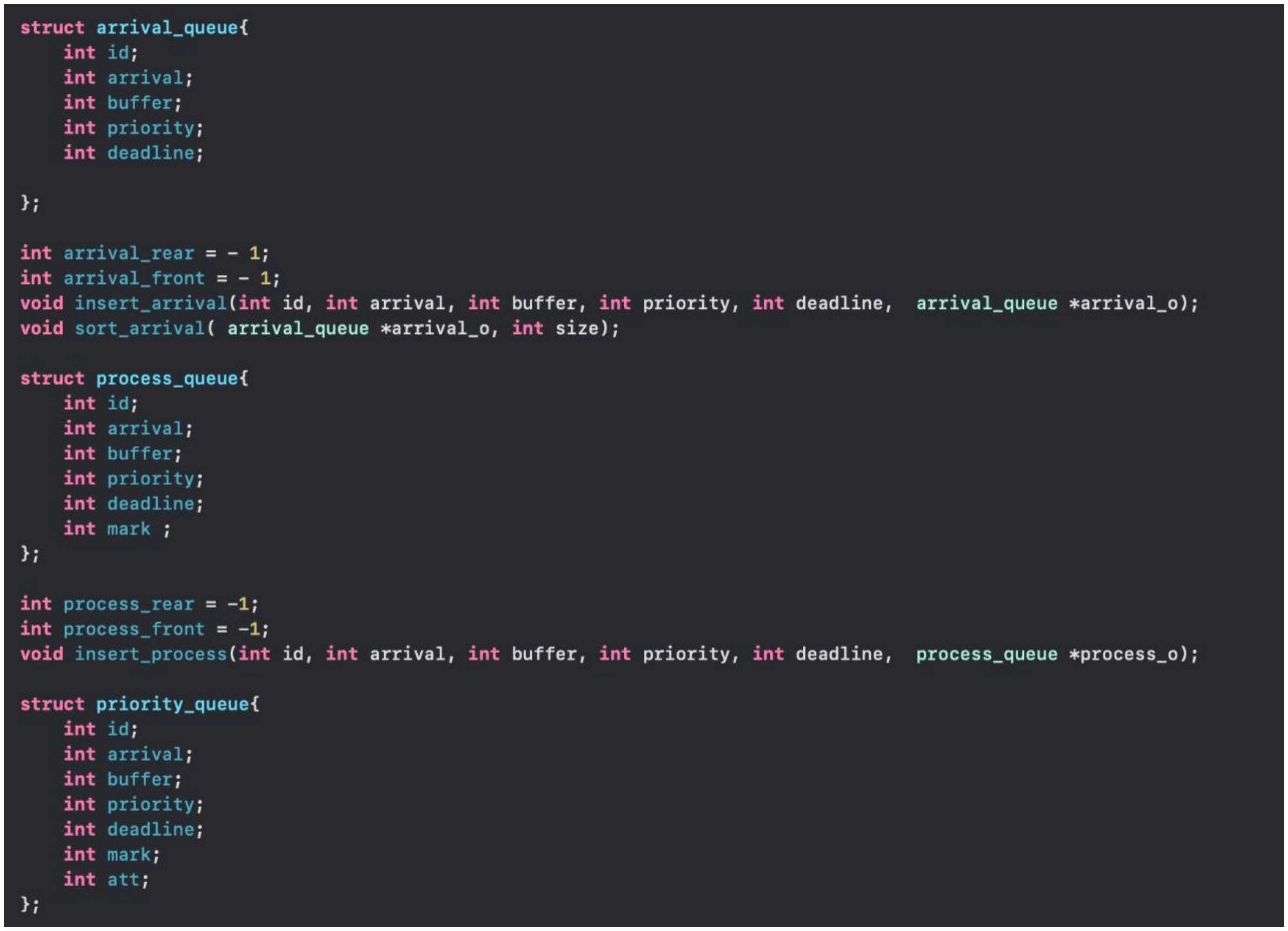

Start
Take Input
Sort Arrival
False
Cpu_iter < 100
True
Insert into priority Queue
Sort priority Queue
Rank Processes
Execute Highest Ranked process
Cpu_iter++
False
If Buffer == 0
True
Remove process from priority queue
Add turnaround time and profit
Print Profit
Print Average Turnaround Time
STOP

VI. IMPLEMENTATION



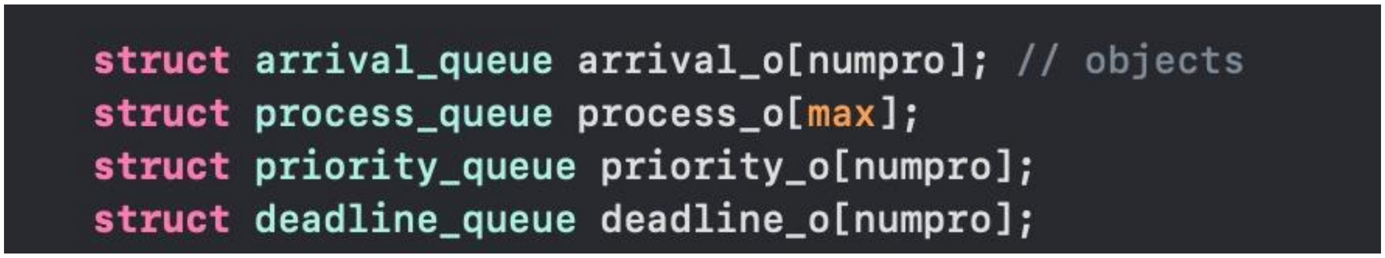
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <iostream>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <pthread.h>
using namespace std;
```

First Import all the require libraries



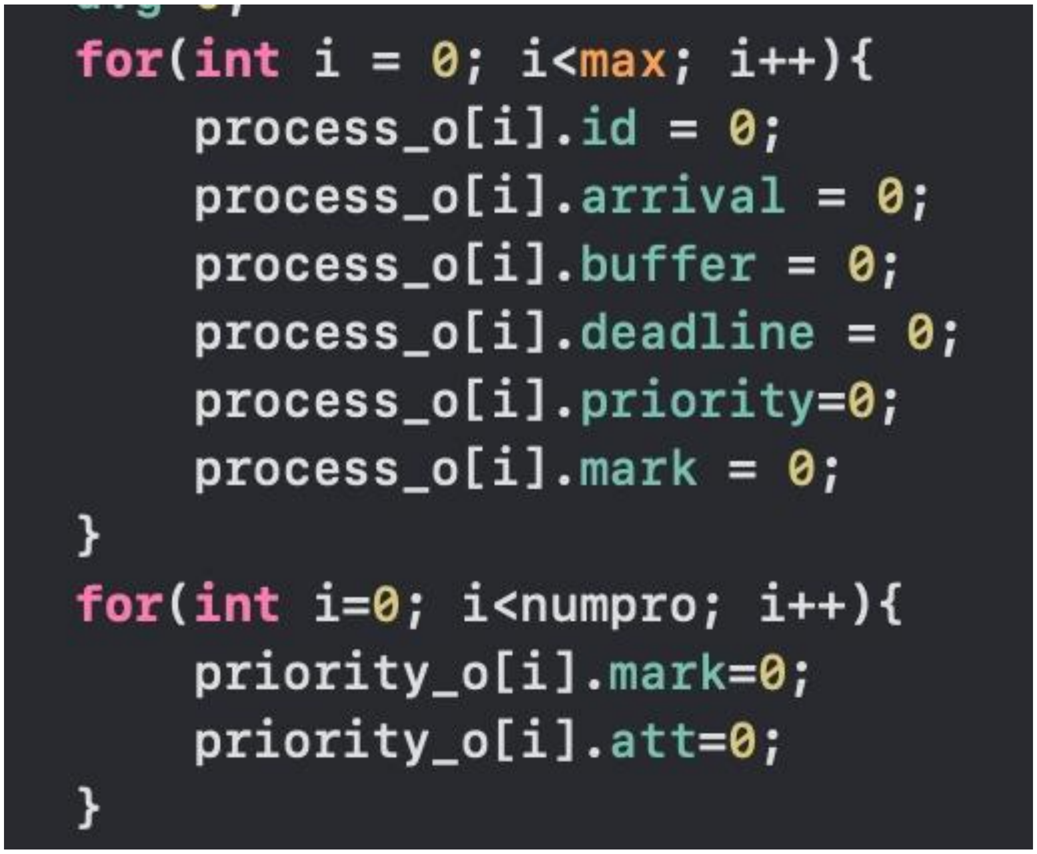
```
struct arrival_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;

};

int arrival_rear = - 1;
int arrival_front = - 1;
void insert_arrival(int id, int arrival, int buffer, int priority, int deadline,  arrival_queue *arrival_o);
void sort_arrival( arrival_queue *arrival_o, int size);

struct process_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int mark ;
};

int process_rear = -1;
int process_front = -1;
void insert_process(int id, int arrival, int buffer, int priority, int deadline,  process_queue *process_o);

