

IMPLEMENTING BLOCK DEVICE DRIVER FOR REAL TIME OPERATING SYSTEM

DISSERTATION-II

*Submitted in partial fulfilment of the
Requirement of the award of*

**Degree of
MASTER OF TECHNOLOGY
IN
ELECTRONICS AND ELECTRICAL ENGINEERING
(EMBEDDED SYSTEM)**

Submitted by
GEETIKA
Registration No: 11501196

Under the Guidance of
MR. KRANTHI KUMAR PULLURI



L LOVELY
P ROFESSIONAL
U NIVERSITY

Transforming Education Transforming India

**School of Electronics and Electrical Engineering
Lovely Professional University
Punjab**

APRIL-2017

TOPIC APPROVAL PERFORMA

School of Electronics and Electrical Engineering

Program : P175::M.Tech. (Electronics and Communication Engineering) [Full Time]

COURSE CODE : ECE521

REGULAR/BACKLOG : Regular

GROUP NUMBER : EEERGD0205

Supervisor Name : Kranthi Kumar Pulluri **UID :** 14907

Designation : Assistant Professor

Qualification : _____

Research Experience : _____

SR.NO.	NAME OF STUDENT	REGISTRATION NO	BATCH	SECTION	CONTACT NUMBER
1	Geetika	11501196	2015	E1514	08556036511

SPECIALIZATION AREA : Embedded Systems

Supervisor Signature: _____

PROPOSED TOPIC : Design of Device Driver using RTOS.

Qualitative Assessment of Proposed Topic by PAC		
Sr.No.	Parameter	Rating (out of 10)
1	Project Novelty: Potential of the project to create new knowledge	7.00
2	Project Feasibility: Project can be timely carried out in-house with low-cost and available resources in the University by the students.	7.33
3	Project Academic Inputs: Project topic is relevant and makes extensive use of academic inputs in UG program and serves as a culminating effort for core study area of the degree program.	7.33
4	Project Supervision: Project supervisor's is technically competent to guide students, resolve any issues, and impart necessary skills.	7.33
5	Social Applicability: Project work intends to solve a practical problem.	8.00
6	Future Scope: Project has potential to become basis of future research work, publication or patent.	7.67

PAC Committee Members		
PAC Member 1 Name: Anshul Mahajan	UID: 11495	Recommended (Y/N): Yes
PAC Member 2 Name: Dushyant Kumar Singh	UID: 13367	Recommended (Y/N): Yes
PAC Member 3 Name: Cherry Bhargava	UID: 12047	Recommended (Y/N): NA
PAC Member 4 Name: Anshul Mahajan	UID: 11495	Recommended (Y/N): Yes
DAA Nominee Name: Manie Kansal	UID: 15692	Recommended (Y/N): NA

Final Topic Approved by PAC: Design of Device Driver using RTOS.

Overall Remarks: Approved

PAC CHAIRPERSON Name: 11211::Prof. Bhupinder Verma

Approval Date: 15 Oct 2016

CANDIDATE'S DECLARATION

I Geetika, student of M.tech (Electronics and Electrical Engineering) under school of Electronics and Electrical Engineering of Lovely Professional University, Punjab, hereby declare that all the information furnished in this dissertation report is an authentic record of my own work carried out under the supervision of “Mr. Kranthi Kumar Pulluri” Assistant Professor, School of Electronics and Electrical Engineering. The matter presented in this dissertation has not been submitted to Lovely Professional University or to any other university or institute for the award of any degree.

Signature of the Student

Reg. No. 11501196

Date:

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Signature of the Supervisor

The M.tech Viva-Voce Examination of (Dissertation-II) has been held on _____ and found satisfactory/Not satisfactory.

Signature of the Internal Examiner

Signature of the External Examiner

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my mentor “**Mr. Kranthi Kumar Pulluri**” Assistant Professor for her guidance, encouragement, and support throughout the course of this work. It was an invaluable learning experience for me to be one of her students. From her, I have gained not only extensive knowledge but also a sincere research attitude. I express my gratitude to Mr. Kranthi Kumar Pulluri (Assistant Professor), of Electronics and Electrical Engineering for his invaluable suggestions and constant encouragement all through the research work. My thanks are extended to my friends in “Embedded System” who built an academic and friendly research environment that made my study at Lovely Professional University, Phagwara most memorable and fruitful.

I would also like to acknowledge the entire teaching and non-teaching staff of Electronics and Electrical Department for establishing a working environment and for constructive discussions. Finally, I am always indebted to all my family members especially my parents, for their endless love and blessings.

NAME: GEETIKA

REG. NO: 11501196

CERTIFICATE

This is to certify that the declaration statement made by this student is correct to the best of my knowledge and belief. She is doing Dissertation-II work under my guidance and supervision. The present work is the result of their original investigation, effort, and study. No part of the work has ever been submitted for any other degree at any University. The Dissertation-II work is fit for the submission and partial fulfilment of the conditions for the award of M.Tech degree in Electronics and Electrical Engineering from Lovely Professional University, Phagwara.

Signature and Name of the Mentor:

Designation:

School of Electronics & Electrical Engineering
Lovely Professional University
Phagwara, Punjab

Date:

ABSTRACT

In this we review and discuss the implementation of block device driver using real time operating system. Real-time systems play a considerable role in our society, and they cover a spectrum from the very simple to the very complex. Examples of current real-time systems include the control of domestic appliances like washing machines and televisions, the control of automobile engines, telecommunication switching systems, military command and control systems, industrial process control, flight control systems, and space shuttle and aircraft avionics. All of these involve gathering data from the environment, processing of gathered data, and providing timely response. A concept of time is the distinguishing issue between real-time and non-real-time systems. When a usual design goal for non-real-time systems is to maximize system's throughput, the goal for real-time system design is to guarantee, that all tasks are processed within a given time. The taxonomy of time introduces special aspects for real-time system research. All the domestic systems that we use in the society are real-time systems and these follow timeliness. Real-time Operating system is to be designed for these real-time systems. These systems get the external inputs and process these into output with time constraints. If the timeliness is not maintained, then the system is said to be failure. As a programmer, you are able to make your own choices about your driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is "flexible," we like this word because it emphasizes that the role of a device driver is providing mechanism, not policy. And there are three types of time constraints i) Hard ii) Soft iii) Firm. Real-time Programme is p and it gets inputs in every t interval of time and event requires c computational time. Dead line for completing the computation is D . Predictability is a main concept in real-time system. Design issues in Real-time system: All the behaviour of the system must be predictable. To achieve this all the components must be time bounded. All the inputs are received by the systems and processed for outputs within a specified time else the system results in failure. Real-time behaviour is mandatory while designing a critical safety system.

LIST OF FIGURES

Figure No	Figure Name	Page No
1.1	Real time operating system	2
1.2	Block diagram of RTOS	5
1.3	Classification of RTOS	6
1.4	Architecture of RTOS	8
1.5	Real time KERNEL	11
1.6	Monolithic Kernel	13
1.7	Microkernel	16
1.8	RTOS Kernel Services	16
1.9	States Transition	19
5.1	Device Driver Interface	29
5.2	Principle interface between a device driver and Linux kernel	29
5.3	Buffer cache Block Device request	32
5.4	Linked List of Disks	35
7.1	Module inserted	44
7.2	List of inserted modules	45
7.3	Sbd module inserted	45
7.4	Format the drive	46
7.5	Name the drive	46
7.6	Mount the Drive	47
7.7	New Drive Created	47
7.8	Unmount the drive	48
7.9	Removable module	49
7.10	Check Module using cat /proc/devices	49

7.11	Module removed	50
7.12	Drive removed	50
7.13	Permanent Drive Created	51
7.14	User and Group that can access the drive is geetika	52
7.15	Make group drivenew	53
7.16	Add users to group	54
7.17	Provide accessibility and permission	55

CONTENTS

CANDIDATE'S DECLARATION	I
ACKNOWLEDGEMENT	II
CERTIFICATE	III
ABSTRACT	IV
LIST OF FIGURES	V-VI
CHAPTER-1 INTRODUCTION.....	(1-21)
1.1 Evolution of real time operating system	1
1.2 Method	2
1.3 Similarities between rtos and gpos.....	3
1.4 Difference between rtos and gpos.....	3
1.5 What is rtos.....	3
1.6 Why RTOS	4
1.7 Classification of RTOS	5
1.8 Misconceptions of RTOS.....	6
1.9 Features of RTOS	7
1.10 RTOS architecture	8
1.11 Kernel.....	8
1.12 Task management	18
1.13 Building modern rtos	19
1.13.1 Strategic decision.....	20
CHAPTER-2 TERMINOLOGY	(22-23)
2.1 Linux.....	22
2.2 Virtual Memory	22
2.3 Linux Device Driver.....	22
2.4 Character Device.....	23

2.5	Block Device.....	23
2.6	Network Device.....	23
CHAPTER-3 SCOPE OF STUDY.....		(24)
CHAPTER-4 LITERATURE REVIEW		(25-26)
CHAPTER-5 RESEARCH METHODOLOGY		(27-35)
5.1	Real time features of linux scheduling.. ..	27
5.2	Virtual Memory	27
5.3	Linux Device Driver.....	27
5.4	Interfacing device driver with kernel.....	30
5.5	Types of device driver.....	31-35
5.5.1	Character Device.....	31
5.5.2	Block Device.....	31
5.5.3	Network Device.....	35
CHAPTER-6 EQUIPMENT,MATERIAL,AND EXPERIMENTAL SETUP.....		(36-58)
6.1	Converting virtual memory as drive	37
	41
CHAPTER-7 RESULT AND PERFORMANCE EVALUATION.....		(43-54)
7.1	Converting virtual memory as drive(temporarily).....	43
7.2	Converting virtual memory as drive(permanently).....	49
7.3	Security features of drive.....	51
CHAPTER-8 CONCLUSION AND FUTURE SCOPE		(54)
LIST OF REFERENCES/BIBLIOGRAPHY		(56)

Chapter – 1

Introduction

1.1 EVOLUTION OF RTOS (Real Time Operating System):

In the beginning of registering, designers made programming applications that included low-level machine code to instate and connect with the framework's equipment straightforwardly. This tight joining between the product and equipment brought about non-convenient applications. A little change in the equipment may bring about modifying a great part of the application itself. Clearly, these frameworks were troublesome and exorbitant to keep up.

As the product business advanced, working frameworks that gave the fundamental programming establishment to figuring frameworks developed and encouraged the reflection of the hidden equipment from the application code. Moreover, the advancement of working frameworks moved the outline of programming applications from substantial, solid applications to more secluded, interconnected applications that could keep running on top of the working framework environment. Throughout the years, numerous variants of working frameworks developed. These ran from universally useful working frameworks (GPOS, for example, UNIX and Microsoft Windows, to littler and more reduced constant working frameworks, for example, VxWorks. Each is quickly talked about next.

In the 70s, when medium sized and centralized server registering was in its prime, UNIX was produced to encourage multi-client access to costly, constrained accessibility processing frameworks. UNIX permitted numerous clients playing out an assortment of assignments to share these vast and exorbitant PCs. multi-client gets to was extremely proficient: one client could print documents, for instance, while another composed projects. In the long run, UNIX was ported to a wide range of machines, from microcomputers to supercomputers.

In the 80s, Microsoft presented the Windows working framework, which underlined the individualized computing environment. Focused for private and business clients communicating with PCs through a graphical UI, the Microsoft Windows working framework drove the individualized computing era. Later in the decade, energy began working for the up and coming

era of figuring: the post-PC, inserted processing time. To address the issues of implanted registering, business RTOS, for example, VxWorks, were produced. Albeit some useful likenesses exist amongst RTOS and GPOS, numerous essential contrasts happen also.



Figure1.1: Real Time Operating System

1.2 METHOD:

The definition, some or the greater part of the accompanying techniques are employed. The RTOS performs few assignments, in this way guaranteeing the undertakings will dependably be executed before the due date. The RTOS drops or decreases certain capacities when they can't be executed inside the time requirements ("stack shedding"). The RTOS screens input reliably and in an opportune way. The RTOS screens assets and can intrude on foundation forms as expected to guarantee continuous execution. The RTOS envisions potential demands and liberates enough of the framework to permit opportune response to the client and demand. The RTOS monitors the amount of every asset (CPU time per time cut, RAM, interchanges data transmission, and so forth.) may perhaps be utilized as a part of the most pessimistic scenario by the right now running undertakings, and declines to acknowledge another errand unless it "fits" in the rest of the un-designated assets.

1.3 SIMILARITIES BETWEEN RTOS & GPOS:

Some center utilitarian similitudes between a run of the mill RTOS and GPOS include:

- Some level of multitasking,
- Software and equipment asset administration,
- Arrangement of basic OS administrations to applications, and
- Abstracting the equipment from the product application.

1.4 DIFFERENCES BETWEEN RTOS & GPOS:

Then again, some key practical contrasts that set RTOS apart from GPOS include:

- Better dependability in implanted application settings,
- The capacity to scale up or down to address application issues,
- Speedier execution,
- Decreased memory necessities,
- Planning arrangements custom-made for ongoing inserted frameworks,
- Bolster for diskless inserted frameworks by permitting executables to boot and keep running from ROM or RAM, and
- Better transportability to various equipment stages

1.5 WHAT IS RTOS?

RTOS includes two segments, to be specific, "Ongoing" and "Working System". A continuous working framework (RTOS) is a working framework that ensures a specific ability inside a predetermined time requirement. "Delicate" continuous working framework, the sequential

construction system would keep on functioning however the creation yield may be lower as articles neglected to show up at their assigned time, bringing about the robot to be briefly inefficient. Some constant working frameworks are made for an exceptional application and others are more broadly useful. To some degree, any universally useful working framework, for example, windows can be assessed constant working framework qualities. That is, regardless of the possibility that a working framework doesn't qualify, it might have attributes that empower it to be considered as an answer for a specific constant application issue.

1.6 WHY RTOS?

The RTOS innovation is decades old yet going solid, but still reacting to new patterns the advancements so as it could meet today's inserted framework requirements. Years ago chip brought forth another age in inserted frameworks advancement. Before long, engineers started creating custom errand schedulers to oversee equipment assets to keep the applications on the programmable chips running. And that time onwards the real time operating system just kept growing.

RTOS, are sometime unmistakable, but sometime not- nucleus, deos, ThreadX, VxWorks, VRTX, pSOS, OS-9, QNX, RMX, SMX ,LynxOS and some other – created during 1980 started supplanting undertaking scheduling and created them less so create implanted items.

Designers were OK with low level computing construct, and C dialect was to a great extent marked down because the requirement profoundly advance product based on such frameworks. Quickly coming ahead – effective multi-core processors , Gigabytes of memory, heaps of I/O, representation, and systems administration choices have changed inserted frameworks into an altogether different creature. Virtualization is being utilized in implanted frameworks as a way to merge complex designs and give partition between ongoing and non-constant frameworks.

For a long time, the RTOS has to a great extent flown under the installed radar, yet at the same time serves a basic part in numerous frameworks that require hard processing due dates be met. Various RTOS organizations have added to the advancement of installed frameworks in the course of recent years with troublesome innovations, molding the RTOS as we probably are aware it today. RTOS is not a required part of all ongoing application in inserted frameworks.

An installed framework in a straightforward electronic rice cooker does not require RTOS. Be that as it may, as the intricacy of uses grows past straightforward errands, advantages of having a RTOS far exceed the partner costs. Installed frameworks are turning out to be more intricate equipment shrewd with each era. Also, as more components are put into them in every cycle, application programs running on the installed framework stages will turn out to be progressively mind boggling to be overseen as they endeavor to meet the framework reaction prerequisites. A RTOS will be compelling to permit the continuous applications to be outlined and extended all the more effectively while meeting the exhibitions required.

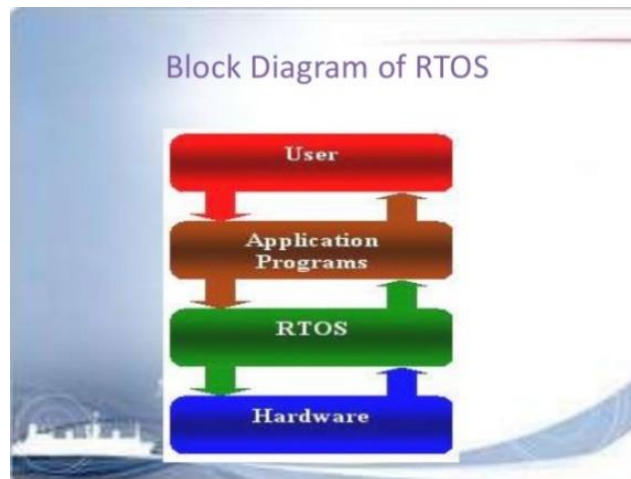


Figure 1.2: Block diagram of RTOS

1.7 Classification of RTOS:

RTOS's are comprehensively ordered into three sorts, in particular, Hard Real Time RTOS, Firm Real Time RTOS and Soft Real Time RTOS as depicted beneath:

- Hard continuous: level of resistance for missed due dates is greatly little or zero. A missed due date has calamitous outcomes for the framework.
- Firm constant: missing a due date may bring about an unsatisfactory quality lessening.
- Delicate continuous: due dates might be missed and can be recouped from. Decrease in framework quality is worthy.

