A New Virtual Machine Scheduling Algorithm For Cloud Computing Systems

Ge Lu
Michelle Chartier
Yeong-Jye Huang

COEN 283 – Operating Systems
December 4th, 2012

Table of Contents

Abstract

Current virtual machine companies provide services that do not take into account priority. While fees may be on a tiered basis, products received differ only in terms of size or available features rather than higher priority. The proposed algorithm attempts to resolve this by taking both time and priority into account when scheduling virtual machine requests from the user. Each virtual machine request will have a priority ranking, low, medium, or high, and a CPU value (1, 2, or 4) and is queued into its respective queue (low, medium, or high then 1, 2, or 4 CPU). The high queue will be serviced in a first come first served manner with each serviced request being moved to the end of its queue. After a predetermined time, the matching queue from the lower priority has its priority elevated and is moved to the higher queue to be serviced. If there is no requests in the high queue, medium priority is treated as high priority and is processed accordingly. If a request comes in that has high priority, the medium will return to medium priority and processed accordingly.

2. Introduction

Objective

The objective of this paper is to propose a new virtual machine scheduling algorithm. This new algorithm is directed at cloud computing systems in order to maximize efficiency. It schedules virtual machine requests by users according to their priority, which is determined by the account level purchased by the user. In addition, the proposed algorithm improves resource utilization by increasing the CPU utilization. This algorithm is the HPF_MRU algorithm (Highest Priority First and Maximize Resource Utilization).

What is the problem

There are many cloud computing options available for free or purchase, each with their own varying levels of services and fees. Each cloud user that pays yearly (rather than month-to-month), pays a fee, or pays a higher fee, should be granted a higher priority. Currently, this is not the case. Rather, some existing scheduling algorithms grant users that pay on a month-to-month basis the same priority as those who pay on a yearly basis. The proposed algorithm seeks to prioritize the scheduling of the virtual machine according to the users' chosen priority level.

Why this is a project related to this class

COEN 283, Operating Systems, is primarily focused on instilling operating system best practices in students – this includes: algorithms, performance (including user perception of that performance), and priority scheduling. This project is concerned with providing an algorithm that is not only efficient, but also utilizes the majority of the resources allocated to it, and provides the user with a satisfactory experience. Since a cloud always contains several computing nodes, it frequently has multiple virtual machine requests arriving simultaneously. This results in a need to provide a scheduling algorithm that is able to quickly respond to many user requests while achieving its goal of priority awareness and maximum resource allocation. Such an algorithm can make a significant difference in how the user perceives the performance of the system and can affect user satisfaction (Tanenbaum, 2008).

Why other approaches are inadequate

One existing scheduling algorithm can be found in the Eucalyptus Private Cloud System. This system's default scheduling algorithm is Greedy Scheduler and Round Robin Scheduler. According to Kyi and Naing (2011), the weakness of Greedy Scheduler and Round Robin Scheduler result in maximized execution times if longer jobs are requested. The solution that Kyi and Naing propose attempts to resolve this downfall of the existing algorithm. However, even after revision, their algorithm makes the implicit assumption that all virtual machine processes are equally important (priority levels are the same). However, in practice, users pay varying levels of fees and subscription fees for increased service in return. Thus, these users should get their service before others (i.e., those who have free accounts).

Why you think your approach is better

The proposed approach is addresses a concern that does not currently get addressed – scheduling virtual machine service requests according to priority while maintaining a high resource utilization rate and thus increasing efficiency. An increase in efficiency helps to reduce redundant energy, time, or resource waste which often leads to the slowing of a system.
Also, this approach considers the situation that virtual machine requests with high priority should be

allocated to the computing nodes and get the service first. Meanwhile, the scheduler monitors the free resources and maximizes the resource utilization.
Finally, this approach is one that keeps many stakeholders in mind - the environment, the company (providing the service), and, of course, the user him or herself.
The environment is impacted in a positive way through the increased efficiency of the algorithm. The more resource utilization that occurs, the more efficient the system becomes. The more efficient a system is, the less power and electricity there is that gets wasted. With electricity a non-renewable energy source, it is essential that it is used wisely.
The company providing the service also greatly benefits from an approach that focuses on efficiency and priority. Through this ability to provide a higher quality of standard for premium accounts, companies would be able to greater monetize their services and provide yet another bracket of service for the discerning user – one which can guarantee higher service priority than non-premium accounts.
Lastly, but most importantly, the user is positively impacted as they are the direct receiver of the service and get impacted directly if his or her premium service if unsatisfactory.

Statement of the problem

Users who pay more, or pay by subscription, for virtual machine service should expect to get higher priority service than lower paying or non-subscribed user accounts.

Area or scope of investigation

The project has been divided into 3 parts – preparation, implementation, and verification. The scope of this project will include all three parts.

The components of these sections are as follows:

1. Preparation
   Come up with the initial virtual machine scheduling algorithm concept
   Create proposal detailing the priority scheduling algorithm for the cloud environment

2. Implementation
   Allocate each virtual machine request to the queue which represents its priority – high, medium, low
   Ensure that the computing node is always allocated the virtual machine with the highest priority
   Elevate the priority of a queue after a fixed time period
   Maximize the CPU utilization through continual benchmarking during development

3. Validation and Verification
   Run the system multiple times to ensure consistent results
   Conduct boundary testing
   Run the system before and after and compare CPU usage

3. Theoretical bases and literature review

Definition of the problem

Virtual machines in cloud computing systems do not take into consideration the user's priority whether that priority be due to higher fees paid or yearly versus monthly payment plan.

Theoretical background of the problem

The background of the problem is centered around three existing algorithms: Priority scheduling, Round-Robin scheduling, and First Come First Served. Each one of these algorithms and their (most prominent) fault(s) will be described in brief in the following sections. Finally, a brief section will be designated for the discussion of the processes of virtual machine requests.

Priority scheduling is when each process has a priority number; this priority number is what is used to determine when it will be scheduled (highest priority comes first) either preemptively or non-preemptively (Amir & Schlossnagle, 2000). However, if only high priority processes are run, then this would result in starvation for all lower priority processes – that is, they would never get run (ibid).

Round-Robin scheduling is said to be the simplest scheduling algorithm for a preemptive scheduler (Wikipedia, 2012). Round-Robin scheduling was designed for time-sharing systems and is similar to First Come First Served scheduling with the exception that, after a certain time-frame, the active process is interrupted and the next process in the queue is serviced (Özdogan, 2011). Round-Robin scheduling is implemented using a FIFO queue, a small unit of time is defined (quantum) and each new process is added to the end of the queue (ibid). Each process is executed in turn and releases the CPU voluntarily if withing the 1 quantum time frame, if not the OS is interrupted, the process moved to the end of the queue, and the next process is executed (ibid). Round-Robin has its own issues, however. For example, using an incorrect quantum size can lead to too many process switches (which in turn lowers the CPU efficiency) or it can lead to poor response time for short interactive requests from the user (Özdogan, 2011).

First Come First Serve (FCFS) scheduling is another very simple scheduling algorithm; however, unlike Round-Robin scheduling, FCFS scheduling is nonpreemptive; thus it does not release the CPU until it terminates or requests I/O (Özdogan, 2011). FCFS scheduling may result in long average wait times, convoy effect, and low CPU and I/O resource utilization (Agrawal, 2008).

Finally, a brief overview of the processes of virtual machine requests is an important topic to elaborate on. Kyi and Naing (2011) described the process for requests for the Eucalyptus System in relation to five components: cluster controller, cloud administer, node controllers, cloud user, cloud system (as a whole). In the initial request for service, the cluster controller provides virtual machine images as Iaas services to the cloud administer to prepare as templates (ibid). Once a virtual machine request is received from the user, the virtual machine is allocated to the appropriate node controllers and begins to run on computer nodes (ibid). The user then performs whatever function s/he needed from the virtual machine and shuts down the machine when finished; the virtual machine instance is then terminated from the cloud system.

Related research to solve the problem

The research that was conducted was the initial perusal, and reading, of two IEEE papers per team member. The papers chosen by the team are listed below (and are also listed in the reference section at the end of this document).

Analysis and Enhancement for Interactive-Oriented Virtual Machine Scheduling.
An Efficient Approach For Virtual Machines Scheduling On a Private Cloud Environment.
User-driven Scheduling Of Interactive Virtual Machines.
Affinity-aware Proportional Share Scheduling for Virtual Machine System.
The Multi-processor Load Balance Scheduler Based on XEN
Efficiently Scheduling Multi-core Guest Virtual Machines on Multi-core Hosts in Network Simulation

Advantage(s)/disadvantage(s) of the research

Advantages:

The aforementioned papers provide an excellent means of understanding the problem domain. These papers also provide great ideas for optimization of the existing scheduling algorithm.

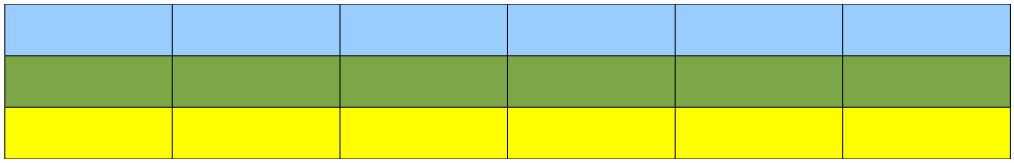
Disadvantage:

One disadvantage of the research was it was difficult to find something on the specific topic. The papers that were chosen did not take into consideration virtual machine requests with different priorities.

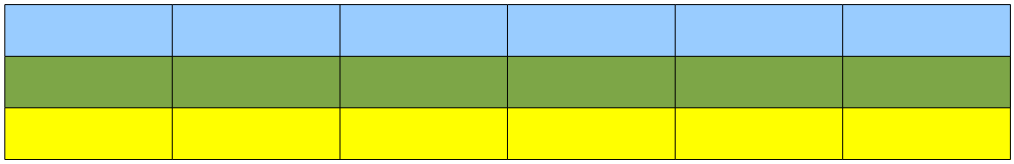
Your solution to solve this problem

The solution to this lack of priority awareness in virtual machine scheduling is solved by the proposed algorithm. This proposed algorithm is based on the algorithm presented in the paper *An Efficient Approach For Virtual Machines Scheduling On a Private Cloud Environment* which uses three, single queues to represent the priority levels of the user requests. The proposed algorithm, however, provides:

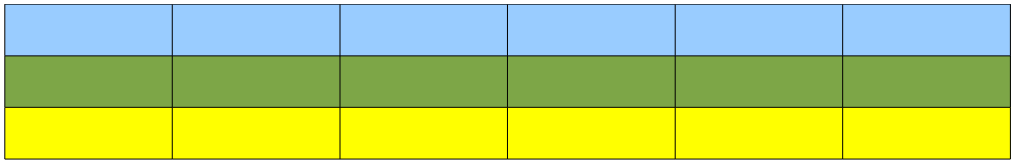
- 3 queue-sets for priority
  - These queue-sets will be ranked according to priority – high, medium, and low.
- 3 queues per queue-set for the number of CPUs
  - Each queue in a queue-set will indicate the number of CPUs to run the user request on. The CPUs will be 1 (yellow), 2 (green), and 4 (blue).