struct priority_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int mark;
    int att;
};
```

Create necessary structures for all the queues -> arrival, process, priority, and define necessary insert, remove, sort and display functions as needed



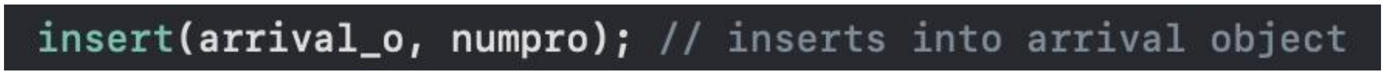
```
    struct arrival_queue arrival_o[numpro]; // objects
    struct process_queue process_o[max];
    struct priority_queue priority_o[numpro];
    struct deadline_queue deadline_o[numpro];
```

Create array objects for all the structures



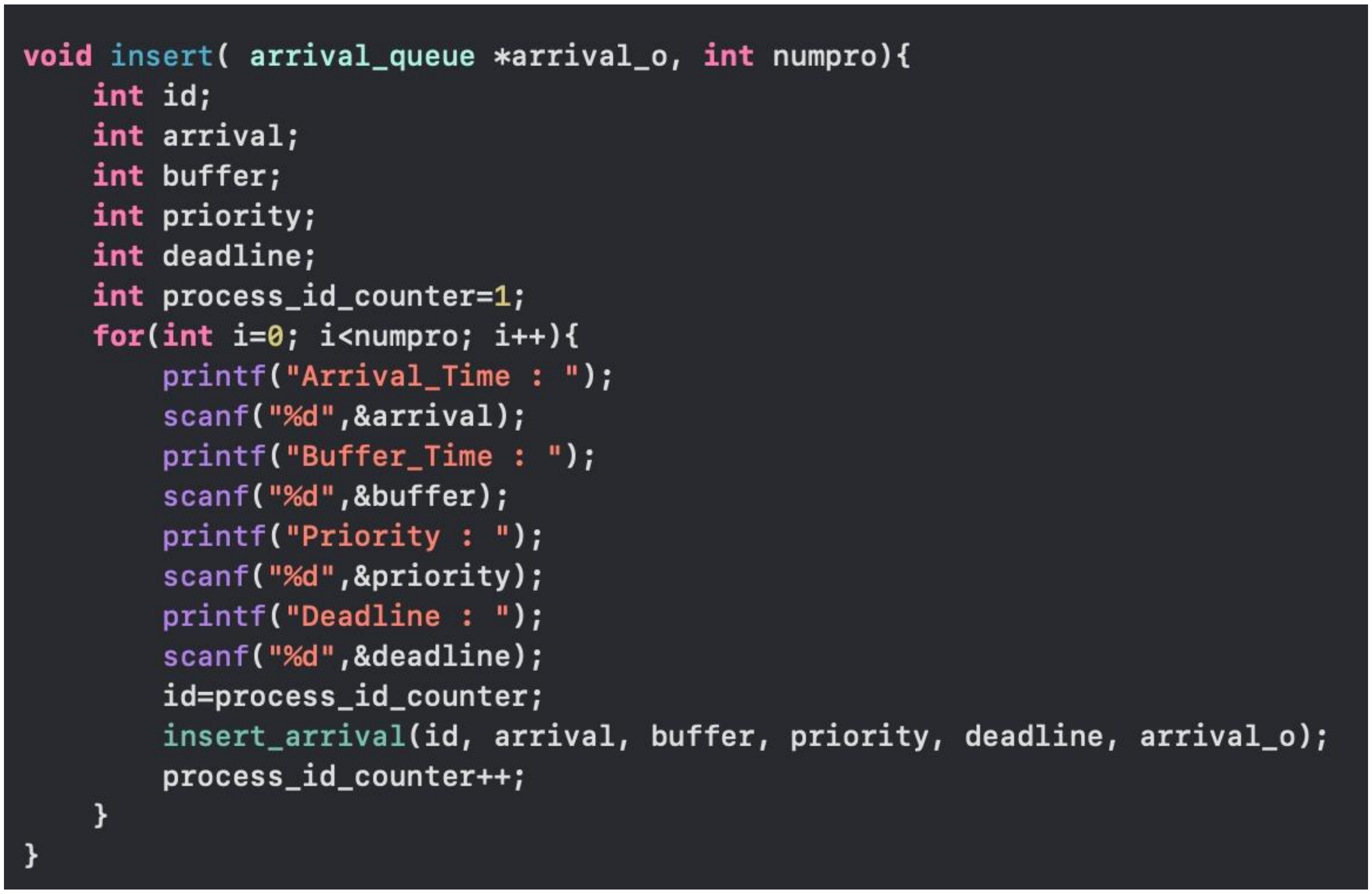
```
for(int i = 0; i<max; i++){
    process_o[i].id = 0;
    process_o[i].arrival = 0;
    process_o[i].buffer = 0;
    process_o[i].deadline = 0;
    process_o[i].priority=0;
    process_o[i].mark = 0;
}
for(int i=0; i<numpro; i++){
    priority_o[i].mark=0;
    priority_o[i].att=0;
}
```

Initialize as per requirement



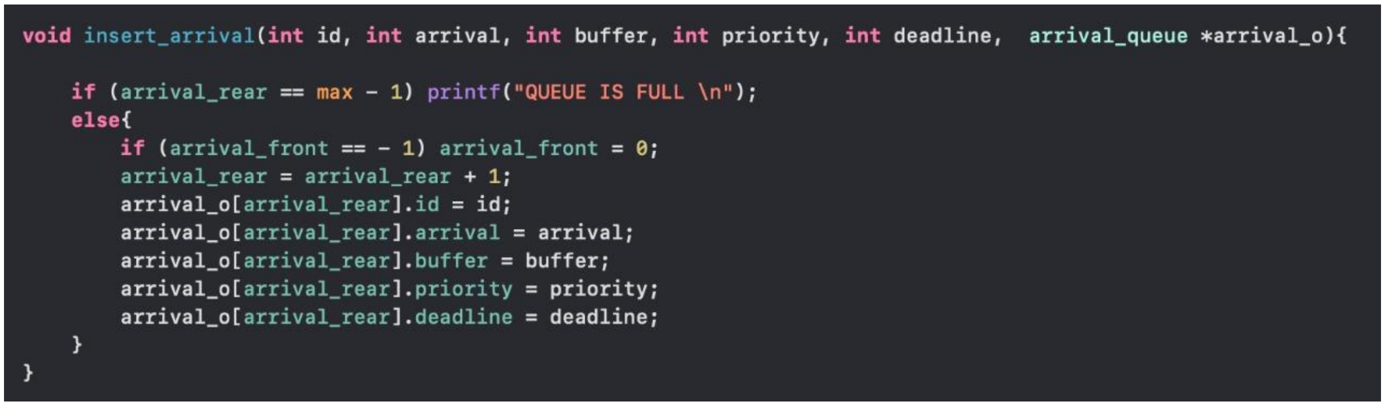
```
insert(arrival_o, numpro); // inserts into arrival object
```

Call the insert function to take input



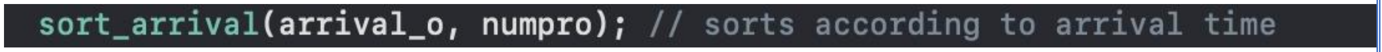
```c
void insert( arrival_queue *arrival_o, int numpro){
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int process_id_counter=1;
    for(int i=0; i<numpro; i++){
        printf("Arrival_Time : ");
        scanf("%d",&arrival);
        printf("Buffer_Time : ");
        scanf("%d",&buffer);
        printf("Priority : ");
        scanf("%d",&priority);
        printf("Deadline : ");
        scanf("%d",&deadline);
        id=process_id_counter;
        insert_arrival(id, arrival, buffer, priority, deadline, arrival_o);
        process_id_counter++;
    }
}
```

Insert Function takes input and calls insert_arrival to insert the input into arrival_queue



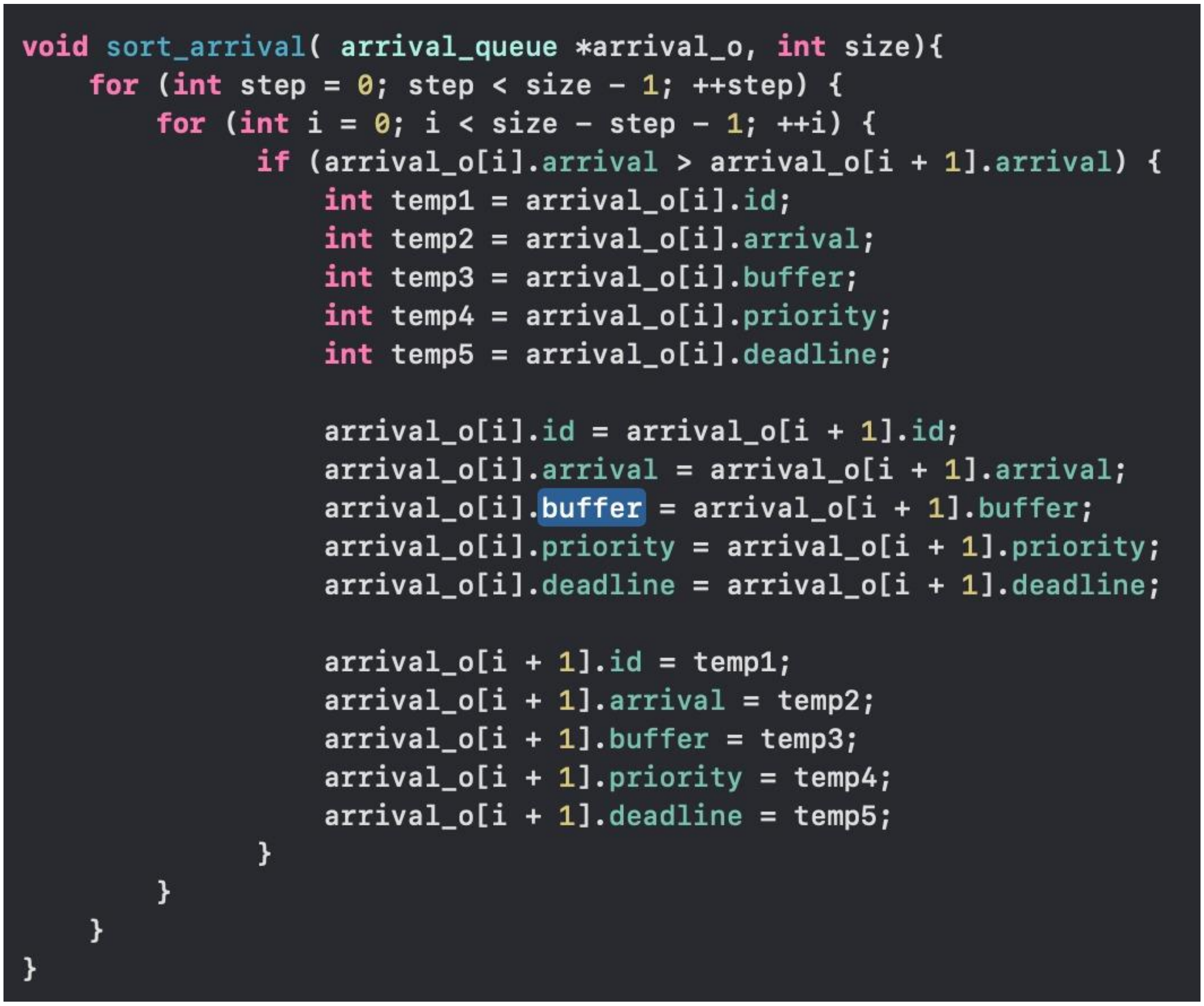
```c
void insert_arrival(int id, int arrival, int buffer, int priority, int deadline,  arrival_queue *arrival_o){

    if (arrival_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (arrival_front == - 1) arrival_front = 0;
        arrival_rear = arrival_rear + 1;
        arrival_o[arrival_rear].id = id;
        arrival_o[arrival_rear].arrival = arrival;
        arrival_o[arrival_rear].buffer = buffer;
        arrival_o[arrival_rear].priority = priority;
        arrival_o[arrival_rear].deadline = deadline;
    }
}
```

Insert Arrival function inserts all the required attributes into the arrival queue



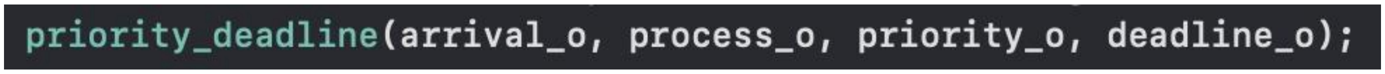
```
sort_arrival(arrival_o, numpro); // sorts according to arrival time
```

BACK IN MAIN FUNCTION → SORT THE ARRIVAL QUEUE



```
void sort_arrival( arrival_queue *arrival_o, int size){
    for (int step = 0; step < size - 1; ++step) {
        for (int i = 0; i < size - step - 1; ++i) {
              if (arrival_o[i].arrival > arrival_o[i + 1].arrival) {
                  int temp1 = arrival_o[i].id;
                  int temp2 = arrival_o[i].arrival;
                  int temp3 = arrival_o[i].buffer;
                  int temp4 = arrival_o[i].priority;
                  int temp5 = arrival_o[i].deadline;

                  arrival_o[i].id = arrival_o[i + 1].id;
                  arrival_o[i].arrival = arrival_o[i + 1].arrival;
                  arrival_o[i].buffer = arrival_o[i + 1].buffer;
                  arrival_o[i].priority = arrival_o[i + 1].priority;
                  arrival_o[i].deadline = arrival_o[i + 1].deadline;

                  arrival_o[i + 1].id = temp1;
                  arrival_o[i + 1].arrival = temp2;
                  arrival_o[i + 1].buffer = temp3;
                  arrival_o[i + 1].priority = temp4;
                  arrival_o[i + 1].deadline = temp5;
              }
        }
    }
}
```

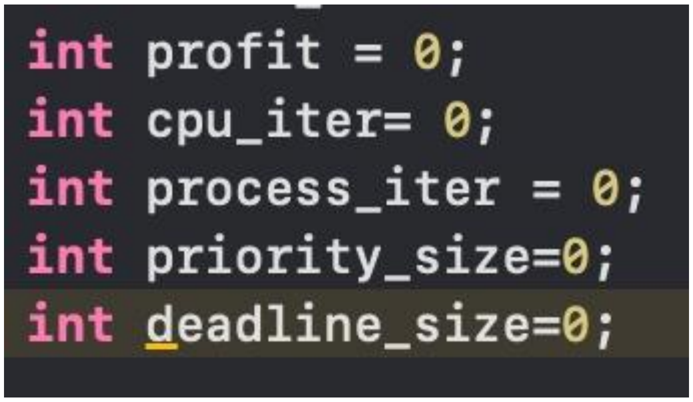
Sort Arrival Function