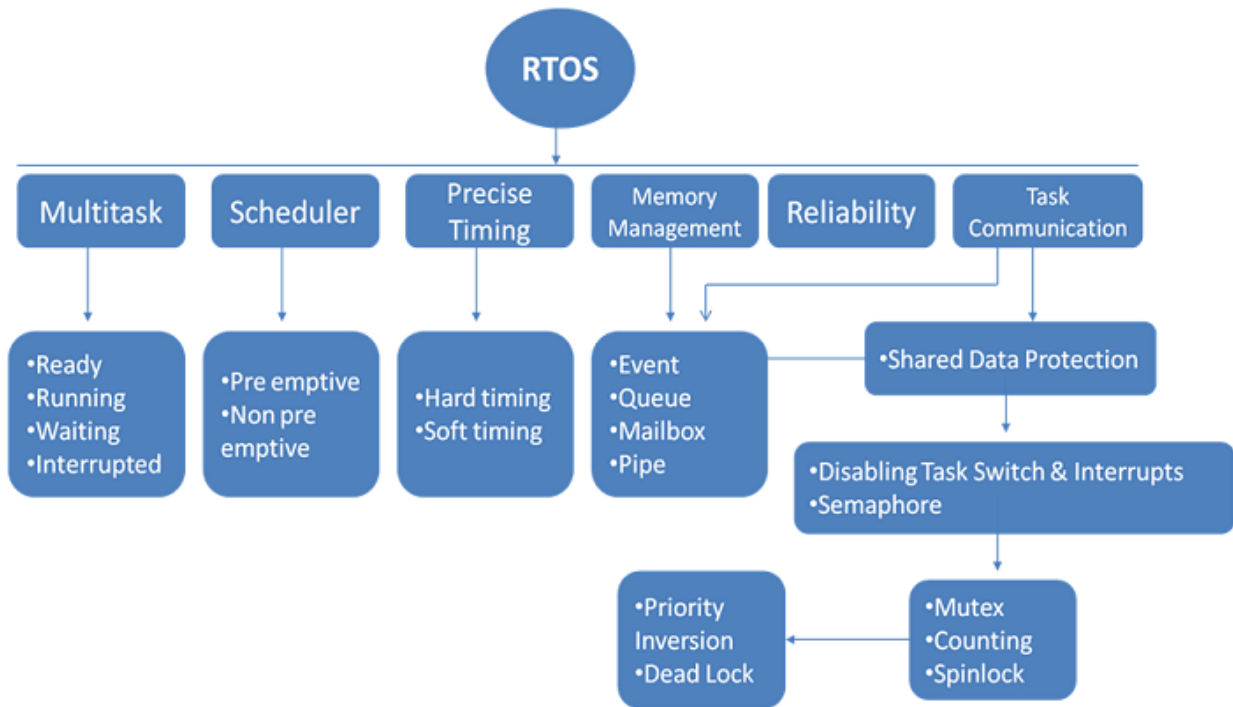


Figure 1.3: Classifications of Rtos

1.8 MISCONCEPTION OF RTOS:

- RTOS must be fast. The responsiveness of a RTOS relies on upon its deterministic conduct and not on its preparing speed. The capacity of RTOS to reaction to occasions inside a course of events does not suggest it is quick.
- RTOS present extensive measure of overhead on CPU. A RTOS commonly just require between 1% to 4% of a CPU time.
- All RTOS are the same. RTOS are by and large intended for 3 sorts of continuous frameworks (i.e. hard, firm and delicate). Likewise, they are further arranged by sorts of equipment gadgets (e.g. 8-bit, 16-bit, 32-bit MPU) upheld.

1.9 RTOS FEATURES:

The plan of a RTOS is basically a harmony between giving a sensibly rich list of capabilities for application improvement and sending and, not giving up consistency and convenience. A fundamental RTOS will be furnished with the accompanying components:

i. Multitasking and Pre-emptibility:

A RTOS must be multi-entrusted and pre-emptible to bolster numerous assignments continuously applications. The scheduler ought to have the capacity to acquire any assignment in the framework and distribute the asset to the errand that necessities it most even at pinnacle stack.

ii. Assignment Priority:

Seizure characterizes the ability to recognize the undertaking that needs an asset the most and dispenses it the control to get the asset. In RTOS, such ability is accomplished by appointing singular assignment with the proper need level. Along these lines, it is critical for RTOS to be furnished with this component.

iii. Solid and Sufficient Inter Task Communication Mechanism:

For various errands to convey in an auspicious way and to guarantee information respectability among each other, solid and adequate between assignment correspondence and synchronization systems are required.

iv. Need Inheritance:

To permit applications with stringent need prerequisites to be actualized, RTOS must have an adequate number of need levels when utilizing need planning.

v. Control of Memory Management:

To guarantee unsurprising reaction to an interfere with, a RTOS ought to give approach to undertaking to bolt its code and information into genuine memory.

1.10 ARCHITECTURE OF RTOS:

The design of a RTOS is reliant on the many-sided quality of its sending. Great RTOSs are adaptable to meet distinctive arrangements of prerequisites for various applications. For basic applications, a RTOS as a rule includes just a portion. For more intricate implanted frameworks, a RTOS can be a mix of different modules, including the piece, organizing convention stacks, and different parts as represented in Figure 2 below.

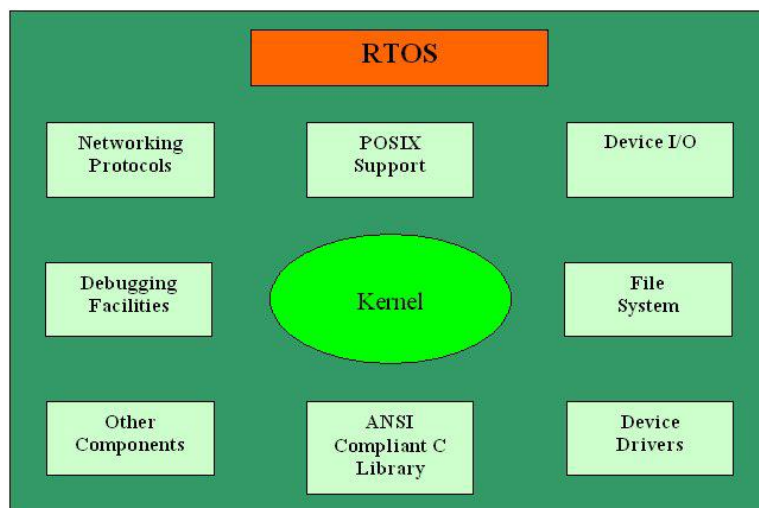


Figure 1.4: Architecture of RTOS

1.11 KERNEL:

A working framework by and large comprises of two sections: part space (portion mode) and client space (client mode). Part is the littlest and focal segment of a working framework. Its administrations incorporate overseeing memory and gadgets furthermore to give an interface to programming applications to utilize the assets. A bit is a focal part of a working framework. It goes about as an interface between the client applications and the equipment. The sole point of the part is to deal with the correspondence between the product (client level applications) and the

equipment (CPU, circle memory and so on).Extra administrations, for example, overseeing security of projects and multitasking might be incorporated relying upon design of working framework. The bit is the center of a working framework. It is the product in charge of running projects and giving secure access to the machine's equipment. Since there are many projects, and assets are restricted, the bit likewise chooses when and to what extent a program ought to run. This is called booking. Getting to the equipment straightforwardly can be extremely intricate, since there are a wide range of equipment outlines for a similar sort of segment. Pieces more often than not actualize some level of equipment reflection (an arrangement of guidelines widespread to all gadgets of a specific sort) to conceal the basic multifaceted nature from applications and give a spotless and uniform interface. This assists application software engineers with developing projects without knowing how to program for particular gadgets. The bit depends upon programming drivers that make an interpretation of the nonexclusive charge into guidelines particular to that gadget.

A working framework part is not entirely expected to run a PC. Projects can be specifically stacked and executed on the "exposed metal" machine, gave that the creators of those projects will manage with no equipment deliberation or working framework bolster. This was the typical working technique for some early PCs, which were reset and reloaded between the running of various projects. In the long run, little subordinate projects, for example, program loaders and debuggers were normally left in-center between runs, or stacked from read-just memory. As these were produced, they framed the premise of what turned out to be early working framework pieces. The "exposed metal" approach is as yet utilized today on numerous computer game consoles and installed frameworks, yet by and large, more current frameworks utilize parts and working frameworks. The main processes of kernel are:

- Process management

- Memory management

- Device management

- Interrupt handling

- I/O communication

-File system...etc

Is Linux a kernel or an operating system? All things considered, there is a distinction amongst bit and OS. Portion as depicted above is the heart of OS which deals with the center components of an OS while if some helpful applications and utilities are included over the part, then the total bundle turns into an OS. Along these lines, it can without much of a stretch be said that a working framework comprises of a bit space and a client space.

In this way, we can state that Linux is a piece as it does exclude applications like record framework utilities, windowing frameworks and graphical desktops, framework manager summons, content tools, compilers and so on. Along these lines, different organizations include these sort of utilizations over linux bit and give their working framework like Ubuntu, suse,redHat etc.

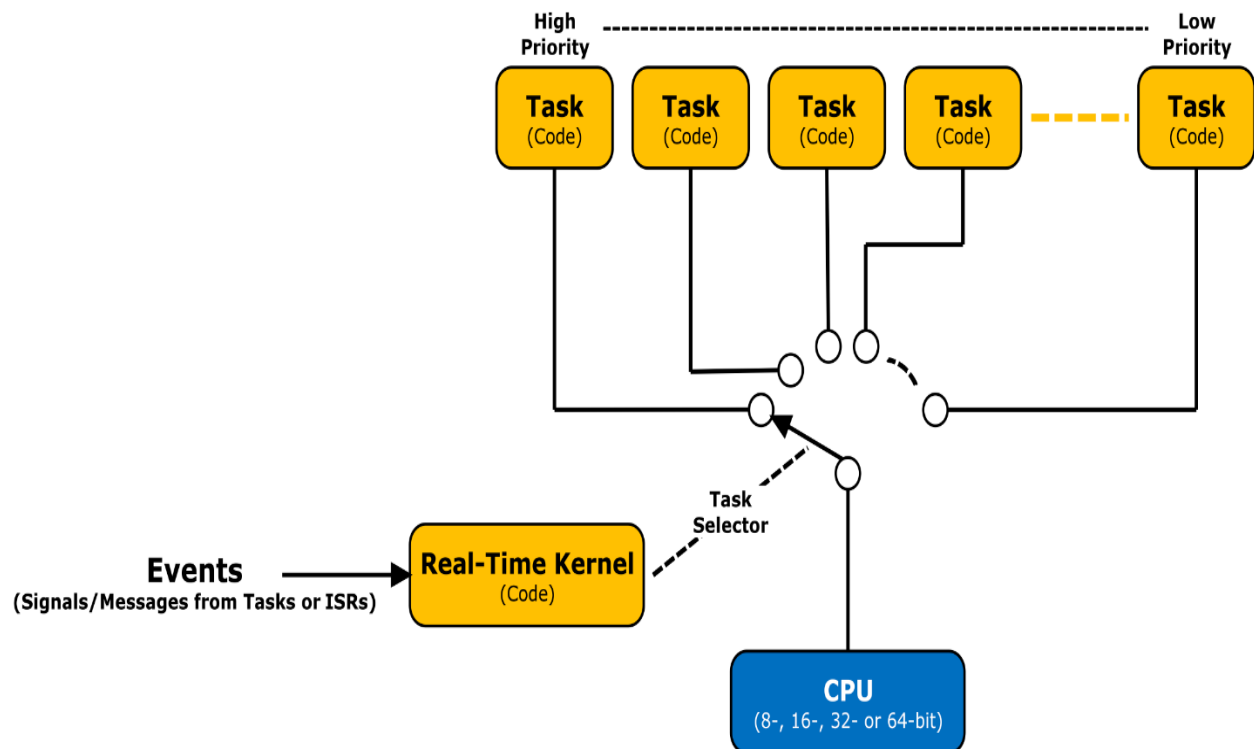


Figure 1.5: Real Time Kernel

There are three general classifications of piece models accessible, in particular:

MONOLITHIC KERNEL: It runs all essential framework administrations (i.e. process and memory administration, intrude on taking care of and I/O correspondence, document framework, and so forth) in bit space. Accordingly, solid bits give rich and intense reflections of the hidden equipment. Measure of setting switches and informing included are incredibly diminished which makes it run speedier than microkernel. Cases are Linux and Windows. In a solid piece, all OS administrations keep running alongside the primary portion string, in this manner additionally dwelling in a similar memory range. This approach gives rich and intense equipment get to. A few designers, for example, UNIX engineer Ken Thompson, keep up that it is "less demanding to execute a solid kernel" than microkernels. The principle burdens of solid bits are the conditions between framework segments – a bug in a gadget driver may crash the whole framework – and the way that substantial bits can turn out to be exceptionally hard to keep up.

Solid bits, which have generally been utilized by Unix-like working frameworks, contain all the working framework center capacities and the gadget drivers (little projects that enable the working framework to connect with equipment gadgets, for example, plate drives, video cards and printers). This is the conventional outline of UNIX frameworks. A solid bit is one single program that contains the majority of the code important to play out each bit related undertaking. Each part which is to be gotten to by most projects which can't be placed in a library is in the bit space: Device drivers, Scheduler, Memory taking care of, File frameworks, Network stacks. Numerous framework calls are given to applications, to enable them to get to each one of those administrations. A solid piece, while at first stacked with subsystems that may not be required, can be tuned to a point where it is as quick as or speedier than the one that was particularly intended for the equipment, albeit more pertinent in a general sense. Current solid portions, for example, those of Linux and FreeBSD, both of which fall into the classification of Unix-like working frameworks, include the capacity to load modules at runtime, along these lines permitting simple expansion of the piece's abilities as required, while limiting the measure of code running in bit space. In the solid piece, a few favorable circumstances rely on these focuses: Since there is less programming included it is quicker.

As it is one single bit of programming it ought to be littler both in source and ordered structures.

Less code by and large means less bugs which can mean less security issues.

Most work in the solid part is done by means of framework calls. These are interfaces, typically kept in a forbidden structure, that get to some subsystem inside the portion, for example, circle operations. Basically calls are made inside projects and a checked duplicate of the demand is gone through the framework call. Thus, not far to go by any stretch of the imagination. The solid Linux bit can be made greatly little not just in light of its capacity to powerfully stack modules additionally in view of its simplicity of customization. Truth be told, there are a few forms that are sufficiently little to fit together with countless and different projects on a solitary floppy circle and still give a completely practical working framework (a standout amongst the most prevalent of which is muLinux). This capacity to scale down its portion has likewise prompted a fast development in the utilization of Linux in implanted frameworks.

These sorts of pieces comprise of the center elements of the working framework and the gadget drivers with the capacity to load modules at runtime. They give rich and capable reflections of the hidden equipment. They give a little arrangement of basic equipment reflections and utilize applications called servers to give greater usefulness. This specific approach characterizes an abnormal state virtual interface over the equipment, with an arrangement of framework calls to execute working framework administrations, for example, prepare administration, simultaneousness and memory administration in a few modules that keep running in manager mode. This plan has a few blemishes and confinements:

Coding in portion can challenge, to some extent since one can't utilize normal libraries (like a full-included libc), and in light of the fact that one needs to utilize a source-level debugger like gdb. Rebooting the PC is regularly required. This is not only an issue of comfort to the designers. When investigating is harder, and as troubles end up noticeably more grounded, it turns out to be more probable that code will be "buggier". Bugs in one a player in the portion have solid reactions; since each capacity in the bit has every one of the benefits, a bug in one capacity can degenerate information structure of another, absolutely irrelevant piece of the part, or of any running project. Pieces regularly turn out to be extensive and hard to keep up. Regardless of the possibility that the modules adjusting these operations are separate from the entire, the code combination is tight and hard to do effectively. Since the modules keep running in a similar address space, a bug can cut down the whole framework. Solid bits are not convenient; in this

manner, they should be changed for each new design that the working framework is to be utilized on.