Queue-set 1 (High priority)

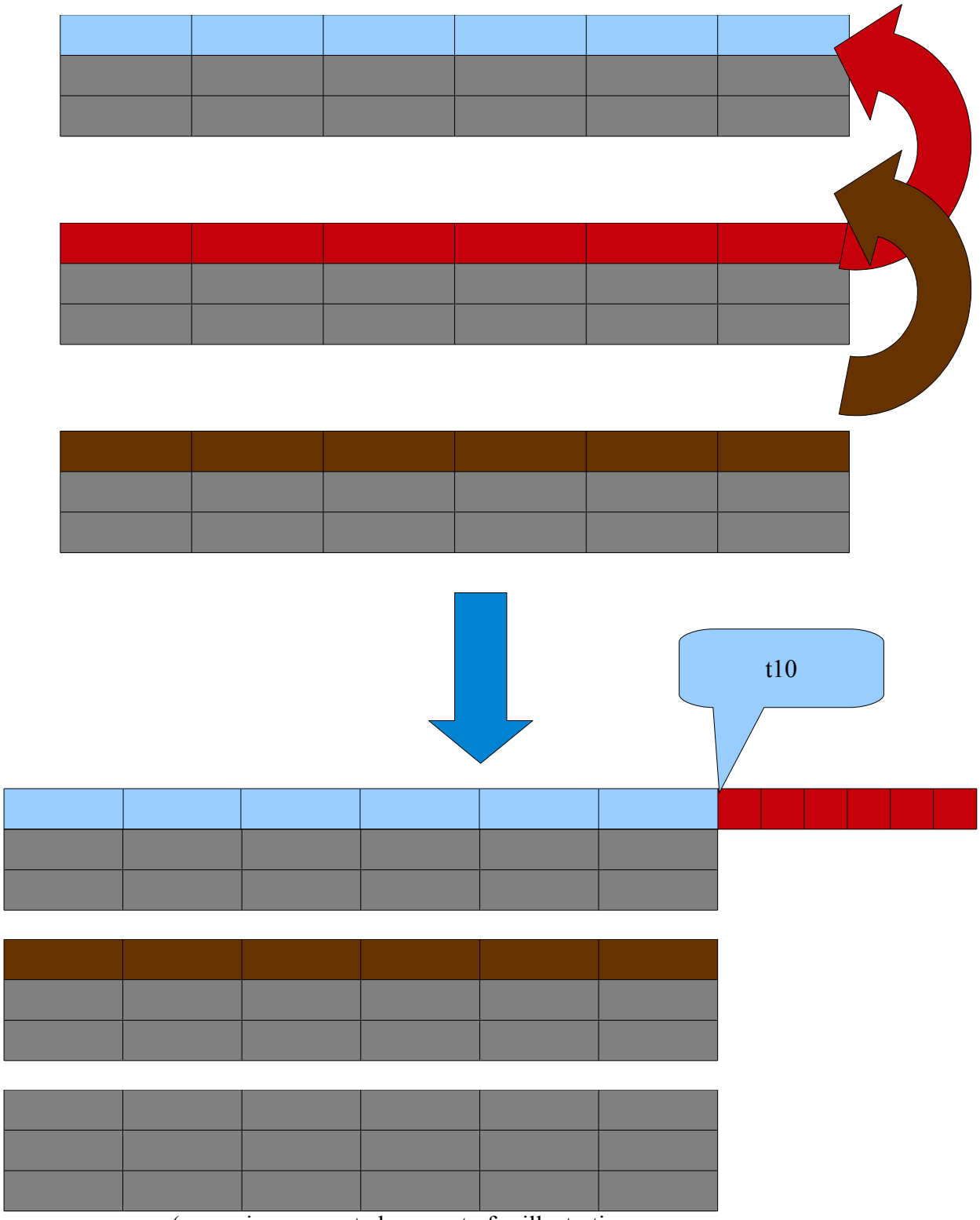


Queue-set 2 (Medium priority)



Queue-set 3 (Low priority)

For each CPU queue in the high priority queue-set, the user request that arrives first is processed first – first come first served. If, however, there are medium priority requests waiting to be processed, then the high priority requests are processed for a specific time frame, e.g., *t10,* then medium priority requests are elevated to high priority and linked to the end of the current high priority list. So also, if there is a low priority request waiting, after a specific time interval, the low priority queue is elevated to medium priority where it waits for its turn to be elevated to high priority:



(queue is represented as empty for illustration purposes.
In real life, requests would probably continue to arrive.)

How your solution is different from others

The proposed algorithm differs primarily from other algorithms as it takes into account user priority

levels as well as efficiency through maximizing the CPU utilization (~90%).

Why your solution is better

This solution to virtual machine scheduling in a cloud computing environment is better than previously proposed or current scheduling algorithms as its primary focus is both efficiency (through high CPU utilization) and incorporating the priority ranking of each user request. Thus, not only will higher priority requests get higher priority when scheduled, but overall, the system will be more efficient while doing so. This algorithm will target a CPU utilization rate of ~90% (+/- 10%). Real-world systems estimate a range of 40%-90% (Yu, 2009).

4. Hypothesis

Positive Hypothesis

Using one data set, the HPF_MRU algorithm can optimize the priority scheduling algorithm by increasing the CPU utilization percentage.

Negative Hypothesis

The proposed algorithm will not result in a decrease in the CPU utilization rate as compared to the same data with no efficiency optimization.

Multiple Hypotheses

The HPF_MRU scheduling algorithm can be used for virtual machine scheduling in cloud system

The high CPU utilization rate will result in a more efficient system.

5. Methodology

How to generate/collect input data

Input data, or test data, will be manually, and sporadically, input using the test data created for the project. This test data contains requests for high, medium, and low priority users. The data will be input via a standard console input window.

Each request will contain only the information about the task to be scheduled. The virtual machine and scheduler will administer to the remaining information – the user ID, priority, arrival time, number of CPUs required, and the time required to complete the request.

How to solve the problem

Algorithm design

The HPF_MRU algorithm will utilize first come, first served scheduling and will handle the priorities of the user requests. The design consists of:

- 3 queue-sets
- 1 queue-set per priority level

- 3 queues per queue set
- 1 queue per CPU number – 1, 2, 4.

Language used

ANSI C

Tools used

Xcode (for code creation), Astah Community (for UML modeling), MacBook Pro, MacBook Air, AutoCAD.

A prototype

An evolutionary prototype is currently being developed by which to guide the project, the design of the algorithm, and the team.

How to generate output

The output will be generated by the program after every user request. That is, when a request has been entered, the user will receive immediate feedback in the form of a console output. This output will include the pertinent information about the request – i.e., "User <name> arrived at <time> and processing length was <time length>".

How to test against hypotheses

In order to determine if the hypothesis has been validated or disproven, the CPU utilization will be checked. This will entail recording the initial state of the system's CPU(s) using a computer's CPU, such as with an Activity Monitor (Mac OS X for systems with 4+ cores), and then, while running the simulation, recording the system's CPU usage.

How to prove correctness

Correctness will be proven by checking that the user requests have been processed in the correct order. That is, the first request is to be processed first, if that request is of a low priority, when a high priority request arrives and is queued, it will replace the earlier but low priority request. This information will be available by means of a "show" functionality and will present the following information via the console:
Current time is <time>
Total user requests: <#ofRequests>
Completed user requests: <completedRequests>
User <user_1> arrived at <time_1> and finished at <time_1>
User <user_2> arrived at <time_2> and finished at <time_2>
…
User <user_n> arrived at <time_n> and finished at <time_n>

In addition, to verify the program's output is correct, six typical virtual machines (scheduled by hand) were used as the test input. The CPU usage was charted and depicted with a diagram. The input data and the program's output are found in Appendix F.

6. implementation

Code (See Appendix B)

Design document and flowchart (for flowchart, see Appendix A)

The design of the implementation of the algorithm was a sequential C program – neither threads nor any Object-Oriented Concepts were utilized. The code was tested against the three test data sets and the results were analyzed.

7. data analysis and discussion

Output generation

Three different sets of test data was created for the purpose of simulating three different types of input situations. This test data was the basis for testing the proposed algorithm.

The three data sets can be broken down in the following three ways to represent three different user situations:

Test Data 1

- Simulates the situation where user requests for virtual machines come randomly.

Test Data 2

- Simulates the same virtual machine requests from users but in a different order than Test Data 1.

Test Data 3

- Same arrival order as Test Data 1 with the exception that the needed time of each virtual machine request is ten times longer than the needed time of each virtual machine request in Test Data 1
- Simulated by decreasing the time quantum to 1/10.

After running each set of the test data, the results were visually depicted using a simple line graph and compared against the hypotheses.

For each graph, the horizontal axis (X) represents the number of the virtual machines while the vertical axis (Y) represents the percentage of the CPU utilization.

Results of Test Data 1 (See Appendix D)

Results of Test Data 2 (See Appendix D)

Results of Test Data 3 (See Appendix D)

Output analysis

In each diagram, there are two lines, a red and a blue line. The blue line represents the utilization percentage of the CPU when using a traditional priority scheduling algorithm. The red line, on the other hand, represents the utilization percentage of the CPU when using the HPF_MRU algorithm (the proposed *Highest Priority First and Maximize Resource Utilization* algorithm).

Analyzing the following pairs, it can be seen that:

- 1, 2: Greater numbers of virtual machine requests with smaller CPU requirements in the waiting queue result in a distinct promotion.
- 1, 3: Based on the fact that the blue lines are identical, a decrease of the time quantum of system will lead to a greater CPU utilization.

Compare output against hypothesis

In appendix E, there is a graph for each of the three test data sets: Test data 1, test data 2, and test data 3. In each graph, there are two lines, a blue line and a red line. The blue line depicts the results of the test data set without efficiency whereas the red line depicts the test data set with efficiency. Reviewing the results of each graph, it can be seen that the blue line supports hypothesis 1, that a priority scheduling algorithm can be used in a cloud-based computing system. Now, in reviewing the red line, the second hypothesis is supported, that the HPF_MRU algorithm can increase the CPU utilization percentage from that of the basic scheduling algorithm.

Abnormal case explanation

An abnormal case for this situation, virtual machines in a cloud based system, would be if two or more users request the services of a virtual machine at exactly the same time. This situation is not the normal situation that is accounted for by the algorithm for this situation could not occur. When two or more users request the services of a virtual machine, the request is first received by the cluster which in turn schedules the requests to different nodes. The new algorithm is focused on the schedule on the node level and not the cluster. Thus, since the cluster is not able to assign two different requests at the exact same time to the node, the algorithm is unaffected by this abnormal case.