```
priority_deadline(arrival_o, process_o, priority_o, deadline_o);
```

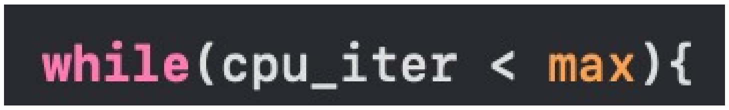
CALL THE PRIORITY_DEADLINE FUNCTION

INSIDE THE PRIORITY DEADLINE FUNCTION



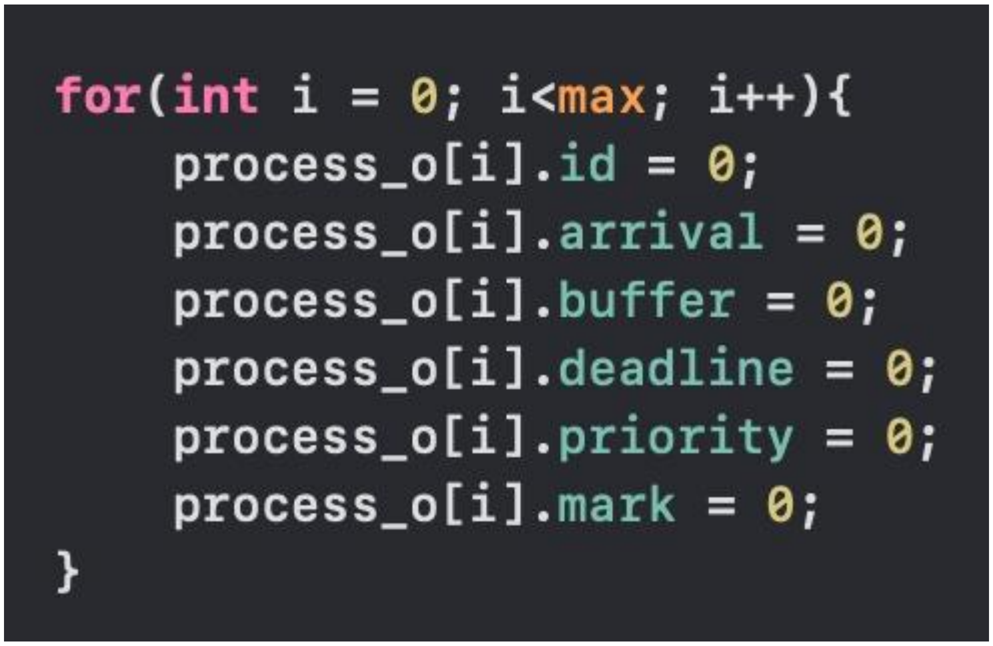
```
int profit = 0;
int cpu_iter= 0;
int process_iter = 0;
int priority_size=0;
int deadline_size=0;
```

DECLARING SOME IMPORTANT VARIABLES



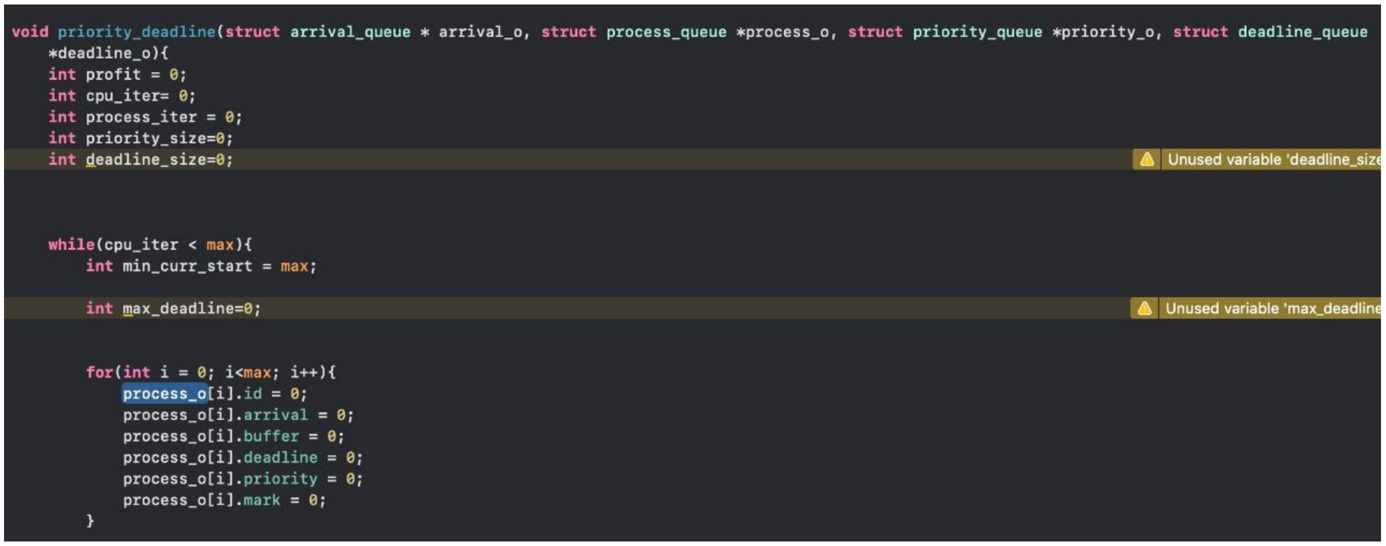
```
while(cpu_iter < max){
```

BEGIN A WHILE LOOP WHICH SIMULATES CPU EXECUTION



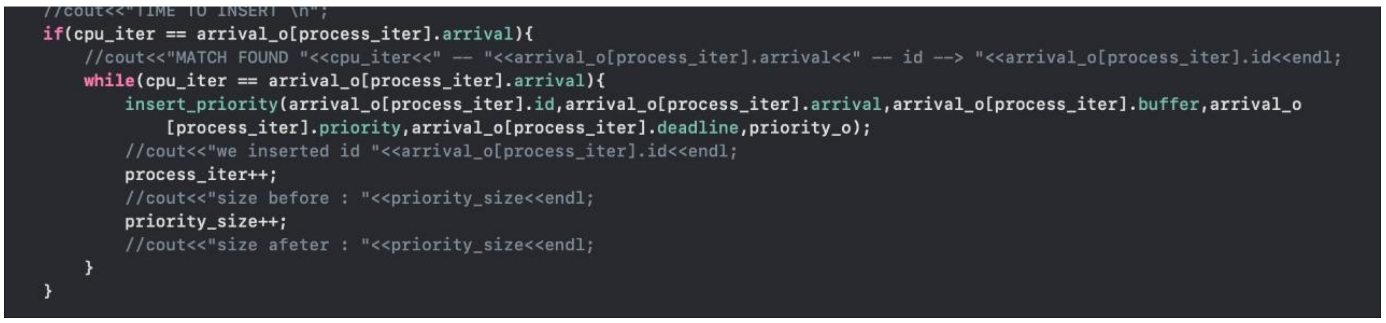
```
for(int i = 0; i<max; i++){
    process_o[i].id = 0;
    process_o[i].arrival = 0;
    process_o[i].buffer = 0;
    process_o[i].deadline = 0;
    process_o[i].priority = 0;
    process_o[i].mark = 0;
}
```

REFRESHING THE PROCESS QUEUE (IMAGINARY TIME CHART)



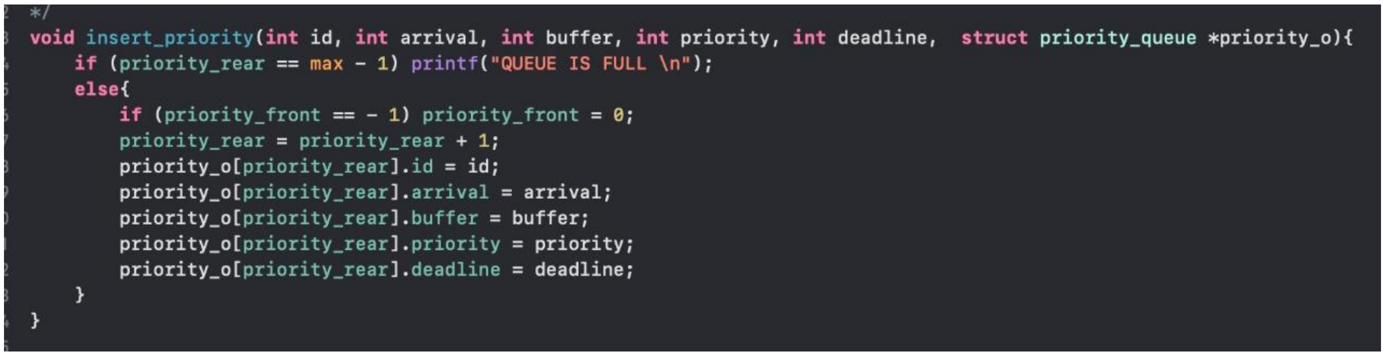
```
void priority_deadline(struct arrival_queue * arrival_o, struct process_queue *process_o, struct priority_queue *priority_o, struct deadline_queue
    *deadline_o){
    int profit = 0;
    int cpu_iter= 0;
    int process_iter = 0;
    int priority_size=0;
    int deadline_size=0;                                                        ⚠ Unused variable 'deadline_size



    while(cpu_iter < max){
        int min_curr_start = max;

        int max_deadline=0;                                                     ⚠ Unused variable 'max_deadline