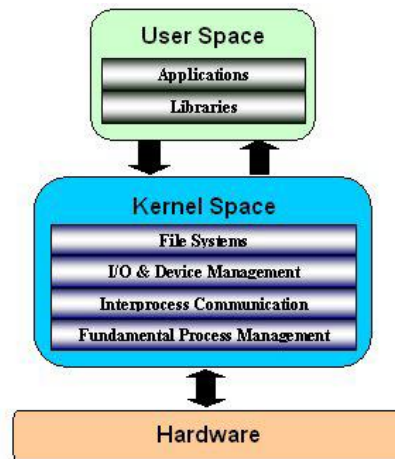


Figure 1.6: Monolithic Kernel

MICROKERNEL: It runs just fundamental process correspondence (informing) and I/O control. The other framework administrations (document framework, organizing, and so on) live in client space as daemons/servers. Along these lines, miniaturized scale bits give a littler arrangement of basic equipment deliberations. It is more steady than solid as the portion is unaffected regardless of the possibility that the servers fizzled (i.e. File System). Cases are AmigaOS and QNX. Microkernel is the term portraying a way to deal with working framework outline by which the usefulness of the framework is moved out of the conventional "bit", into an arrangement of "servers" that impart through an "insignificant" piece, leaving as meager as conceivable in "framework space" and however much as could reasonably be expected in "client space". A microkernel that is intended for a particular stage or gadget is just perpetually going to have what it needs to work. The microkernel approach comprises of characterizing a basic reflection over the equipment, with an arrangement of primitives or framework calls to actualize insignificant OS administrations, for example, memory administration, multitasking, and between process correspondence. Different administrations, including those typically gave by the piece, for example, systems administration, are executed in client space programs, alluded to as servers. Microkernels are simpler to keep up than solid parts, yet the extensive number of framework calls and setting switches may back off the framework since they commonly create more overhead than plain capacity calls.

Just parts which truly require being in a special mode are in piece space: IPC (Inter-Process Communication), fundamental scheduler, or planning primitives, essential memory taking care of, essential I/O primitives. Numerous basic parts are currently running in client space: The entire scheduler, memory dealing with, record frameworks, and system stacks. Miniaturized scale pieces were developed as a response to conventional "solid" bit plan, whereby all framework usefulness was placed in a one static program running in an uncommon "framework" method of the processor. In the microkernel, just the most crucial of errands are performed, for example, having the capacity to get to a few (not really all) of the equipment, oversee memory and arrange message going between the procedures. A few frameworks that utilization miniaturized scale bits are HURD and QNX. On account of QNX and Hurd client sessions can be whole depictions of the framework itself or perspectives as it is alluded to. The very pith of the microkernel engineering shows some of its points of interest:

Upkeep is for the most part simpler.

Patches can be tried in a different occurrence, and after that swapped into assume control over a generation case.

Quick advancement time and new programming can be tried without rebooting the bit.

More determination when all is said in done, in the event that one case goes feed wire, it is frequently conceivable to substitute it with an operational mirror.

Most miniaturized scale parts utilize a message passing arrangement or something to that affect to deal with solicitations starting with one server then onto the next. The message passing framework for the most part works on a port premise with the microkernel. For instance, if a demand for more memory is sent, a port is opened with the microkernel and the demand sent through. Once inside the microkernel, the means are like framework calls. The method of reasoning was that it would get seclusion the framework design, which would involve a cleaner framework, simpler to troubleshoot or progressively change, adjustable to clients' needs, and additionally performing. They are a piece of the working frameworks like AIX, BeOS, Hurd, Mach, macOS, MINIX, QNX. And so on. Albeit smaller scale parts are little independent from anyone else, in blend with all their required assistant code they are, actually, regularly bigger than solid pieces. Supporters of solid portions additionally bring up that the two-layered structure

of microkernel frameworks, in which a large portion of the working framework does not connect specifically with the equipment, makes a not-immaterial cost as far as framework proficiency. These sorts of bits typically give just the insignificant administrations, for example, characterizing memory address spaces, Inter-handle correspondence (IPC) and the procedure administration. Alternate capacities, for example, running the equipment procedures are not taken care of straightforwardly by small scale portions. Defenders of miniaturized scale parts bring up those solid portions have the weakness that a blunder in the piece can make the whole framework crash. Be that as it may, with a microkernel, if a bit procedure crashes, it is as yet conceivable to keep a crash of the framework all in all by just restarting the administration that brought about the blunder. Different administrations gave by the piece, for example, systems administration are executed in client space programs alluded to as servers. Servers enable the working framework to be changed by essentially beginning and halting projects. For a machine without systems administration bolster, for example, the systems administration server is not begun. The errand of moving all through the bit to move information between the different applications and servers makes overhead which is inconvenient to the productivity of smaller scale pieces in correlation with solid portions. Disservices in the microkernel exist in any case. Some are: Bigger running memory impression More programming for interfacing is required, there is a potential for execution misfortune. Informing bugs can be harder to settle because of the more extended excursion they need to take versus the erratic duplicate in a solid piece. Prepare administration when all is said in done can be extremely confused. The drawbacks for smaller scale parts are greatly setting based. For instance, they function admirably for little single reason (and basic) frameworks on the grounds that if very few procedures need to run, then the intricacies of process administration are adequately relieved. A microkernel permits the usage of the rest of the piece of the working framework as a typical application program written in an abnormal state dialect, and the utilization of various working frameworks on top of the same unaltered kernel.[21] It is likewise conceivable to powerfully switch among working frameworks and to have more than one dynamic at the same time.

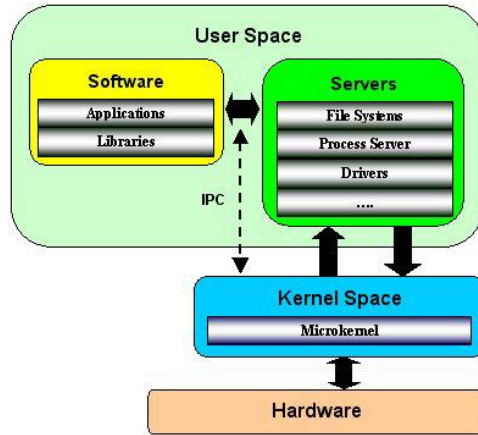


Figure 1.7: Microkernel

A RTOS for the most part abstains from actualizing the portion as a substantial solid program. The piece is created rather as a miniaturized scale part with included configurable functionalities. This usage gives coming about advantage in increment framework configurability, as each inserted application requires a particular arrangement of framework administrations as for its attributes.

The part of a RTOS gives a reflection layer between the application programming and equipment. This reflection layer includes six primary sorts of regular administrations gave by the part to the application programming. Figure 5 demonstrates the six basic administrations of a RTOS bit.

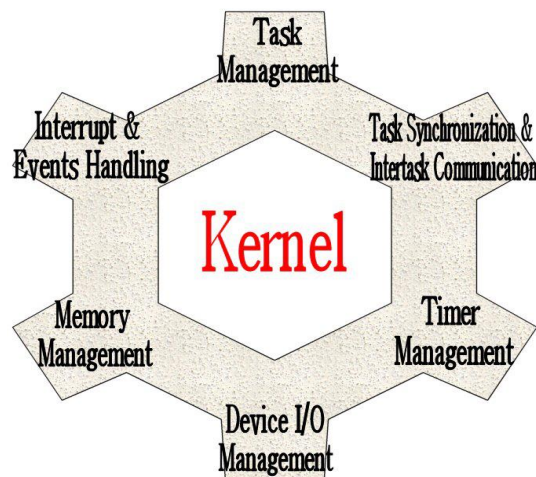


Figure 1.8: RTOS Kernel Services

HYBRID KERNEL: They are like miniaturized scale parts, with the exception of they incorporate some extra code in piece space to expand execution. These portions speak to a trade off that was actualized by a few designers before it was exhibited that unadulterated miniaturized scale pieces can give elite. These sorts of pieces are expansions of smaller scale parts with a few properties of solid bits. Dissimilar to solid bits, these sorts of parts can't stack modules at runtime all alone. Cross breed pieces are miniaturized scale parts that have some "superfluous" code in portion space all together for the code to run more rapidly than it would were it to be in client space. Mixture pieces are a bargain between the solid and microkernel plans. This suggests running a few administrations, (for example, the system stack or the filesystem) in piece space to diminish the execution overhead of a conventional microkernel, yet at the same time running bit code, (for example, gadget drivers) as servers in client space.

Numerous customarily solid parts are currently at any rate including (if not effectively abusing) the module capacity. The most understood of these bits is the Linux portion. The particular bit basically can have parts of it that are incorporated with the center portion parallel or pairs that heap into memory on request. Note that a code corrupted module can possibly destabilize a running part. Many individuals wind up plainly confounded on this moment that examining small scale bits. It is conceivable to compose a driver for a microkernel in a totally isolate memory space and test it before "going" live. At the point when a piece module is stacked, it gets to the solid segment's memory space by adding to it what it needs, in this manner, opening the entryway to conceivable contamination. A couple preferences to the measured (or) Hybrid part are:

Quicker advancement time for drivers that can work from inside modules. No reboot required for testing (gave the piece is not destabilized).

On request capacity as opposed to investing energy recompiling an entire piece for things like new drivers or subsystems.

Quicker mix of outsider innovation (identified with advancement however applicable unto itself in any case).

Modules, by and large, speak with the piece utilizing a module interface or something to that affect. The interface is summed up (albeit specific to a given working framework) so it is not

generally conceivable to utilize modules. Frequently the gadget drivers may require more adaptability than the module interface manages. Basically, it is two framework calls and regularly the wellbeing watches that lone must be done once in the solid piece now might be done twice. A portion of the hindrances of the secluded approach are:

With more interfaces to go through, the likelihood of expanded bugs exists (which infers greater security gaps).

Keeping up modules can mistake for a few managers when managing issues like image contrasts.

EXOKERNELS: Exokernels are a still-trial way to deal with working framework outline. They vary from alternate sorts of portions in that their usefulness is restricted to the assurance and multiplexing of the crude equipment, giving no equipment deliberations on top of which to create applications. This detachment of equipment security from equipment administration empowers application designers to decide how to make the most effective utilization of the accessible equipment for every particular program.

Exokernels in themselves are to a great degree little. Be that as it may, they are joined by library working frameworks (see likewise unikernel), giving application engineers the functionalities of an ordinary working framework. A noteworthy preferred standpoint of exokernel-based frameworks is that they can consolidate numerous library working frameworks, each sending out an alternate API, for instance one for abnormal state UI advancement and one for continuous control.

1.12 TASK MANAGEMENT:

Task administration permits developers to plan their product as various separate "pieces" of codes with every taking care of a particular objective and due date. This administration envelops instrument, for example, scheduler and dispatcher that makes and keep up task objects.

To accomplish simultaneousness continuously application program, the application is break down into little, schedulable, and successive program units known as "Task". Continuously setting, task is the essential unit of execution and is represented by three time-basic properties; discharge time, due date and execution time. Discharge time alludes to the point in time from

which the task can be executed. Due date is the point in time by which the task must finish. Execution time signifies the time the task takes to execute.

A task protest is characterized by the accompanying arrangement of parts:

- Task Control piece (Task information structures dwelling in RAM and just open by RTOS)
- Task Stack (Data characterized in program dwelling in RAM and open by stack pointer)
- Task Routine (Program code dwelling in ROM)

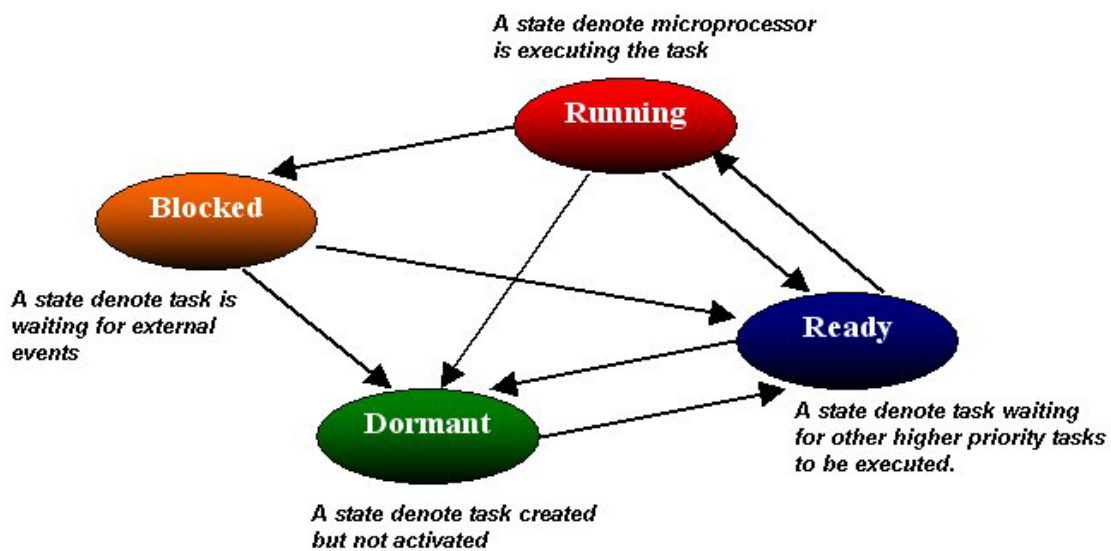


Figure 1.9: States Transition

1.13 BUILDING A MODERN REAL TIME OPERATING SYSTEM:

As inserted frameworks advance and new measures, processors, stages, I/O, and applications rise, the RTOS keeps on developing and adjust to consolidate these new capacities and assume an indispensable part in the improvement of the up and coming era of installed frameworks. The

RTOS is no more extended just about overseeing memory, I/O, and assignment planning with a couple devices tossed in. It's about total and powerful incorporated improvement and investigate situations, rich representation, and system capacities, security highlights, and the capacity to speak with other virtualized programming situations while keeping up constant determinism, high unwavering quality, and effectiveness.

1.13.1 A STRATEGIC DECISION:

The basic design of a RTOS is an imperative model, yet so are different variables. These include:

1. **Flexible decision of planning calculations** - RTOS bolster a decision of booking calculations (FIFO, round robin, sporadic, and so forth.). We can allot those calculations on a for every string premise, or the RTOS drive into relegating one calculation to all strings in your framework.
2. **Guaranteed CPU accessibility** - The RTOS bolster a parceling scheduler that gives errands an ensured rate of CPU time, paying little mind to what different assignments, including higher-need undertakings, are doing.
3. **Graphical UIs** - The RTOS utilize primitive illustrations libraries or does it give propelled design capacities, for example, multi-layer interfaces, multi-headed showcases, and a genuine windowing framework. We effortlessly alter the GUI's look-and-feel. The GUI show and info different dialects (Chinese, Korean, Japanese, English, Russian, and so on.) all the while.
4. **Tools for remote diagnostics** - Because downtime is heinous for some installed frameworks, the RTOS seller ought to give diagnostics instruments that can break down a framework's conduct without intruding on administrations that the framework gives.
5. **Open advancement stage** - The RTOS seller give an improvement situation in view of an open stage like Eclipse, which gives us "a chance to connect to" our most loved outsider apparatuses for displaying, form control, et cetera.
6. **Internet abilities** - The RTOS bolster an a la mode suite of pre integrated convention stacks, for example, IPv4, IPv6, IPsec, SCTP, and IP separating with NAT. It additionally bolsters an installed Web program. The program ought to have an adaptable impression and be fit for

rendering standard Web pages on little screens. It ought to likewise bolster norms, for example, HTML 4.01, SSL 3.0, CSS 1 and 2, JavaScript, WAP, and WML.

CHAPTER 2

Terminology

2.1 LINUX :

Linux is a full-highlighted UNIX usage. The primary plan foundation of the Linux portion is the throughput, while constant and consistency is not an issue. The principle disable to considering Linux as a constant framework is that the piece is not preemptable; that is, while the processor executes portion code, no different procedure or occasion can seize part execution. Although Linux is not a continuous framework, it has a few components, effectively incorporated into the standard source code or appropriated as fix documents, intended to give ongoing to Linux. These are the elements portrayed in this segment.

2.2 VIRTUAL MEMORY:

Continuous application errands must be kept from being swapped out of memory on the grounds that the irregular and long postponements presented when RAM is depleted and swapping is required are unfortunate in an ongoing framework. To bolster this, Linux gives the `mlock()` and `mlockall()` capacities that incapacitate paging for the predefined scope of memory, or for the entire procedure separately. Along these lines, all the "bolted" memory will remain in RAM until the procedure exits or opens the memory. `mlock()` and `mlockall()` are incorporated into the POSIX constant augmentations.

2.3 LINUX DEVICE DRIVER:

The real part of the Linux is being made by device driver. For different sections of the working framework, the work is being done in an environment which is special which bring about fiasco on the off chance that they misunderstand things. The communication among working framework , the controlled device equipment being controlled by device driver. The instance for which, the utilization is being made of general device interface. The interest points is being dealt by driver and particular things happen. The controller chip that is being driven have particular device drivers. The control and status registers (CSRs) are particularly for equipment controller based and different for different devices.

2.4 CHARACTER DEVICES:

Character gadgets, minimal complex of Linux's gadgets, are gotten to as reports, applications use standard system calls to open them, read from them, stay in contact with them and close them decisively as if the gadget were a record. This is bona fide paying little respect to the likelihood that the gadget is a modem being used by the PPP daemon to interface a Linux structure onto a framework.

