CHAPTER 1

INTRODUCTION

1.1 Device Drivers:

One of the many advantages of free operating systems, as typified by Linux, is that their internals are open for all to view. The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity.

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

The open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

1.2 Role of Device Driver:

This privileged role of the driver allows the driver programmer to choose exactly how the device should appear. Different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. You could implement memory mapping on the device independently of its hardware capabilities, or you could provide a user library to help application programmers implement new policies on top of the available primitives, and so forth. One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple

1.3 Loadable Modules:

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel while the system is up and running.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types of modules, including, but not limited to, device drivers. Each module is made up of object code that can be dynamically linked to the running kernel by the **insmod** program and can be unlinked by the **rmmod** program.

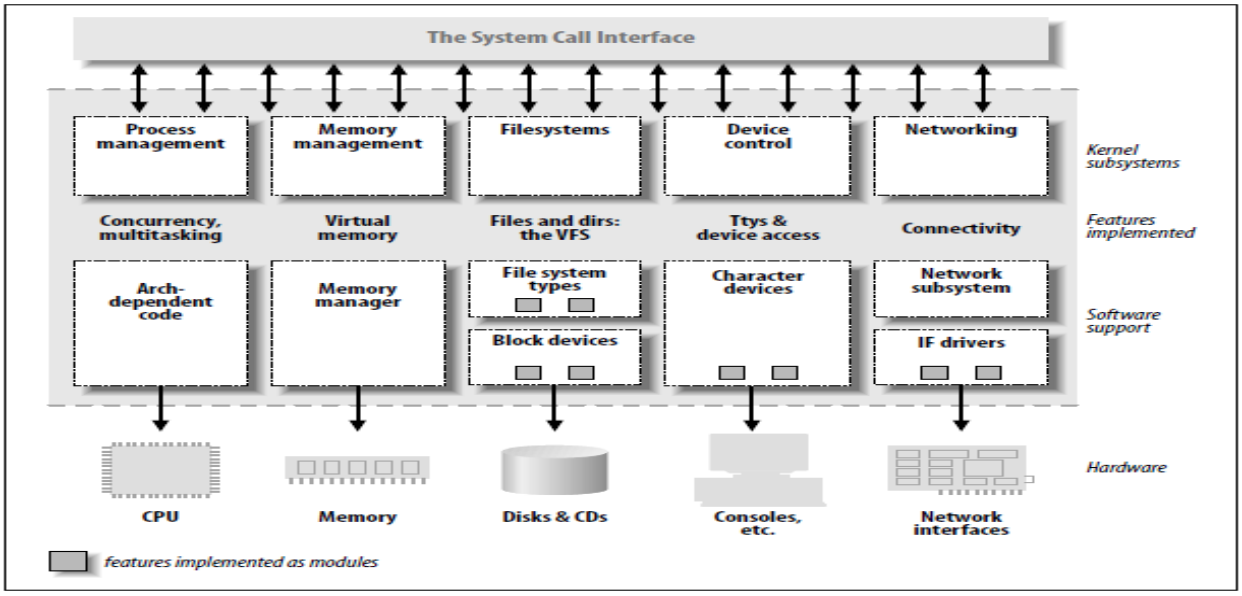


Fig 1. A split view of the kernel

1.4 Types of Device Drivers:

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. The 3 classes are:

- **Character devices:**

A character (char) device is one that can be accessed as a stream of bytes (like a file). A char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless,

char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

- **Block Devices:**

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a file system node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

- **Network Interfaces:**

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

CHAPTER 2

LITERATURE SURVEY

To Understand Linux device drivers, categorizing the drivers according to their interfaces is necessary. Considering all device drivers, bus drivers and virtual drivers that constitute the driver directories (/sound and /drivers) in the Linux kernel. Following are the categorization done according to the device driver developments that have occurred over a period of time

Sl No	Year	OS	Name	Features
1	1990	Windows 3.0	Virtual Device Drivers [1]	-Creating and maintaining a variety of virtual machines. -Execute different programs in different processor contexts. -Replaced the real-mode drivers with new virtual device drivers (VDD).
2	1993	Windows NT	Authorized Drivers [2]	-Lengthy Code for Drivers -Fully managed on its own -Less priority for interrupts -Hardware Interfaced Directly
3	1995	Windows 95	Windows Driver Model[3]	-Single Driver Model -Written in Higher Level Language C -Compatible with previous versions -Compiled using DDK
4	2002	Linux 1.8.3	CPN Mouse [4]	-Filter drivers specific for mouse -Investigation of Advanced Interaction Techniques -Non Generic Method of interfacing API -Limitations: Supports only limited CPN and no other framework for drivers
5	2004	Linux 2.3.0	Framework for enabling collaboration of dual drivers [4]	-Supporting multiple mouse -New Framework developed -Client Server Arch Using Sockets -Keep track of user intrface component states
6	2007	Linux 2.6.32	SDG Toolkit for Interface Drivers [4]	-Interrupts Introduced for multiple mouse interface in drivers -Easy and Rapid Prototyping