Statistic regression

Discussion

The conclusions of this study is enlightening, the design of algorithms do not have to be either usable or efficient but they can be both usable *and* efficient. Encouraging companies to provide users with another form of customization and features while decreasing waste of the CPU is a smart adaptation of technology to today's world. In addition, it would help to reduce the use of non-renewable resources such as electricity.

8. conclusions and recommendations

Summary and conclusions

Overall, scheduling virtual machines in the cloud lack an algorithm that combines priority and efficiency. This combination would be an algorithm that can maintain the order in which virtual machine requests are received while also maintaining a high CPU utilization rate. To this end, the HPF_MRU algorithm was created. The priority consideration is achieved with elements of the FCFS and Round Robin algorithm. Virtual machine requests that come in first will have priority over those who come in second (FCFS). Furthermore, higher priority users (determined by the plan's cost, benefits chosen, etc.) have priority over medium or low priority users. Once a request has been processed, it is rotated to the end of the queue to be serviced again when its time comes around again (Round-Robin).

The algorithm focused on efficiency by increasing the utilization of the CPU. This increased utilization of the CPU is achieved by decreasing the size of the time quantum of the system. As the time quantum decreases, the CPU utilization percentage increases. As compared with a basic priority scheduling algorithm, the HPF_MRU algorithm has a greater utilization percentage. Thus, the HPF_MRU algorithm can make the scheduling of virtual machines in a cloud-based system more efficient while retaining the highest priority first model.

Recommendations for future studies

Based on the research conducted prior to embarking on this project, there does not seem to be cloud-scheduling algorithms that are focused on priority and efficiency. Creating a scheduling algorithm that captures the user's practical needs and requirements would be extrememly useful in virtual machine systems.

In addition, as there do not appear to be any differences in cloud user plans with regard to priority, this would a means of encouraging commerce competition for the most efficient CPU. This in turn would help companies to focus on the efficiency of their CPU in addition to their other considerations.

9. References:

Amir, Y. & Schlossnagle, T.(2000). *[Lecture Slides].* Retrieved from http://www.cs.jhu.edu/~yairamir/cs418/os2/sld030.htm

Özdogan, C. (2011). *[Lecture Notes].* Retrieved from http://siber.cankaya.edu.tr/OperatingSystems/ceng328/node125.html

Agrawal, G. (2008). *[Lecture Slides].* Retrieved from http://www.cse.ohio-state.edu/~agrawal/660/Slides/jan18.pdf

Yu, B. (2009). *[Lecture Slides].* Retrieved from *http://csit.udc.edu/~byu/UDC3529315/Lecture5.pdf*

Tanenbaum, A. S. (2008). *Modern Operating Systems.* Upper Saddle River, NJ: Pearson Education, Inc.

IEEE Papers

Xia et al. (2008). *Analysis and Enhancement for Interactive-Oriented Virtual Machine Scheduling.* Retrieved from https://0-ieeexplore.ieee.org.sculib.scu.edu

Kyi, H. M., Naing, T. T. (2011). *An Efficient Approach For Virtual Machines Scheduling On a Private Cloud Environment.* Retrieved from https://0-ieeexplore.ieee.org.sculib.scu.edu

Lin, B., Dinda, P., A., Lu, D. (2004). *User-driven Scheduling Of Interactive Virtual Machines.* Retrieved from https://0-ieeexplore.ieee.org.sculib.scu.edu

Chen, H., Jin, H., Hu, K. (2010). *Affinity-aware Proportional Share Scheduling for Virtual Machine System.* Retrieved from https://0-ieeexplore.ieee.org.sculib.scu.edu

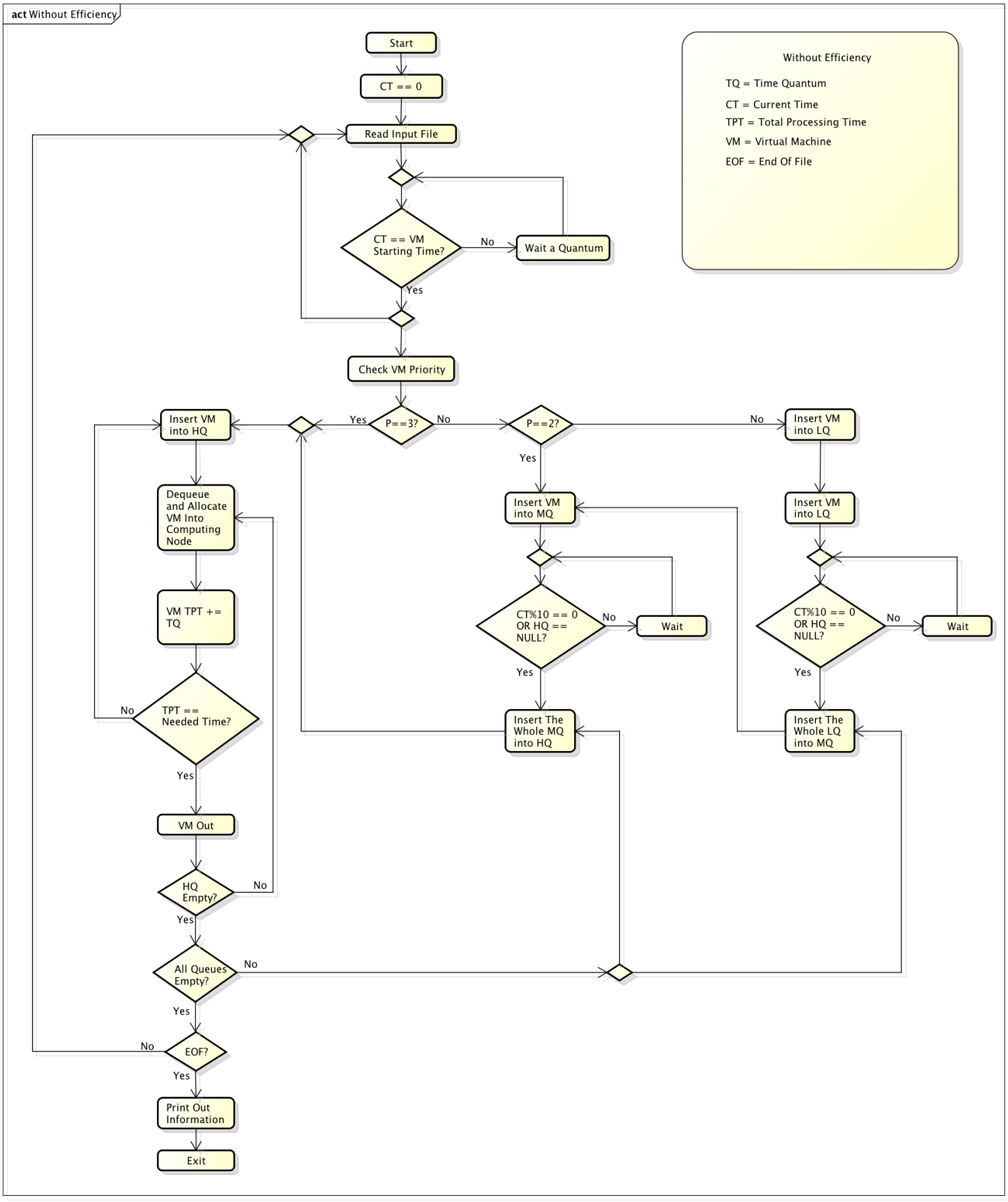
Zheng, Z., Zhang, Y., Wang, Z., Zhang, X. (2012). *The Multi-processor Load Balance Scheduler Based on XEN.* IEEE

Yoginath, S. B. & Perumalla, K. S. (2011). *Efficiently Scheduling Multi-core Guest Virtual Machines on Multi-core Hosts in Network Simulation.* IEEE

