**Operating System : Course Project**

Implementation of Preemptive Scheduling Algorithm

(Rohit Pujari : 75 & Vivek Pundkar : 77)

21.11.2021

# INTRODUCTION

When you turn on your computer, it's nice to think that you're in control. There's the trusty mouse, which you can move anywhere on the screen, summoning up your music library or internet browser at the slightest whim. Although it's easy to feel like a director in front of your desktop or laptop, there's a lot going on inside, and the real person behind the curtain handling the necessary tasks is the operating system.

# WHAT IS OPERATING SYSTEM?

An operating system is a program that acts as an interface between the computer user and computer hardware, and controls the execution of programs.

The operating system (OS) manages all of the software and hardware on the computer. It performs basic tasks such as file, memory and process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Most of the time, there are several different computer programs running at the same time, and they all need to access your computer’s central processing unit (CPU), memory and storage. The operating system coordinates all of this to make sure each program gets what it needs.

# FUNCTIONS OF OPERATING SYSTEM

* Processor management
* Memory management
* Device management
* Storage management
* Application interface
* User interface
* System security management

# HOW OPERATING SYSTEM WORKS ?

* An operating system works in the middle of software and hardware. When a user gives an instruction or command through a software it passed to the operating system.
* The operating system allocates memory from the RAM and gives that instruction to the CPU for executing it. Multi-tasking is also done in this way.
* Certain programs in a computer are given high, medium and low priorities and the instructions are executed according to these priorities.
* For example, whenever explorer in your Windows operating system crashes you will press 'Alt+Ctrl+Del' for task manager and we use task manager to end the non-responding program. In Windows, task manager is given a high priority and that's why it opens even if explorer is crashed.

# WHAT IS PROCESS ?

* Process is the execution of a program that performs the actions specified in that program.
* Process operations can be easily controlled with the help of PCB(Process Control Block). You can consider it as the brain of the process, which contains all the crucial information related to processing like process id, priority, state, CPU registers, etc.

## PROCESS ARCHITECTURE

* **Stack:** The Stack stores temporary data like function parameters, returns addresses, and local variables.
* **Heap:** Allocates memory, which may be processed during its run time.
* **Data:** It contains the variable.
* **Text:** Text Section includes the current activity, which is represented by the value of the Program Counter.

## 

# WHAT IS PROCESS MANAGEMENT?

Process management involves various tasks like creation, scheduling, termination of processes, and a dead lock. Process is a program that is under execution, which is an important part of modern-day operating systems.

The OS must allocate resources that enable processes to share and exchange information. It also protects the resources of each process from other methods and allows synchronization among processes.

It is the job of OS to manage all the running processes of the system. It handles operations by performing tasks like process scheduling and such as resource allocation.

# PROCESS SCHEDULING

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

* Job queue − This queue keeps all the processes in the system.
* Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

# TYPES OF PROCESS SCHEDULING

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Long-Term Scheduler** | **Short-Term Scheduler** | **Medium-Term Scheduler** |
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