2.5 BLOCK DEVICES:

Square gadgets also support being gotten the chance to like archives. The frameworks used to give the correct course of action of archive operations for the opened square outstanding record are particularly the same as for character gadgets. Linux keeps up the game plan of enlisted square gadgets as the blkdevs vector. It, like the chrdevs vector, is requested using the gadget's genuine gadget number. Its passageways are moreover device_struct data structures. Not in any manner like character gadgets, there are classes of square gadgets. SCSI gadgets are one such class and IDE gadgets are another. The class registers itself with the Linux part and gives report operations to the bit. The gadget drivers for a class of square gadget give class specific interfaces to the class.

Circle drives give a more never-ending system to securing data, keeping it on turning plate platters. To form data, an unobtrusive head charges minute particles on the platter's surface. The data is examined by a head, which can perceive whether a particular minute particle is spellbound.

2.6 NETWORK DEVICES:

A system gadget is, so far as Linux's system subsystem is concerned, a substance that sends and gets packs of data. This is customarily a physical gadget, for instance, an ethernet card. Some system gadgets however are modifying just, for instance, the loopback gadget which is used for sending data to yourself.

CHAPTER 3

Scope of study

When we hear "Working System" the primary ones that strike a chord are those we encounter/use in our everyday life, say, Windows XP, Linux, Ubuntu, Windows 7 for Computer frameworks, Android for mobiles and numerous more . We predominantly realize that working frameworks are for PCs. Most of the advanced electronic gadgets run some kind of working frameworks inside. There are many working frameworks produced for small scale controllers as well. Be that as it may, here it is commonplace as REAL TIME OPERATING SYSTEM. The expression 'Ongoing' demonstrates that the reaction of the working frameworks is snappy. Microcontrollers don't have much space for code. Along these lines the working frameworks have less degree to be progressed. They attempt to give in any event the base extent of threading, planning and checking of numerous assignments for little frameworks.

Typically, Real Time Operating Systems are a section or a part of the entire program that chooses the following errand, undertaking need, handles the assignment messages and arranges the majority of the errands. A RTOS is a perplexing idea. A Real time working framework handles a few assignments or schedules to be run. The bit of the working framework doles out CPU thoughtfulness regarding a specific undertaking for a timeframe. It additionally checks the errand need, organizes the back rubs from undertakings and calendars.

The essential functionalities a RTOS are:

- RTOS Services
- Synchronization and informing
- Scheduler

Chapter – 4

Literature Review

Brockmeyer et al. [1] have associated with memory estimation and define the problem for memory mapping and dynamic management in memory, mapping techniques in all types of memories.

Apurva et al [2] have focused the analysis that the criteria Success Ratio & Effective CPU Utilization Time are of prime focus in over Loaded condition. the Criteria Success Ratio is to control the deadline meet in real time System & The Criteria Effective CPU Utilization Time to control the Performance efficiency of the Processors.

Ch. Ykman-Couvreur [3] et al have described the content of tasks assignment, tasks management, resource management in design –time and Run-time. resource utilization at design time is important because of satisfying all user and process constraints are real challenge so paper focus on design-time application exploration which includes the utilization of resources for developing embedded software development. in case of an embedded platform, while for a general purpose platform, minimized the cost of component. Programming effort can be implemented and defined as the design effort necessary to get a wide variety of applications running within specifications. Custom design for a single product or application is not much feasible and available for longer time.

Grant Martin [4] in his paper, he has reviewed the design challenges and issues faced by developers in market in all stages like hardware and software applications. Application development is a need for programming models and intercommunication Application programming interfaces(APIs) use to allow software applications to be without difficulty of communication and configured for many different available architectures without repetitive rewriting, while at the same time ensuring well-organized production code. Paper focus on Synchronization and control of tasks scheduling may be used by RTOS (Real time Operating System) or other scheduling methods, and the selection of programming and threading models, there is some lacking either in symmetric model or asymmetric model, has a high risk on how best optimal control tasks or thread running.

Oliver Arnold et al [5] have analyzed to design and implement a new heterogeneous multiprocessor system which associated with two features like (1) dynamic memory and (2) power management for high performance and power consumption is described.

O. Moreira et al [6] have described and analyzed on the basis of two models being differentiated as (1) caches managed by hardware and (2) the memory stream managed by software. Analysis of the two models and return the comparative analysis and design of these classifications fall in the same set of requirements under recent technicalities, work place, speed, power consumption efficiency is being done in this paper.

H. Nikolov et al [7] have focused on real time operating system architecture with microkernel implementations of chips used for embedded system. Paper imitate the implementation of microkernel in programming language of C and C++.

M. Horowitz et al [8] Literature paper described the real-challenges in dynamic mapping when system become over Loaded and satisfies with tasks management, resource management and process management.

D. A. Patterson et al [9] literature paper he briefly introduced on the architecture of Operating system which includes the single processor scheduling polices, multiprocessor scheduling policies, deadlock detection, deadlock avoidance, challenges in real time operating system. Book focus on the basic architecture of operating system like process management, memory management, file and disk management.

M. H. Wiggers [10] have analyzed the hard real-time system, measure the significant disparity between EDF- based Scheduling policy. Paper focus on wide range of soft real - time applications to be scheduled.

M.kaldevi et al [11] have analyzed, described and interact with real time Operating System (RTOS) based applications that get deadlines to facilitated and provided logically correct results. Paper focused on multitasking operating system features and application which work on time deadlines and functioning in real time constraints. To achieve the real-time constraints in RTS (Real time system) for scheduling the tasks, different scheduling algorithms were used. Paper analyzed designed of RTS using priority based preemptive scheduling and the execution of high priority tasks.

CHAPTER 5

RESEARCH METHODOLOGY

5.1 REAL TIME FEATURES OF LINUX SCHEDULING:

Indeed, even in early Linux portion discharges, the scheduler was ongoing POSIX good. It bolsters two settled need booking approaches, each with needs from [0 to 99]:

> SCHED_FIFO—when confronted with two equivalent need errands, SCHED_FIFO permits the main assignment to finish before booking the other undertaking.

> SCHED_RR—when confronted with two equivalent need errands, SCHED_RR time cuts. A ton of work has been done to enhance the execution of the scheduler through a watchful plan that respects another scheduler code and structure. Linux portion v2.4.19 and past, and in addition the temperamental part advancement tree (2.5.x) incorporates another scheduler which replaces the old scheduler code with an enhanced "O (1) scheduler" created by Ingo Molnar. This new scheduler can deal with countless with no overhead debasement.

5.2 VIRTUAL MEMORY:

Continuous application errands must be kept from being swapped out of memory on the grounds that the irregular and long postponements presented when RAM is depleted and swapping is required are unfortunate in an ongoing framework. To bolster this, Linux gives the mlock() and mlockall() capacities that incapacitate paging for the predefined scope of memory, or for the entire procedure separately. Along these lines, all the "bolted" memory will remain in RAM until the procedure exits or opens the memory. mlock() and mlockall() are incorporated into the POSIX constant augmentations.

5.3 LINUX DEVICE DRIVER:

The real part of the Linux is being made by device driver. For different sections of the working framework, the work is being done in an environment which is special which bring about fiasco on the off chance that they misunderstand things. The communication among working framework , the controlled device equipment being controlled by device driver. The instance for which, the utilization is being made of general device interface. The interest points is being dealt by driver and particular things happen. The controller chip that is being

driven have particular device drivers. The control and status registers (CSRs) are particularly for equipment controller based and different for different devices.

The utilization of the CSRs for beginning and stopping the device, for introduction and determination of any issues to have it. Rather than writing the program to deal with the controllers in each applications, Linux keeps the program. Handling or dealing with the product with an equipment containing the controller is termed as device driver. The Linux piece gadget drivers are, essentially, a common library of favored, memory tenant, low level hardware dealing with timetables. It is Linux's gadget drivers that handle the eccentricities of the gadgets they are supervising.

One of the basic components of is that it abstracts the treatment of gadgets. All hardware gadgets look like standard records; they can be opened, closed, perused and formed using the same, standard, system calls that are used to control archives. Every gadget in the structure is addressed by a gadget extraordinary record, for example the essential IDE hover in the system is addressed by `/dev/hda`.

For square (plate) and character gadgets, these gadget unprecedented archives are made by the `mknod` summon and they depict the gadget using major and minor gadget numbers. Orchestrate gadgets are moreover addressed by gadget unprecedented reports yet they are made by Linux as it finds and instates the framework controllers in the structure. All gadgets controlled by a comparable gadget driver have a run of the mill genuine gadget number. The minor device numbers are utilized to recognize distinctive devices and their controllers, for instance every parcel on the essential IDE plate has an alternate minor device number. Thus, `/dev/hda2`, the second parcel of the essential IDE circle has a noteworthy number of 3 and a minor number of 2. Linux maps the device uncommon record go in framework calls (say to mount a document framework on a square device) to the device driver utilizing the real device number and various framework tables, for instance the character device table, `chrdevs`.

Linux underpins three sorts of equipment device: character, piece and system. Character devices are perused and composed specifically without buffering, for instance the framework's serial ports `/dev/cua0` and `/dev/cua1`. Piece devices must be composed to and read from in products of the square size, ordinarily 512 or 1024 bytes. Square devices are gotten to by means of the cradle reserve and might be haphazardly gotten to, that is to state, any piece can be perused or composed regardless of where it is on the device. Piece devices can be gotten to by means of their device unique document however more regularly they are gotten

to by means of the record framework. Just a square device can bolster a mounted record framework.

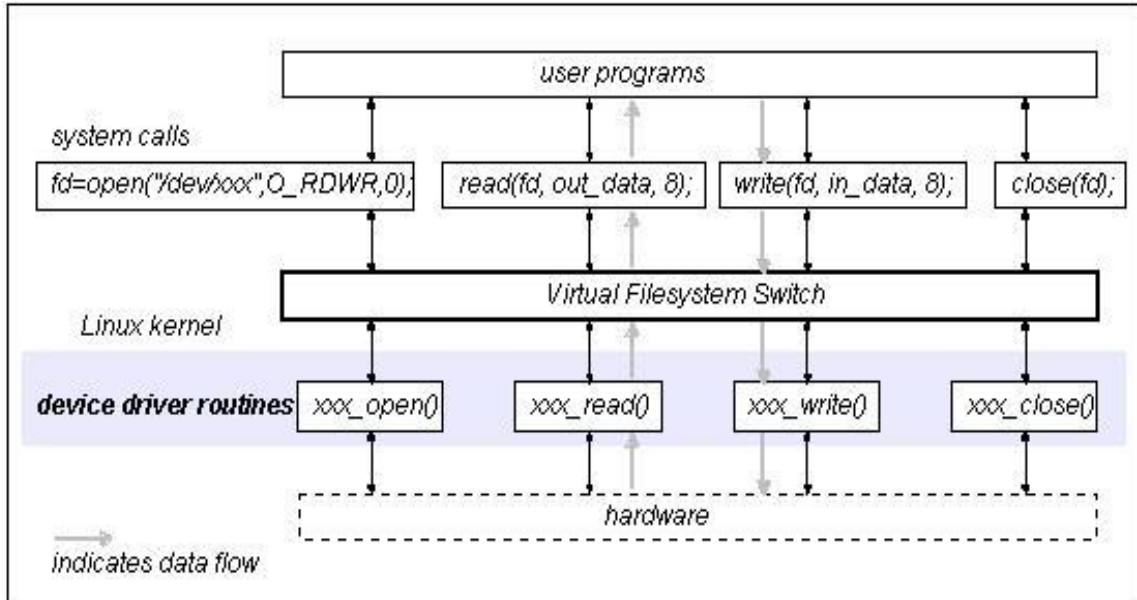


Figure 5.1: Device Driver Interface

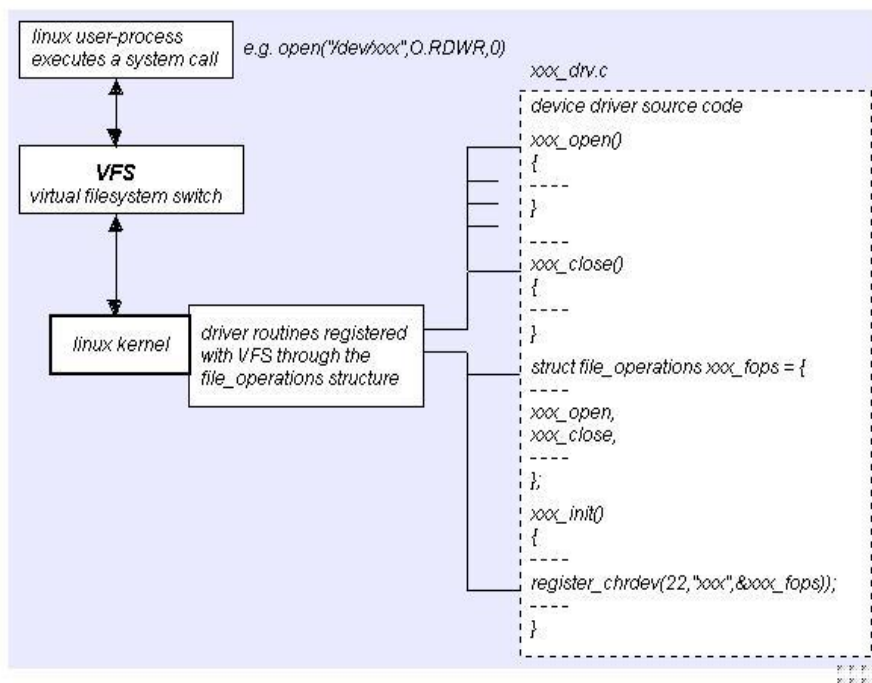


Figure 5.2: Principal Interface between a device driver and Linux kernel

There are an extensive variety of gadget drivers in the Linux bit (that is one of Linux's qualities) notwithstanding they all share some fundamental properties:

Bit code: Device drivers are a bit of the part and, as other code inside the bit, in case they turn out severely they can really hurt the system. A truly made driver may even crash the system, maybe degrading report structures and losing data.

Part interfaces: Device drivers must give a standard interface to the Linux partition or to the subsystem that they are a bit of. For example, the terminal driver gives a report I/O interface to the Linux partition and a SCSI gadget driver gives a SCSI gadget interface to the SCSI subsystem which, along these lines, gives both record I/O and bolster store interfaces to the part.

Part frameworks and organizations: Device drivers make use of standard segment organizations, for instance, memory dissemination, meddle with transport and hold up lines to work.

Loadable: most of the Linux gadget drivers can be stacked on demand as bit modules when they are required and exhausted when they are never again being used. This makes the piece greatly flexible and viable with the system's benefits.

Configurable: Linux gadget drivers can be consolidated with the part. Which gadgets are developed is configurable when the bit is requested.

Dynamic: As the system boots and each gadget driver is presented it scans for the hardware gadgets that it is controlling. It doesn't have any kind of effect if the gadget being controlled by a particular gadget driver does not exist. For this circumstance the gadget driver is basically overabundance and causes no fiendishness isolated from including a tiny bit of the system's memory.

5.4 INTERFACING DEVICE DRIVERS WITH KERNEL:

The Linux portion must have the ability to coordinate with them in standard ways. Each class of gadget driver, character, piece and framework, gives customary interfaces that the bit uses when requesting organizations from them. These fundamental interfaces suggest that the part can treat frequently through and through various gadgets and their gadget drivers absolutely the same. For example, SCSI and IDE hovers bear on contrastingly yet the Linux part uses a comparative interface to them two.

Linux is especially effective, each time a Linux portion boots it may encounter assorted physical gadgets and thusly require unmistakable gadget drivers. Linux licenses you to consolidate gadget drivers at bit develop time by methods for its course of action scripts. Right when these drivers are instated at boot time they may not discover any hardware to control. Distinctive drivers can be stacked as bit modules when they are required. To adjust to this dynamic nature of gadget drivers, gadget drivers enroll themselves with the part as they are instated. Linux keeps up tables of enrolled gadget drivers as a part of its interfaces with them. These tables fuse pointers to timetables and information that reinforce the interface with that class of gadgets.

5.5 TYPES OF DEVICE DRIVERS:

5.5.1 CHARACTER DEVICES:

Character gadgets, minimal complex of Linux's gadgets, are gotten to as reports, applications use standard system calls to open them, read from them, stay in contact with them and close them decisively as if the gadget were a record. This is bona fide paying little respect to the likelihood that the gadget is a modem being used by the PPP daemon to interface a Linux structure onto a framework. As a character gadget is presented its gadget driver registers itself with the Linux bit by including a segment into the chrdevs vector of device_struct data structures. The gadget's huge gadget identifier (for example 4 for the tty gadget) is used as a record into this vector. The critical gadget identifier for a gadget is settled.