        for(int i = 0; i<max; i++){
            process_o[i].id = 0;
            process_o[i].arrival = 0;
            process_o[i].buffer = 0;
            process_o[i].deadline = 0;
            process_o[i].priority = 0;
            process_o[i].mark = 0;
        }
```

ZOOM OUT PHOTO OF THE WHILE LOOP



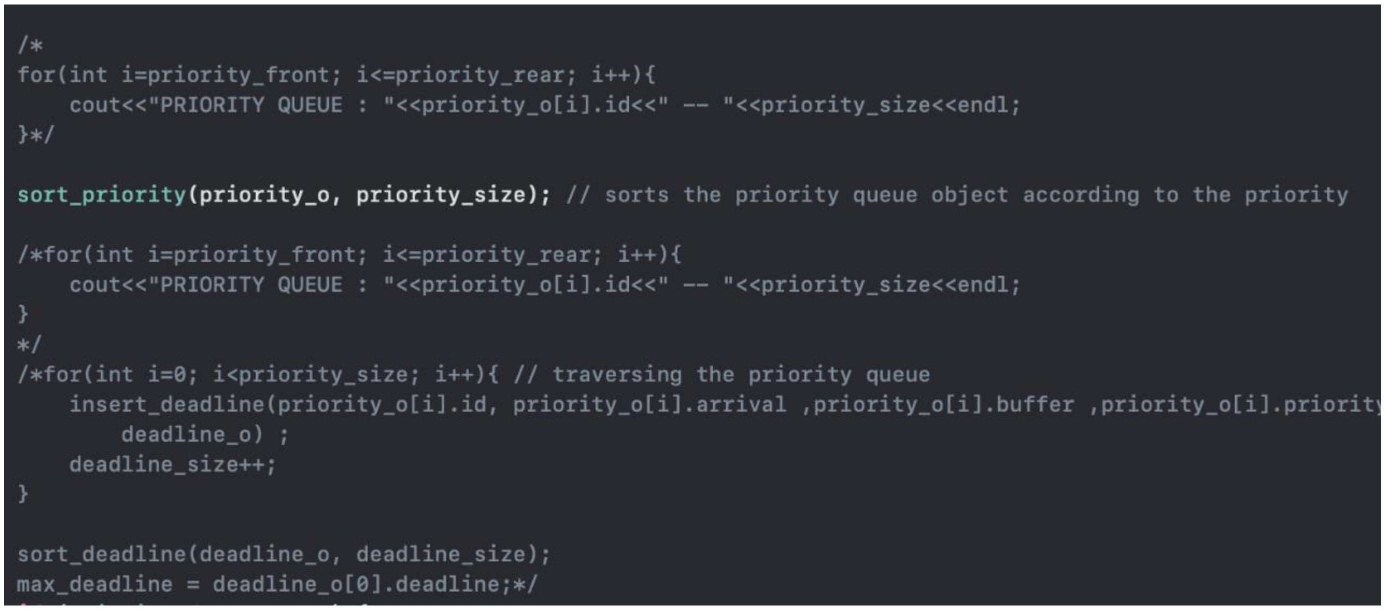
```
    //cout<<"TIME TO INSERT \n";
    if(cpu_iter == arrival_o[process_iter].arrival){
        //cout<<"MATCH FOUND "<<cpu_iter<<" -- "<<arrival_o[process_iter].arrival<<" -- id --> "<<arrival_o[process_iter].id<<endl;
        while(cpu_iter == arrival_o[process_iter].arrival){
            insert_priority(arrival_o[process_iter].id,arrival_o[process_iter].arrival,arrival_o[process_iter].buffer,arrival_o
                [process_iter].priority,arrival_o[process_iter].deadline,priority_o);
            //cout<<"we inserted id "<<arrival_o[process_iter].id<<endl;
            process_iter++;
            //cout<<"size before : "<<priority_size<<endl;
            priority_size++;
            //cout<<"size afeter : "<<priority_size<<endl;
        }
    }
```

INSERT PROCESS INTO PRIORITY QUEUE ACCORDING TO ARRIVAL TIME



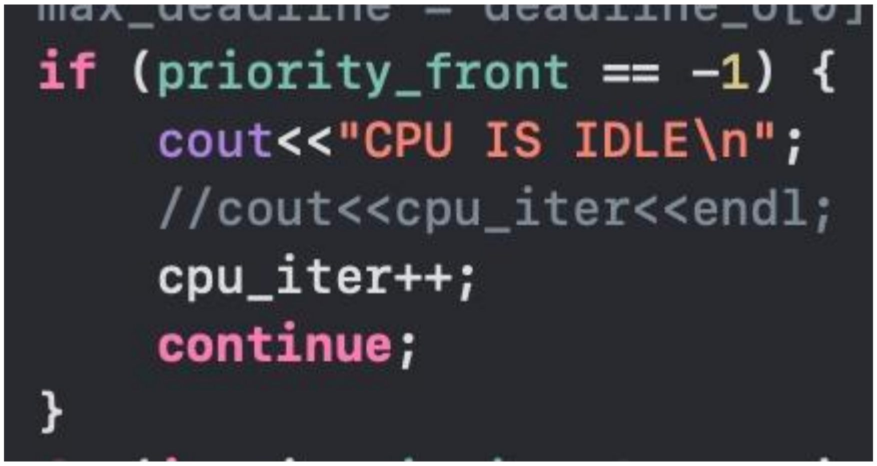
```
*/
void insert_priority(int id, int arrival, int buffer, int priority, int deadline,  struct priority_queue *priority_o){
    if (priority_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (priority_front == - 1) priority_front = 0;
        priority_rear = priority_rear + 1;
        priority_o[priority_rear].id = id;
        priority_o[priority_rear].arrival = arrival;
        priority_o[priority_rear].buffer = buffer;
        priority_o[priority_rear].priority = priority;
        priority_o[priority_rear].deadline = deadline;
    }
}
```

INSERT PRIORITY FUNCTION



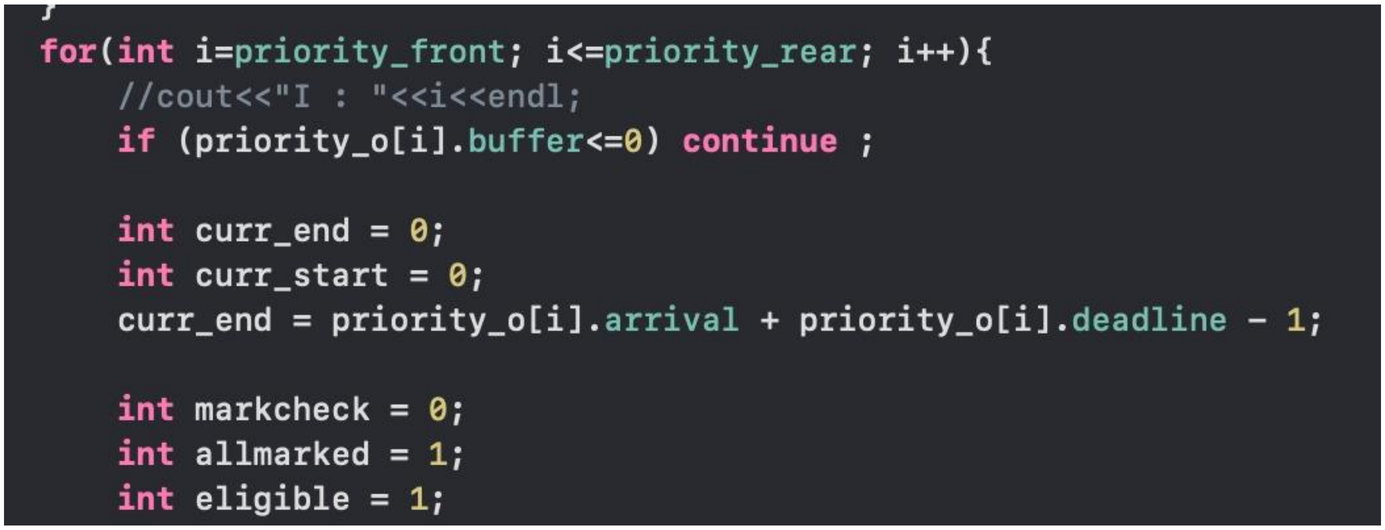
```
/*
for(int i=priority_front; i<=priority_rear; i++){
    cout<<"PRIORITY QUEUE : "<<priority_o[i].id<<" -- "<<priority_size<<endl;
}*/

sort_priority(priority_o, priority_size); // sorts the priority queue object according to the priority

/*for(int i=priority_front; i<=priority_rear; i++){
    cout<<"PRIORITY QUEUE : "<<priority_o[i].id<<" -- "<<priority_size<<endl;
}
*/
/*for(int i=0; i<priority_size; i++){ // traversing the priority queue
    insert_deadline(priority_o[i].id, priority_o[i].arrival ,priority_o[i].buffer ,priority_o[i].priority
        deadline_o) ;
    deadline_size++;
}

sort_deadline(deadline_o, deadline_size);
max_deadline = deadline_o[0].deadline;*/
```

SORT PRIORITY FUNCTION AND DEBUGGING



```
if (priority_front == -1) {
    cout<<"CPU IS IDLE\n";
    //cout<<cpu_iter<<endl;
    cpu_iter++;
    continue;
}
```

IN CASE THE PRIORITY QUEUE IS EMPTY CONTINUE WITH NEXT TIME UNIT

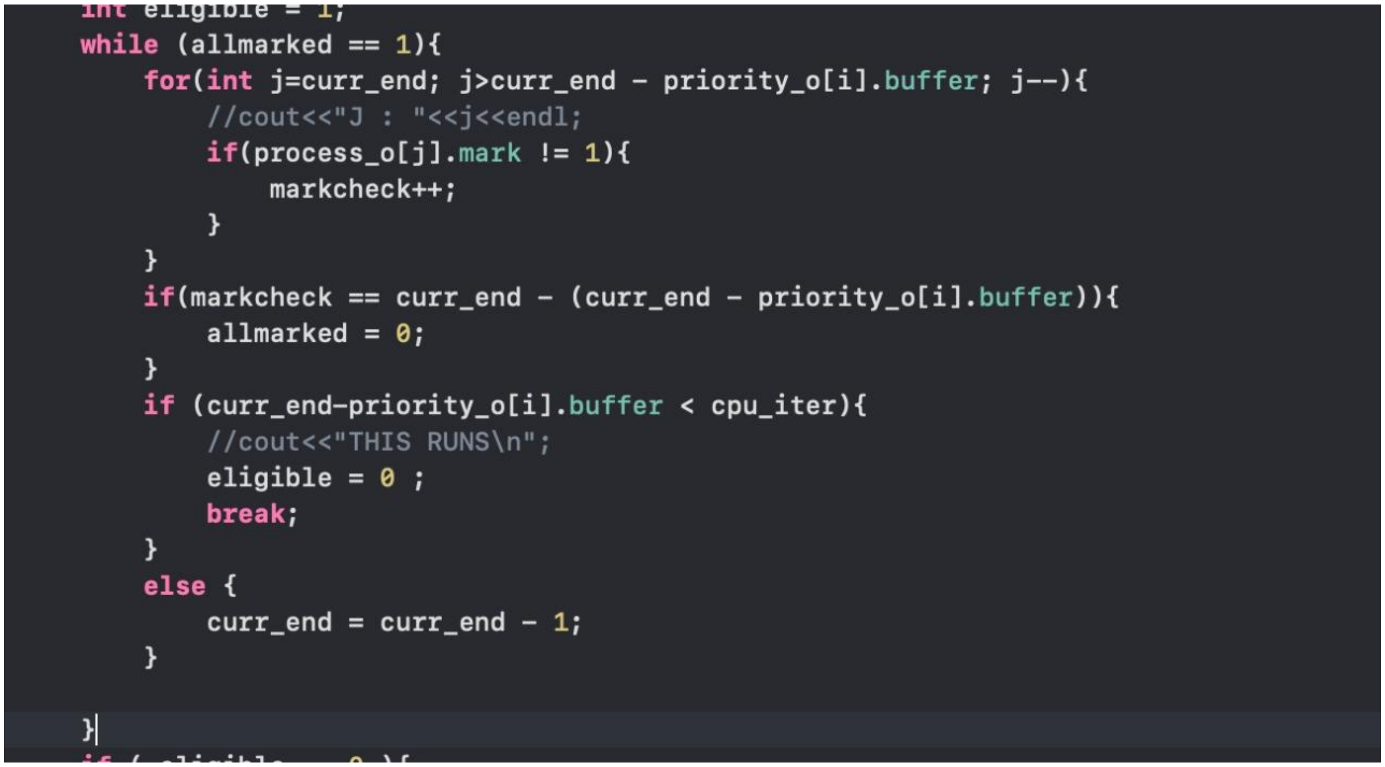