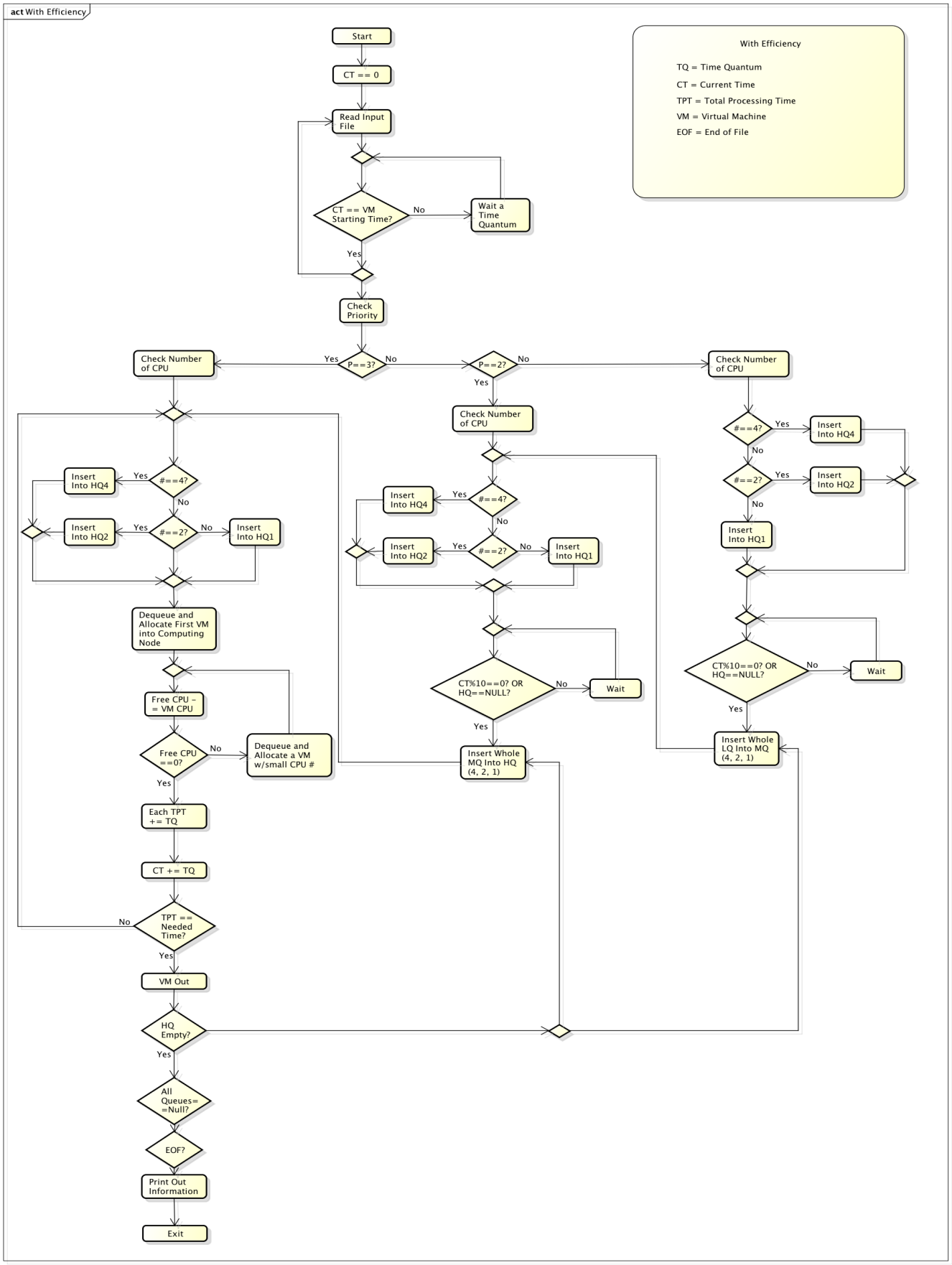
10. Appendices

A: Program flowchart

Without Efficiency



With Efficiency



B: Program source code with documentation

```
/*  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdint.h>

#define NODE_CPU 4 /* this is the total cpu in the computing node */
#define TIME_SEGMENT 1 /* this is the time for computing node to run each VM */
#define PERIOD 10 /* used to elevate the priority */

/* global variables */
int current_time = 0; /* used to record the current time */


typedef struct node
{
   int user_ID;
   int arrival_time;
   int need_time; /* the total time needed to finish the whole VM request */
   int total_process_time; /* time that is used to process this VM request */
   int finish_time; /* the finish time of this request */
   int CPU; /* # of CPU core */

   struct node *next; /* link to the next node in the queue */
} NODE;

typedef struct queue
{
   int count_4; /* record the # of nodes in this queue with a 4-core CPU */
   int count_2; /* record the # of nodes in this queue with a 2-core CPU */
   int count_1; /* record the # of nodes in this queue with a 1-core CPU */
   int count_total;

   NODE *front_4; /* used to point the first node in this queue with a 4-core CPU */
   NODE *rear_4; /* used to point the last node in this queue with a 4-core CPU */
   NODE *front_2; /* used to point the first node in this queue with a 2-core CPU */
   NODE *rear_2; /* used to point the last node in this queue with a 2-core CPU */
   NODE *front_1; /* used to point the first node in this queue with a 1-core CPU */
   NODE *rear_1; /* used to point the last node in this queue with a 1-core CPU */
} QUEUE;


QUEUE* CreateQueue()
{
```

```
    QUEUE* q = (QUEUE*)malloc(sizeof(QUEUE));
    q->front_4 = NULL;
    q->rear_4 = NULL;
    q->count_4 = 0;
    q->front_2 = NULL;
    q->rear_2 = NULL;
    q->count_2 = 0;
    q->front_1 = NULL;
    q->rear_1 = NULL;
    q->count_1 = 0;
    q->count_total = q->count_4 + q->count_2 + q->count_1;
    return q;
}

/* insert the vm into the queue */
void Enqueue(QUEUE *q, int u_ID, int a_t, int n_t, int cpu, int p_t)
{
    NODE *newNode = (NODE*)malloc(sizeof(NODE));

    newNode->user_ID = u_ID;
    newNode->arrival_time = a_t;
    newNode->need_time = n_t;
    newNode->CPU = cpu;
    newNode->total_process_time = p_t;
    newNode->next = NULL;

    if (cpu == 4) {
        if (q->front_4 == NULL)
            q->front_4 = newNode;
        else
            q->rear_4->next = newNode;
        q->rear_4 = newNode;
        q->count_4++;
    }
    else if (cpu == 2) {
        if (q->front_2 == NULL)
            q->front_2 = newNode;
        else
            q->rear_2->next = newNode;
        q->rear_2 = newNode;
        q->count_2++;
    }
    else {
        if (q->front_1 == NULL)
            q->front_1 = newNode;
        else
            q->rear_1->next = newNode;
        q->rear_1 = newNode;
        q->count_1++;
```

```
    }
    q->count_total = q->count_4 + q->count_2 + q->count_1;
}

/* delete a vm from the queue */
void Dequeue(QUEUE *q, int cpu)
{
    NODE *temp;

    if (cpu == 4) {
        temp = q->front_4;

        if (q->count_4 == 1)
            q->rear_4 = NULL;
        q->front_4 = q->front_4->next;
        q->count_4--;
    }
    else if (cpu == 2) {
        temp = q->front_2;

        if (q->count_2 == 1)
            q->rear_2 = NULL;
        q->front_2 = q->front_2->next;
        q->count_2--;
    }
    else {
        temp = q->front_1;

        if (q->count_1 == 1)
            q->rear_1 = NULL;
        q->front_1 = q->front_1->next;
        q->count_1--;
    }

    q->count_total = q->count_4 + q->count_2 + q->count_1;
    free(temp);
}

void DisplayQueue(QUEUE *q)
{
    NODE *temp_4 = q->front_4;
    NODE *temp_2 = q->front_2;
    NODE *temp_1 = q->front_1;

    while (temp_4 != NULL) {
        printf("Here is the information of VM with 4-core CPU: \n");
        printf("User %d arrived at %d have processed %d time.\n", temp_4->user_ID, temp_4-
>arrival_time, temp_4->total_process_time);
        temp_4 = temp_4->next;
```

```
    }
    while (temp_2 != NULL) {
        printf("Here is the information of VM with 2-core CPU: \n");
        printf("User %d arrived at %d have processed %d time.\n", temp_2->user_ID, temp_2-
>arrival_time, temp_2->total_process_time);
        temp_2 = temp_2->next;
    }
    while (temp_1 != NULL) {
        printf("Here is the information of VM with 1-core CPU: \n");
        printf("User %d arrived at %d have processed %d time.\n", temp_1->user_ID, temp_1-
>arrival_time, temp_1->total_process_time);
        temp_1 = temp_1->next;
    }
}

void Round_Robin(QUEUE *q, int cpu)
{
    if (cpu == 4) {
        q->rear_4->next = q->front_4;
        q->rear_4 = q->rear_4->next;
        q->front_4 = q->front_4->next;
        q->rear_4->next = NULL;
    }
    else if (cpu == 2) {
        q->rear_2->next = q->front_2;
        q->rear_2 = q->rear_2->next;
        q->front_2 = q->front_2->next;
        q->rear_2->next = NULL;
    }
    else {
        q->rear_1->next = q->front_1;
        q->rear_1 = q->rear_1->next;
        q->front_1 = q->front_1->next;
        q->rear_1->next = NULL;
    }
}

/* the hq means the queue with a higher priority, the lq means the queue with a lower priority, hp and lp
are the relative concepts */
/* the condition of doind the elecate is (1) t = every t10, (2), the highest priority queue-set is empty */
/* to do the elevate, the lq can NOT be empty */
void Elevate_Priority(QUEUE *hq, QUEUE *lq)
{
    if (lq->front_4 != NULL) {
        if (hq->front_4 != NULL) {
            hq->rear_4->next = lq->front_4;
            hq->rear_4 = lq->rear_4;
            lq->front_4 = NULL;
            lq->rear_4 = NULL;
```

```
    }
    else {
        hq->front_4 = lq->front_4;
        hq->rear_4 = lq->rear_4;
        lq->front_4 = NULL;
        lq->rear_4 = NULL;
    }
    hq->count_4 += lq->count_4;
    hq->count_total += lq->count_4;
    lq->count_4 = 0;
    lq->count_total = 0;
}

if (lq->front_2 != NULL) {
    if (hq->front_2 != NULL) {
        hq->rear_2->next = lq->front_2;
        hq->rear_2 = lq->rear_2;
        lq->front_2 = NULL;
        lq->rear_2 = NULL;
    }
    else {
        hq->front_2 = lq->front_2;
        hq->rear_2 = lq->rear_2;
        lq->front_2 = NULL;
        lq->rear_2 = NULL;
    }
    hq->count_2 += lq->count_2;
    hq->count_total += lq->count_2;
    lq->count_2 = 0;
    lq->count_total = 0;
}

if (lq->front_1 != NULL) {
    if (hq->front_1 != NULL) {
        hq->rear_1->next = lq->front_1;
        hq->rear_1 = lq->rear_1;
        lq->front_1 = NULL;
        lq->rear_1 = NULL;
    }
    else {
        hq->front_1 = lq->front_1;
        hq->rear_1 = lq->rear_1;
        lq->front_1 = NULL;
        lq->rear_1 = NULL;
    }
    hq->count_1 += lq->count_1;
    hq->count_total += lq->count_1;
    lq->count_1 = 0;
    lq->count_total = 0;
```

```
    }
}

int get_the_early_request_free_4(QUEUE *q)
{
    if (q->front_4 != NULL && q->front_2 != NULL && q->front_1 != NULL) {
        if (q->front_4->arrival_time < q->front_2->arrival_time && q->front_4->arrival_time < q-
>front_1->arrival_time) {
            return 4;
        }
        else if (q->front_2->arrival_time < q->front_1->arrival_time && q->front_2->arrival_time < q-
>front_4->arrival_time) {
            return 2;
        }
        else if (q->front_1->arrival_time < q->front_4->arrival_time && q->front_1->arrival_time < q-
>front_2->arrival_time) {
            return 1;
        }
    }
    else if (q->front_4 == NULL && q->front_2 != NULL && q->front_1 != NULL) {
        if (q->front_2->arrival_time < q->front_1->arrival_time) {
            return 2;
        }
        else if (q->front_1->arrival_time < q->front_2->arrival_time) {
            return 1;
        }
    }
    else if (q->front_4 != NULL && q->front_2 == NULL && q->front_1 != NULL) {
        if (q->front_4->arrival_time < q->front_1->arrival_time) {
            return 4;
        }
        else if (q->front_1->arrival_time < q->front_4->arrival_time) {
            return 1;
        }
    }
    else if (q->front_4 != NULL && q->front_2 != NULL && q->front_1 == NULL) {
        if (q->front_2->arrival_time < q->front_4->arrival_time) {
            return 2;
        }
        else if (q->front_4->arrival_time < q->front_2->arrival_time) {
            return 4;
        }
    }
    else if (q->front_4 == NULL && q->front_2 == NULL && q->front_1 != NULL) {
        return 1;
    }
    else if (q->front_4 != NULL && q->front_2 == NULL && q->front_1 == NULL) {
        return 4;
    }
```

```
    else if (q->front_4 == NULL && q->front_2 != NULL && q->front_1 == NULL) {
        return 2;
    }
    else {
        return -1;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// the following part is doing the scheduling without efficiency

typedef struct queue_wo
{
    int count; /* record the # of nodes in this queue */

    NODE *front; /* used to point the first node in this queue */
    NODE *rear; /* used to point the last node in this queue */
} QUEUE_WO;


QUEUE_WO* CreateQueue_WO()
{
    QUEUE_WO* q = (QUEUE_WO*)malloc(sizeof(QUEUE_WO));
    q->front = NULL;
    q->rear = NULL;
    q->count = 0;

    return q;
}

/* insert the vm into the queue */
void Enqueue_WO(QUEUE_WO *q, int u_ID, int a_t, int n_t, int cpu)
{
    NODE *newNode = (NODE*)malloc(sizeof(NODE));

    newNode->user_ID = u_ID;
    newNode->arrival_time = a_t;
    newNode->need_time = n_t;
    newNode->CPU = cpu;
    newNode->total_process_time = 0;
    newNode->next = NULL;

    if (q->front == NULL)
            q->front = newNode;
    else
        q->rear->next = newNode;
    q->rear = newNode;
    q->count++;
```

```
}

/* delete a vm from the queue */
void Dequeue_WO(QUEUE_WO *q)
{
   NODE *temp;

   temp = q->front;

   if (q->count == 1)
      q->rear = NULL;
   q->front = q->front->next;
   q->count--;

   free(temp);
}

void DisplayQueue_WO(QUEUE_WO *q)
{
   NODE *temp = q->front;

   while (temp != NULL) {
      printf("Here is the information of VM: \n");
      printf("User %d arrived at %d have processed %d time.\n", temp->user_ID, temp->arrival_time,
temp->total_process_time);
      temp = temp->next;
   }
}

void Round_Robin_WO(QUEUE_WO *q)
{
   q->rear->next = q->front;
   q->rear = q->rear->next;
   q->front = q->front->next;
   q->rear->next = NULL;
}

/* the hq means the queue with a higher priority, the lq means the queue with a lower priority, hp and lp
are the relative concepts */
/* the condition of doind the elecate is (1) t = every t10, (2), the highest priority queue-set is empty */
/* to do the elevate, the lq can NOT be empty */
void Elevate_Priority_WO(QUEUE_WO *hq, QUEUE_WO *lq)
{
   if (lq->front != NULL) {
      if (hq->front != NULL) {
         hq->rear->next = lq->front;
         hq->rear = lq->rear;
         lq->front = NULL;
         lq->rear = NULL;
```

```
        }
        else {
            hq->front = lq->front;
            hq->rear = lq->rear;
            lq->front = NULL;
            lq->rear = NULL;
        }
    }
    hq->count += lq->count;
    lq->count = 0;
}

// the above part is doing the scheduling without efficiency
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


/* the following part are used to create a queue for storing the finished VM */
typedef struct queue_2
{
    int count; /* record the # of nodes in this queue */

    NODE *front; /* used to point the first node in this queue */
    NODE *rear; /* used to point the last node in this queue */
} QUEUE_2;

QUEUE_2* CreateFinishQueue()
{
    QUEUE_2* q = (QUEUE_2*)malloc(sizeof(QUEUE_2));
    q->front = NULL;
    q->rear = NULL;
    q->count = 0;

    return q;
}

/* insert the vm into the finish queue */