5.5.2 BLOCK DEVICES:

Square gadgets also support being gotten the chance to like archives. The frameworks used to give the correct course of action of archive operations for the opened square outstanding record are particularly the same as for character gadgets. Linux keeps up the game plan of enlisted square gadgets as the blkdevs vector. It, like the chrdevs vector, is requested using the gadget's genuine gadget number. Its passageways are moreover device_struct data structures. Not in any manner like character gadgets, there are classes of square gadgets. SCSI gadgets are one such class and IDE gadgets are another. The class registers itself with the Linux part and gives report operations to the bit. The gadget drivers for a class of square gadget give class specific interfaces to the class.

Along these lines, for example, a SCSI gadget driver needs to offer interfaces to the SCSI subsystem which the SCSI subsystem uses to give record operations to this gadget to the

portion. Each piece gadget driver must give an interface to the support store and furthermore the customary record operations interface. Each piece gadget driver fills in its passageway in the blk_dev vector of blk_dev_struct data structures. The record into this vector is, afresh, the gadget's noteworthy number. The blk_dev_struct data structure involves the address of a request routine and a pointer to a summary of interest data structures, each one addressing a request from the bolster hold for the driver to examine or make a shut out of data.

Each time the support reserve wishes to examine or make a shut out of data to or from an enrolled gadget it incorporates a request data structure onto its blk_dev_struct. Figure exhibits that each request has a pointer to no less than one buffer_head data structures, everybody a request to scrutinize or make a shut out of data. The buffer_head structures are dashed (by the support hold) and there may be a strategy watching out for the piece operation to this cradle to wrap up. Each request structure is assigned from a static once-over, the all_requests list. In case the request is being added to an unfilled request list, the driver's request limit is called to start taking care of the request line. By and large, the driver will simply prepare each request on the request list.

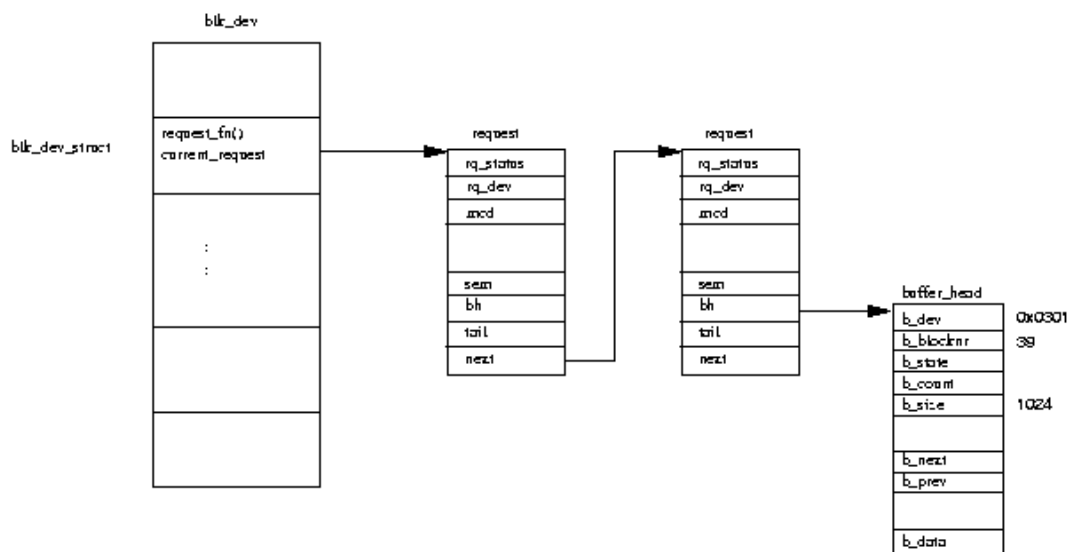


Figure 5.3: Buffer cache Block device request

Once the gadget driver has completed a request it must clear each of the buffer_head structures from the request structure, stamp them as remarkable and open them. This opening of the buffer_head will stir any strategy that has been napping sitting tight for the piece operation to wrap up. An instance of this would be the place a record name is being settled and the EXT2 document framework must read the piece of data that contains the accompanying EXT2 list entry from the square gadget that holds the

record framework. The technique ponders the buffer_head that will contain the index section until the gadget driver stirs it. The request data structure is separate as free so it can be used as a piece of another square inquire.

Circle drives give a more never-ending system to securing data, keeping it on turning plate platters. To form data, an unobtrusive head charges minute particles on the platter's surface. The data is examined by a head, which can perceive whether a particular minute particle is spellbound.

A circle drive involves no less than one platters, each made of finely cleaned glass or mud composites and secured with a fine layer of iron oxide. The platters are affixed to a central shaft and turn at a reliable speed that can move something like 3000 and 10,000 RPM depending upon the model. Balance this with a floppy plate which just contorts at 360 RPM. The circle's scrutinized/make heads are accountable for examining and creating data and there is a couple for each platter, one set out toward each surface. The read/create heads don't physically touch the surface of the platters, rather they drift on a thin (10 millionths of an inch) cushion of air. The read/form heads are moved over the surface of the platters by an actuator. Most of the read/make heads are attached together, they all move over the surfaces of the platters together.

Each surface of the platter is apportioned into thin, concentric circles called tracks. Track 0 is the fringe track and the most critical numbered track is the track closest to the central shaft. A barrel is the game plan of all tracks with a comparative number. So most of the fifth tracks from each side of every platter in the circle is known as chamber 5. As the amount of barrels is the same as the amount of tracks, you consistently watch plate geometries portrayed similar to chambers. Each track is confined into divisions. A section is the most diminutive unit of data that can be created to or perused from a hard circle and it is similarly the plate's square gauge. A run of the mill range size is 512 bytes and the fragment size was set when the plate was sorted out, generally when the circle is created.

A circle is normally depicted by its geometry, the amount of barrels, heads and divisions. For example, at boot time Linux depicts one of my IDE plates as:

```
hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63
```

This infers it has 1050 chambers (tracks), 16 heads (8 platters) and 63 divisions for each track. With a division, or square, size of 512 bytes this gives the plate a limit farthest point of

529200 bytes. This does not facilitate the circle's communicated point of confinement of 516 Mbytes as a part of the territories are used for plate separating information. A couple circles normally find terrible divisions and re-record the plate to work around them.

Hard circles can be additionally subdivided into bundles. A section is a boundless social occasion of fragments circulated for a particular reason. Distributing circle allows the plate to be used by a couple working system or for a couple purposes. An extensive measure of Linux structures has a lone plate with three packages; one containing a DOS record framework, another an EXT2 document framework and a third for the swap portion. The sections of a hard plate are delineated by a bundle table; each entry depicting where the portion starts and completes the extent that heads, fragments and barrel numbers. For DOS sorted out plates, those organized by fdisk, there are four basic circle packages. Not each of the four entries in the package table must be used. There are three sorts of package reinforced by fdisk, basic, created and reasonable. Expanded allocations are not honest to goodness sections by any methods, they contain any number of wise allotments. Enlarged and real bundles were made as a way around the most distant purpose of four fundamental designations.

In the midst of presentation Linux maps the topology of the hard circles in the structure. It finds what number of hard plates there are and of what sort. Additionally, Linux discovers how the individual circles have been divided. This is by and large addressed by an once-over of gendisk data structures pointed at by the `gendisk_head` list pointer. As each circle subsystem, for example IDE, is presented it produces gendisk data structures addressing the plates that it finds. It does this meanwhile as it registers its record operations and incorporates its passage into the `blk_dev` data structure. Each gendisk data structure has an uncommon genuine gadget number and these match the noteworthy amounts of the square novel gadgets. For example, the SCSI plate subsystem makes a lone gendisk section ('sd') with an imperative number of 8, the huge number of all SCSI circle gadgets. Two gendisk sections, the first for the SCSI circle subsystem and the second for an IDE plate controller. This is `ide0`, the basic IDE controller.

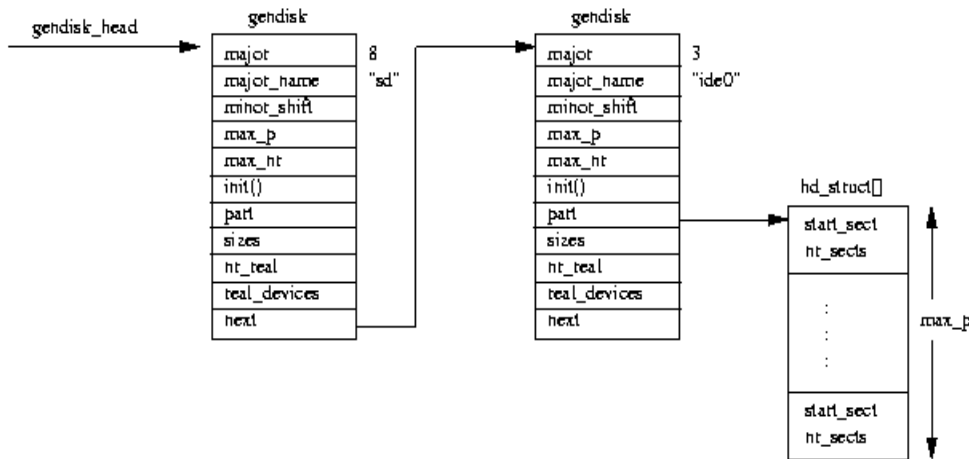


Figure 5.4: Linked Lists of Disks

Notwithstanding the way that the plate subsystems make the `gendisk` segments in the midst of their instatement they are quite recently used by Linux in the midst of bundle checking. Or maybe, every plate subsystem keeps up its own specific data structures which allow it to guide gadget phenomenal major and minor gadget numbers to packages inside physical circles. At whatever point a square gadget is examined from or made to, either through the cushion reserve or record operations, the portion manages the operation to the reasonable gadget using the critical gadget number found in its piece excellent gadget report (for example `/dev/sda2`). It is the individual gadget driver or subsystem that maps the minor gadget number to the certifiable physical gadget.

5.5.3 NETWORK DEVICES:

A system gadget is, so far as Linux's system subsystem is concerned, a substance that sends and gets packs of data. This is customarily a physical gadget, for instance, an ethernet card. Some system gadgets however are modifying just, for instance, the loopback gadget which is used for sending data to yourself. Each system gadget is addressed by a gadget data structure. Arrange gadget drivers select the gadgets that they control with Linux in the midst of system instatement at bit boot time. The gadget data structure contains information about the gadget and the areas of limits that allow the diverse supported system traditions to use the gadget's organizations.

CHAPTER 6

Equipment,Material,and Experimental Setup

Ubuntu is a Debian-based Linux working framework for PCs, tablets and cell phones, where Ubuntu Touch release is utilized; furthermore runs arrange servers, as a rule with the Ubuntu Server version, either on physical or virtual servers, (for example, on centralized computers) as well as with compartments, that is with big business class highlights; keeps running on the most famous models, including server-class ARM-based. Ubuntu is distributed by Canonical Ltd, who offer business support.It depends on free programming and named after the Southern African logic of ubuntu (actually, 'human-ness'), which Canonical Ltd. recommends can be approximately made an interpretation of as "mankind to others" or "I am what I am a direct result of who we as a whole are".It utilizes Unity as its default UI for the desktop. Ubuntu is the most prominent working framework running in facilitated situations, so-called "clouds",as it is the most well known server Linux dissemination. Improvement of Ubuntu is driven by UK-based Canonical Ltd., an organization of South African business person Mark Shuttleworth. Canonical creates income through the offer of specialized support and different administrations identified with Ubuntu.The Ubuntu venture is freely dedicated to the standards of open-source programming advancement; individuals are urged to utilize free programming, examine how it works, enhance it, and disseminate it.

We convert the general kernel of Linux ubuntu into the real time kernel using the following steps:

- 2 Download kernel from Kernel.org . We download tar.xz which is 83 mb .
- 3 In the kernel write `uname -r`. It will show the present kernel like 3.2.0-23.
- 4 Install the packages. Write `sudo apt-get install build-essential libssl-dev libncurses5-dev`
- 5 write `cd ~ homedirectory` ,press enter. Write `cd Downloads`.
- 6 Write `ls`.
- 7 Write `sudo su`.
- 8 Then write `cp linux -4.4.16 tar.xz/usr/src`.

- 9 `Cd/usr/src/` .press enter then write `ls`. It will show `linux -4.4.16 tar.xz`
- 10 `tar -xf linux -4.4.16 tar.xz`. Extract the zip file.
- 11 `cd linux-4.4.16`
- 12 copy bootloader file of previous kernel. `Cp/boot/config-3.2.0-23-generic-pae.config`
- 13 `make menuconfig`.
- 14 `Make`
- 15 `make modules_install`
- 16 `make install`
- 17 `update-initramfs -c -k 4.4.16`
- 18 `update-grub`
- 19 `update bootloader`
- 20 grand unified bootloader

By using these steps we can convert a generic kernel to real time kernel.

6.1 CONVERTING VIRTUAL MEMORY AS DRIVE:

DRIVER LAYOUT:

The driver must be outlined in type of a module as it is added progressively to the running kernel. The kernel question (.ko record) will be made after arrangement which we can include utilizing the `insmod` utility.

`Insmod` utility: Allocation of driver structure is done utilizing `blk_dev` as a part of this schedule.

- Driver introduction
- Using `blk_get_device` the rundown of block device is filtered.
- The wanted device found is added to driver's device list.
- Device empowered.

- Using blk_request_regions allotment done of IO port for the predefined device.
- From the block device's arrangement space required data is perused and put away into private protest of device.
- For instatement of a module three stages are performed:

They are:

1. Sys_create_module(): This framework call demands memory distribution in kernel space utilizing vmalloc().

Status - 1 returned when memory is not accessible.

Status 0 returned when memory is accessible.

2. get_kern_syms(): This capacity settle all the symbols,variables and capacities embedded into image table. It gets every one of the announcements and definitions.

3. module_init(): This capacity instates the module.

rmmod utility: rmmod is partner of init schedule.

- De-designation is done here.
- The device structure is expelled from the memory.
- Driver structure de-assigned.

There are two capacities to evacuate the module:

1. Sys_delete_module(): This capacity checks whether the module is on utilize or not. Dereferencing of variables,symbols,definitions are pronounced in image table.

2. module_cleanup(): This capacity free the memory.

Ventures for Inserting a Module:

1. make:To make .ko,.o records
2. insmod file.ko:major_num=252 num=2(here num speaks to the quantity of minor numbers) on the off chance that 252 is occupied , powerfully it will designate the significant number for the module
3. lsmod | head 5:to check the module was embedded or not

4. `dmesg | tail - 10`:to get the major and minor numbers for the embedded module

5. incorporate the API:`gcc - o hi hello.c`

we will get hi double

6. We need to make the device unique documents

`mknod sc0 c major_num minor_num`

case: `mknod 0 c 252 0`(for first device)

contingent upon the minor numbers we need to make those numerous device unique documents

7. Now run the API parallel means. `/hi` it will solicit the way from device extraordinary document .

Ventures To Remove A Module:

8. expel the module:`rmmod file.ko`

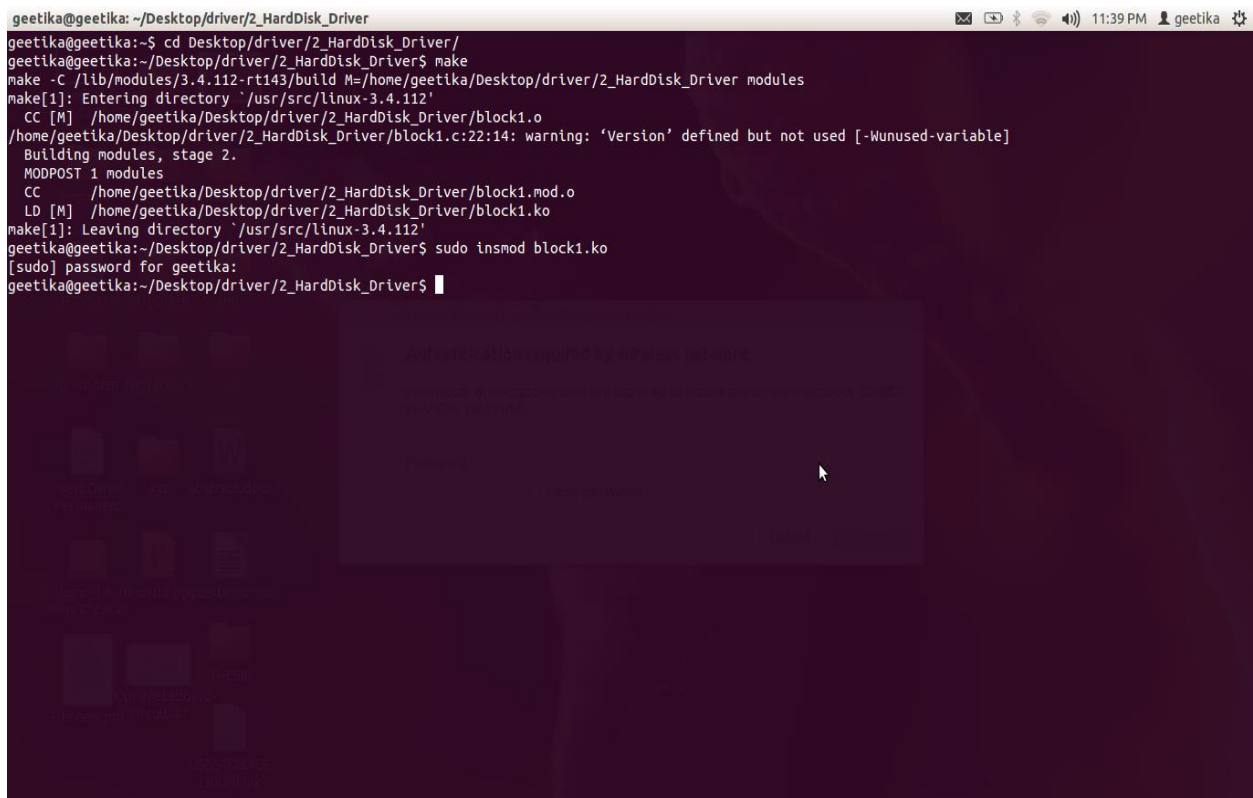
CHAPTER 7

RESULT AND PERFORMANCE EVALUATION

7.1 Converting virtual memory as drive(temporarily)

Make : It is utilized for aggregating the kernel record and will make the protest document and kernel question document.