```
}
for(int i=priority_front; i<=priority_rear; i++){
    //cout<<"I : "<<i<<endl;
    if (priority_o[i].buffer<=0) continue ;

    int curr_end = 0;
    int curr_start = 0;
    curr_end = priority_o[i].arrival + priority_o[i].deadline - 1;

    int markcheck = 0;
    int allmarked = 1;
    int eligible = 1;
```

TRAVERSING THE PRIORITY QUEUE.

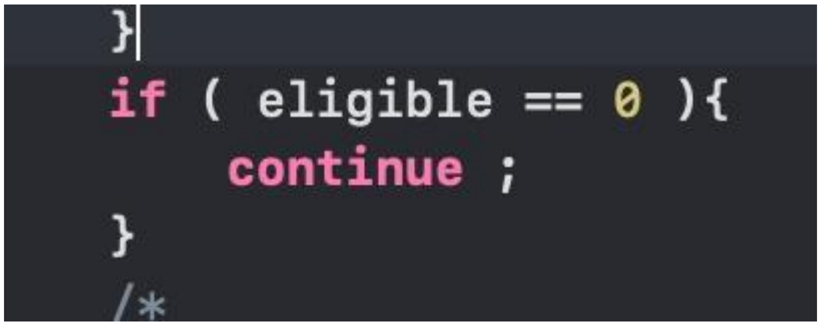
IF STATEMENT FOR THE CASE WHERE BUFFER IS 0

CURR_END IS USED FOR MARKING THE AREA OF THE PROCESS IN THE IMAGINARY TIME CHART (PROCESS QUEUE)



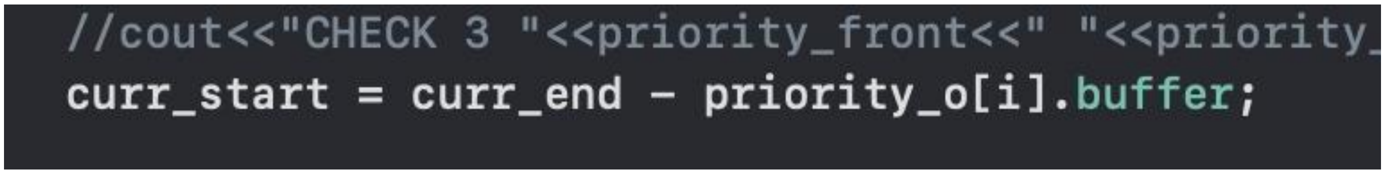
```
    int eligible = 1;
    while (allmarked == 1){
        for(int j=curr_end; j>curr_end - priority_o[i].buffer; j--){
            //cout<<"J : "<<j<<endl;
            if(process_o[j].mark != 1){
                markcheck++;
            }
        }
        if(markcheck == curr_end - (curr_end - priority_o[i].buffer)){
            allmarked = 0;
        }
        if (curr_end-priority_o[i].buffer < cpu_iter){
            //cout<<"THIS RUNS\n";
            eligible = 0 ;
            break;
        }
        else {
            curr_end = curr_end - 1;
        }

    }
    if ( eligible == 0 ){
```

WHILE LOOP CHECKS IF ALL THE SLOTS REQUIRED FOR THE PROCESS ARE FREE AND NOT TAKEN BY ANY OTHER PROCESS. IF THE SLOTS ARE FREE THEN THEN FURTHER PROGRAM IS EXECUTED, IF NO SLOT IS FREE FOR CURRENT CURR_END, CURR_END IS DECREMENTED, AND IF CURR_END GOES BEYOND (BEFORE) THE CURRENT CPU_ITER, IT IS DETERMINED THAT THIS PROCESS CANNOT BE ACCOMODATED AT THIS UNIT OF TIME, AND WILL BE CONSIDERED LATER



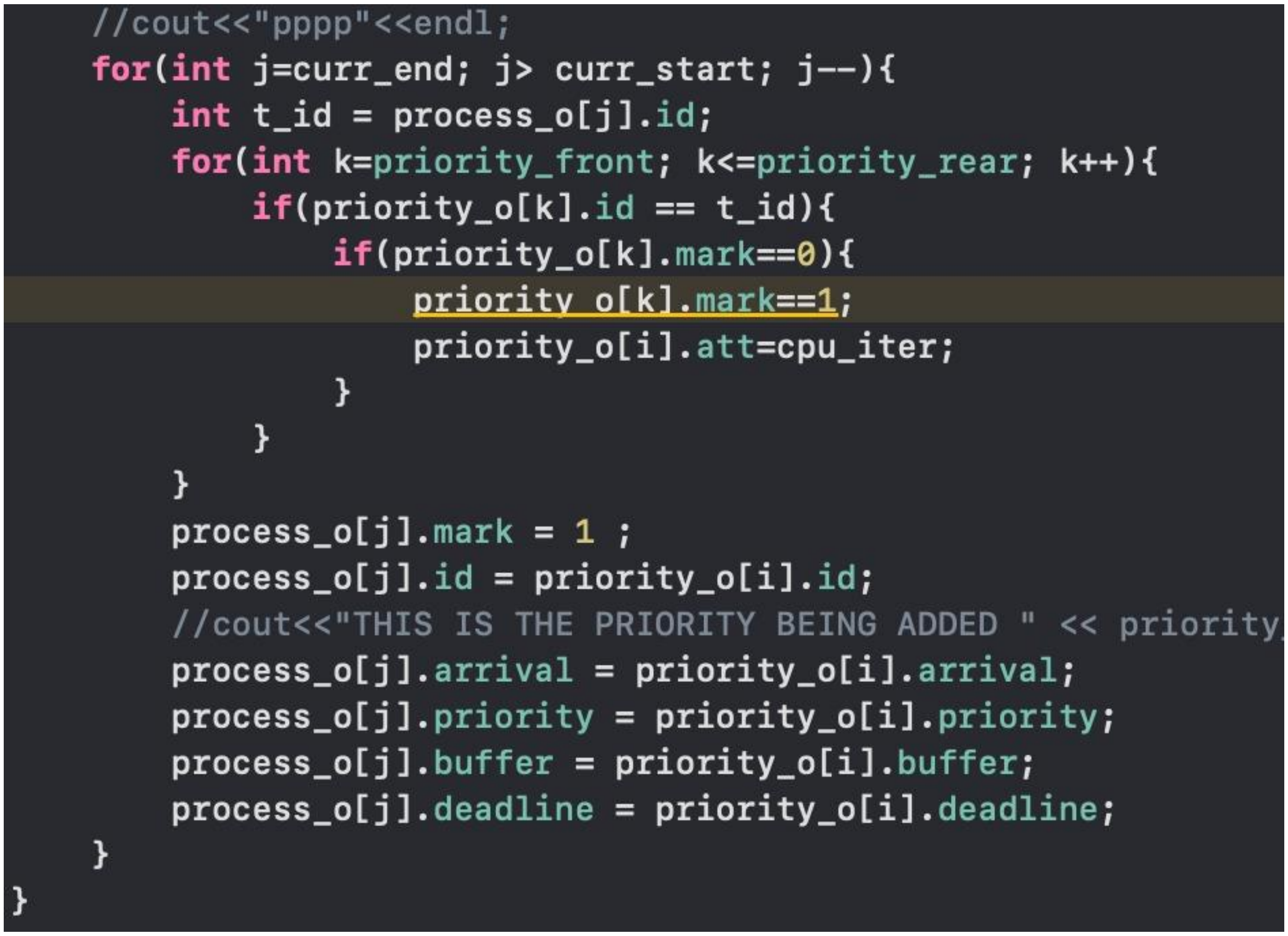
```
}
if ( eligible == 0 ){
    continue ;
}
/*
```

CONTINUING FROM THE PREVIOUS WHILE LOOP, IF THE PROCESS IS NOT ELIGIBLE (NO SLOTS ARE AVAILABLE), IT IS SKIPPED FOR THIS TIME UNIT AND WILL BE CONSIDERED LATER IF TIME PERMITS



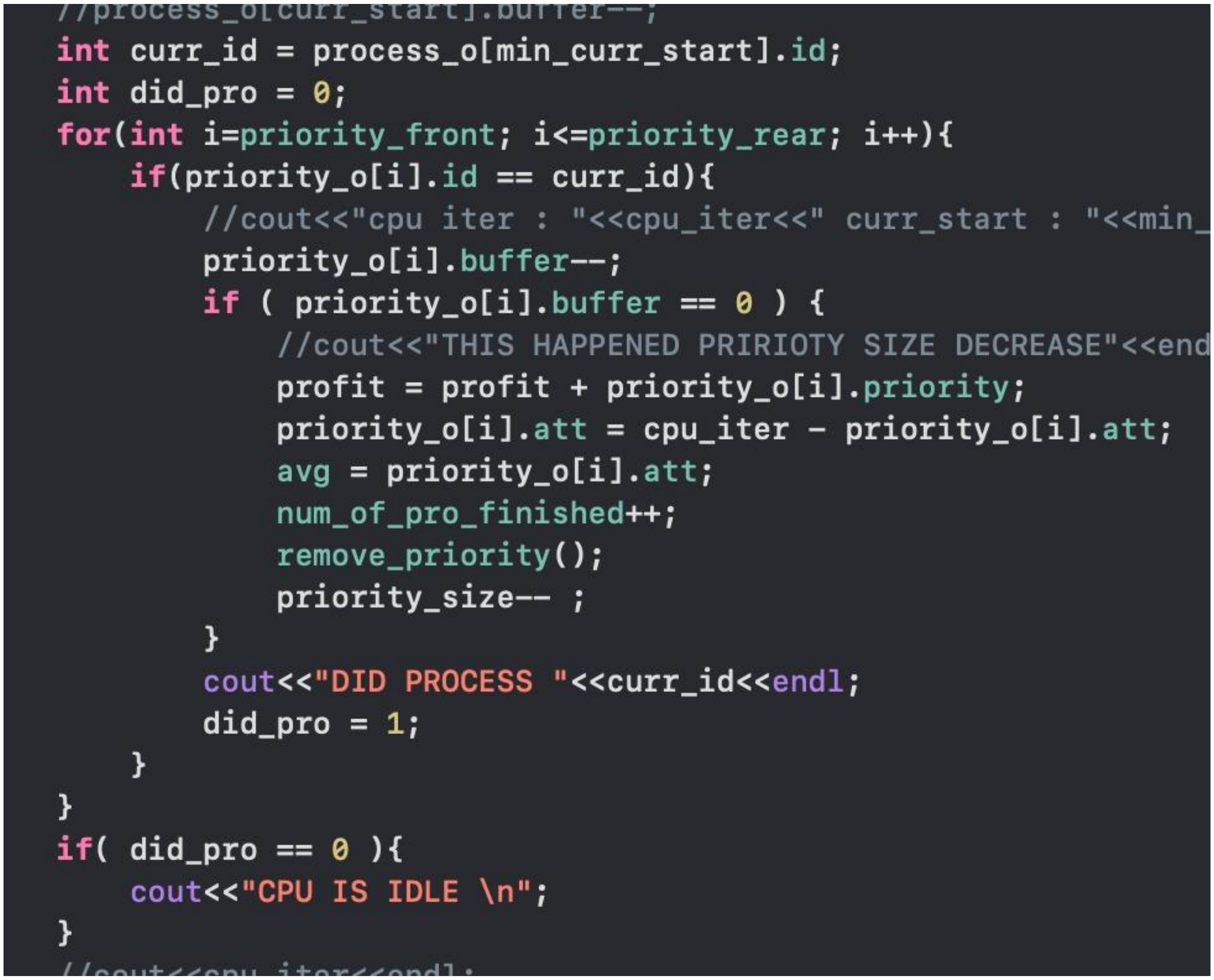
```
//cout<<"CHECK 3 "<<priority_front<<" "<<priority_
curr_start = curr_end - priority_o[i].buffer;
```

A CURR_START IS DEFINED FOR THE CURR_END TO DEFINE A RANGE FOR THE ENTIRE BUFFER FOR THE REQUIRE PROCESS



```
    //cout<<"pppp"<<endl;
    for(int j=curr_end; j> curr_start; j--){
        int t_id = process_o[j].id;
        for(int k=priority_front; k<=priority_rear; k++){
            if(priority_o[k].id == t_id){
                if(priority_o[k].mark==0){
                    priority_o[k].mark==1;
                    priority_o[i].att=cpu_iter;
                }
            }
        }
        process_o[j].mark = 1 ;
        process_o[j].id = priority_o[i].id;
        //cout<<"THIS IS THE PRIORITY BEING ADDED " << priority
        process_o[j].arrival = priority_o[i].arrival;
        process_o[j].priority = priority_o[i].priority;
        process_o[j].buffer = priority_o[i].buffer;
        process_o[j].deadline = priority_o[i].deadline;
    }
}
```

THE PROCESS QUEUE IS TRAVERSED, AND THE SLOTS ARE FILLED WITH THE CURRENT PROCESS.