void EnFinishQueue(QUEUE_2 *q, int u_ID, int a_t, int f_t, int cpu)
{
    NODE *newNode = (NODE*)malloc(sizeof(NODE));

    newNode->user_ID = u_ID;
    newNode->arrival_time = a_t;
    newNode->finish_time = f_t;
    newNode->CPU = cpu;
    newNode->next = NULL;

    if (q->front == NULL)
        q->front = newNode;
```

```
    else
        q->rear->next = newNode;
    q->rear = newNode;
    q->count++;
}

void DisplayFinishQueue(QUEUE_2 *q)
{
    NODE *temp = q->front;

    printf("Here is the information of finished VM: \n");
    while (temp != NULL) {
        printf("User %d arrived at %d and finished at %d .\n", temp->user_ID, temp->arrival_time, temp-
>finish_time);
        temp = temp->next;
    }
}

void DO_WITHOUT_EFFICIENCY()
{
    QUEUE_WO *HQ = CreateQueue_WO();
    QUEUE_WO *MQ = CreateQueue_WO();
    QUEUE_WO *LQ = CreateQueue_WO();

    QUEUE_2 *FINISH_QUEUE = CreateFinishQueue();

    char command[64];
    int uid, priority, arrival_time, need_time, cpu;
    int command_count = 0;
    int i, j;
    int a = 0;
    int flag = 1;
    int total_system_cpu = 0;
    int total_use_cpu = 0;

    current_time = 0;

    FILE *fin = fopen("/Users/GE/Desktop/test_in.txt", "r");
    if (fin == NULL) {
        printf("ERROR, NO SUCH FILE1\n");
        exit(1);
    }

    int user_array[64][5];

    for (i = 0; i < 64; i++) {
        for (j = 0; j < 5; j++) {
            user_array[i][j] = -1;
        }
```

```
    }

    while (!feof(fin)) {
        fgets(command, 63, fin);
        if (command[0] != '#') {
            sscanf(command, "%d%d%d%d%d", &user_array[a][0], &user_array[a][1], &user_array[a][2],
&user_array[a][3], &user_array[a][4]);
            a++;
            command_count++;
        }
    }

    fclose(fin);

    a = 0;

    while (flag == 1) {
        arrival_time = user_array[a][2];
        uid = user_array[a][0];
        priority = user_array[a][1];
        need_time = user_array[a][3];
        cpu = user_array[a][4];

        if (arrival_time == current_time) {
            if (priority == 3) {
                Enqueue_WO(HQ, uid, arrival_time, need_time, cpu);
            }
            else if (priority == 2){
                Enqueue_WO(MQ, uid, arrival_time, need_time, cpu);
            }
            else {
                Enqueue_WO(LQ, uid, arrival_time, need_time, cpu);
            }
            a++;
        }

        if (HQ->count == 0 || (current_time !=0 && current_time % PERIOD == 0)) {
            Elevate_Priority_WO(HQ, MQ);
            Elevate_Priority_WO(MQ, LQ);
        }

        HQ->front->total_process_time += TIME_SEGMENT;
        total_system_cpu += 4;
        total_use_cpu += HQ->front->CPU;

        if (HQ->front->total_process_time == HQ->front->need_time) {
            EnFinishQueue(FINISH_QUEUE, HQ->front->user_ID, HQ->front->arrival_time,
current_time+1, HQ->front->CPU);
            Dequeue_WO(HQ);
```

```
        }
        else{
            Round_Robin_WO(HQ);
        }

        if (a == command_count && HQ->count == 0 && MQ->count == 0 && LQ->count == 0) {
            flag = 0;
        }

        current_time += TIME_SEGMENT;
    }

    DisplayFinishQueue(FINISH_QUEUE);
    printf("total system cpu is: %d\n", total_system_cpu);
    printf("total use cpu is: %d\n", total_use_cpu);
    printf("percentage is: %lf\n", (double)total_use_cpu/(double)total_system_cpu);

    free(FINISH_QUEUE);
    free(HQ);
    free(MQ);
    free(LQ);
}



void DO_WITH_EFFICIENCY()
{
    QUEUE *HQ = CreateQueue();
    QUEUE *MQ = CreateQueue();
    QUEUE *LQ = CreateQueue();

    QUEUE_2 *FINISH_QUEUE = CreateFinishQueue();

    char command[64];
    int uid, priority, arrival_time, need_time, cpu;
    int command_count = 0;
    int i, j;
    int a = 0;
    int flag = 1;
    int total_system_cpu = 0;
    int total_use_cpu = 0;
    int early_request;

    NODE *temp1 = (NODE*)malloc(sizeof(NODE));
    NODE *temp2 = (NODE*)malloc(sizeof(NODE));
    NODE *temp3 = (NODE*)malloc(sizeof(NODE));
    NODE *temp4 = (NODE*)malloc(sizeof(NODE));
```

```
    current_time = 0;

    FILE *fin = fopen("/Users/GE/Desktop/test_in.txt", "r");
    if (fin == NULL) {
        printf("ERROR, NO SUCH FILE1\n");
        exit(1);
    }

    int user_array[64][5];

    for (i = 0; i < 64; i++) {
        for (j = 0; j < 5; j++) {
            user_array[i][j] = -1;
        }
    }

    while (!feof(fin)) {
        fgets(command, 63, fin);
        if (command[0] != '#') {
            sscanf(command, "%d%d%d%d%d", &user_array[a][0], &user_array[a][1], &user_array[a][2],
&user_array[a][3], &user_array[a][4]);
            a++;
            command_count++;
        }
    }

    fclose(fin);

    a = 0;

    while (flag == 1) {
        arrival_time = user_array[a][2];
        uid = user_array[a][0];
        priority = user_array[a][1];
        need_time = user_array[a][3];
        cpu = user_array[a][4];

        if (arrival_time == current_time) {
            if (priority == 3) {
                Enqueue(HQ, uid, arrival_time, need_time, cpu, 0);
            }
            else if (priority == 2){
                Enqueue(MQ, uid, arrival_time, need_time, cpu, 0);
            }
            else {
                Enqueue(LQ, uid, arrival_time, need_time, cpu, 0);
            }
            a++;
        }
```

```
    if (HQ->count_total == 0 || (current_time !=0 && current_time % PERIOD == 0)) {
        Elevate_Priority(HQ, MQ);
        Elevate_Priority(MQ, LQ);
    }


    early_request = get_the_early_request_free_4(HQ);

    if (early_request == 4) {

        HQ->front_4->total_process_time += TIME_SEGMENT;
        total_system_cpu += 4;
        total_use_cpu += HQ->front_4->CPU;


        if (HQ->front_4->total_process_time == HQ->front_4->need_time) {
            EnFinishQueue(FINISH_QUEUE, HQ->front_4->user_ID, HQ->front_4->arrival_time,
current_time+1, HQ->front_4->CPU);
            Dequeue(HQ, 4);
        }
        else{
            Round_Robin(HQ, 4);
        }
    }
    else if (early_request == 2) {
        total_system_cpu += 4;

        *temp1 = *(HQ->front_2);
        Dequeue(HQ, 2);

        if (HQ->front_1 == NULL) {
            if (HQ->front_2 != NULL) {
                *temp2 = *(HQ->front_2);
                Dequeue(HQ, 2);

                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                total_use_cpu += 4;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1->CPU,
temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
```

```
                EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
            }
            else {
                Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2->CPU,
temp2->total_process_time);
            }
        }
        else {
            temp1->total_process_time += TIME_SEGMENT;
            total_use_cpu += 2;
            if (temp1->total_process_time == temp1->need_time) {
                EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
            }
            else {
                Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1->CPU,
temp1->total_process_time);
            }
        }
    }
    else {
        if (HQ->front_2 != NULL) {
            if (HQ->front_2->arrival_time < HQ->front_1->arrival_time) {
                *temp2 = *(HQ->front_2);
                Dequeue(HQ, 2);

                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                total_use_cpu += 4;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                    Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                }
            }
            else {
```

```
                *temp2 = *(HQ->front_1);
                Dequeue(HQ, 1);
                if (HQ->front_1 != NULL) {
                    *temp3 = *(HQ->front_1);
                    Dequeue(HQ, 1);

                    temp1->total_process_time += TIME_SEGMENT;
                    temp2->total_process_time += TIME_SEGMENT;
                    temp3->total_process_time += TIME_SEGMENT;
                    total_use_cpu += 4;

                    if (temp1->total_process_time == temp1->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                    }
                    else {
                        Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                    }
                    if (temp2->total_process_time == temp2->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                    }
                    else {
                        Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                    }
                    if (temp3->total_process_time == temp3->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                    }
                    else {
                        Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                    }
                }
                else {
                    temp1->total_process_time += TIME_SEGMENT;
                    temp2->total_process_time += TIME_SEGMENT;
                    total_use_cpu += 3;

                    if (temp1->total_process_time == temp1->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                    }
                    else {
                        Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                    }
```

```
                    if (temp2->total_process_time == temp2->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                    }
                    else {
                        Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                    }
                }
            }
            else {
                *temp2 = *(HQ->front_1);
                Dequeue(HQ, 1);
                if (HQ->front_1 != NULL) {
                    *temp3 = *(HQ->front_1);
                    Dequeue(HQ, 1);

                    temp1->total_process_time += TIME_SEGMENT;
                    temp2->total_process_time += TIME_SEGMENT;
                    temp3->total_process_time += TIME_SEGMENT;
                    total_use_cpu += 4;

                    if (temp1->total_process_time == temp1->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                    }
                    else {
                        Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                    }
                    if (temp2->total_process_time == temp2->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                    }
                    else {
                        Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                    }
                    if (temp3->total_process_time == temp3->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                    }
                    else {
                        Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                    }
                }
                else {
```

```
                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                total_use_cpu += 3;

                if (temp1->total_process_time == temp1->need_time) {
                   EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                   Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
                   EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                   Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp3->total_process_time);
                }
             }
          }
       }
    }
    ////////////////////////////////////////////////////////////////////////
    else if (early_request == 1) {
       total_system_cpu += 4;

       *temp1 = *(HQ->front_1);
       Dequeue(HQ, 1);

       if (HQ->front_1 == NULL) {
          if (HQ->front_2 != NULL) {
             *temp2 = *(HQ->front_2);
             Dequeue(HQ, 2);

             temp1->total_process_time += TIME_SEGMENT;