Sudo insmod block1.ko: The driver is embedded.

A terminal window with a dark background and light text. The window title is 'geetika@geetika: ~/Desktop/driver/2_HardDisk_Driver'. The terminal shows the following commands and output:

```
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ cd Desktop/driver/2_HardDisk_Driver/
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ make
make -C /lib/modules/3.4.112-rt143/build M=/home/geetika/Desktop/driver/2_HardDisk_Driver modules
make[1]: Entering directory '/usr/src/linux-3.4.112'
  CC [M] /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.o
/home/geetika/Desktop/driver/2_HardDisk_Driver/block1.c:22:14: warning: 'Version' defined but not used [-Wunused-variable]
Building modules, stage 2.
MODPOST 1 modules
  CC /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.mod.o
  LD [M] /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.ko
make[1]: Leaving directory '/usr/src/linux-3.4.112'
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ sudo insmod block1.ko
[sudo] password for geetika:
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$
```

Figure 7.1: Module inserted

Cat /proc/devices: we can check the drivers here. Our driver myblkdev is added.

```
geetika@geetika: ~/Desktop/driver/2_HardDisk_Driver
geetika@geetika:~$ cd Desktop/driver/2_HardDisk_Driver/
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ make
make -C /lib/modules/3.4.112-rt143/build M=/home/geetika/Desktop/driver/2_HardDisk_Driver modules
make[1]: Entering directory `/usr/src/linux-3.4.112'
  CC [M] /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.o
/home/geetika/Desktop/driver/2_HardDisk_Driver/block1.c:22:14: warning: 'Version' defined but not used [-Wunused-variable]
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.mod.o
  LD [M] /home/geetika/Desktop/driver/2_HardDisk_Driver/block1.ko
make[1]: Leaving directory `/usr/src/linux-3.4.112'
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ sudo insmod block1.ko
[sudo] password for geetika:
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 5 ttyprintk
 6 lp
 7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
81 video4linux
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
252 usbmon
252 usbmon
```

Figure 7.2: List of inserted modules

```
geetika@geetika: ~/Desktop/driver/2_HardDisk_Driver
81 video4linux
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
252 usbmon
252 usbmon
253 bsg
254 rtc

Block devices:
 1 ramdisk
259 blkext
 7 loop
 8 sd
 9 md
11 sr
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
251 sbd
252 device-mapper
253 virtblk
254 mdp
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$
```

Figure 7.3: sbd module inserted

Format: Now after the driver is inserted, go to home search 'disk utility'. Click on it the drive is now shown. So now format the drive. Click on 'format'.

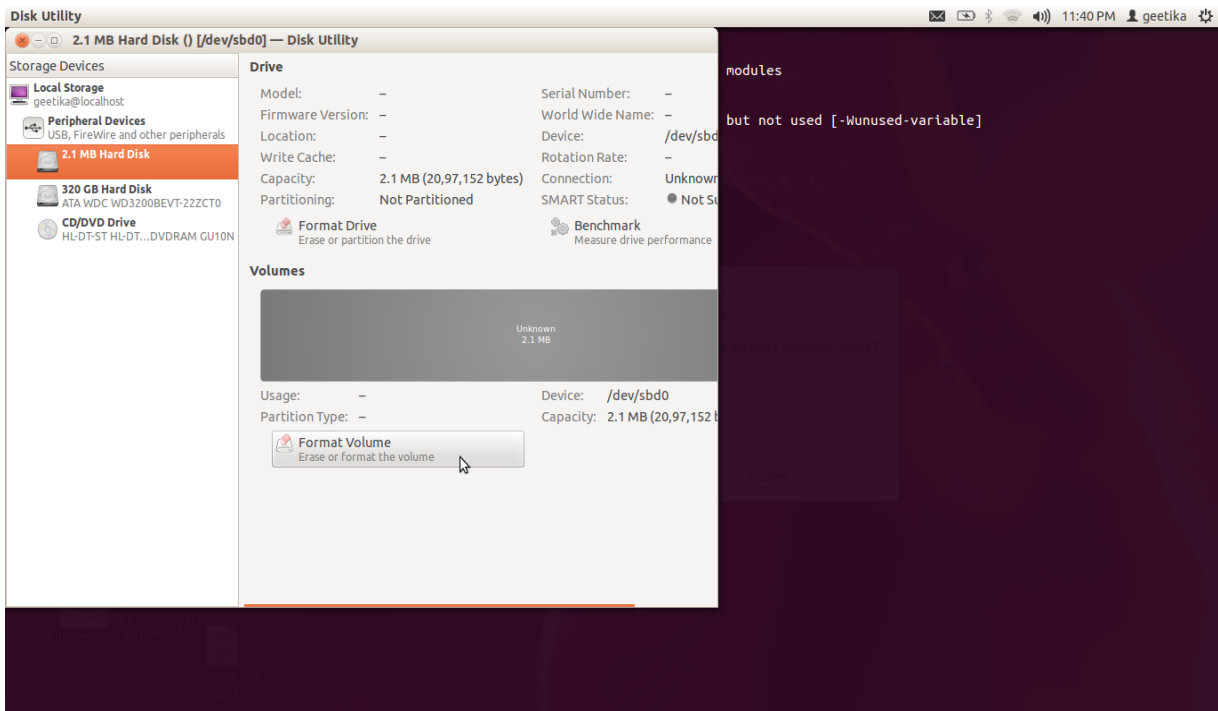


Figure 7.4: Format the Drive

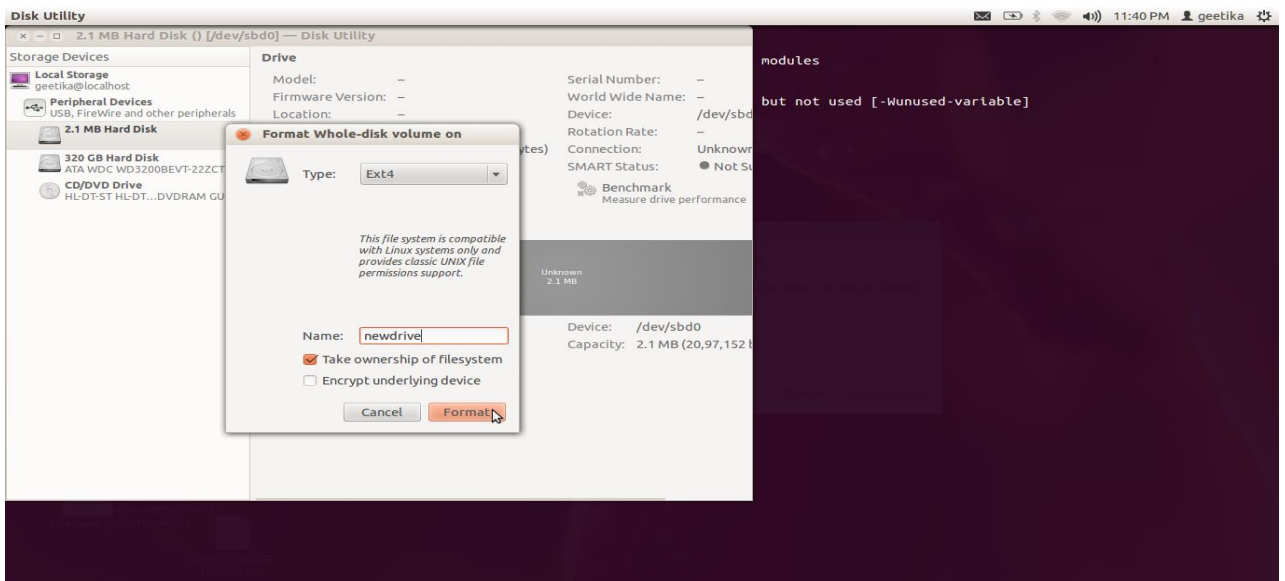


Figure 7.5 : Name the drive

Mount: Now mount the drive. Make a folder in /media and mount in that folder.

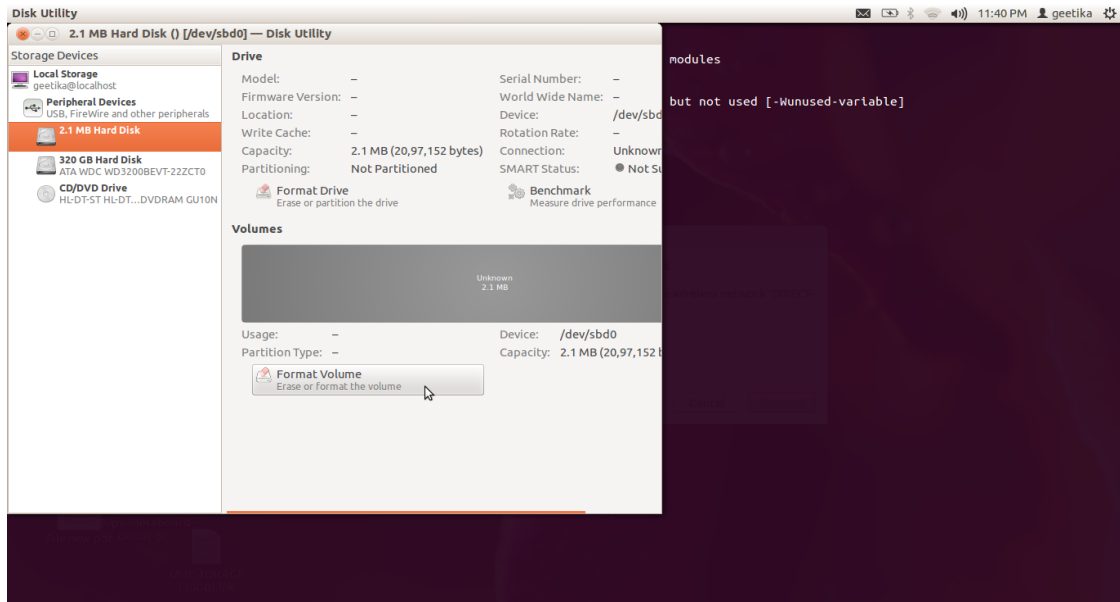


Figure 7.6: Mount the drive

Now our hard disk's virtual memory is converted to drive. We can use this drive like normal drives and store data.

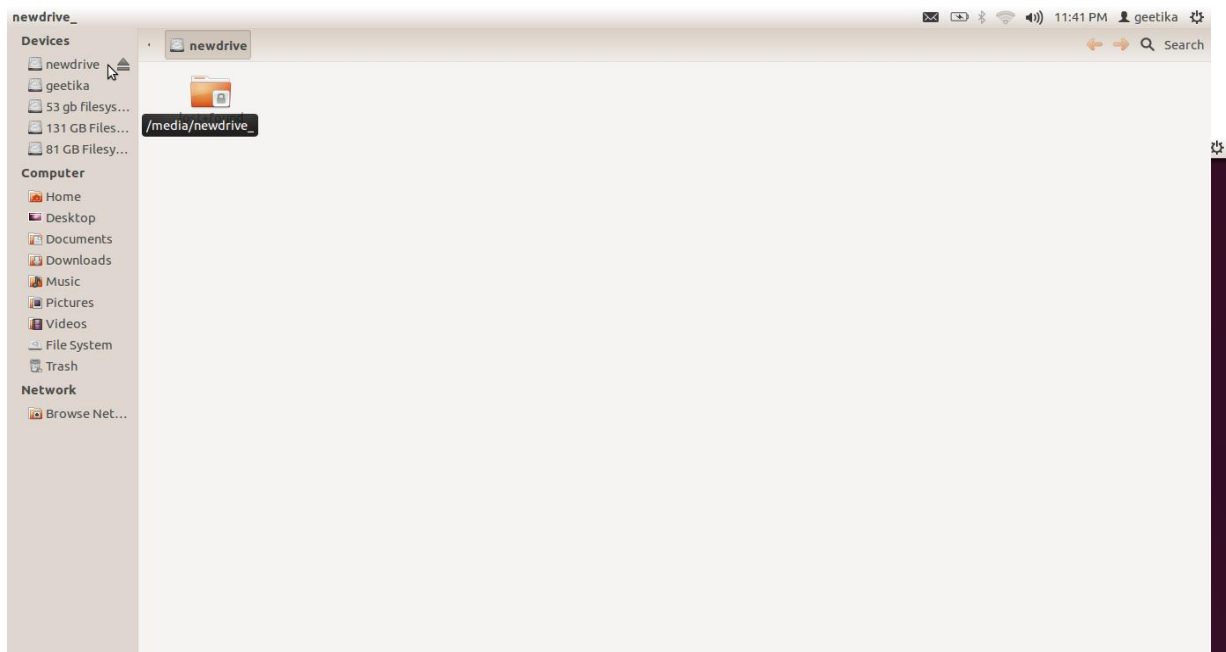


Figure 7.7: New Drive created

Unmount: Now unmount the drive to remove it manually or it will be automatically removed after reboot.

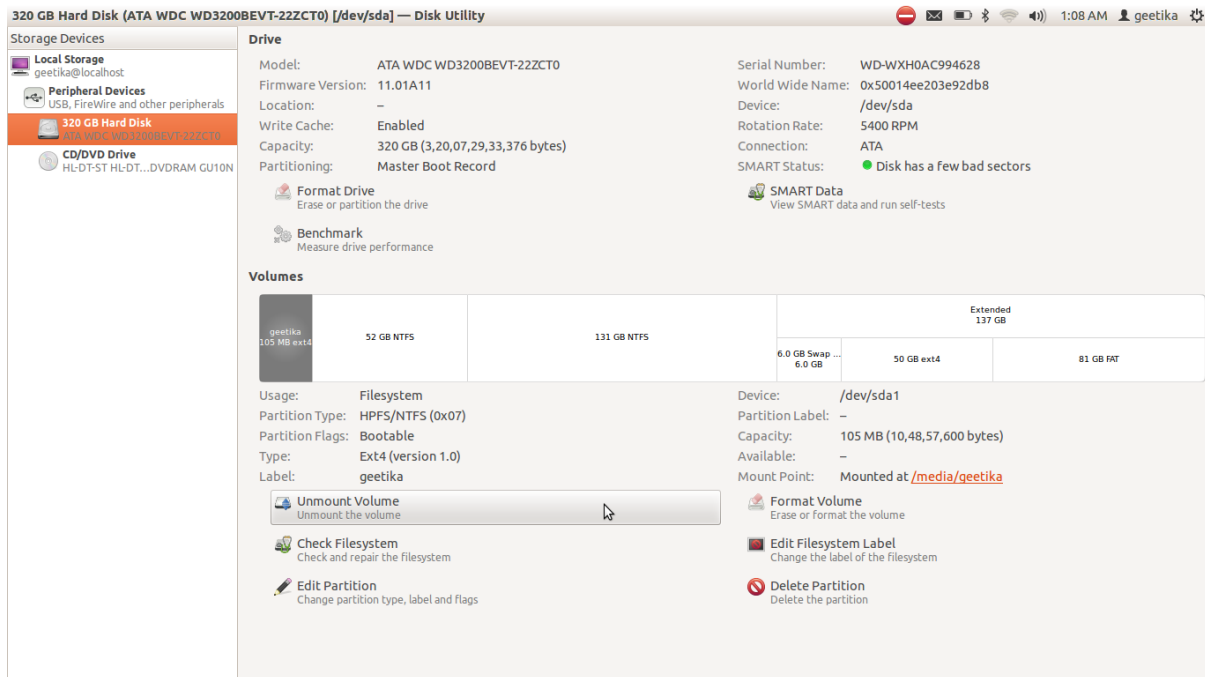


Figure 7.8: Unmount the Drive

Rmmod: Remove the module from the kernel in order to delete it.