```
    int curr_id = process_o[min_curr_start].id;
    int did_pro = 0;
    for(int i=priority_front; i<=priority_rear; i++){
        if(priority_o[i].id == curr_id){
            //cout<<"cpu iter : "<<cpu_iter<<" curr_start : "<<min_
            priority_o[i].buffer--;
            if ( priority_o[i].buffer == 0 ) {
                //cout<<"THIS HAPPENED PRIRIOTY SIZE DECREASE"<<end
                profit = profit + priority_o[i].priority;
                priority_o[i].att = cpu_iter - priority_o[i].att;
                avg = priority_o[i].att;
                num_of_pro_finished++;
                remove_priority();
                priority_size-- ;
            }
            cout<<"DID PROCESS "<<curr_id<<endl;
            did_pro = 1;
        }
    }
    if( did_pro == 0 ){
        cout<<"CPU IS IDLE \n";
    }
```

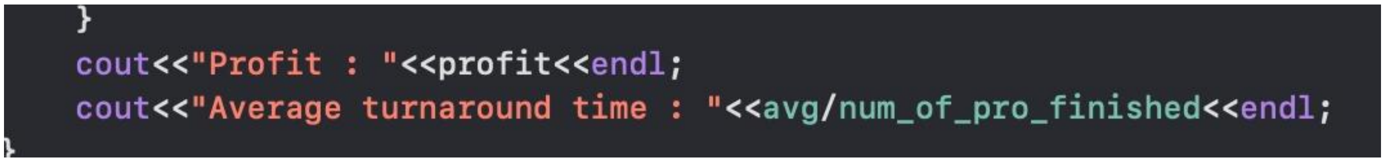
THE PROCESS WHICH IS RANKED THE HIGHEST IS IDENTIFIED, THE PRIORITY QUEUE IS TRAVERSED, THE CORRESPONDING BUFFER IS DECREMENTED, IF THE BUFFER IS ZERO “

THE PROCESS IS EJECTED FROM THE QUEUE

TURN AROUND TIME IS CALCULATED.

NUM OF PROCESSES EXECUTED IS ALSO INCREMENTED.

IF NO PROCESS AVAILABLE RIGHT NOW THEN IT PRINTS “CPU IS IDLE”

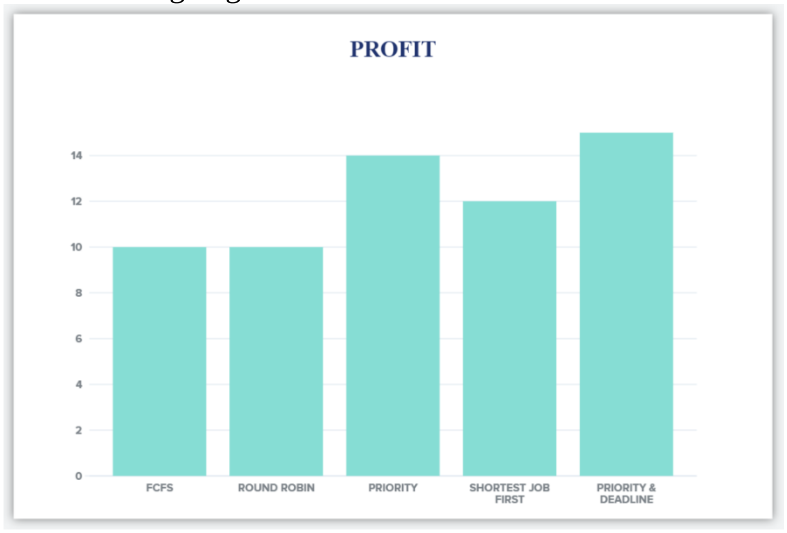


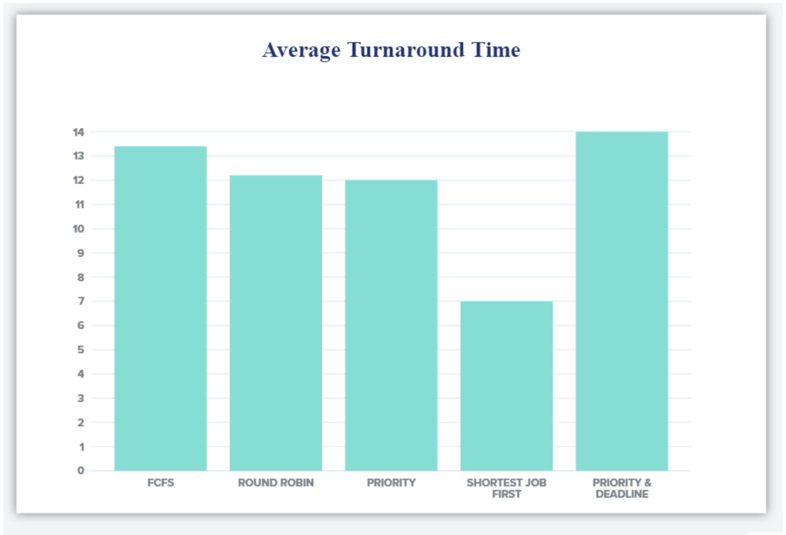
```
    }
    cout<<"Profit : "<<profit<<endl;
    cout<<"Average turnaround time : "<<avg/num_of_pro_finished<<endl;
}
```

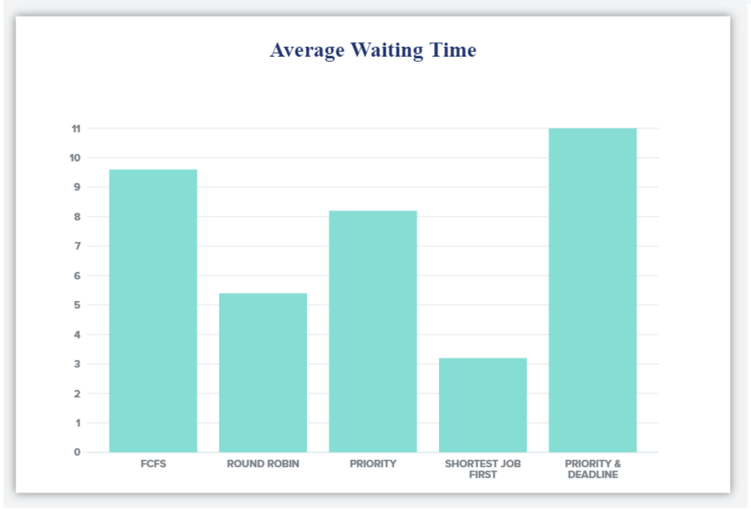
FINAL PROFIT AND AVERAGE TURNAOROUND ARE PRINTED

VII. RESULTS AND DISCUSSION

These are the graphs which compare the Priority and Deadline Algorithm with the several other CPU Scheduling Algorithms:-







As we can see, the Priority & Deadline Algorithm fetches us the highest profit (first graph). This is because it optimizes the priority and the deadline to create the best possible time table and or schedule to execute the processes.

The average turn around time, and the waiting time, however, is quite high for the Priority & Deadline Algorithm. This is a result of the extensively pre-emptive nature of this algorithm, that is, after beginning the execution, the process is set aside in favor of other processes for long periods of time. This is a sacrifice which has to be made for the optimized schedule.

VIII. CONCLUSION, LIMITATIONS AND SCOPE FOR FUTURE WORK

Finally, Priority & Deadline Algorithm allows for a system where the higher priority processes are paid attention to but by fitting them into a precise schedule which leaves time for the lower priority processes to be executed along with the higher priority processes all in moderation.

A limitation is that many of the processes with a lower priority and a shorter deadline get ignored because of their execution time being given to a process with a higher priority and a shorter deadline.
Another limitation is that sometimes due to the everchanging process queue (because new processes are added every unit of time) some processes which begin execution, never really finish execution because they get sidelined midway, and then as a result of their deadline expiring, they end up never finishing execution. This means that all the units of time which were spent executing a part of that process were a waste.

In future work, if we could form a system which made sure that the processes which began execution, finished execution also, so that there is not wastage of time (or garbage execution). We might be able to achieve this by, instead of refreshing the process queue at every unit of time, we could decide a robin time higher that 1 unit, after which the process queue will be refreshed..

This an algorithm not without its flaws, but I believe that the idea is an interesting one, and is worth considering.
Even with it flaws it can provide functioning, and if polished even further, it could be a CPU Scheduling Algorithm which every developer uses.

IX. REFERENCES

1. C. Mattihalli, "Designing and Implementing of Earliest Deadline First Scheduling Algorithm on Standard Linux," *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, Hangzhou, 2010, pp. 901-906, doi: 10.1109/GreenCom-CPSCom.2010.82.

2. J. Teraiya and A. Shah, "Comparative Study of LST and SJF Scheduling Algorithm in Soft Real-Time System with its Implementation and Analysis," *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Bangalore, 2018, pp. 706-711, doi: 10.1109/ICACCI.2018.8554483.

3. R. I. Davis and A. Burns, "Robust Priority Assignment for Fixed Priority Real-Time Systems," *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Tucson, AZ, 2007, pp. 3-14, doi: 10.1109/RTSS.2007.11.

4. J. Li, G. Zheng, H. Zhang and G. Shi, "Task Scheduling Algorithm for Heterogeneous Real-time Systems Based on Deadline Constraints," *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China, 2019, pp. 113-116, doi: 10.1109/ICEIEC.2019.8784641.

5. F. Zhang and A. Burns, "Improvement to Quick Processor-Demand Analysis for EDF-Scheduled Real-Time Systems," *2009 21st Euromicro Conference on Real-Time Systems*, Dublin, 2009, pp. 76-86, doi: 10.1109/ECRTS.2009.20.

6. Zhu Xiangbin and Tu Shiliang, "An improved dynamic scheduling algorithm for multiprocessor real-time systems," *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, Chengdu, China, 2003, pp. 710-714, doi: 10.1109/PDCAT.2003.1236397.

7. https://www.youtube.com/watch?v=zPtI8q9gvX8

8. C. Suman and G. Kumar, "Performance Enhancement of Real Time System using Dynamic Scheduling Algorithms," *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, Bombay, India, 2019, pp. 1-6, doi: 10.1109/I2CT45611.2019.9033843.

9. V. Salmani, M. Naghibzadeh, A. Habibi and H. Deldari, "Quantitative Comparison of Job-level Dynamic Scheduling Policies in Parallel Real-time Systems," *TENCON 2006 - 2006 IEEE Region 10 Conference*, Hong Kong, 2006, pp. 1-4, doi: 10.1109/TENCON.2006.343971.

10. https://www.youtube.com/watch?v=ejPXTOcMRPA

APPENDIX

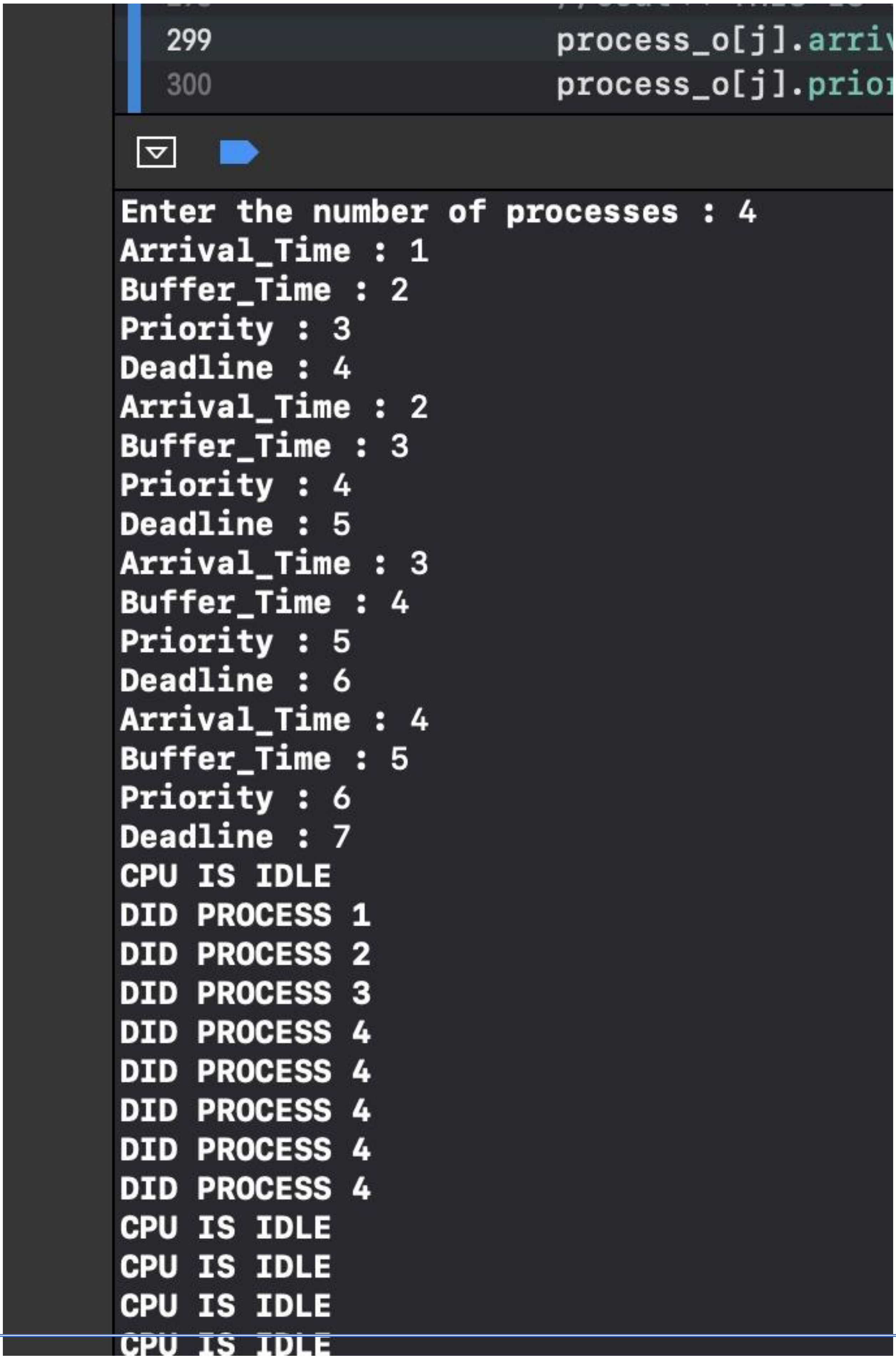
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <iostream>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <pthread.h>
using namespace std;

#define max 100

int num_of_pro_finished;
int avg;

struct arrival_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;

};

int arrival_rear = - 1;
int arrival_front = - 1;
void insert_arrival(int id, int arrival, int buffer, int priority, int deadline,  arrival_queue *arrival_o);