             temp2->total_process_time += TIME_SEGMENT;
             total_use_cpu += 3;

             if (temp1->total_process_time == temp1->need_time) {
                EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
             }
             else {
                Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1->CPU,
temp1->total_process_time);
             }
             if (temp2->total_process_time == temp2->need_time) {
```

```
                EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
            }
            else {
                Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2->CPU,
temp2->total_process_time);
            }
        }
        else {
            temp1->total_process_time += TIME_SEGMENT;
            total_use_cpu += 1;
            if (temp1->total_process_time == temp1->need_time) {
                EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
            }
            else {
                Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1->CPU,
temp1->total_process_time);
            }
        }
    }
    else {
        if (HQ->front_2 != NULL) {
            if (HQ->front_2->arrival_time < HQ->front_1->arrival_time) {
                *temp2 = *(HQ->front_2);
                Dequeue(HQ, 2);
                *temp3 = *(HQ->front_1);
                Dequeue(HQ, 1);

                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                temp3->total_process_time += TIME_SEGMENT;
                total_use_cpu += 4;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                    Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
```

```
                }
                if (temp3->total_process_time == temp3->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                }
                else {
                    Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                }
            }
            else {
                *temp2 = *(HQ->front_1);
                Dequeue(HQ, 1);
                ////
                if (HQ->front_1 != 0) {
                    if (HQ->front_2->arrival_time < HQ->front_1->arrival_time) {
                        *temp3 = *(HQ->front_2);
                        Dequeue(HQ, 2);

                        temp1->total_process_time += TIME_SEGMENT;
                        temp2->total_process_time += TIME_SEGMENT;
                        temp3->total_process_time += TIME_SEGMENT;
                        total_use_cpu += 4;

                        if (temp1->total_process_time == temp1->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                        }
                        else {
                            Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time,
temp1->CPU, temp1->total_process_time);
                        }
                        if (temp2->total_process_time == temp2->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                        }
                        else {
                            Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time,
temp2->CPU, temp2->total_process_time);
                        }
                        if (temp3->total_process_time == temp3->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                        }
                        else {
                            Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time,
temp3->CPU, temp3->total_process_time);
                        }
                    }
```

```
                else {
                    *temp3 = *(HQ->front_1);
                    Dequeue(HQ, 1);

                    if (HQ->front_1 != NULL) {
                        *temp4 = *(HQ->front_1);
                        Dequeue(HQ, 1);

                        temp1->total_process_time += TIME_SEGMENT;
                        temp2->total_process_time += TIME_SEGMENT;
                        temp3->total_process_time += TIME_SEGMENT;
                        temp4->total_process_time += TIME_SEGMENT;
                        total_use_cpu += 4;

                        if (temp1->total_process_time == temp1->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                        }
                        else {
                            Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time,
temp1->CPU, temp1->total_process_time);
                        }
                        if (temp2->total_process_time == temp2->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                        }
                        else {
                            Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time,
temp2->CPU, temp2->total_process_time);
                        }
                        if (temp3->total_process_time == temp3->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                        }
                        else {
                            Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time,
temp3->CPU, temp3->total_process_time);
                        }
                        if (temp4->total_process_time == temp4->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp4->user_ID, temp4->arrival_time,
current_time+1, temp4->CPU);
                        }
                        else {
                            Enqueue(HQ, temp4->user_ID, temp4->arrival_time, temp4->need_time,
temp4->CPU, temp4->total_process_time);
                        }
                    }
                    else {
                        temp1->total_process_time += TIME_SEGMENT;
```

```
                        temp2->total_process_time += TIME_SEGMENT;
                        temp3->total_process_time += TIME_SEGMENT;
                        total_use_cpu += 3;

                        if (temp1->total_process_time == temp1->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                        }
                        else {
                            Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time,
temp1->CPU, temp1->total_process_time);
                        }
                        if (temp2->total_process_time == temp2->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                        }
                        else {
                            Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time,
temp2->CPU, temp2->total_process_time);
                        }
                        if (temp3->total_process_time == temp3->need_time) {
                            EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                        }
                        else {
                            Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time,
temp3->CPU, temp3->total_process_time);
                        }
                    }
                }
            }
            else {
                *temp3 = *(HQ->front_2);
                Dequeue(HQ, 2);

                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                temp3->total_process_time += TIME_SEGMENT;
                total_use_cpu += 4;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
```

```
                    EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                    Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                }
                if (temp3->total_process_time == temp3->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                }
                else {
                    Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                }
            }
        }
    }
    else {
        *temp2 = *(HQ->front_1);
        Dequeue(HQ, 1);
        if (HQ->front_1 != NULL) {
            *temp3 = *(HQ->front_1);
            Dequeue(HQ, 1);

            if (HQ->front_1 != NULL) {
                *temp4 = *(HQ->front_1);
                Dequeue(HQ, 1);

                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                temp3->total_process_time += TIME_SEGMENT;
                temp4->total_process_time += TIME_SEGMENT;
                total_use_cpu += 4;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                    Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
```

```
>CPU, temp2->total_process_time);
                    }
                    if (temp3->total_process_time == temp3->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                    }
                    else {
                        Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                    }
                    if (temp4->total_process_time == temp4->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp4->user_ID, temp4->arrival_time,
current_time+1, temp4->CPU);
                    }
                    else {
                        Enqueue(HQ, temp4->user_ID, temp4->arrival_time, temp4->need_time, temp4-
>CPU, temp4->total_process_time);
                    }
                }
                else {
                    temp1->total_process_time += TIME_SEGMENT;
                    temp2->total_process_time += TIME_SEGMENT;
                    temp3->total_process_time += TIME_SEGMENT;
                    total_use_cpu += 3;

                    if (temp1->total_process_time == temp1->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                    }
                    else {
                        Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                    }
                    if (temp2->total_process_time == temp2->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                    }
                    else {
                        Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp2->total_process_time);
                    }
                    if (temp3->total_process_time == temp3->need_time) {
                        EnFinishQueue(FINISH_QUEUE, temp3->user_ID, temp3->arrival_time,
current_time+1, temp3->CPU);
                    }
                    else {
                        Enqueue(HQ, temp3->user_ID, temp3->arrival_time, temp3->need_time, temp3-
>CPU, temp3->total_process_time);
                    }
```

```
                }
            }
            else {
                temp1->total_process_time += TIME_SEGMENT;
                temp2->total_process_time += TIME_SEGMENT;
                total_use_cpu += 2;

                if (temp1->total_process_time == temp1->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp1->user_ID, temp1->arrival_time,
current_time+1, temp1->CPU);
                }
                else {
                    Enqueue(HQ, temp1->user_ID, temp1->arrival_time, temp1->need_time, temp1-
>CPU, temp1->total_process_time);
                }
                if (temp2->total_process_time == temp2->need_time) {
                    EnFinishQueue(FINISH_QUEUE, temp2->user_ID, temp2->arrival_time,
current_time+1, temp2->CPU);
                }
                else {
                    Enqueue(HQ, temp2->user_ID, temp2->arrival_time, temp2->need_time, temp2-
>CPU, temp3->total_process_time);
                }
            }
        }
    }
}

    if (a == command_count && HQ->count_total == 0 && MQ->count_total == 0 && LQ-
>count_total == 0) {
        flag = 0;
    }

    current_time += TIME_SEGMENT;
}

DisplayFinishQueue(FINISH_QUEUE);
printf("total system cpu is: %d\n", total_system_cpu);
printf("total use cpu is: %d\n", total_use_cpu);
printf("percentage is: %lf\n", (double)total_use_cpu/(double)total_system_cpu);

free(FINISH_QUEUE);
free(HQ);
free(MQ);
free(LQ);
}

int main()
{
```

```
    printf("<----Priority Scheduling without efficiency---->\n");
    DO_WITHOUT_EFFICIENCY();

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

    printf("<----Priority Scheduling with efficiency---->\n");
    DO_WITH_EFFICIENCY();

    return 0;
}
```

