

DEVICE DRIVER DIAGNOSIS & TROUBLESHOOTING

Submitted in partial fulfillment of the requirements for the award of Bachelor of
Engineering Degree in Electrical and Electronics Engineering

By

Mathankumar J(38140027)

Suresh A(38140053)



DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited with Grade “A” by NAAC

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119

MAY 2022



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **MATHANKUMAR J** (Reg. No. 38140027) and **SURESH A** (Reg. No. 38140053) who Carried out the project entitled "**DEVICE DRIVER DIAGNOSIS AND TROUBLESHOOTING**" under our supervision from November 2021 to May 2022.

N SARAVANAN

Dr.S.JAYAPRAKASH

FRANCIS PRAVIN ANTONY MICHEALRAJ

Internal Guide

External Guide

Dr. V. SIVACHIDAMBARANATHAN
Head of the Department

Submitted for Viva voce Examination held on _____

Name:

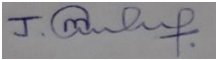
Signature:

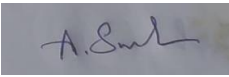
Internal Examiner

External Examiner

DECLARATION

We **MATHANKUMAR J** (Reg. No. 38140027) and **SURESH A** (Reg. No. 38140053) hereby declare that the Project Report entitled “**DEVICE DRIVER DIAGNOSIS AND TROUBLESHOOTING**” done by us under the guidance of **Dr. S.JAYAPRAKASH** is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Electrical and Electronics Engineering.

1. 

2. 

DATE : 20/04/2022

PLACE: CHENNAI.

SIGNATURE OF THE CANDIDATES

ACKNOWLEDGEMENT

We are pleased to acknowledge our sincere thanks to Board of Management of **SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. We are grateful to them.

We convey our thanks to **Dr. N. M. Nandhita., Dean, School of Electrical and Electronics Engineering** and **Dr. V. Sivachidambaranathan, Head of the Department, Dept. of Electrical and Electronics Engineering** for providing us necessary support and details at the right time during the progressive reviews.

We would like to express our sincere and deep sense of gratitude to our External Project Guides **N SARAVANAN., Technical Specialist, HCL., And FRANCIS PRAVIN ANTONY MICHEAL RAJ., Technical Staff., HCL** And Internal Project Guide **Dr. S. JAYAPRAKASH** for their valuable guidance, suggestions and constant encouragement paved way for the successful completion of our project work.

We wish to express our thanks to all Teaching and Non-teaching staff members of the Department of Electrical and Electronics Engineering who were helpful in many ways for the completion of the project.

ABSTRACT

A Device driver diagnosis and Troubleshooting is proposed for diagnose the problem in driver and verifying the function in different drive and reporting gcov based information as a HTML output using Icov. The device driver is a software which is mounted in between kernel and hardware. The Application will reduce the man hours to debug. using these applications we can easily understand the code flow and also we can easily understand the positive and negative code path and capture additional debug information during error scenario. A debugfs is a special type of file system used for debugging purposes. debugfs is a simple-to-use RAM-based file system and It exists as a simple way for kernel developers to make information available to user space. The virtual box is a software will handle more than one virtual machine inside it. ubuntu 18.04 is was designed to download the upstream kernel and generate the various debug information. It consists of PC with Virtual box installed, Ubuntu 18.04 ,nvme drive ,serial cable etc,.

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	V
	LIST OF FIGURES	IX
	LIST OF TABLES	X
1	INTRODUCTION	
1.1	GENERAL INTRODUCTION	11
1.2	BLOCK DIAGRAM	12
1.2.1	BLOCK DISCRIBUTION	13
1.3	HARDWARE REQUIREMENT	14
1.4	SOFTWARE REQUIREMENT	15
2	LITERATURE SURVEY	
2.1	EXISTING SYSTEM	16
2.1.1	COHESION ANALYSIS IN LINUX KERNEL	16
2.1.2	A COMREHENSIVE STUDY OF KERNEL	17
2.1.3	PERFORMANCE AND SCALABILITY OF LIGHTWEIGHT MULTI-KERNEL	18