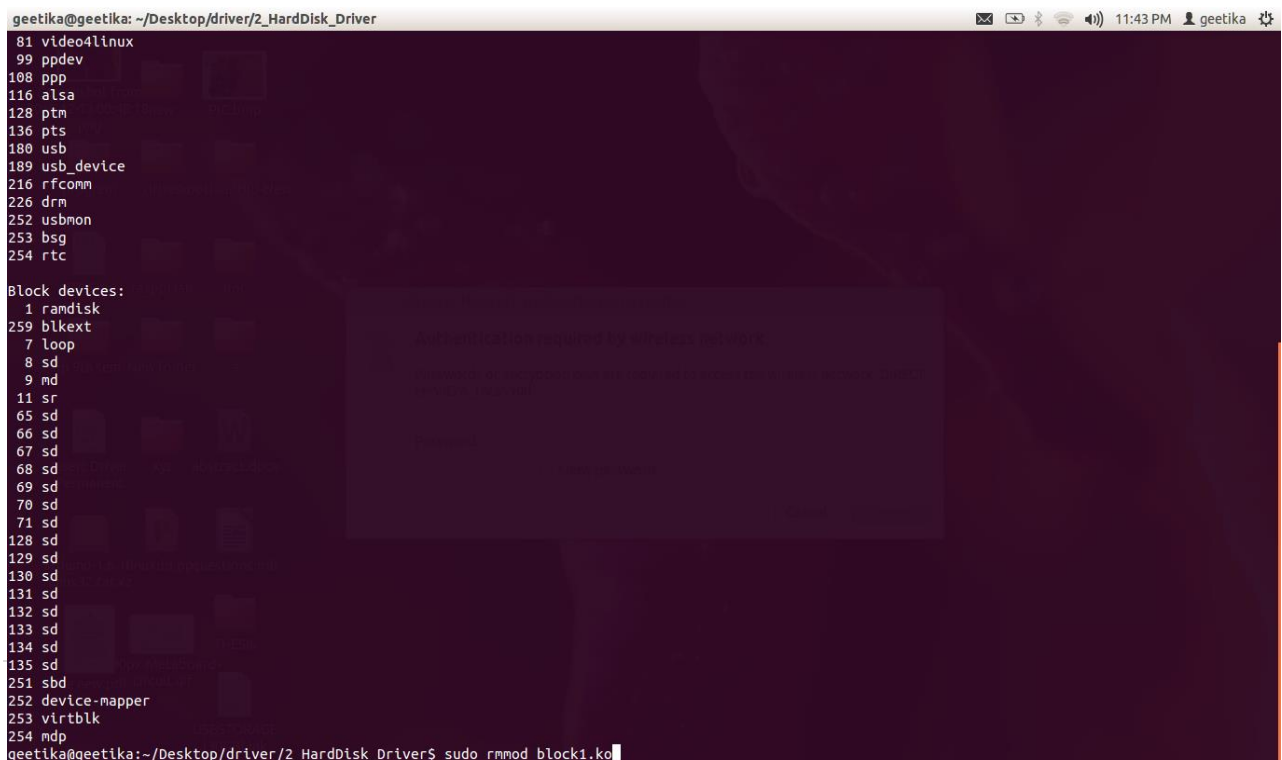


Figure 7.9 : Remove module

```
geetika@geetika: ~/Desktop/driver/2_HardDisk_Driver
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
252 usbmon
253 bsg
254 rtc

Block devices:
 1 ramdisk
259 blkext
 7 loop
 8 sd
 9 md
11 sr
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
251 sbd
252 device-mapper
253 virtblk
254 mdp
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ sudo rmmod block1.ko
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$ cat /proc/devices
```

Figure 7.10: Check module using cat /proc/devices

```
geetika@geetika: ~/Desktop/driver/2_HardDisk_Driver
29 fb
81 video4linux
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
226 drm
252 usbmon
253 bsg
254 rtc

Block devices:
 1 ramdisk
259 blkext
 7 loop
 8 sd
 9 md
11 sr
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
252 device-mapper
253 virtblk
254 mdp
geetika@geetika:~/Desktop/driver/2_HardDisk_Driver$
```

Figure 7.11: Module removed

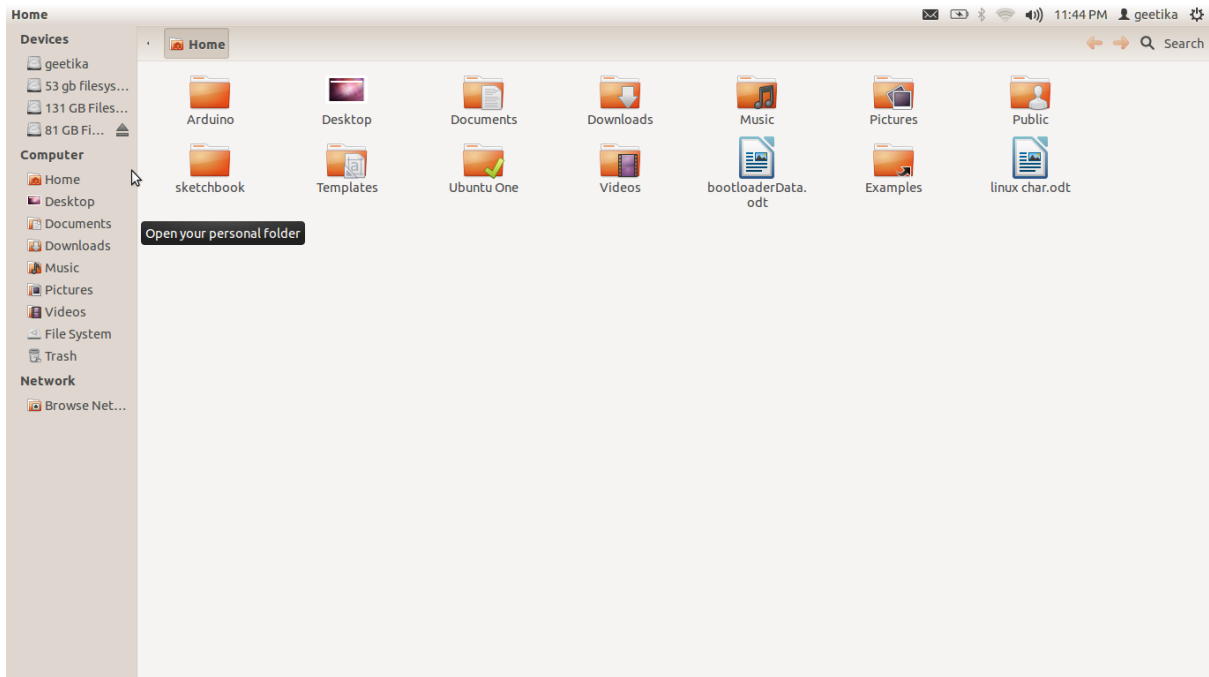


Figure 7.12: Drive removed

7.2 CONVERTING VIRTUAL MEMORY AS DRIVE(PERMANENTLY):

1. `$sudo cp hello.ko /lib/modules/3.4.112-rt143/kernel/drivers/`

2. `$sudo gedit /etc/modules`

here write your driver name for example if your driver is hello.ko then write

hello in /etc/modules

3. `$sudo update-initramfs -u`

4. `$sudo gedit /etc/modules-load.d/modules.conf`

5. `$sudo cp hello.ko /lib/modules/`

6. `$sudo depmod -a`

7. `$reboot`

8. `$sudo rmmod hello.ko`

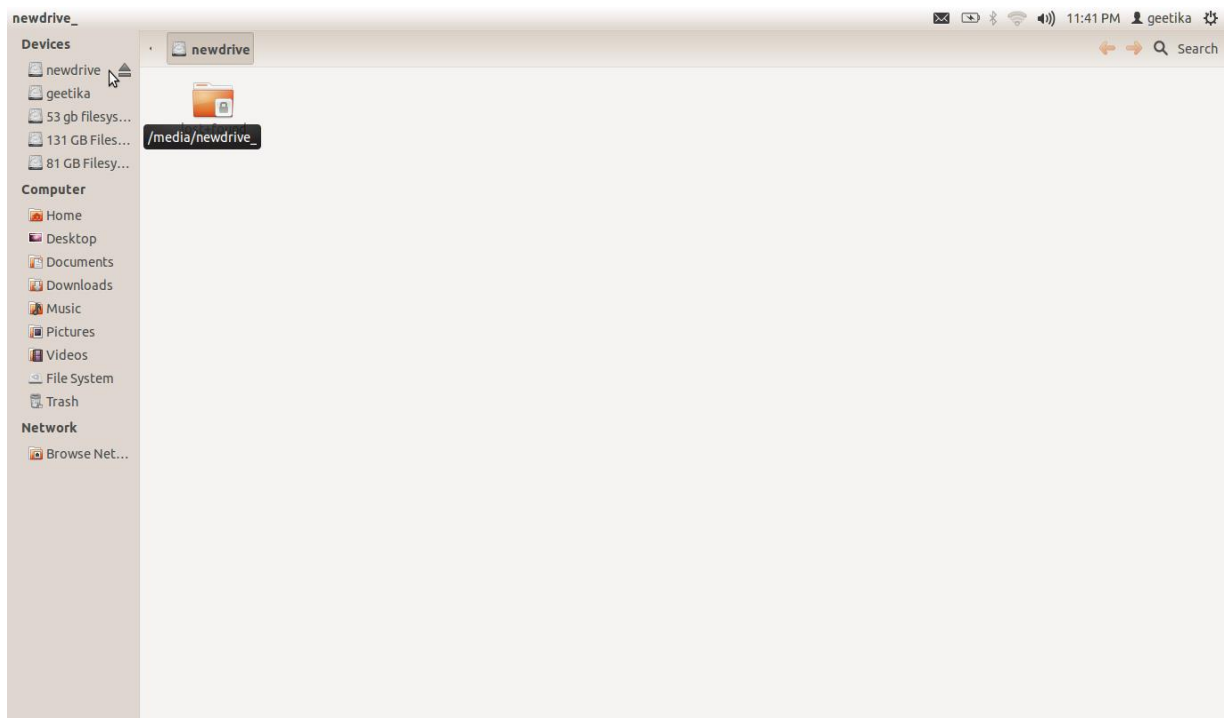


Figure 7.13: Permanent drive created

This drive won't be automatically removed after reboot. In order to remove the drive we can use this command and remove the drive.

7.3 SECURITY FEATURES OF DRIVE

If “geetika” is current user in which we have created the drive then that won't be accessible to other users. Now if we want to make it accessible to other user then we can follow the steps.

1. Check the accessibility of the drive by users.

```
geetika@geetika: /media
geetika@geetika:~$ cd /media
geetika@geetika:/media$ ls -l
total 44
drwx----- 21 geetika geetika 32768 Jan  1  1970 271A-01FB
drwxr-xr-x  2 root    root    4096 Nov  4  15:50 mydrive
drwxr-xr-x  2 root    root    4096 Nov  9  11:14 newdrive
drwxr-xr-x  2 root    root    4096 Dec 13  13:34 windows
geetika@geetika:/media$
```

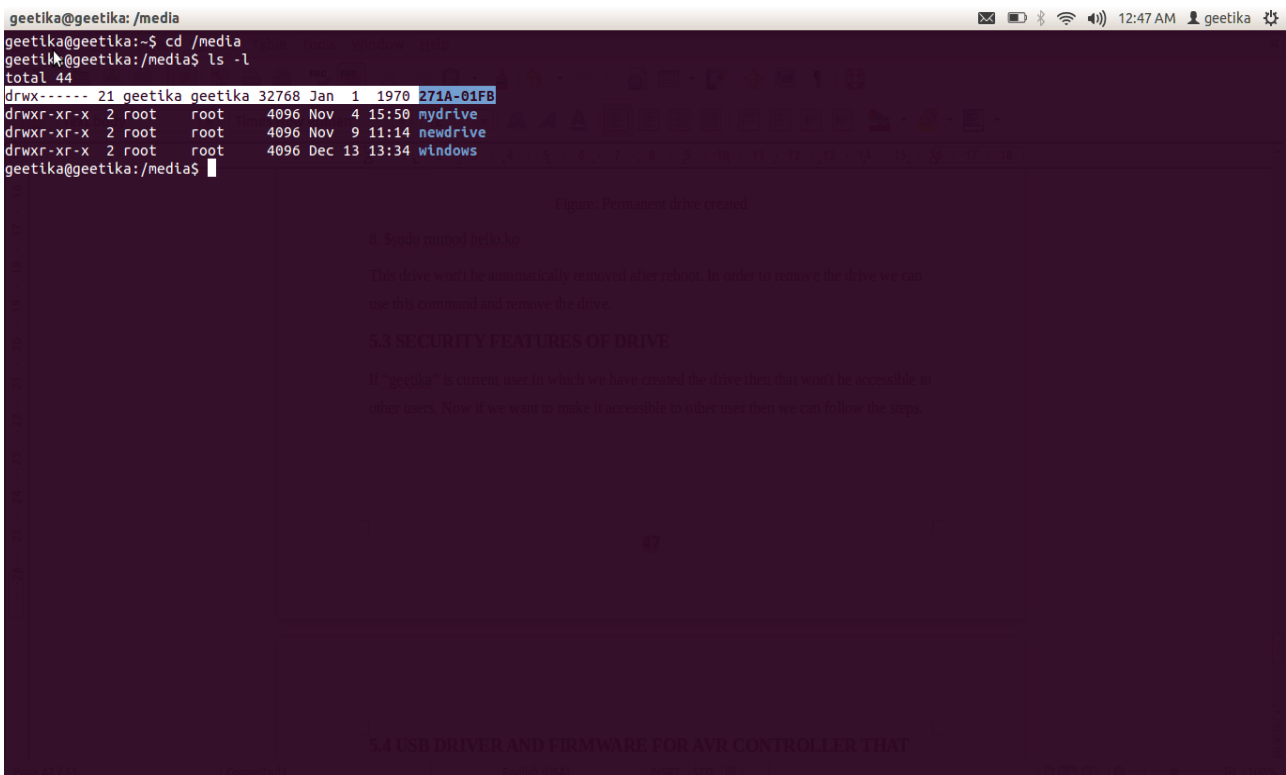


Figure 7.14: User and Group that can access the drive is geetika

2. To make the drive accessible to other users create and add a group.

```
geetika@geetika: /media
geetika@geetika:~$ cd /media
geetika@geetika:/media$ ls -l
total 44
drwx----- 21 geetika geetika 32768 Jan  1  1970 271A-01FB
drwxr-xr-x  2 root   root   4096 Nov  4 15:50 mydrive
drwxr-xr-x  2 root   root   4096 Nov  9 11:14 newdrive
drwxr-xr-x  2 root   root   4096 Dec 13 13:34 windows
geetika@geetika:/media$ sudo groupadd drivenew
```

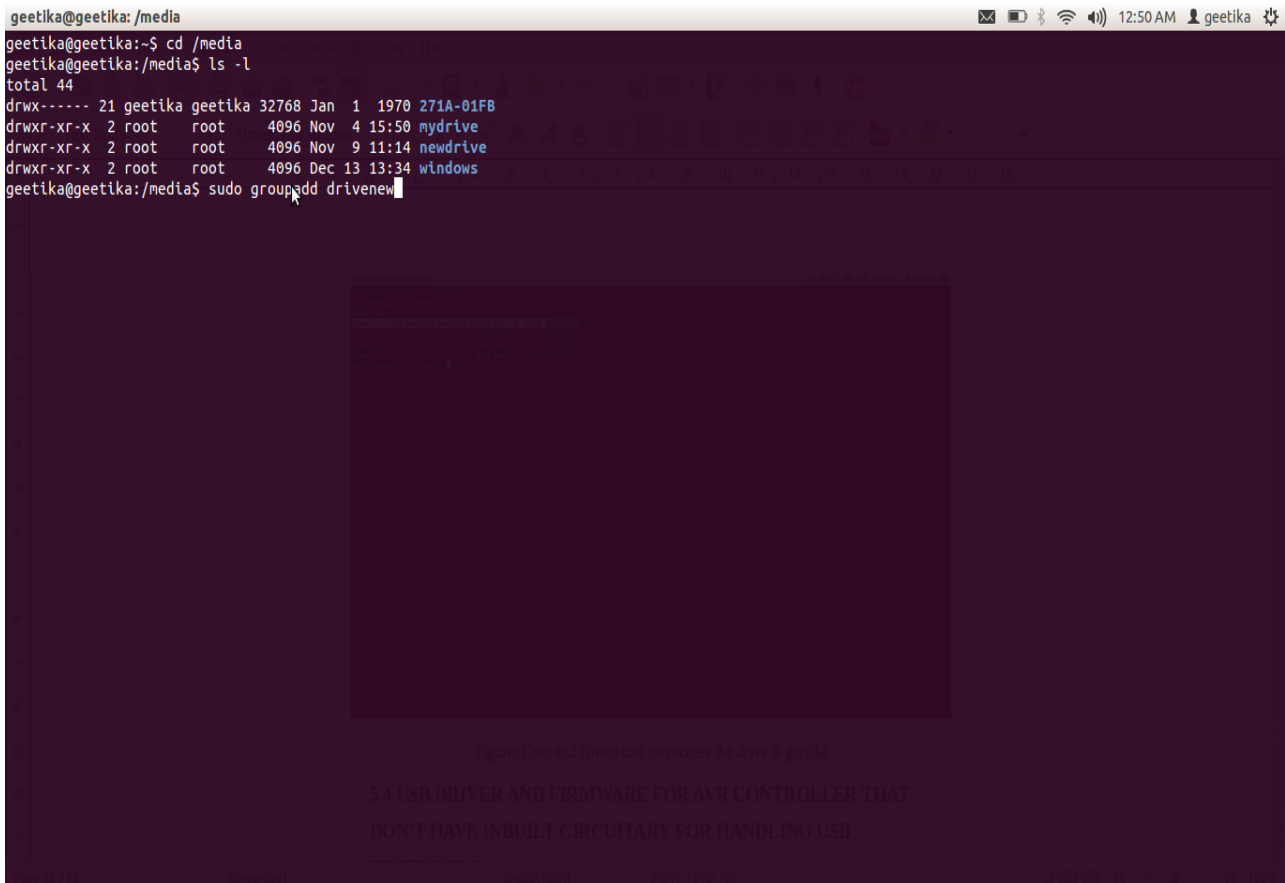


Figure 7.15: Make group drivenew

3. Now add the user which we want to make accessible to that group . Add both geetika and test users to the drivenew group.

```
geetika@geetika: /media
geetika@geetika:/media$ sudo usermod -aG drivenew geetika
geetika@geetika:/media$ sudo usermod -aG drivenew test
geetika@geetika:/media$ groups geetika
geetika : geetika adm dialout cdrom sudo dip plugdev lpadmin sambashare drivenew
geetika@geetika:/media$ groups test
test : test sudo drivenew
geetika@geetika:/media$
```