7	2010	Linux Multiseat 5.0	Linux Multipoint [4]	-Multiple Interrupts for mouse and multiple interaction -Driver developed in distinction of multiple mouse cursors -Drivers developed with C# -Easier and Rapid prototyping
8	2013	Linux 4.3.6	Embedded Compact Interrupt Mouse Device Driver [5]	-Encapsulation and Abstraction the functionality of its underlying hardware -Makes functionality availability to OS -Doesn't take Hardware implementation into consideration

The latest kernel of the linux is 4.5.0 which is still unstable and is believed to have a capability of supporting multiple mouse device driver support which means operating multiple instances of mouse at the same time and this multiple cursors appears on the screen wherein each operates individually [6]

CHAPTER 3

PROBLEM STATEMENT AND EVALUATION PLAN

Problem Statement:

To develop a device driver for mouse on Linux operating system involving high priority interrupt as to be counter incremented in the background and display the same to the user by reading the contents from the memory and displaying it on the monitor terminal.

Evaluation Plan:

The main objective is to enable interrupt based counter for mouse in Linux. By the end of this project implementation, advantages and disadvantages of the different ways of enabling a interrupt based device driver for mouse should be known. Generating the interrupts on the number of times the mouse moved and getting it counted in the incremental counter gives success.. Apart from just counting the interrupts generated by the mouse, objective of this project is to also do polling if a function during data transfer, read the data and write the data from the buffer on to the memory/ monitor and clear the buffer for next content of data.

CHAPTER 4

METHODOLOGY

Device drivers are typically written in C, using the Driver Development Kit (DDK). There are functional and object-oriented ways to program drivers, depending on the language chosen to write in. It is generally not possible to program a driver in Visual Basic or other high-level languages.

Because drivers operate in kernel mode, there are no restrictions on the actions that a driver may take. A driver may read and write to protected areas of memory, it may access I/O ports directly, and can generally do all sorts of very powerful things. This power makes drivers exceptionally capable of crashing an otherwise stable system. The following picture below depicts the basic methodology of the project