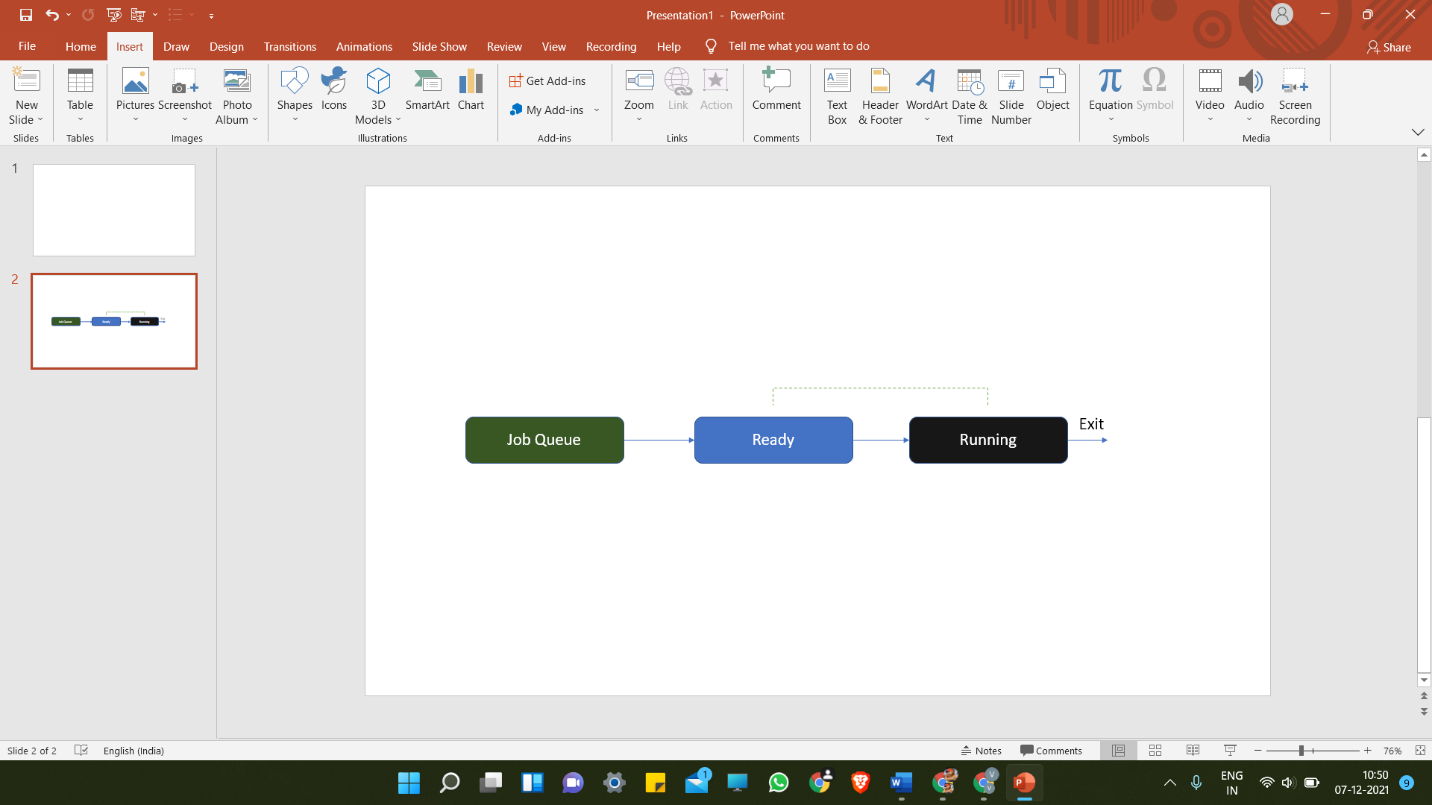
# PREEMPTIVE SCHEDULING ALGORITHM

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

**Algorithms based on preemptive scheduling :**

* Round Robin (RR)
* Shortest Remaining Time First (SRTF)
* Priority (preemptive version)

**Time Quantum**

A preemptive scheduler will allow a particular process to run for a short amount of time called a quantum (or time slice). After this amount of time, the process is placed back in the ready queue and another process is placed into the run state (i.e., the scheduler ensures that the processes take turns to run). 

* The size of a quantum has to be selected carefully. Each time the operating system makes a scheduling decision, it is itself using the processor. This is because the operating system comprises one or more processes and it has to use system processing time to do its own computations in order to decide which process to run and actually move them from state to state;
* If the time quanta are too short, the operating system has to perform scheduling activities more frequently, and thus, the overheads are higher as a proportion of total system processing resource.
* On the other hand, if the quanta are too long, the other processes in the ready queue must wait longer between turns, and there is a risk that the users of the system will notice a lack of responsiveness in the applications to which these processes belong.

**Round Robin (RR)**

Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also known as time quanta)are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive).

Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept.

Real time Example:

If the time slot is 100 milliseconds, and job1 takes a total time of 250 ms to complete, the round-robin scheduler will suspend the job after 100 ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100 ms each), job1 will get another allocation of CPU time and the cycle will repeat. This process continues until the job finishes and needs no more time on the CPU.

Job1 = Total time to complete 250 ms (quantum 100 ms).

First allocation = 100 ms.

Second allocation = 100 ms.

Third allocation = 100 ms but job1 self-terminates after 50 ms.

Total CPU time of job1 = 250 ms

Consider the following table with the arrival time and execute time of the process with the quantum time of 100 ms to understand the round-robin scheduling:

|  |  |  |
| --- | --- | --- |
| **Process name** | **Arrival time** | **Execute time** |
| P0 | 0 | 250 |
| P1 | 50 | 170 |
| P2 | 130 | 75 |
| P3 | 190 | 100 |
| P4 | 210 | 130 |
| P5 | 350 | 50 |

|  |  |  |
| --- | --- | --- |
| **Process name** | **Arrival time** | **Execute time** |
| P1 | 0 | 5 |
| P2 | 1 | 4 |
| P3 | 2 | 2 |
| P4 | 4 | 1 |

------ \* -------

Ready Queue

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P1 | P4 | P2 | P1 |  |

Running Queue(GANTT chart)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P1 | P4 | P2 | P1 |  |