C: Test Data Sets

Test Data 1

User ID	Priority	Arrival Time	Need Time	CPU
1	3	0	10	2
30	2	5	22	4
2	3	8	5	2
15	1	12	30	1
45	1	14	8	4
36	2	20	18	4
24	3	36	10	1
11	3	40	16	1
9	1	41	8	2
31	2	46	12	4
59	2	50	13	2
42	1	52	16	1
103	2	55	30	4
99	3	58	40	4
33	3	64	7	2

Test Data 2

User ID	Priority	Arrival Time	Need Time	CPU
30	2	0	22	4
31	2	4	12	4
36	2	6	18	4
1	3	12	10	2
2	3	18	5	2
15	1	22	30	1
103	2	25	30	4
33	3	34	7	2
45	1	44	8	4
24	3	46	10	1
11	3	50	16	1
9	1	51	8	2
59	2	54	13	2
42	1	56	16	1
99	3	64	40	4

Test Data 3

User ID	Priority	Arrival Time	Need Time	CPU
1	3	0	100	2
30	2	5	220	4
2	3	8	50	2
15	1	12	300	1
45	1	14	80	4
36	2	20	180	4
24	3	36	100	1
11	3	40	160	1
9	1	41	80	2
31	2	46	120	4
59	2	50	130	2
42	1	52	160	1
103	2	55	300	4
99	3	58	400	4
33	3	64	70	2

D: Input/output listing

Input 1 with Corresponding Output

ID	Priority	Arrival Time	Need Time	CPU	Priority Scheduling without efficiency	Priority Scheduling with efficiency
1	3	0	10	2	<----Priority Scheduling without efficiency----> Here is the information of finished VM: User 1 arrived at 0 and finished at 11 . User 2 arrived at 8 and finished at 18 . User 45 arrived at 14 and finished at 87 . User 30 arrived at 5 and finished at 124 . User 24 arrived at 36 and finished at 134 . User 36 arrived at 20 and finished at 140 . User 33 arrived at 64 and finished at 146 . User 9 arrived at 41 and finished at 151 . User 31 arrived at 46 and finished at 169 . User 59 arrived at 50 and finished at 176 . User 11 arrived at 40 and finished at 183 . User 42 arrived at 52 and finished at 200 . User 15 arrived at 12 and finished at 223 . User 103 arrived at 55 and finished at 235 . User 99 arrived at 58 and finished at 245 . total system cpu is: 980 total use cpu is: 678 percentage is: 0.691837	<----Priority Scheduling with efficiency----> Here is the information of finished VM: User 1 arrived at 0 and finished at 10 . User 2 arrived at 8 and finished at 24 . User 24 arrived at 36 and finished at 50 . User 11 arrived at 40 and finished at 56 . User 15 arrived at 12 and finished at 60 . User 59 arrived at 50 and finished at 75 . User 33 arrived at 64 and finished at 84 . User 9 arrived at 41 and finished at 85 . User 42 arrived at 52 and finished at 95 . User 30 arrived at 5 and finished at 129 . User 45 arrived at 14 and finished at 130 . User 31 arrived at 46 and finished at 147 . User 36 arrived at 20 and finished at 156 . User 103 arrived at 55 and finished at 188 . User 99 arrived at 58 and finished at 198 . total system cpu is: 792 total use cpu is: 688 percentage is: 0.868687
30	2	5	22	4		
2	3	8	5	2		
15	1	12	30	1		
45	1	14	8	4		
36	2	20	18	4		
24	3	36	10	1		
11	3	40	16	1		
9	1	41	8	2		
31	2	46	12	4		
59	2	50	13	2		
42	1	52	16	1		
103	2	55	30	4		
99	3	58	40	4		
33	3	64	7	2		

Input 2 with Corresponding Output

User ID	Priority	Arrival Time	Need Time	CPU	Priority Scheduling without efficiency	Priority Scheduling with efficiency
30	2	0	22	4	<----Priority Scheduling without efficiency----> Here is the information of finished VM: User 2 arrived at 18 and finished at 46 . User 1 arrived at 12 and finished at 79 . User 30 arrived at 0 and finished at 87 . User 31 arrived at 4 and finished at 89 . User 33 arrived at 34 and finished at 103 . User 45 arrived at 44 and finished at 146 . User 24 arrived at 46 and finished at 151 . User 9 arrived at 51 and finished at 156 . User 36 arrived at 6 and finished at 159 . User 59 arrived at 54 and finished at 183 . User 11 arrived at 50 and finished at 191 . User 42 arrived at 56 and finished at 201 . User 103 arrived at 25 and finished at 228 . User 15 arrived at 22 and finished at 232 . User 99 arrived at 64 and finished at 245 . total system cpu is: 980 total use cpu is: 678 percentage is: 0.691837	<----Priority Scheduling with efficiency----> Here is the information of finished VM: User 2 arrived at 18 and finished at 49 . User 30 arrived at 0 and finished at 58 . User 31 arrived at 4 and finished at 59 . User 24 arrived at 46 and finished at 74 . User 33 arrived at 34 and finished at 78 . User 1 arrived at 12 and finished at 80 . User 9 arrived at 51 and finished at 85 . User 59 arrived at 54 and finished at 90 . User 11 arrived at 50 and finished at 95 . User 15 arrived at 22 and finished at 104 . User 42 arrived at 56 and finished at 107 . User 36 arrived at 6 and finished at 125 . User 45 arrived at 44 and finished at 130 . User 103 arrived at 25 and finished at 164 . User 99 arrived at 64 and finished at 181 . total system cpu is: 724 total use cpu is: 687 percentage is: 0.948895
31	2	4	12	4		
36	2	6	18	4		
1	3	12	10	2		
2	3	18	5	2		
15	1	22	30	1		
103	2	25	30	4		
33	3	34	7	2		
45	1	44	8	4		
24	3	46	10	1		
11	3	50	16	1		
9	1	51	8	2		
59	2	54	13	2		
42	1	56	16	1		
99	3	64	40	4		

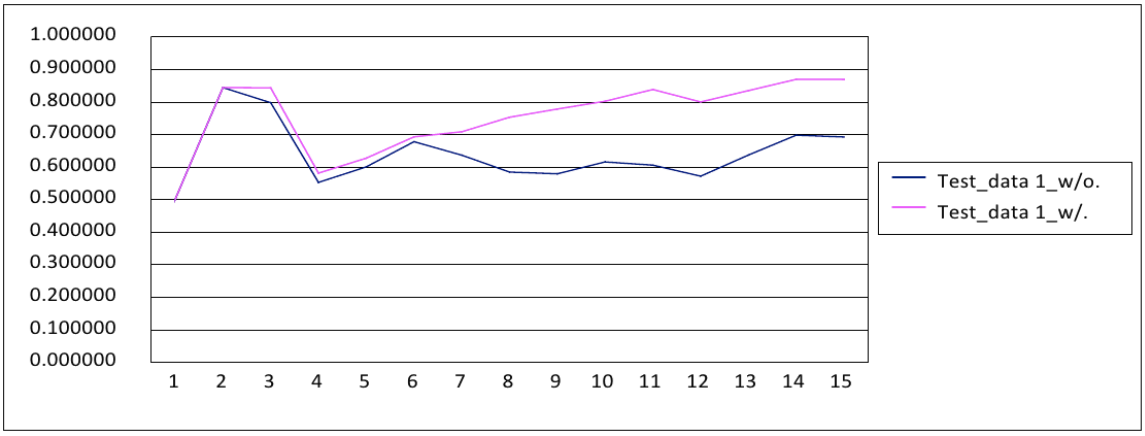
Input 3 with Corresponding Output

User ID	Priority	Arrival Time	Need Time	CPU	Priority Scheduling without efficiency	Priority Scheduling with efficiency
1	3	0	100	2	<----Priority Scheduling without efficiency----> Here is the information of finished VM: User 2 arrived at 8 and finished at 632 . User 33 arrived at 64 and finished at 1081 . User 45 arrived at 14 and finished at 1150 . User 9 arrived at 41 and finished at 1203 . User 1 arrived at 0 and finished at 1206 . User 24 arrived at 36 and finished at 1376 . User 31 arrived at 46 and finished at 1571 . User 59 arrived at 50 and finished at 1653 . User 11 arrived at 40 and finished at 1853 . User 42 arrived at 52 and finished at 1874 . User 36 arrived at 20 and finished at 1936 . User 30 arrived at 5 and finished at 2081 . User 15 arrived at 12 and finished at 2340 . User 103 arrived at 55 and finished at 2350 . User 99 arrived at 58 and finished at 2450 . total system cpu is: 9800 total use cpu is: 6780 percentage is: 0.691837	<----Priority Scheduling with efficiency----> Here is the information of finished VM: User 24 arrived at 36 and finished at 152 . User 11 arrived at 40 and finished at 216 . User 2 arrived at 8 and finished at 236 . User 42 arrived at 52 and finished at 254 . User 15 arrived at 12 and finished at 375 . User 1 arrived at 0 and finished at 386 . User 45 arrived at 14 and finished at 860 . User 31 arrived at 46 and finished at 1061 . User 36 arrived at 20 and finished at 1301 . User 30 arrived at 5 and finished at 1419 . User 103 arrived at 55 and finished at 1583 . User 99 arrived at 58 and finished at 1683 . User 33 arrived at 64 and finished at 1714 . User 9 arrived at 41 and finished at 1724 . User 59 arrived at 50 and finished at 1773 . total system cpu is: 7092 total use cpu is: 6780 percentage is: 0.956007
30	2	5	220	4		
2	3	8	50	2		
15	1	12	300	1		
45	1	14	80	4		
36	2	20	180	4		
24	3	36	100	1		
11	3	40	160	1		
9	1	41	80	2		
31	2	46	120	4		
59	2	50	130	2		
42	1	52	160	1		
103	2	55	300	4		
99	3	58	400	4		
33	3	64	70	2		

E: Results From Test Data Sets

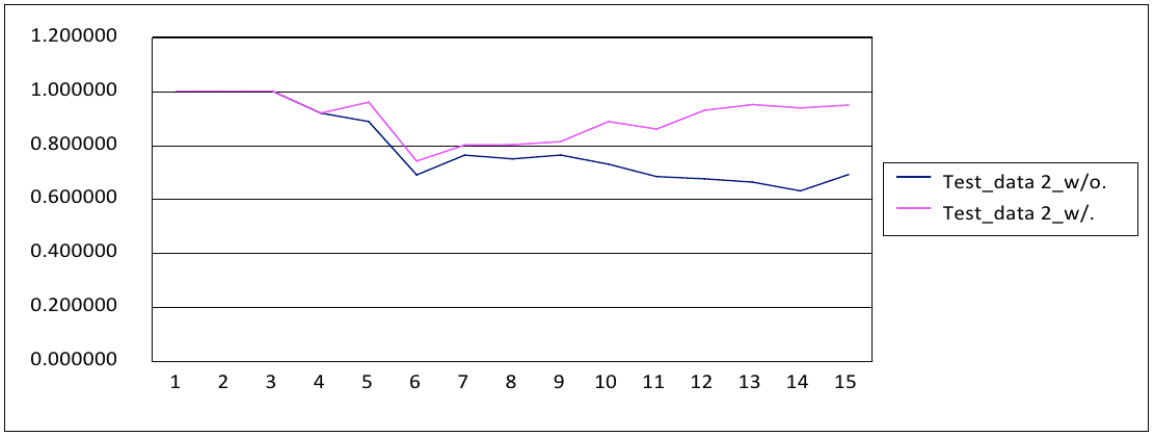
Results for Test Data 1

Number of Virtual Machines (X-Axis)	CPU Utilization Percentage w/o Efficiency (Y-Axis)	CPU Utilization Percentage w/ Efficiency (Y-Axis)
1	0.5	0.5
2	0.84375	0.84375
3	0.797297	0.842857
4	0.552239	0.580769
5	0.6	0.626712
6	0.677419	0.692308
7	0.635922	0.707921
8	0.584034	0.752577
9	0.57874	0.777778
10	0.615108	0.801802
11	0.605263	0.837838
12	0.571429	0.799587
13	0.636364	0.834437
14	0.697479	0.86911
15	0.691837	0.868687