2.1.4	DETECTING STRUCT MEMBER-RELATED MEMORY LEAKS USING ERROR CODE ANALYSIS IN LINUX KERNEL	18
2.1.5	ANALYSIS OF EMBEDDED LINUX USING KERNEL ANALYSIS SYSTEM	19
2.1.6	TROUBLESHOOTING HARDWARE PROBLEMS IN LINUX	19

3 AIM AND SCOPE & SOFTWARE TOOLS

3.1	PROJECT AIM	21
3.2	PROJECT SCOPE	21
3.3	VIRTUAL BOX	21
3.4	UBUNTU 18.04	23
3.5	VIRTUAL BOX EXTENSION PACK	23
3.6	GCOV	24
3.7	LCOV	25
3.8	KGDB	28
3.9	KDB	28
3.10	DEBUGFS	29
3.11	NVME	31

4	KERNEL & DEVICE DRIVERS	
4.1	USER SPACE	33
4.2	KERNEL SPACE	34
4.2.1	PROCESS MANAGEMENT	38
4.2.2	MEMORY MANAGEMENT	40
4.2.3	NETWORKING UNIT	42
4.2.4	VIRTUAL FILE SYSTEM	44
4.2.5	INTER PROCESS COMMUNICATION	45
4.2.6	DEVICE DRIVERS	46
4.2.6.1	CHARACTER DRIVER	47
4.2.6.1.1	NVME CONTROLLER	48
4.2.6.2	BLOCK DRIVER	48
4.2.6.2.1	NVME NAMESPACE	50
5.	CONCLUSION	52

LIST OF FIGURES FIGURE

Fig.No.	TITLE	PAGE No
1.1	Block Diagram of Proposed System	12
3.1	Lcov output	26
3.2	Block diagram for KGDB	28
4.1	Mapping of Virtual memory	42
4.2	output of USB driver	50

LIST OF TABLES

Table.No.	TITLE	PAGE No.
3.1	Difference between GCOV and LCOV	27
3.2	Difference between KGDB and KDB	29
4.1	Difference between Kernel mode and User mode	35
4.2	Types of System calls	37

CHAPTER 1

1.1 GENERAL INTRODUCTION

A device driver (sometimes known as a 'driver') is a piece of software that controls a certain type of computer equipment. It connects the operating system and other programmes to the physical device via a software interface. Character, Block, Network, and USB drivers are all available in GNU/Linux. We'll only look at character drivers in this section. The most common drivers are character drivers. They allow unrestricted access to hardware devices. Character drivers can be thought of as a long sequence of bytes, similar to conventional files but only accessible in sequential order. At the very least, character drivers support the `open()`, `close()`, `read()`, and `write()` methods. The text terminal, This category includes `/dev/console`, serial consoles (`/dev/stty*`), and audio/video drivers. A driver must be present in order for a device to be usable. So, let's look at how an application uses a driver to access data from a device. The following four significant entities will be discussed. Troubleshooting is a type of problem-solving technique that is frequently used to fix failing items or processes on a machine or system. It's a logical, systematic search for the root of a problem in order to solve it and restore functionality to a product or process. To diagnose the symptoms, troubleshooting is required. Eliminating probable sources of an issue is the process of determining the most likely cause.

Finally, troubleshooting necessitates proof that the solution returns the product or process to its previous state of operation. Troubleshooting is the process of identifying and diagnosing "trouble" in a system's management flow that is caused by a failure of some sort. The issue is first presented as malfunctioning symptoms, and troubleshooting is the process of identifying and correcting the sources of these symptoms. A system's expected, wanted, or intended behaviour can be characterised in terms of its expected, desired, or intended behaviour (usually, for artificial systems, its purpose). The system's events or inputs are anticipated to produce specified results or outputs. (Take, for example, picking "print" from the drop-down menu.)

Various computer programmes are designed to result in hardcopy being produced by a given device). A symptom is any unusual or unwanted activity. Troubleshooting is the

process of determining the exact cause of a condition. The inability of the product or method to yield any results is a common symptom. (For example, nothing was printed.) Corrective action can then be made to avoid similar failures in the future.