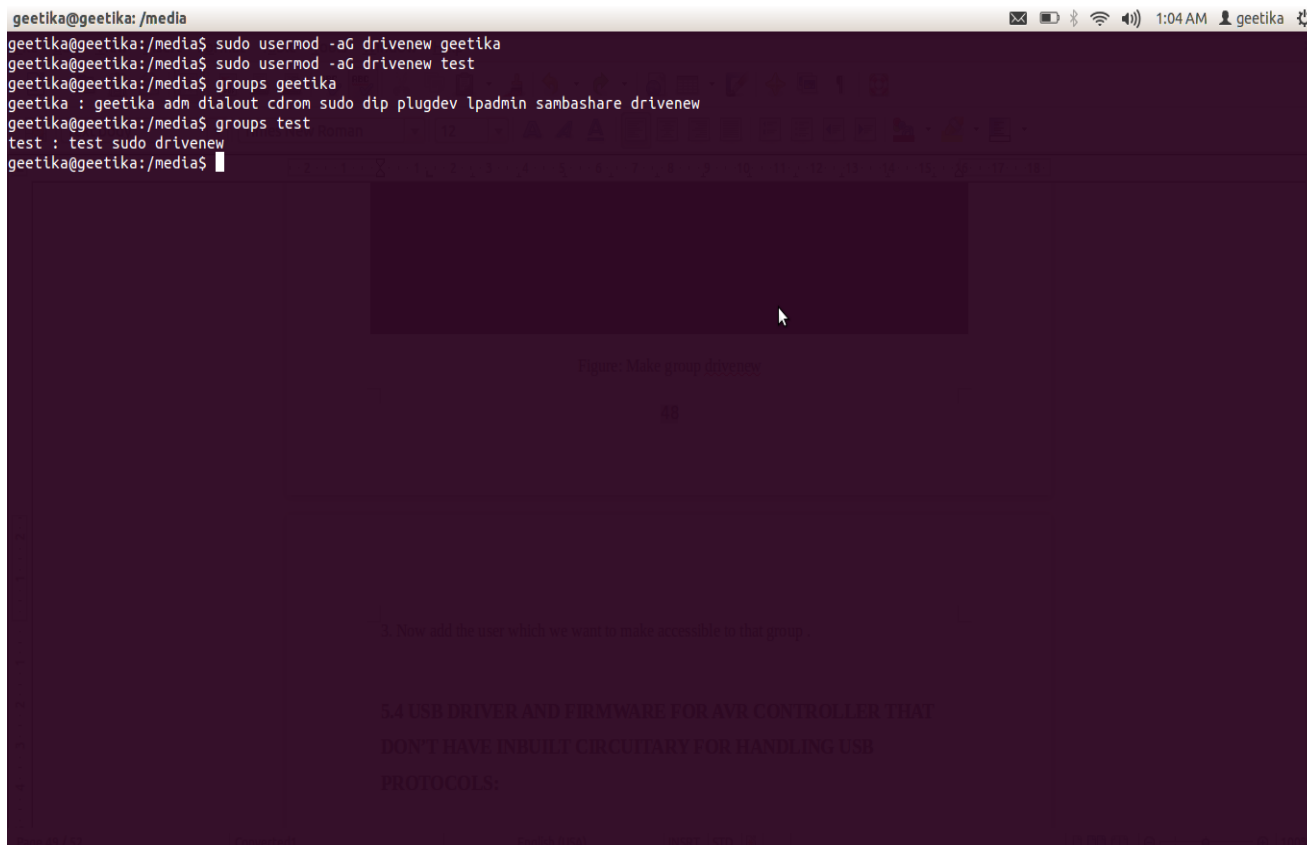
The image shows a terminal window with a dark background and light text. The terminal output shows the following commands and their results: 1. 'sudo usermod -aG drivenew geetika' is executed. 2. 'sudo usermod -aG drivenew test' is executed. 3. 'groups geetika' is executed, showing the output: 'geetika : geetika adm dialout cdrom sudo dip plugdev lpadmin sambashare drivenew'. 4. 'groups test' is executed, showing the output: 'test : test sudo drivenew'. The terminal window has a title bar that reads 'geetika@geetika: /media' and a system tray on the right showing icons for mail, battery, Bluetooth, Wi-Fi, and volume, along with the time '1:04 AM' and the user 'geetika'.

Figure 7.16: Add users to the group.

4. Give the accessibility of drive now to test user using `sudo chown test:drivenew newdrive` and change the permission using `sudo chmod g+rwx newdrive`.

```
geetika@geetika: /media
geetika@geetika:/media$ sudo usermod -aG drivenew geetika
geetika@geetika:/media$ sudo usermod -aG drivenew test
geetika@geetika:/media$ groups geetika
geetika : geetika adm dialout cdrom sudo dip plugdev lpadmin sambashare drivenew
geetika@geetika:/media$ groups test
test : test sudo drivenew
geetika@geetika:/media$ sudo chown test:drive newdrive
```

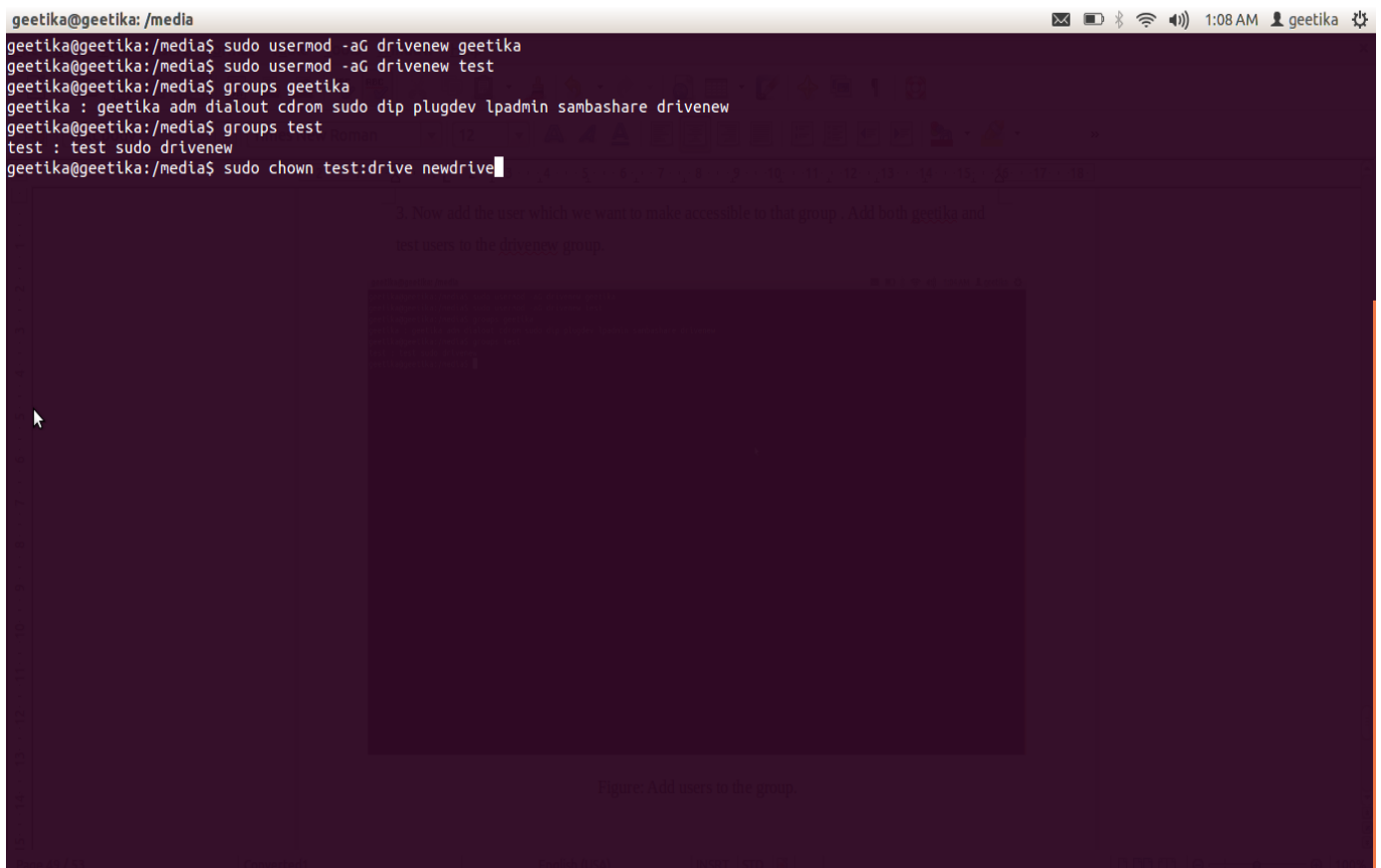


Figure 7.17: Provide accessibility and permission

The drive is now accessible to test user as well and it can use it like user geetika. But the file of one user cannot be written or executed by other user, but this can also be changed further by setting the permission.

CHAPTER 8

CONCLUSION AND FUTURE SCOPE

When the virtual memory of the system is being converted into the drive it was temporary initially but by applying furthermore commands it can be made permanent as well which will not automatically unmount itself on reboot. And talking about the security feature, the drive made in one user having its complete control can be made accessible to the other users as well. The main purpose is accomplished that if when our hard disk's memory is out of space and we are not having any secondary storage as well but we need to store some data ,in that case we can convert the virtual memory of the hard drive to the drive and access it like all the other drives and store our data there.

As a matter of fact, as we all know the system's efficiency can always be improved thus we can make more efficient coding so that the function may occur even more faster and we can always enhance the security features as and when required. Linux as we all know provides the brilliant security features which helps us for better implementation.

LIST OF REFERENCES/BIBLIOGRAPHY:

1. Brockmeyer ,“ Dynamic memory management”,IEEE Proceedings-F, 140(2),pp.107-113,2006
2. Apurva ,” Deadline meet in the real time system”, BIHTL,2015
3. Moshe PEELEH “ Priority simulation in real time system ”,IEEE Proceedings-P,134,pp.201-220,2014
4. Ch Ykman-Couvreur, (2013 IEEE) “Design time application exploration in real time system”,IEEE Proceedings-I,201,pp.123-211,2011
5. Grant Martin (2014 IEEE) “Synchronization and control of task scheduling in rtos”.
6. Oliver Arnold (2015 IEEE) “Memory and power management in rtos”.
7. H.Nikolov (2015 IEEE) “RTOS architecture with microkernel implementation”.
8. M.Horowitz (2013 IEEE) “Real time challanges in dynamic mapping”.
9. D.A.Patterson (2014 IEEE) “basic architecture of operating system .”
10. M.H.Wiggers (2014 IEEE) “Analysis of hard real time system”
11. M.Kaldevi (2011 IEEE ICC) “Design RTS using priority based preemptive scheduling”
12. Tanenbaum, Andrew (2008). Modern Operating Systems. Upper Saddle River, NJ: Pearson/Prentice Hall. p. 160. ISBN 978-0-13-600663-3.
13. "RTOS Concepts".
14. "Context switching time". Segger Microcontroller Systems. Retrieved 2009-12-20.
15. "RTOS performance comparison on emb4fun.de".
16. CS 241, University of Illinois
17. "2014 Embedded Market Study" (PDF).EE Live!. UBM. p. 44.

APPENDIX

```
/*
 * A sample, extra-simple block driver. Updated for kernel 2.6.31.
 *
 */

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/vmalloc.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>

MODULE_LICENSE("Dual BSD/GPL");
static char *Version = "1.4";

static int major_num = 0;
module_param(major_num, int, 0);
static int logical_block_size = 512;
module_param(logical_block_size, int, 0);
static int nsectors = 4096; /* How big the drive is */
module_param(nsectors, int, 0);

/*
```



```

* We can tweak our hardware sector size, but the kernel talks to us
* in terms of small sectors, always.
*/
#define KERNEL_SECTOR_SIZE 512

/*
* Our request queue.
*/
static struct request_queue *Queue;

/*
* The internal representation of our device.
*/
static struct sbd_device
{
    unsigned long size;
    spinlock_t lock;
    u8 *data;
    struct gendisk *gd;
} Device;

/*
* Handle an I/O request.
*/
static void sbd_transfer(struct sbd_device *dev, sector_t sector,
    unsigned long nsect, char *buffer, int write) {
    unsigned long offset = sector * logical_block_size;
    unsigned long nbytes = nsect * logical_block_size;

    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "sbd: Beyond-end write (%ld %ld)\n", offset,
nbytes);

```

```

        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}

static void sbd_request(struct request_queue *q) {
    struct request *req;

    req = blk_fetch_request(q);
    while (req != NULL) {
        // blk_fs_request() was removed in 2.6.36 - many thanks to
        // Christian Paro for the heads up and fix...
        //if (!blk_fs_request(req)) {
        if (req == NULL || (req->cmd_type != REQ_TYPE_FS)) {
            printk (KERN_NOTICE "Skip non-CMD request\n");
            __blk_end_request_all(req, -EIO);
            continue;
        }
        sbd_transfer(&Device, blk_rq_pos(req), blk_rq_cur_sectors(req),
                    req->buffer, rq_data_dir(req));
        if ( ! __blk_end_request_cur(req, 0) ) {
            req = blk_fetch_request(q);
        }
    }
}

```

/*

* The HDIO_GETGEO ioctl is handled in blkdev_ioctl(), which

* calls this. We need to implement getgeo, since we can't

* use tools such as fdisk to partition the drive otherwise.

```
static int __init sbd_init(void) {
    /*
     * Set up our internal device.
     */
    Device.size = nsectors * logical_block_size;
    spin_lock_init(&Device.lock);
    Device.data = vmalloc(Device.size);
    if (Device.data == NULL)
        return -ENOMEM;
    /*
     * Get a request queue.
     */
    Queue = blk_init_queue(sbd_request, &Device.lock);
    if (Queue == NULL)
        goto out;
    blk_queue_logical_block_size(Queue, logical_block_size);
    /*
     * Get registered.
     */
    major_num = register_blkdev(major_num, "sbd");
    if (major_num < 0) {
        printk(KERN_WARNING "sbd: unable to get major number\n");
        goto out;
    }
    /*
     * And the gendisk structure.
     */
    Device.gd = alloc_disk(16);
    if (!Device.gd)
```

```

        goto out_unregister;
Device.gd->major = major_num;
Device.gd->first_minor = 0;
Device.gd->fops = &sbd_ops;
Device.gd->private_data = &Device;
strcpy(Device.gd->disk_name, "sbd0");
set_capacity(Device.gd, nsectors);
Device.gd->queue = Queue;
add_disk(Device.gd);

return 0;

out_unregister:
    unregister_blkdev(major_num, "sbd");
out:
    vfree(Device.data);
    return -ENOMEM;
}

static void __exit sbd_exit(void)
{
    del_gendisk(Device.gd);
    put_disk(Device.gd);
    unregister_blkdev(major_num, "sbd");
    blk_cleanup_queue(Queue);
    vfree(Device.data);
}

module_init(sbd_init);
module_exit(sbd_exit);

```