void sort_arrival( arrival_queue *arrival_o, int size);

struct process_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int mark ;
};

int process_rear = -1;
int process_front = -1;
void insert_process(int id, int arrival, int buffer, int priority, int deadline,  process_queue *process_o);
```

```
struct priority_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int mark;
    int att;
};

int priority_rear = -1;
int priority_front = -1;
void insert_priority(int id, int arrival, int buffer, int priority, int deadline,  struct priority_queue
*priority_o);
void sort_priority( priority_queue *priority_o, int size);
void remove_priority();

struct deadline_queue{
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
};
int deadline_rear = -1;
int deadline_front = -1;
void insert_deadline(int id, int arrival, int buffer, int priority, int deadline,  struct deadline_queue
*deadline_o);
void sort_deadline( deadline_queue *deadline_o, int size);


void priority_deadline(struct arrival_queue * arrival_o, struct process_queue *process_o, struct
priority_queue *priority_o, struct deadline_queue *deadline_o);
void insert(arrival_queue *arrival_o, int numpro);

//void display_arrival();
//void delete_arrival();


/*
void display_process();
void delete_process();

void insert_priority(int x);
void display_priority();
void delete_priority();


void insert_deadline(int x);
void display_deadline();
void delete_deadline();*/

//int compare (const void * elem1, const void * elem2);

int main(){

    int numpro;
    printf("Enter the number of processes : "); scanf("%d",&numpro); // number of processes

    struct arrival_queue arrival_o[numpro]; // objects
    struct process_queue process_o[max];
    struct priority_queue priority_o[numpro];
    struct deadline_queue deadline_o[numpro];
    num_of_pro_finished=0;
    avg=0;
    for(int i = 0; i<max; i++){
        process_o[i].id = 0;
        process_o[i].arrival = 0;
        process_o[i].buffer = 0;
        process_o[i].deadline = 0;
        process_o[i].priority=0;
        process_o[i].mark = 0;
    }
    for(int i=0; i<numpro; i++){
        priority_o[i].mark=0;
        priority_o[i].att=0;
    }
    insert(arrival_o, numpro); // inserts into arrival object
    for(int i=0; i<numpro; i++){
        //cout<<"ARRIVAL QUEUE : "<<arrival_o[i].id<<" "<<arrival_o[i].arrival<<endl;
    }
    sort_arrival(arrival_o, numpro); // sorts according to arrival time
    priority_deadline(arrival_o, process_o, priority_o, deadline_o);
}



void priority_deadline(struct arrival_queue * arrival_o, struct process_queue *process_o, struct
priority_queue *priority_o, struct deadline_queue *deadline_o){
    int profit = 0;
    int cpu_iter= 0;
    int process_iter = 0;
    int priority_size=0;
    int deadline_size=0;



    while(cpu_iter < max){
        int min_curr_start = max;

        int max_deadline=0;
        for(int i = 0; i<max; i++){
            process_o[i].id = 0;
            process_o[i].arrival = 0;
            process_o[i].buffer = 0;
            process_o[i].deadline = 0;
            process_o[i].priority = 0;
            process_o[i].mark = 0;
        }

        //cout<<"TIME TO INSERT \n";
        if(cpu_iter == arrival_o[process_iter].arrival){
            //cout<<"MATCH FOUND "<<cpu_iter<<" -- "<<arrival_o[process_iter].arrival<<" -- id -->
"<<arrival_o[process_iter].id<<endl;
            while(cpu_iter == arrival_o[process_iter].arrival){

insert_priority(arrival_o[process_iter].id,arrival_o[process_iter].arrival,arrival_o[process_iter].buffer,arr
ival_o[process_iter].priority,arrival_o[process_iter].deadline,priority_o);
                //cout<<"we inserted id "<<arrival_o[process_iter].id<<endl;
                process_iter++;
                //cout<<"size before : "<<priority_size<<endl;
                priority_size++;
                //cout<<"size afeter : "<<priority_size<<endl;
            }
        }

        /*
        for(int i=priority_front; i<=priority_rear; i++){
            cout<<"PRIORITY QUEUE : "<<priority_o[i].id<<" -- "<<priority_size<<endl;
        }*/

        sort_priority(priority_o, priority_size); // sorts the priority queue object according to the priority

        /*for(int i=priority_front; i<=priority_rear; i++){
            cout<<"PRIORITY QUEUE : "<<priority_o[i].id<<" -- "<<priority_size<<endl;
        }
        */
        /*for(int i=0; i<priority_size; i++){ // traversing the priority queue
            insert_deadline(priority_o[i].id, priority_o[i].arrival ,priority_o[i].buffer
,priority_o[i].priority,priority_o[i].deadline, deadline_o) ;
            deadline_size++;
        }

        sort_deadline(deadline_o, deadline_size);
        max_deadline = deadline_o[0].deadline;*/
        if (priority_front == -1) {
            cout<<"CPU IS IDLE\n";
            //cout<<cpu_iter<<endl;
            cpu_iter++;
            continue;
```

```
}
for(int i=priority_front; i<=priority_rear; i++){
    //cout<<"I : "<<i<<endl;
    if (priority_o[i].buffer<=0) continue ;

    int curr_end = 0;
    int curr_start = 0;
    curr_end = priority_o[i].arrival + priority_o[i].deadline - 1;

    int markcheck = 0;
    int allmarked = 1;
    int eligible = 1;
    while (allmarked == 1){
        for(int j=curr_end; j>curr_end - priority_o[i].buffer; j--){
            //cout<<"J : "<<j<<endl;
            if(process_o[j].mark != 1){
                markcheck++;
            }
        }
        if(markcheck == curr_end - (curr_end - priority_o[i].buffer)){
            allmarked = 0;
        }
        if (curr_end-priority_o[i].buffer < cpu_iter){
            //cout<<"THIS RUNS\n";
            eligible = 0 ;
            break;
        }
        else {
            curr_end = curr_end - 1;
        }

    }
    if ( eligible == 0 ){
        continue ;
    }
    /*
    while (process_o[curr_end].mark == 1){
        cout<<"CHECK 2"<<endl;

        curr_end = curr_end-1;
    }*/
    //cout<<"CHECK 3 "<<priority_front<<" "<<priority_rear<<endl;
    curr_start = curr_end - priority_o[i].buffer;

    if (curr_end < min_curr_start ) min_curr_start = curr_end ;
    for(int j=priority_front; j<=priority_rear; j++){
        //cout<<priority_o[j].id<<" priority : "<<priority_o[j].priority<<endl;
    }
    //cout<<"pppp"<<endl;
    for(int j=curr_end; j> curr_start; j--){
        int t_id = process_o[j].id;
```