1.2 BLOCK DIAGRAM

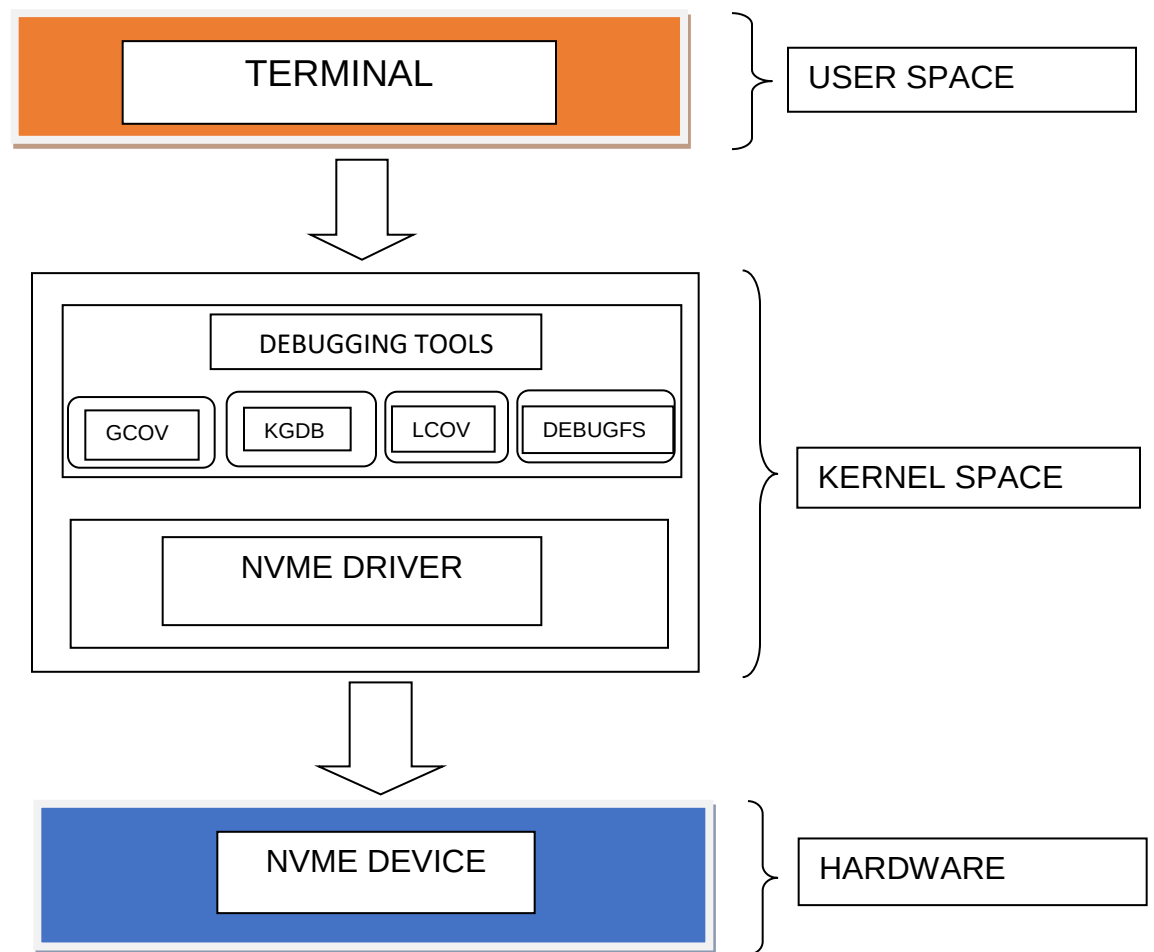


Fig.no. 1.1 Block diagram of proposed system

1.2.1 BLOCK DESCRIPTION

TERMINAL

The command-line, console, or shell are all terms used to describe the Linux terminal. It's our computer's text interface. With this system application, we can comprehend commands and write scripts. It may appear tough at first, but once we get the hang of it, it will be simple to use.

By inputting commands in the terminal with the Linux command line, we may manually execute commands. Shell Scripts can also be used to automate commands. A Linux terminal is a common way to communicate with a Linux server. Although using the command line appears to be a challenging chore, it is much easier if we start with the basics and build our skills from there.