0 2 4 6 8 9 11 12

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process No | Arrival Time | Burst Time | Completion Time | TAT | WT | RT |
| P1 | 0 | 5 | 12 | 12 | 7 | 0 |
| P2 | 1 | 4 | 11 | 10 | 6 | 1 |
| P3 | 2 | 2 | 6 | 4 | 2 | 2 |
| P4 | 4 | 1 | 9 | 5 | 4 | 4 |

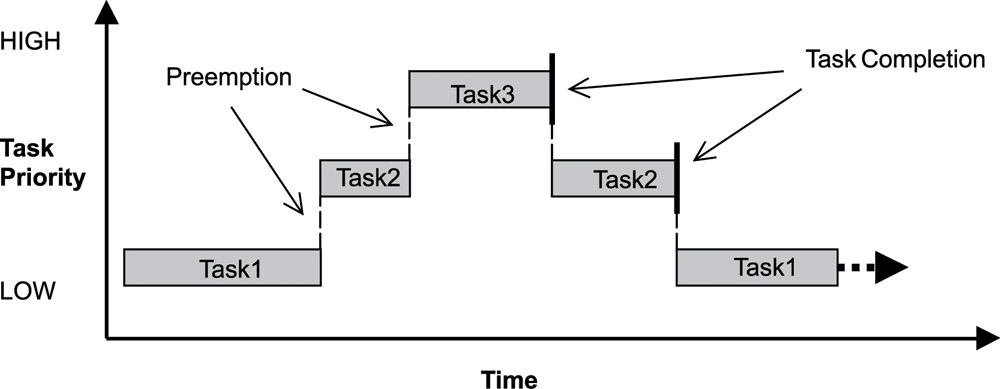
                    Average Waiting Time = (7+6+2+4)/4 = 19/4 units

**Priority (preemptive version)**

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and non-preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.



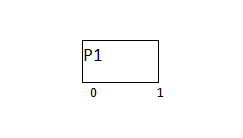
Example

There are 7 processes P1, P2, P3, P4, P5, P6 and P7 given. Their respective priorities, Arrival Times and Burst times are given in the table below.

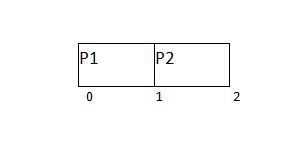
|  |  |  |  |
| --- | --- | --- | --- |
| **Process Id** | **Priority** | **Arrival Time** | **Burst Time** |
| 1 | 2(L) | 0 | 1 |
| 2 | 6 | 1 | 7 |
| 3 | 3 | 2 | 3 |
| 4 | 5 | 3 | 6 |
| 5 | 4 | 4 | 5 |
| 6 | 10(H) | 5 | 15 |
| 7 | 9 | 15 | 8 |

### GANTT chart Preparation

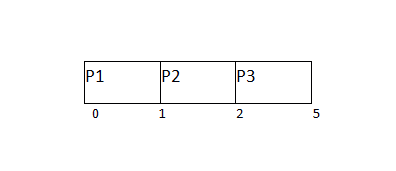
At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).



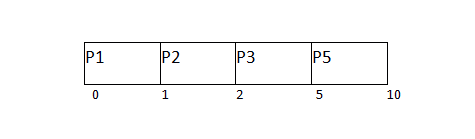
At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



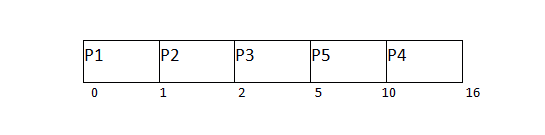
The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the CPU.



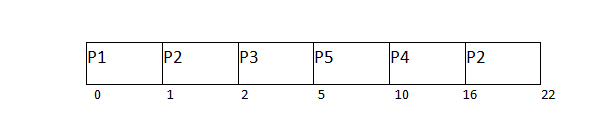
During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so PS can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.



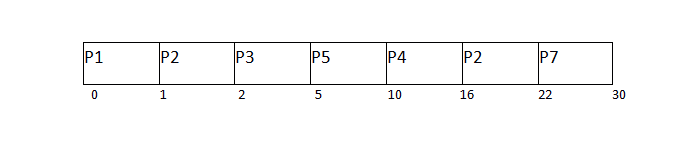
Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



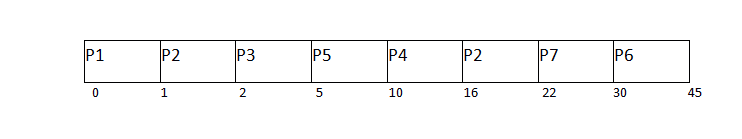
Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Process Id** | **Priority** | **Arrival Time** | **Burst Time** | **Completion Time** | **Turnaround Time** | **Waiting Time** |
| 1 | 2 | 0 | 1 | 1 | 1 | 0 |
| 2 | 6 | 1 | 7 | 22 | 21 | 14 |
| 3 | 3 | 2 | 3 | 5 | 3 | 0 |
| 4 | 5 | 3 | 6 | 16 | 13 | 7 |
| 5 | 4 | 4 | 5 | 10 | 6 | 1 |
| 6 | 10 | 5 | 15 | 45 | 40 | 25 |
| 7 | 9 | 6 | 8 | 30 | 24 | 16 |