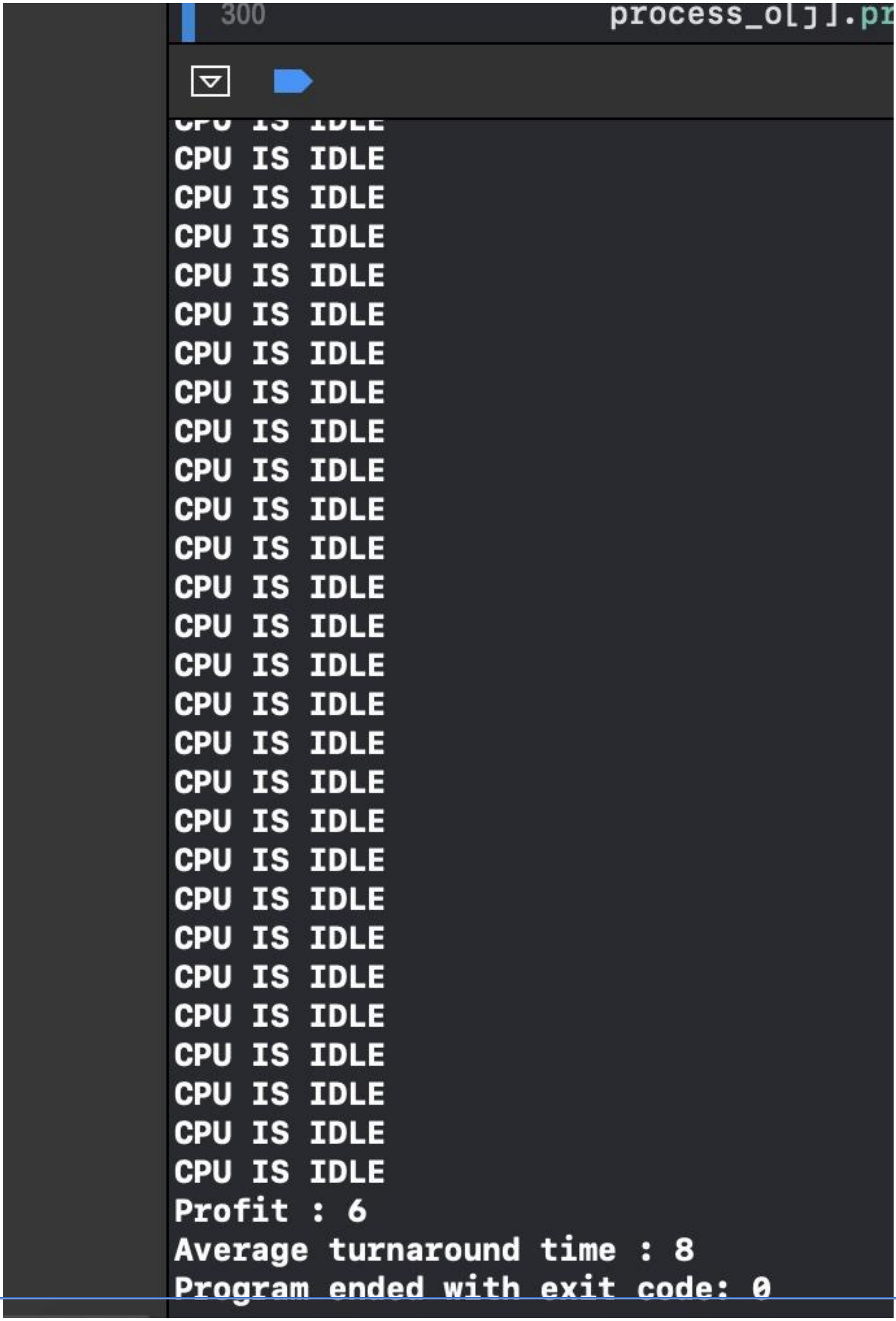
```
                for(int k=priority_front; k<=priority_rear; k++){
                    if(priority_o[k].id == t_id){
                        if(priority_o[k].mark==0){
                            priority_o[k].mark==1;
                            priority_o[i].att=cpu_iter;
                        }
                    }
                }
                process_o[j].mark = 1 ;
                process_o[j].id = priority_o[i].id;
                //cout<<"THIS IS THE PRIORITY BEING ADDED " << priority_o[i].id << " AT
"<<j<<endl;
                process_o[j].arrival = priority_o[i].arrival;
                process_o[j].priority = priority_o[i].priority;
                process_o[j].buffer = priority_o[i].buffer;
                process_o[j].deadline = priority_o[i].deadline;
            }
        }

        for(int i=0; i<max; i++){
            //cout<<process_o[i].id<<" ";
        }
        //cout<<endl;
        //process_o[curr_start].buffer--;
        int curr_id = process_o[min_curr_start].id;
        int did_pro = 0;
        for(int i=priority_front; i<=priority_rear; i++){
            if(priority_o[i].id == curr_id){
                //cout<<"cpu iter : "<<cpu_iter<<" curr_start : "<<min_curr_start -
priority_o[i].buffer<<endl;
                priority_o[i].buffer--;
                if ( priority_o[i].buffer == 0 ) {
                    //cout<<"THIS HAPPENED PRIRIOTY SIZE DECREASE"<<endl;
                    profit = profit + priority_o[i].priority;
                    priority_o[i].att = cpu_iter - priority_o[i].att;
                    avg = priority_o[i].att;
                    num_of_pro_finished++;
                    remove_priority();
                    priority_size-- ;
                }
                cout<<"DID PROCESS "<<curr_id<<endl;
                did_pro = 1;
            }
        }
        if( did_pro == 0 ){
            cout<<"CPU IS IDLE \n";
        }
        //cout<<cpu_iter<<endl;
        cpu_iter++;
    }
    cout<<"Profit : "<<profit<<endl;
    cout<<"Average turnaround time : "<<avg/num_of_pro_finished<<endl;
}

void insert( arrival_queue *arrival_o, int numpro){
    int id;
    int arrival;
    int buffer;
    int priority;
    int deadline;
    int process_id_counter=1;
    for(int i=0; i<numpro; i++){
        printf("Arrival_Time : ");
        scanf("%d",&arrival);
        printf("Buffer_Time : ");
        scanf("%d",&buffer);
        printf("Priority : ");
        scanf("%d",&priority);
        printf("Deadline : ");
        scanf("%d",&deadline);
        id=process_id_counter;
        insert_arrival(id, arrival, buffer, priority, deadline, arrival_o);
        process_id_counter++;
    }
}

void insert_arrival(int id, int arrival, int buffer, int priority, int deadline,  arrival_queue *arrival_o){

    if (arrival_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (arrival_front == - 1) arrival_front = 0;
        arrival_rear = arrival_rear + 1;
        arrival_o[arrival_rear].id = id;
        arrival_o[arrival_rear].arrival = arrival;
        arrival_o[arrival_rear].buffer = buffer;
        arrival_o[arrival_rear].priority = priority;
        arrival_o[arrival_rear].deadline = deadline;
    }
}


void sort_arrival( arrival_queue *arrival_o, int size){
    for (int step = 0; step < size - 1; ++step) {
        for (int i = 0; i < size - step - 1; ++i) {
              if (arrival_o[i].arrival > arrival_o[i + 1].arrival) {
                  int temp1 = arrival_o[i].id;
                  int temp2 = arrival_o[i].arrival;
                  int temp3 = arrival_o[i].buffer;
                  int temp4 = arrival_o[i].priority;
                  int temp5 = arrival_o[i].deadline;

                  arrival_o[i].id = arrival_o[i + 1].id;
```

```
                arrival_o[i].arrival = arrival_o[i + 1].arrival;
                arrival_o[i].buffer = arrival_o[i + 1].buffer;
                arrival_o[i].priority = arrival_o[i + 1].priority;
                arrival_o[i].deadline = arrival_o[i + 1].deadline;

                arrival_o[i + 1].id = temp1;
                arrival_o[i + 1].arrival = temp2;
                arrival_o[i + 1].buffer = temp3;
                arrival_o[i + 1].priority = temp4;
                arrival_o[i + 1].deadline = temp5;
            }
        }
    }
}

void insert_process(int id, int arrival, int buffer, int priority, int deadline,  process_queue *process_o){
    if (process_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (process_front == - 1) process_front = 0;
        process_rear = process_rear + 1;
        process_o[process_rear].id = id;
        process_o[process_rear].arrival = arrival;
        process_o[process_rear].buffer = buffer;
        process_o[process_rear].priority = priority;
        process_o[process_rear].deadline = deadline;
    }
}
/*
void sort_process( process_queue *process_o, int size){
    for (int step = 0; step < size - 1; ++step) {
        for (int i = 0; i < size - step - 1; ++i) {
            if (process_o[i].process > process_o[i + 1].process) {
                int temp1 = process_o[i].id;
                int temp2 = process_o[i].arrival;
                int temp3 = process_o[i].buffer;
                int temp4 = process_o[i].priority;
                int temp5 = process_o[i].deadline;

                process_o[i].id = process_o[i + 1].id;
                process_o[i].arrival = process_o[i + 1].arrival;
                process_o[i].buffer = process_o[i + 1].buffer;
                process_o[i].priority = process_o[i + 1].priority;
                process_o[i].deadline = process_o[i + 1].deadline;

                process_o[i + 1].id = temp1;
                process_o[i + 1].arrival = temp2;
                process_o[i + 1].buffer = temp3;
                process_o[i + 1].priority = temp4;
                process_o[i + 1].deadline = temp5;
            }
        }
    }
}
*/
void insert_priority(int id, int arrival, int buffer, int priority, int deadline,  struct priority_queue
*priority_o){
    if (priority_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (priority_front == - 1) priority_front = 0;
        priority_rear = priority_rear + 1;
        priority_o[priority_rear].id = id;
        priority_o[priority_rear].arrival = arrival;
        priority_o[priority_rear].buffer = buffer;
        priority_o[priority_rear].priority = priority;
        priority_o[priority_rear].deadline = deadline;
    }
}

void remove_priority()
{
    if (priority_front == - 1 || priority_front > priority_rear) return ;
    else priority_front = priority_front + 1;
}

void sort_priority( priority_queue *priority_o, int size){
    //cout<<"1 "<< priority_front << " "<< priority_rear << endl;
    for (int step = priority_front; step <= priority_rear; step++) {
        //cout<<"2"<<endl;
        for (int i = priority_front; i < priority_rear - step ; i++) {
            //cout<<"3"<<endl;
              if (priority_o[i].priority < priority_o[i + 1].priority) {
                  //cout<<"this owrks"<<endl;
                  int temp1 = priority_o[i].id;
                  int temp2 = priority_o[i].arrival;
                  int temp3 = priority_o[i].buffer;
                  int temp4 = priority_o[i].priority;
                  int temp5 = priority_o[i].deadline;

                  priority_o[i].id = priority_o[i + 1].id;
                  priority_o[i].arrival = priority_o[i + 1].arrival;
                  priority_o[i].buffer = priority_o[i + 1].buffer;
                  priority_o[i].priority = priority_o[i + 1].priority;
                  priority_o[i].deadline = priority_o[i + 1].deadline;

                  priority_o[i + 1].id = temp1;
                  priority_o[i + 1].arrival = temp2;
                  priority_o[i + 1].buffer = temp3;
                  priority_o[i + 1].priority = temp4;
                  priority_o[i + 1].deadline = temp5;
              }
        }
    }
}

void insert_deadline(int id, int arrival, int buffer, int priority, int deadline,  struct deadline_queue
*deadline_o){
    if (deadline_rear == max - 1) printf("QUEUE IS FULL \n");
    else{
        if (deadline_front == - 1) deadline_front = 0;
        deadline_rear = deadline_rear + 1;
        deadline_o[deadline_rear].id = id;
        deadline_o[deadline_rear].arrival = arrival;
        deadline_o[deadline_rear].buffer = buffer;
        deadline_o[deadline_rear].priority = priority;
        deadline_o[deadline_rear].deadline = deadline;
    }
}

void sort_deadline( deadline_queue *deadline_o, int size){
    for (int step = 0; step < size - 1; ++step) {
        for (int i = 0; i < size - step - 1; ++i) {
              if (deadline_o[i].deadline < deadline_o[i + 1].deadline) {
                  int temp1 = deadline_o[i].id;
                  int temp2 = deadline_o[i].arrival;
                  int temp3 = deadline_o[i].buffer;
                  int temp4 = deadline_o[i].priority;
                  int temp5 = deadline_o[i].deadline;

                  deadline_o[i].id = deadline_o[i + 1].id;
                  deadline_o[i].arrival = deadline_o[i + 1].arrival;
                  deadline_o[i].buffer = deadline_o[i + 1].buffer;
                  deadline_o[i].priority = deadline_o[i + 1].priority;
                  deadline_o[i].deadline = deadline_o[i + 1].deadline;

                  deadline_o[i + 1].id = temp1;
                  deadline_o[i + 1].arrival = temp2;
                  deadline_o[i + 1].buffer = temp3;
                  deadline_o[i + 1].priority = temp4;
                  deadline_o[i + 1].deadline = temp5;
              }

        }
    }
}
```
OUTPUT :



```
process_o[j].arri
process_o[j].prio
```

```
Enter the number of processes : 4
Arrival_Time : 1
Buffer_Time : 2
Priority : 3
Deadline : 4
Arrival_Time : 2
Buffer_Time : 3
Priority : 4
Deadline : 5
Arrival_Time : 3
Buffer_Time : 4
Priority : 5
Deadline : 6
Arrival_Time : 4
Buffer_Time : 5
Priority : 6
Deadline : 7
CPU IS IDLE
DID PROCESS 1
DID PROCESS 2
DID PROCESS 3
DID PROCESS 4
DID PROCESS 4
DID PROCESS 4
DID PROCESS 4
DID PROCESS 4
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
```



```
                    process_o[j].pr
```

```
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
CPU IS IDLE
Profit : 6
Average turnaround time : 8
Program ended with exit code: 0
```