GCOV

gcov is a coverage test programme. Use it in conjunction with GCC to analyse your programmes in order to assist you write more efficient, faster-running code and uncover areas of your programme that haven't been tested. Gcov can be used as a profiling tool to see where your optimization efforts will have the most impact on your code. You can also use gcov in conjunction with another profiling tool, gprof, to determine which areas of your code consume the most computing time.

KGDB

A kernel debugger is the kgdb. Kgdb is used for kernel debugging in the same way as gdb is used for user application debugging. Set breakpoints in the kernel code path, inspect kernel data structure, and control the kernel code executing flow with the debugger. The debug client for kgdb is gdb, which provides debugging at the C source code level and also recognises the kernel data structure. Although kdb does not require a client, it can only debug at the assembly language level and does not comprehend kernel data structures. As a result, kdb can only dump data structures at the raw memory level.

LCOV

GCC's coverage testing tool gcov has a graphical front-end called lcov. It gathers gcov data for numerous source files and generates HTML pages with the source code annotated with coverage data. It also has overview pages to make navigating the file structure easier. Collect coverage data with lcov and generate HTML pages with genhtml. Data on coverage can be gathered from the currently running Linux kernel or from a user space application.

DEBUGFS

Description The debugfs program is an interactive file system debugger. It can be used to examine and change the state of an ext2, ext3, or ext4 file system. device is the special file corresponding to the device containing the file system (e.g /dev/hdXX).

NVME DEVICE

NVMe technology provides superior storage, superior speed and superior compatibility. Since NVMe utilises PCIe sockets, it transfers 25x more data than the SATA equivalent. Along with more data, NVMe's commands are 2x faster than that of AHCI drivers. In addition, NVMe input/output operations per second (IOPS) exceeds 1 million and is up to 900% faster compared to AHCI drives. NVMe also communicates directly with the system CPU, giving it incredible speeds due to its compatibility. NVMe drives work with all major operating systems regardless of form factor.

1.3 HARDWARE REQUIREMENTS

1. NVMe Driver\RAM Disk Drive.
2. Serial Cable.
3. Spare PC(KGDB).

1.4 SOFTWARE REQUIREMENTS

1. Virtual Box(Version-6.1.26).
2. Ubuntu(Version-18.04).
3. Upstream Kernel(Version-5.15.25).
4. Virtual Box Extension Pack(Version-6.1.26).
5. NVMe-cli Tool.

CHAPTER 2

LITERATURE SURVEY

2.1 EXISTING SYSTEM

In computing, a device driver (also known as a driver) is a computer software that operates or controls a specific type of computer-connected device. A driver is a software interface to hardware devices that allows operating systems and other computer applications to access hardware operations without knowing the specifics of the hardware. Writing a device driver necessitates a thorough understanding of how the hardware and software for a particular platform function interact. Drivers operate in a highly privileged environment because they require low-level access to hardware operations in order to work, and if something goes wrong, they can create system operating concerns.

On current operating systems, however, most user-level software can be stopped without affecting the rest of the system. If a device is incorrectly coded, even drivers running in user mode can cause a system to crash. These variables make diagnosing problems more complex and dangerous.

2.1.1 COHESION ANALYSIS IN LINUX KERNEL

D.Janakiram IIT Madras, Chennai, India, Department of Computer Science and Engineering Strong coupling, such as common coupling, should be used with caution among modules, according to this paper from the Indian Institute of Technology Madras in Chennai, Tamil Nadu. The Linux kernel is examined in terms of these common coupling instances, and the results reveal that modules have an excessive amount of common coupling. It is stated that kernel maintenance will be extremely difficult in the near future unless the kernel is redesigned with the bare minimum of common coupling. We attribute this issue to the kernel's lack of data abstractions. To back up our claim, we looked at cohesion in the Linux kernel at both the module (file) and system level. Although the function level cohesiveness is excellent, the module level cohesion is low, indicating a lack of focus on data abstractions at the module level. We feel that this is an intrinsic flaw in the procedural paradigm, which prioritises utility over data. As a result, we recommend migrating the kernel to an object-oriented paradigm in order to reduce coupling between modules and promote cohesiveness within modules. Because efficiency is one of the most

important considerations for an OS kernel, object-oriented features can be added to the kernel in stages using object-oriented wrappers.