# ROUND ROBIN:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process No | Arrival Time | Burst Time | Completion Time | TAT | WT | RT |
| P1 | 0 | 5 | 12 | 12 | 7 | 0 |
| P2 | 1 | 4 | 11 | 10 | 6 | 1 |
| P3 | 2 | 2 | 6 | 4 | 2 | 2 |
| P4 | 4 | 1 | 9 | 5 | 4 | 4 |

Time quantum: 2

# ALGORITHM :

* Stable sort the processes in order of arrival time in ascending order.
* We will be using a FIFO queue to implement this algorithm
* We will first push the first process from the sorted list into queue.
* We will use a array to check if the process is in the queue or not.
* Keep track of the time using a variable - current\_time
* If the process is getting CPU for the first time, record its start time as current\_time.
* Give quantum unit of time to the process that is at front in the queue and pop this process from the queue.
* If the burst time of this process becomes 0, calculate CT, TAT, WT and RT for it.
* If some process has arrived when this process was executing, insert them into the queue.
* If the current process has burst time remianing, push the process into queue again.
* If the queue is empty, pick the first process from the list that is not completed.
* Keep doing this till all processes are completed.

# IMPLEMENTATION:

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <queue>

#include <string.h>

using namespace std;

struct process {

int pid;

int arrival\_time;

int burst\_time;

int start\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int response\_time;

};

bool compare1(process p1, process p2)

{

return p1.arrival\_time < p2.arrival\_time;

}

bool compare2(process p1, process p2)

{

return p1.pid < p2.pid;

}

int main() {

int n;

int tq;

struct process p[100];

float avg\_turnaround\_time;

float avg\_waiting\_time;

float avg\_response\_time;

float cpu\_utilisation;

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

int total\_response\_time = 0;

int total\_idle\_time = 0;

float throughput;

int burst\_remaining[100];

int idx;

cout << setprecision(2) << fixed;

cout<<"Enter the number of processes: ";

cin>>n;

cout<<"Enter time quantum: ";

cin>>tq;

for(int i = 0; i < n; i++) {

cout<<"Enter arrival time of process "<<i+1<<": ";

cin>>p[i].arrival\_time;

cout<<"Enter burst time of process "<<i+1<<": ";

cin>>p[i].burst\_time;

burst\_remaining[i] = p[i].burst\_time;

p[i].pid = i+1;

cout<<endl;

}

sort(p,p+n,compare1);

queue<int> q;

int current\_time = 0;

q.push(0);

int completed = 0;

int mark[100];

memset(mark,0,sizeof(mark));

mark[0] = 1;

while(completed != n) {

idx = q.front();

q.pop();

if(burst\_remaining[idx] == p[idx].burst\_time) {

p[idx].start\_time = max(current\_time,p[idx].arrival\_time);

total\_idle\_time += p[idx].start\_time - current\_time;

current\_time = p[idx].start\_time;

}

if(burst\_remaining[idx]-tq > 0) {

burst\_remaining[idx] -= tq;

current\_time += tq;

}

else {

current\_time += burst\_remaining[idx];

burst\_remaining[idx] = 0;

completed++;

p[idx].completion\_time = current\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

}

for(int i = 1; i < n; i++) {

if(burst\_remaining[i] > 0 && p[i].arrival\_time <= current\_time && mark[i] == 0) {

q.push(i);

mark[i] = 1;

}

}

if(burst\_remaining[idx] > 0) {

q.push(idx);

}

if(q.empty()) {

for(int i = 1; i < n; i++) {

if(burst\_remaining[i] > 0) {

q.push(i);

mark[i] = 1;

break;

}

}

}

}

avg\_turnaround\_time = (float) total\_turnaround\_time / n;

avg\_waiting\_time = (float) total\_waiting\_time / n;

avg\_response\_time = (float) total\_response\_time / n;

cpu\_utilisation = ((p[n-1].completion\_time - total\_idle\_time) / (float) p[n-1].completion\_time)\*100;

throughput = float(n) / (p[n-1].completion\_time - p[0].arrival\_time);

sort(p,p+n,compare2);

cout<<endl;

cout<<"#P\t"<<"AT\t"<<"BT\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;

for(int i = 0; i < n; i++) {

cout<<p[i].pid<<"\t"<<p[i].arrival\_time<<"\t"<<p[i].burst\_time<<"\t"<<p[i].start\_time<<"\t"<<p[i].completion\_time<<"\t"<<p[i].turnaround\_time<<"\t"<<p[i].waiting\_time<<"\t"<<p[i].response\_time<<"\t"<<"\n"<<endl;