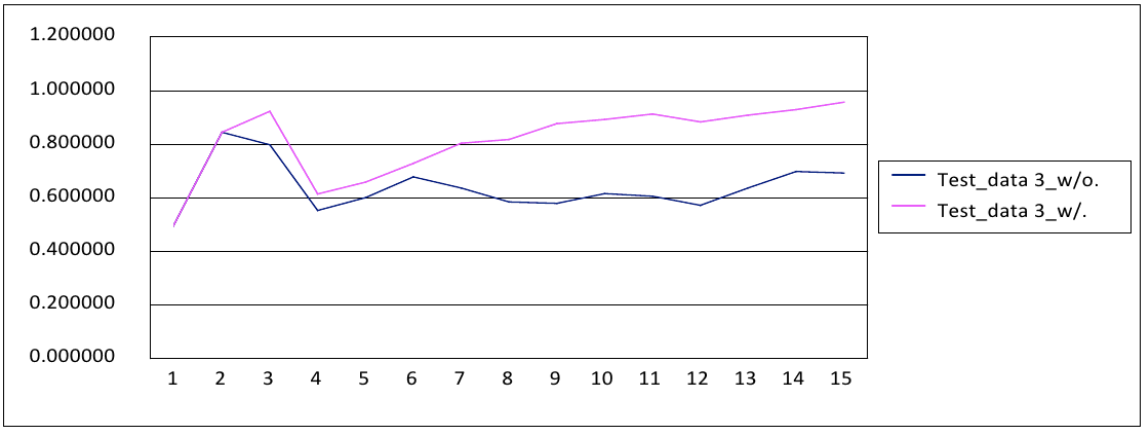
Results for Test Data 2

Number of Virtual Machines (X-Axis)	Test Data 2 w/o Efficiency (Y-Axis)	Test Data 2 w/Efficiency (Y-Axis)
1	1	1
2	1	1
3	1	1
4	0.919355	0.919355
5	0.88806	0.959677
6	0.690722	0.741848
7	0.76378	0.80123
8	0.75	0.802326
9	0.764085	0.813869
10	0.730263	0.888
11	0.684524	0.860294
12	0.676136	0.929688
13	0.664021	0.950758
14	0.631707	0.938406
15	0.691837	0.948895



Results for Test Data 3

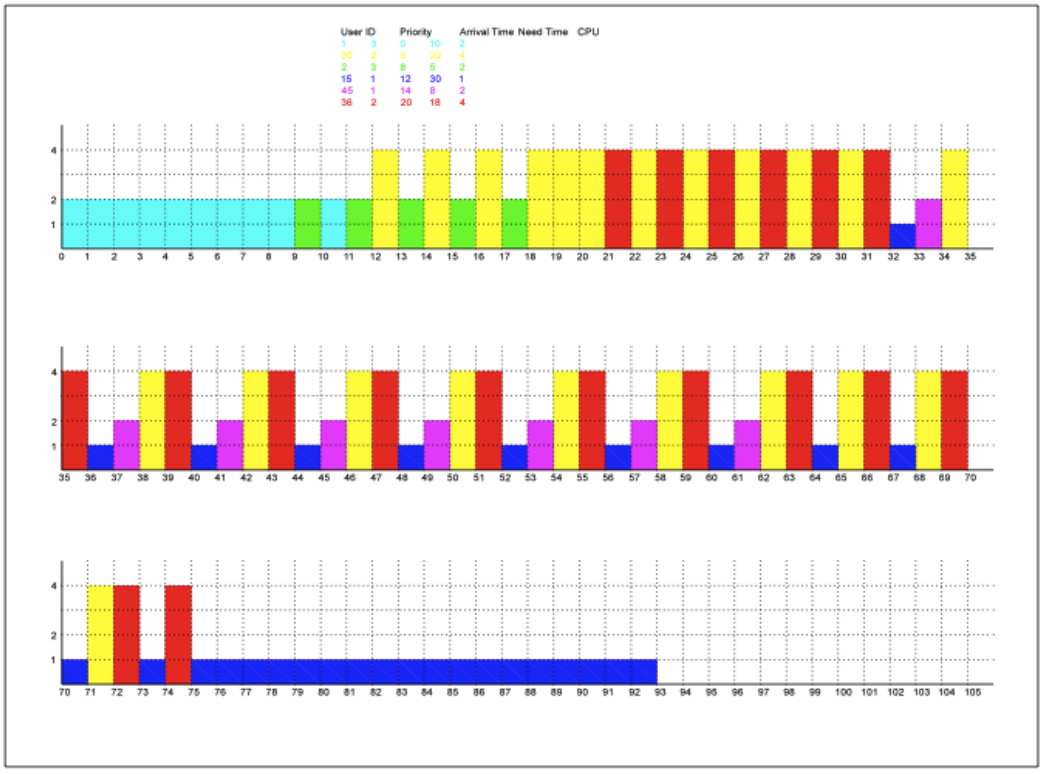
Number of Virtual Machines (X-Axis)	Test Data 3 w/o Efficiency (Y-Axis)	Test Data 3 w/Efficiency (Y-Axis)
1	0.5	0.5
2	0.84375	0.84375
3	0.797297	0.921875
4	0.552239	0.61371
5	0.6	0.657857
6	0.677419	0.727841
7	0.635922	0.803013
8	0.584034	0.816886
9	0.57874	0.876043
10	0.615108	0.891554
11	0.605263	0.911794
12	0.571429	0.882434
13	0.636364	0.907753
14	0.697479	0.928332
15	0.691837	0.956007



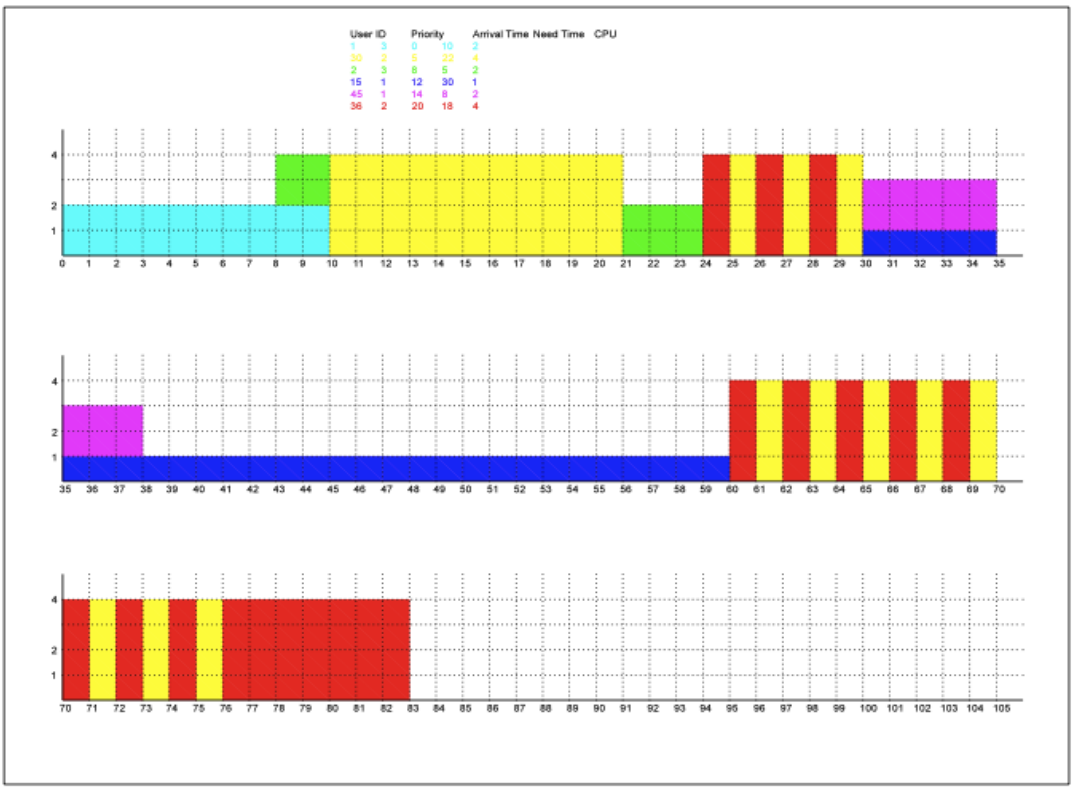
F: Proof of Output Correctness

User ID	Priority	Arrival Time	Need Time	CPU	Priority Scheduling without efficiency	Priority Scheduling with efficiency
1	3	0	10	2	<----Priority Scheduling without efficiency----> Here is the information of finished VM: User 1 arrived at 0 and finished at 11 . User 2 arrived at 8 and finished at 18 . User 45 arrived at 14 and finished at 62 . User 30 arrived at 5 and finished at 72 . User 36 arrived at 20 and finished at 75 . User 15 arrived at 12 and finished at 93 . total system cpu is: 372 total use cpu is: 236 percentage is: 0.634409	<----Priority Scheduling with efficiency----> Here is the information of finished VM: User 1 arrived at 0 and finished at 10 . User 2 arrived at 8 and finished at 24 . User 45 arrived at 14 and finished at 38 . User 15 arrived at 12 and finished at 60 . User 30 arrived at 5 and finished at 76 . User 36 arrived at 20 and finished at 83 . total system cpu is: 332 total use cpu is: 236 percentage is: 0.710843
30	2	5	22	4		
2	3	8	5	2		
15	1	12	30	1		
45	1	14	8	2		
36	2	20	18	4		

Without Efficiency:



With Efficiency:



G: Other Related Material

Team Contributions

Ge Lv (Team Lead)

Came up with idea, implementation, testing, created graphs, Part II submission, presentation

Michelle Chartier

Documentation, Part I submission, converting flowchart to UML diagram, presentation

Yeong-Jye Huang (James)

Created Part I slides, presentation