2.1.2A COMPREHENSIVE STUDY OF KERNEL (ISSUES AND CONCEPTS) IN DIFFERENT OPERATING SYSTEMS

Various operating systems (OS) with various purposes and features have appeared over time, according to Mohammed A. M. Sadeeq,(2021). As a result, they understand how each OS has been built, which helps users make informed judgments about how to configure the OS on their workstations. As a result, a comparative research of various operating systems is required to provide specifics on the same as well as differences in novel types of operating systems in order to remedy their shortcomings. The visual operating system is the focus of this article, which compares iOS, Android, Mac, Windows, and Linux operating systems, as well as their features, limits, and strengths. Linux, Android, and Windows 10 are operating systems that are more stable, compatible, and deployed. Linux, Android, and Windows have grown in popularity to the point that, unlike other operating systems, they have become more user-friendly and have more application programmes. Mac OS X and Windows 10 both have built-in firewalls. Android and Windows, particularly the novelist versions, are the most popular platforms. It's because they're inexpensive, dependable, compatible, safe, and simple to operate. Furthermore, recent advancements in difficulties arising from growing technology and the rise of the cell phone offered numerous features such as high-speed processors, enormous memory, multitasking, high-resolution screens, functioning telecommunication hardware, and so on.

2.1.3 PERFORMANCE AND SCALABILITY OF LIGHTWEIGHT MULTI-KERNEL BASED OPERATING SYSTEMS

Balazs Gerofi (2018), Multi-kernels take advantage of today's multi-core processors to run two operating system (OS) kernels at the same time, usually a Light Weight Kernel (LWK) and a Linux kernel. High performance and scalability are provided by the LWK, while

compatibility is provided by the Linux kernel. Multi-kernels offer promise in meeting tomorrow's extreme-scale computing needs while maintaining robust isolation, resulting in the high performance and scalability required by traditional HPC workloads. McKernel and mOS began as separate research projects to investigate the aforesaid potential. Their design and architecture advantages were previously described in previous work. The results of executing the two LWKs on a 2,048-node system with Intel Xeon Phi processors (KNL) coupled by Intel Omni-Path Fabric are presented in this study. McKernel, mOS, and Linux are compared in terms of performance.

Despite the fact that the two multi-kernel attempts took different approaches to the problem, the results reveal a median performance improvement of 9%, with certain applications improving by as much as 280 percent, demonstrating the efficacy of the multi-kernel strategy. We analyse the strengths of the two different multi-kernel techniques and provide insight into the performance benefits.

2.1.4 DETECTING STRUCT MEMBER-RELATED MEMORY LEAKS USING ERROR CODE ANALYSIS IN LINUX KERNEL

Memory leaks caused by structural members can become a major issue. The Linux kernel is no different. According to our analysis of Linux Kernel patches, struct member leaks accounted for 54.6 percent of all memory leak-related changes in the last two years. When a struct is freed before its dynamically allocated struct members are freed, this happens. To detect these issues in large-scale software, the analysis cost must be reduced for scalability, and the state of a struct and its members must be properly collected. We provide a simple static-analysis method for detecting struct member-related memory leaks in the Linux kernel in this work. Our investigation begins with a path-insensitive analysis to collect alloc/free data.

We introduce error-code analysis, which is an optimization to efficiently feed back the alloc/free information by focusing on the return value of the callee and its use in the caller, to conduct inter-procedural analysis. When we detect a struct free, we look through the information we've gathered to see if there are any members that haven't been freed yet, and issue warnings to them. We tested our method on the Linux Kernel 5.3-rc4 and

discovered two new problems. Linux Kernel developers reviewed and validated both of the problems.

2.1.5 ANALYSIS OF EMBEDDED LINUX USING KERNEL ANALYSIS SYSTEM

Kiduk Kwon (2009) makes a suggestion. Due to recent advancements in the fields of multimedia and networking, the complexity of embedded system software is rapidly increasing. It's becoming more difficult to pinpoint the source of system problems as a result of these developments. This is especially true when the causes of problems are buried in the kernel layer, making comparisons with the user layer more challenging. One of the reasons for the difficulty is that there is no good method available for developers to investigate kernel problems. In this research, we propose the Kernel Analysis System (KAS), which can detect kernel faults and bugs.