}

cout<<"Average Turnaround Time = "<<avg\_turnaround\_time<<endl;

cout<<"Average Waiting Time = "<<avg\_waiting\_time<<endl;

cout<<"Average Response Time = "<<avg\_response\_time<<endl;

}

# 

# PRIORITY :

# ALGORITHM :

completed = 0

current\_time = 0

while(completed != n) {

find process with maximum priority time among process that are in ready queue at current\_time

if(process found) {

if(process is getting CPU for the first time) {

start\_time = current\_time

}

burst\_time = burst\_time - 1

current\_time = current\_time + 1

if(burst\_time == 0) {

completion\_time  = current\_time

turnaround\_time = completion\_time - arrival\_time

waiting\_time = turnaround\_time - burst\_time

response\_time = start\_time - arrival\_time

mark process as completed

completed++

}}

else {

current\_time++

}}

# IMPLEMENTATION:

#include <iostream>

#include <iomanip>

#include <string.h>

using namespace std;

struct process {

    int pid;

    int arrival\_time;

    int burst\_time;

    int priority;

    int start\_time;

    int completion\_time;

    int turnaround\_time;

    int waiting\_time;

    int response\_time;

};

int main() {

    int n;

    struct process p[100];

    float avg\_turnaround\_time;

    float avg\_waiting\_time;

    float avg\_response\_time;

    int total\_turnaround\_time = 0;

    int total\_waiting\_time = 0;

    int total\_response\_time = 0;

    int total\_idle\_time = 0;

    int burst\_remaining[100];

    int is\_completed[100];

    memset(is\_completed,0,sizeof(is\_completed));

    cout<<"Enter the number of processes: ";

    cin>>n;

    for(int i = 0; i < n; i++) {

        cout<<"Enter arrival time of process "<<i+1<<": ";

        cin>>p[i].arrival\_time;

        cout<<"Enter burst time of process "<<i+1<<": ";

        cin>>p[i].burst\_time;

        cout<<"Enter priority of the process "<<i+1<<": ";

        cin>>p[i].priority;

        p[i].pid = i+1;

        burst\_remaining[i] = p[i].burst\_time;

        cout<<endl;

    }

    int current\_time = 0;

    int completed = 0;

    int prev = 0;

    while(completed != n) {

        int idx = -1;

        int mx = -1;

        for(int i = 0; i < n; i++) {

            if(p[i].arrival\_time <= current\_time && is\_completed[i] == 0) {

                if(p[i].priority > mx) {

                    mx = p[i].priority;

                    idx = i;

                }

                if(p[i].priority == mx) {

                    if(p[i].arrival\_time < p[idx].arrival\_time) {

                        mx = p[i].priority;

                        idx = i;

                    }

                }

            }

        }

        if(idx != -1) {

            if(burst\_remaining[idx] == p[idx].burst\_time) {

                p[idx].start\_time = current\_time;

                total\_idle\_time += p[idx].start\_time - prev;

            }

            burst\_remaining[idx] -= 1;

            current\_time++;

            prev = current\_time;

            if(burst\_remaining[idx] == 0) {

                p[idx].completion\_time = current\_time;

                p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

                p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

                p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

                total\_turnaround\_time += p[idx].turnaround\_time;

                total\_waiting\_time += p[idx].waiting\_time;

                total\_response\_time += p[idx].response\_time;

                is\_completed[idx] = 1;

                completed++;

            }

        }

        else {

             current\_time++;

        }

    }

    avg\_turnaround\_time = (float) total\_turnaround\_time / n;

    avg\_waiting\_time = (float) total\_waiting\_time / n;

    avg\_response\_time = (float) total\_response\_time / n;

    cout<<endl<<endl;

    cout<<"#P\t"<<"AT\t"<<"BT\t"<<"PRI\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;

    for(int i = 0; i < n; i++) {

        cout<<p[i].pid<<"\t"<<p[i].arrival\_time<<"\t"<<p[i].burst\_time<<"\t"<<p[i].priority<<"\t"<<p[i].start\_time<<"\t"<<p[i].completion\_time<<"\t"<<p[i].turnaround\_time<<"\t"<<p[i].waiting\_time<<"\t"<<p[i].response\_time<<"\t"<<"\n"<<endl;

    }

    cout<<"Average Turnaround Time = "<<avg\_turnaround\_time<<endl;

    cout<<"Average Waiting Time = "<<avg\_waiting\_time<<endl;

    cout<<"Average Response Time = "<<avg\_response\_time<<endl;