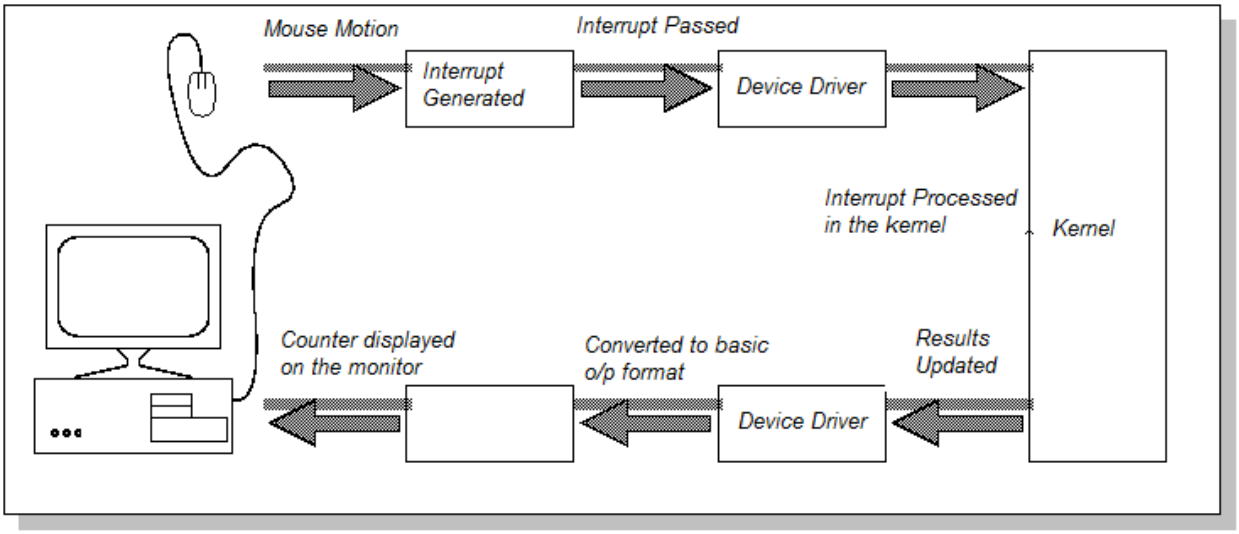


Fig 2 Methodology of the kernel

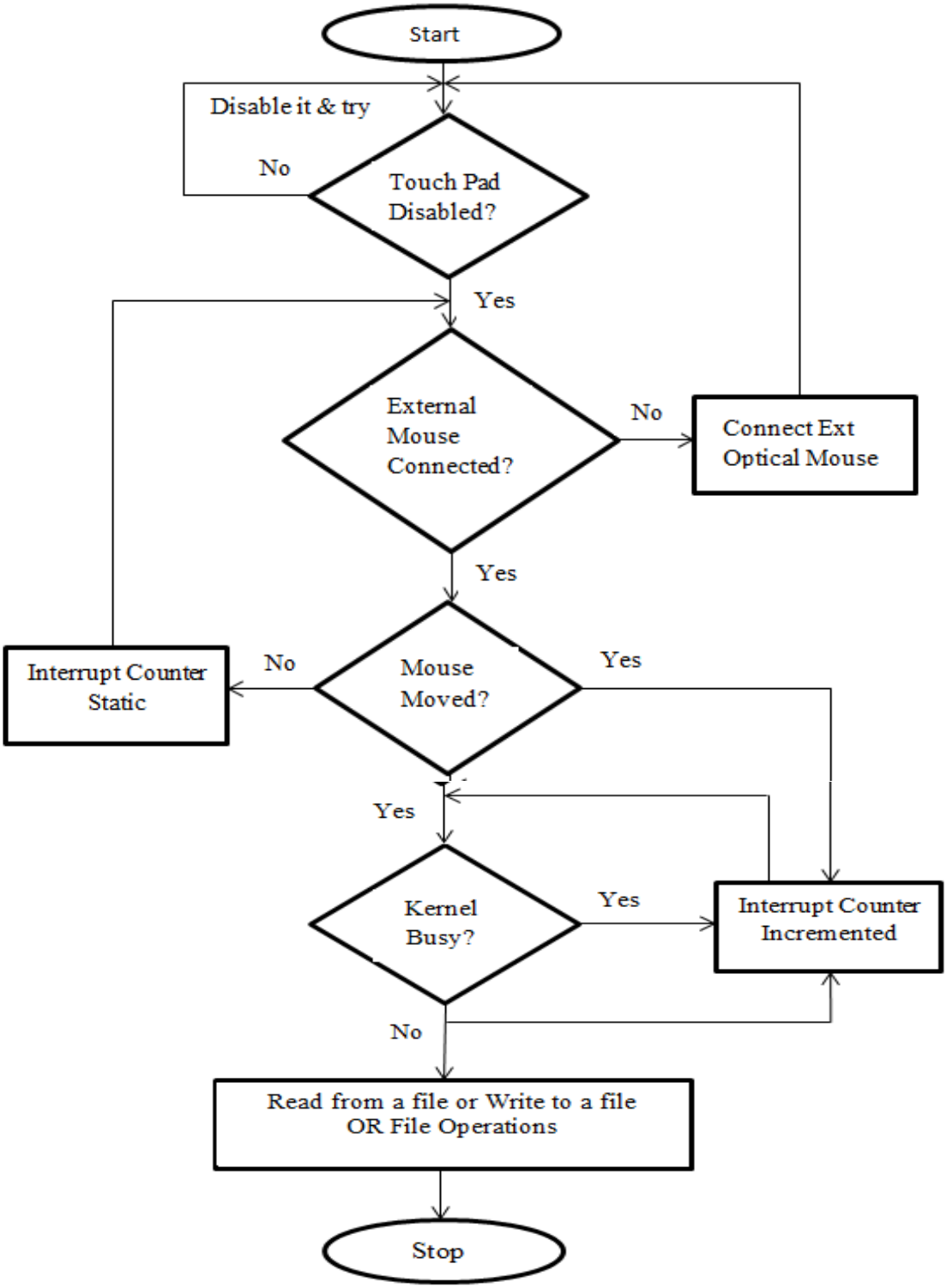
After determining the most feasible way of generating interrupts and incrementing the counter and other aspects, the application is planned to be written in the high level language C. The written device driver code for the mouse would then be cross compiled and the be implemented on the inbuilt hardware. During the following process no external hardware other than USB mouse will be used since there is no requirement of it. This application will be placed within the device drivers already built for the current hardware for the Linux kernel. Knowing the priority and the device id of the currently using mouse, disabling the touch pad/mouse currently using and making it to be used by the external mouse and noting the incremented