As an example, the KAS discovers timer delay in the kernel layer, which is a severe concern for real-time processing systems. It can efficiently locate the issue and provide assistance in determining the source of the issue. We will try to generalise the system in the future to find other problems and causes to assist the developers

2.1.6 TROUBLESHOOTING HARDWARE PROBLEMS IN LINUX

Daniel Oh (Red Hat) suggests deploying mission-critical business applications across a variety of infrastructures, including physical machines, virtualization, private cloud, public cloud, and hybrid cloud. Linux system administrators must know how to manage Linux hardware infrastructure, including software-defined networking, storage, Linux containers, and a variety of tools on Linux servers. On Linux, troubleshooting and resolving hardware-related issues can take some time. Even the most seasoned sysadmins can spend hours troubleshooting unexplained hardware and software issues. The following pointers should make troubleshooting hardware in Linux more faster and easier. Many different factors might cause difficulties with Linux hardware; before attempting to diagnose them, it's a good idea to familiarise yourself with the most prevalent issues and where you're most likely to encounter them.

CHAPTER 3

3.0 AIM AND SCOPE & SOFTWARE TOOLS

3.1 PROJECT AIM

The Linux Diagnostic Tools project aims to develop better diagnostic tools for Linux systems. We're concerned with more than just running diagnostic tests when it comes to diagnosing systems; we're also concerned with capturing first failure data, error log analysis, preventative testing, and system inventory gathering. Analyze the code coverage. Debugfs, problem reproduction, dumps, kgdb, memory concerns, tracing, and other tools are used to diagnose and troubleshoot problems. Packet sniffing, latency measurement, data analysis, and other services are available.

3.2 PROJECT SCOPE

- Reduced man hours in debugging.
- Easily understanding of code flow.
- Verifying the positive and negative code path.
- Capture additional debug information during error scenario.

3.3 VIRTUAL BOX

VirtualBox is free software that allows you to virtualize the x86 computer architecture. It functions as a hypervisor, allowing the user to run another operating system on a virtual machine (VM) (operating system). The "host" OS is the operating system on which VirtualBox runs. The "guest" OS is the operating system that runs in the virtual machine. VirtualBox can run on Windows, Linux, or macOS as its host operating system.

When designing a virtual machine, the user can define how many CPU cores the VM should have, as well as how much RAM and disc space it should have. The VM can be "paused" while it is operating. At that point, the system's execution is paused, and the user can resume utilising it later.

VIRTUAL MACHINE

A virtual machine (VM) is a computer within a computer that operates in a virtual environment. It runs on its host computer's isolated partition, which has its own CPU, memory, operating system (such as Windows, Linux, or macOS), and other resources. End

users can run applications on virtual machines and use them just like they would on their own computers.

VIRTUAL MACHINES WORK

The technique of virtualization allows for the creation of virtual machines. Virtualization is a method of running many virtual machines on a single machine by using software to emulate virtual hardware. The real machine is referred to as the host, while the virtual machines that run on it are referred to as guests. A hypervisor is a piece of software that oversees this procedure. The hypervisor is in charge of managing and provisioning resources from the host to the guests, such as memory and storage. It also schedules operations in virtual machines so that they don't overlap when using resources. Virtual machines (vms) can only function if a hypervisor is used to virtualize and distribute host resources. In virtualization, there are two types of hypervisors.

HOST AND TARGET MACHINES

Host

- A computer system on which all the programming tools run
- Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device.

Target:

- After writing the program, compiled, assembled and linked, it is moved to target
- After development, the code is cross-compiled, translated – cross-assembled, linked into target processor instruction set and located into the target.