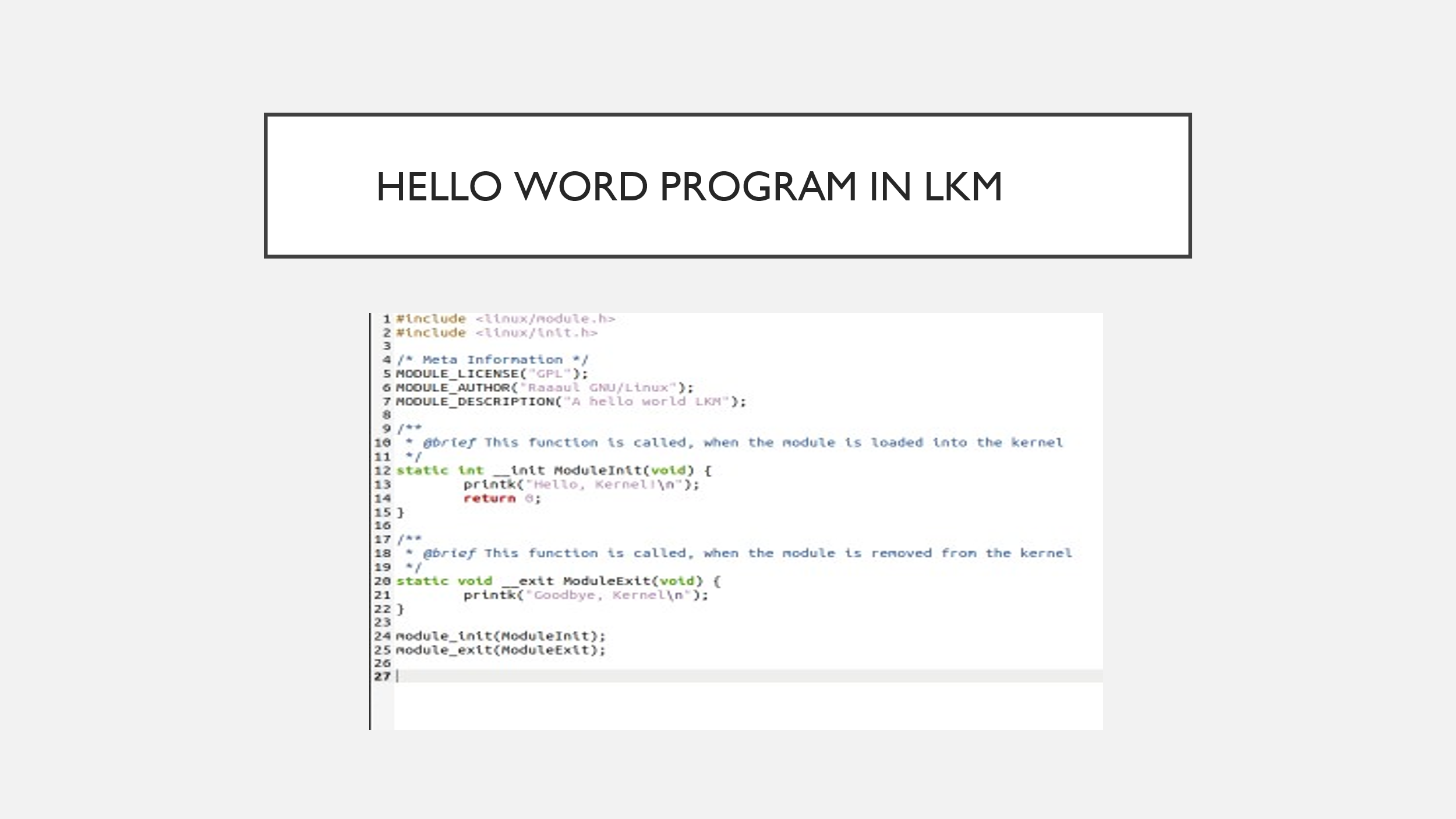
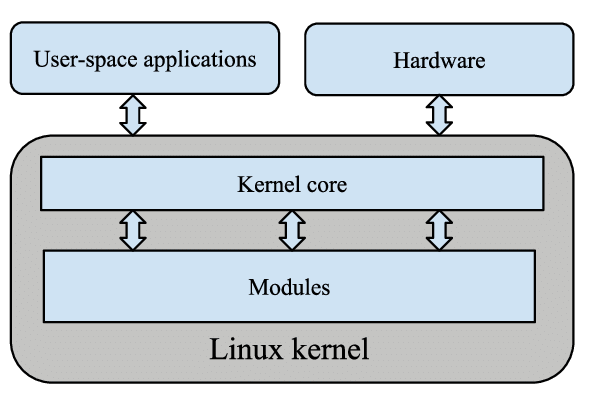
}