counter and then is being printed on the screen. Thus the previous and the current device attached use the same device driver written but alternatively. The disabling of one of the device is done as the user may get confused as to which is being operated, which is being used and for which and whose interrupt the counter would increment. The device driver applications will be constructed using the agile methodology. To begin with, a simple device driver program is being implemented and then improvements are being done as and how required such as counter incrementation, polling, reading or waiting for data, writing the data from buffer to memory or output. Thus the project is implemented part by part. As and when changes are required, they are implemented and final required product is obtained.

CHAPTER 5

ALGORITHM AND FLOWCHART

5.1 Flowchart:



5.2 Algorithm:

Step 1: Initially launch the driver written and insert the module into the kernel by using commands.

Step 2: Check whether the touchpad of the system is disabled. If not, disable it else it will hamper the interrupt count of the originally written device driver.

Step 3: Check whether the external optical mouse has been connected. If not, connect external optical mouse and also verify that the touchpad of the system is being disabled.

Step 4: If the external mouse has been connected through the USB port, check whether the connected mouse is moved.

Step 5: If the external mouse is moved, interrupts are generated and thus the Interrupt Counter is incremented. If External Mouse is not moved, the increment counter is not changed an thus the USB mouse connection is again verified.

Step 6: The kernel is checked whether it is busy. If busy then too the interrupts are being generated and thus interrupt counter increment and thus it polls kernel again until it is free. If the kernel is not busy, then the file operations are done such as Reading or Writing a file.

CHAPTER 6

STEPS TO DESIGN AND DEBUG A DEVICE DRIVER

The following device driver is being designed on VMWare workstation for USB optical optical mouse specifically using Linux Platforms.

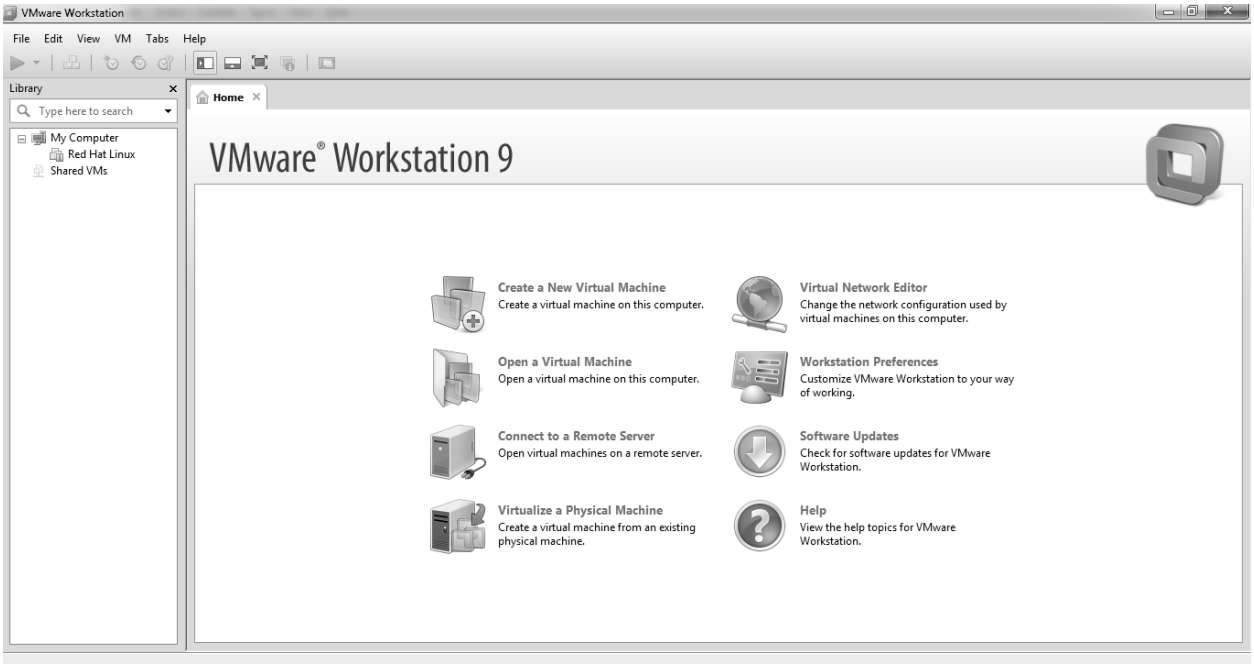


Fig 3. Setting up a VMWare Linux Instance

After creating the virtual instance of linux, enter the username and password for authentication and entry as a root. All the administrator privileges will be provided in the root mode. Writing the program indicates as shown below.