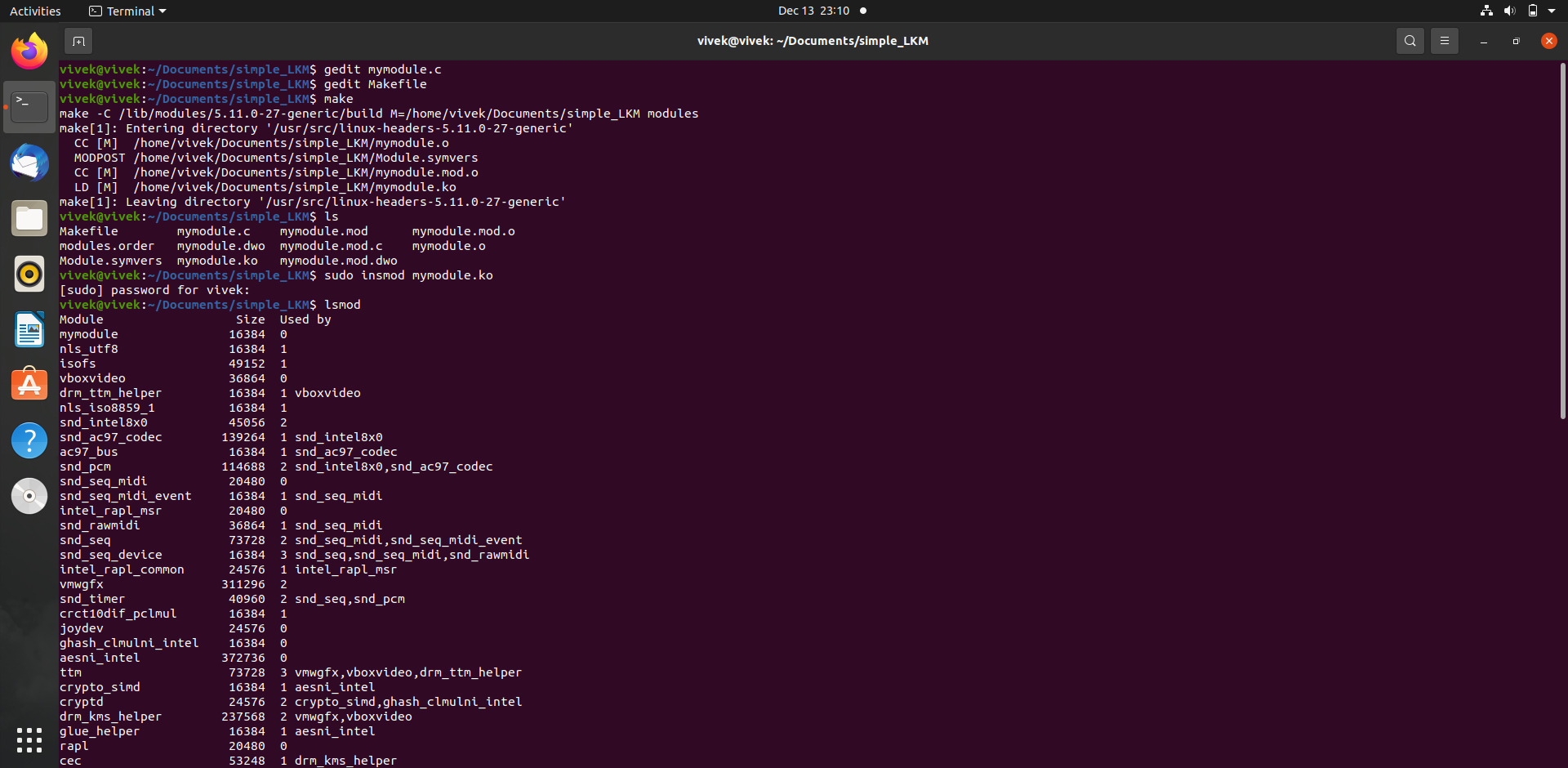
# 

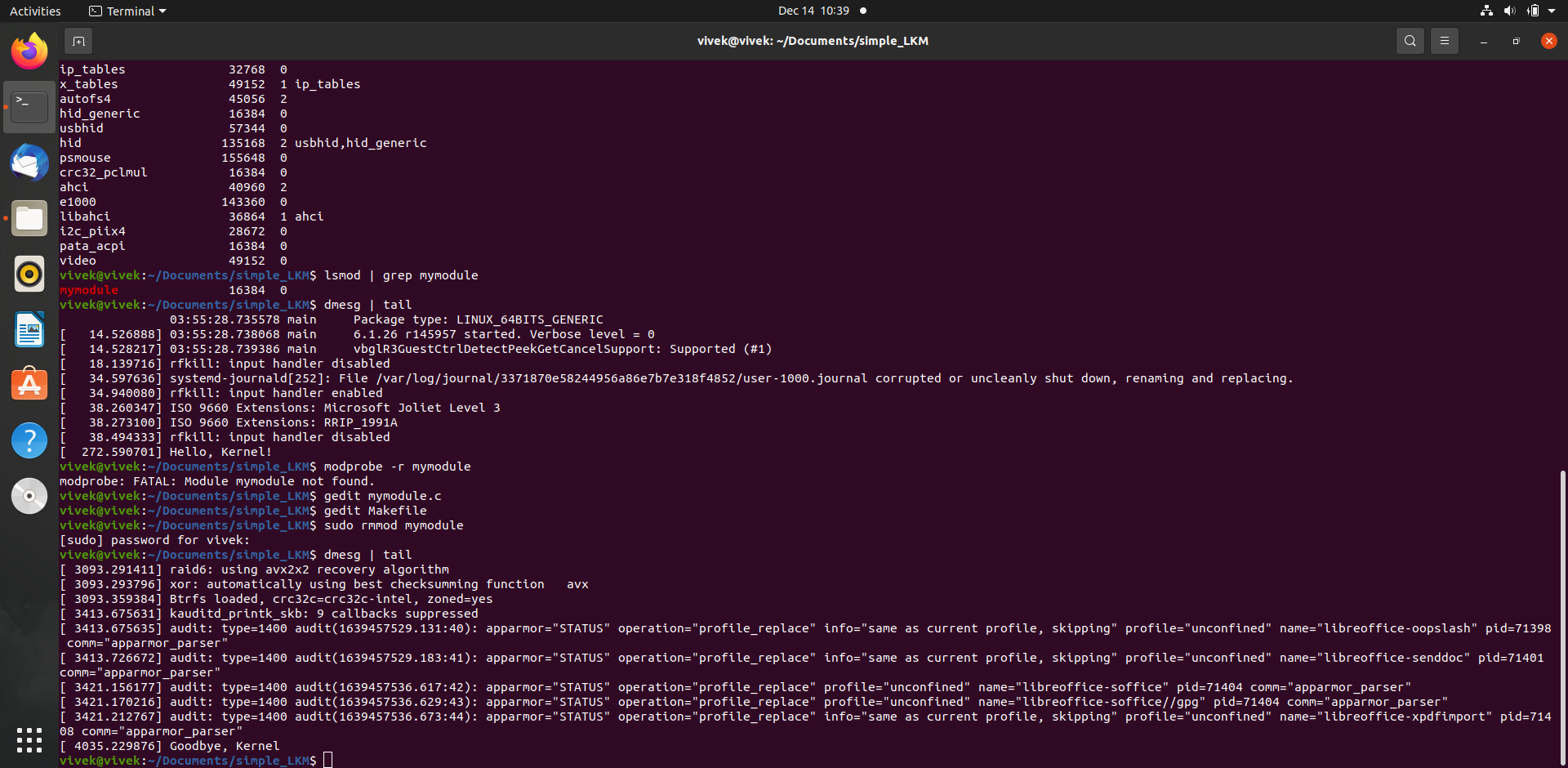
# KERNAL STUDY

A Kernel Module is a small file that may be loaded into the running Kernel, and unloaded, at will (the next generation of the Linux kernel will not allow unloading modules, unless told otherwise).

Many functions of the Kernel may either be compiled into the Kernel directly, or compiled as Modules. This design makes it faster to load the Kernel (no need to load and initialize un-needed Modules), as well as developing Drivers (if you have a bug, just unload the Module, fix it, recompile and load it again).







# REFERENCES

[1]. Abbas, A. A. (2015). Multiprocessor and Real-Time Scheduling Shortest-Job-First (SJF) Scheduling Algorithm. European Journal of Computer

Science and Information Technology, Vol 3, No. 5, 8-20.

[2]. Hasan, T. F. (2014). CPU Scheduling Visualization. Diyala Jounal of Engineering Sciences Vol 07. No. 01 March, 16-29.

[3]. Ji Young Heo, M. P. (2017). Shortest Job First Scheduling for Reducing Jitter in Cyber Physical Systems. International Conference Embedded

Systems, Cyber-physycal Systems, & applications.

[4]. Kuldeep Singh Kaswan, A. (2017). A New Technique for CPU Scheduling: Standard Deviation Based. International Journal of Advanced

Research in Computer Engineering & Technology, Volume 6, Issue 8, August