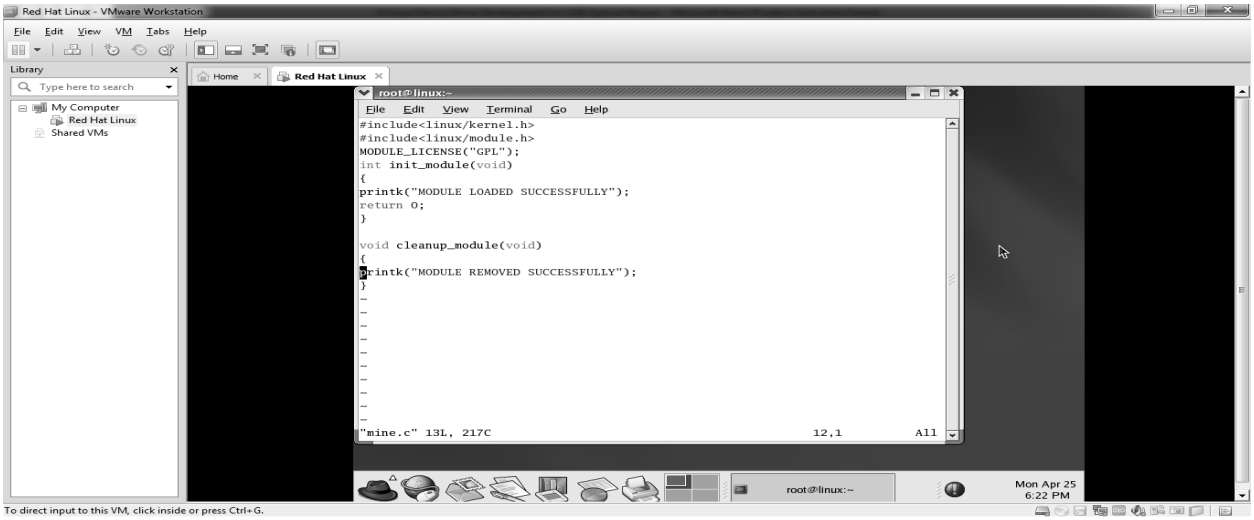


Fig 4. Writing a device driver program

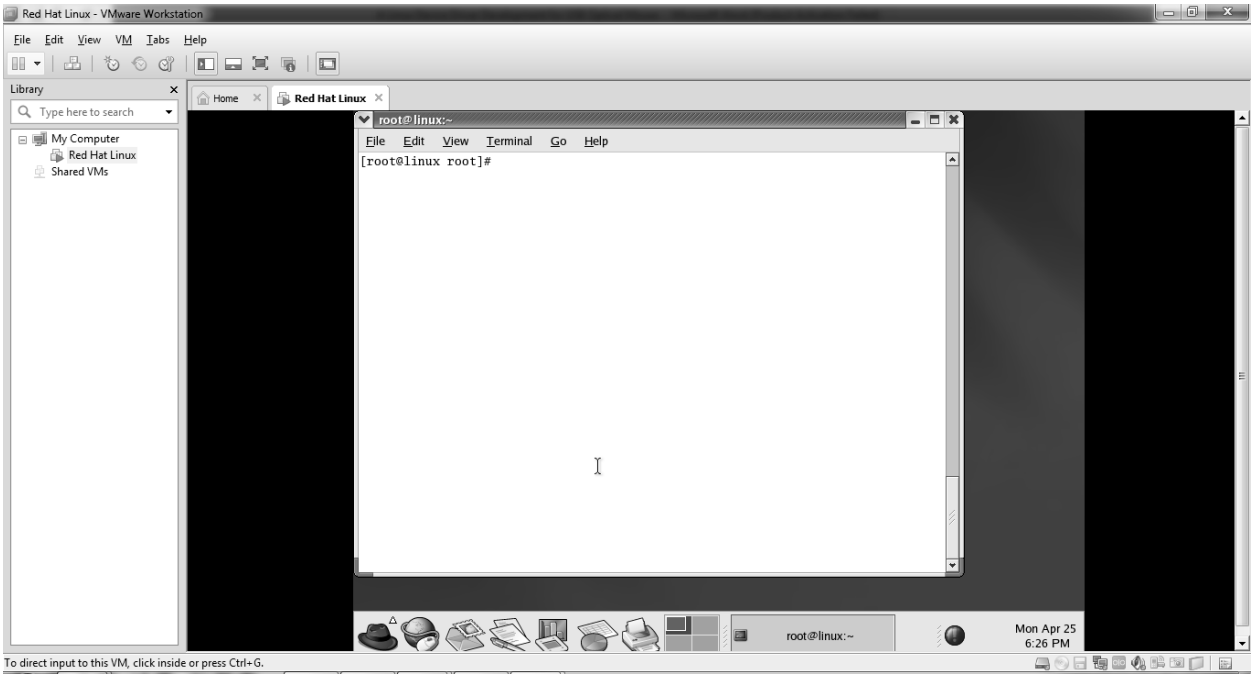


Fig 5: Screen showing execution of the device driver program

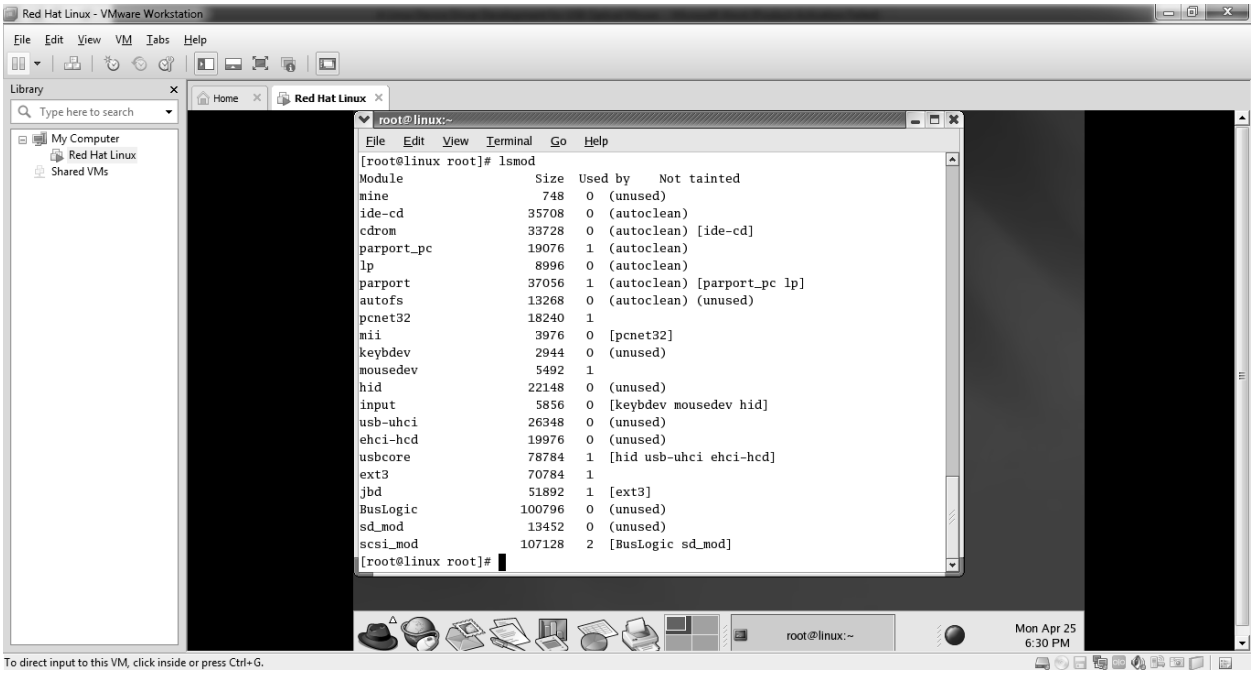


Fig 6: Screen showing various device drivers with sizes, used by, Module name. Module named "Mine" being inserted as the second driver

Finally view the number of interrupts generated, Module name, Parameters of the Device Driver Module being written for the USB Mouse connected externally.

CHAPTER 7

RESULTS AND CONCLUSION

The number of interrupts generated by the external connected mouse will be counted and be displayed on the screen. The result screen also displays the memory region allocated to the device driver. The messages are being copied from the user space and kernel space to vice versa. Also the device driver module author, description and parameters are also being displayed.