3.4 UBUNTU 18.04

Ubuntu is a full Linux operating system that is accessible for free and has both community and professional support. The Ubuntu community is based on the Ubuntu Manifesto's ideas that software should be available for free, that software tools should be accessible to people

in their native language and regardless of disabilities, and that people should be able to customise and alter their software in any way they see fit. Ubuntu will always be free, and there will be no additional charge for the "business edition," as we make our greatest work available to everyone on the same terms. Ubuntu offers the best translations and accessibility infrastructure the Free Software community has to offer, ensuring that Ubuntu is accessible to everyone.

Ubuntu is distributed in reliable and predictable release cycles, with a new version every six months. Every two even years, Ubuntu will issue a long-term support (LTS) release, which will be supported for five years. The interim Ubuntu releases (known as development or non-LTS releases) are each supported for nine months. Ubuntu is completely dedicated to the ideas of open source software development; we encourage users to use, enhance, and share open source software. Ubuntu is appropriate for usage on both the desktop and the server. Intel x86 (IBM-compatible PC), AMD64 (x86-64), ARMv7, ARMv8 (ARM64), IBM POWER8/POWER9 (ppc64el), IBM Z zEC12/zEC13/z14, and IBM LinuxONE Rockhopper I+II/Emporer I+II are all supported in the current Ubuntu edition (s390x).

Starting with the Linux kernel version 5.4 and GNOME 3.28, Ubuntu offers thousands of pieces of software that cover everything from word processing and spreadsheets to internet access, web server software, email software, programming languages and tools, and, of course, various games.

3.5 VIRTUALBOX EXTENSION PACK

VirtualBox Extension Pack is a binary package that extends VirtualBox's capability. The following features are included in the Extension pack:

USB 2.0 and USB 3.0 devices are supported. USB devices inserted into physical USB ports of a host machine, such as network adapters, flash drives, hard discs, webcams, and so on, can be joined to a virtual machine operating on VirtualBox. As a result, a physical USB device can be used in a guest operating system (OS). The VirtualBox Extension Pack is a package that extends VirtualBox's fundamental capabilities. Allows you to use physical USB devices in a virtual environment. Even if drivers aren't available, use the host webcam on a guest operating system. VirtualBox allows you to connect to a remote VM.

3.6 GCOV

gcov is a coverage test programme. Use it in conjunction with GCC to analyse your programmes in order to assist you write more efficient, faster-running code and uncover areas of your programme that haven't been tested. Gcov can be used as a profiling tool to see where your optimization efforts will have the most impact on your code. You can also use gcov in conjunction with another profiling tool, gprof, to determine which areas of your code consume the most computing time.

Profiling tools assist you in analysing the performance of your code. You can get some basic performance information using a profiler like gcov or gprof, such as:

gcov is a coverage test programme. Use it in conjunction with GCC to analyse your programmes in order to assist you write more efficient, faster-running code and uncover areas of your programme that haven't been tested. Gcov can be used as a profiling tool to see where your optimization efforts will have the most impact on your code. You can also use gcov in conjunction with another profiling tool, gprof, to determine which areas of your code consume the most computing time.

Profiling tools assist you in analysing the performance of your code. You can get some basic performance information using a profiler like gcov or gprof, such as:

1. what lines of code are actually executed
2. how much computing time each section of code uses

Once you understand how your code is compiled, you may examine each module to see which ones should be optimised. gcov assists you in determining where you should focus your optimization efforts. Coverage testing is also used by software developers in conjunction with test suites to ensure that product is ready for release. A coverage programme checks to see how much of the programme is exercised by the testsuite. Testsuites can validate that a programme operates as expected. Developers can then choose which kind of test cases should be added to the testsuites in order to improve both testing and the final product.

If you plan to use gcov, you should compile your code without optimization since optimization, which combines several lines of code into one function, may not provide you with as much information as you need to seek for 'hot spots' where the code is consuming a lot of computer time. Similarly, because gcov collects statistics every line (at the lowest

resolution), it works best with a programming style in which each line contains only one statement. The statistics are less useful if you employ complex macros that expand to loops or other control structures, because they only report on the line where the macro call appears. If your complex macros behave like functions, you can overcome this problem by replacing them with inline functions.

gcov generates a sourcefile logfile. gcov displays the number of times each line of sourcefile.c has been executed. These logfiles, combined with gprof, can be used to fine-tune the performance of your programmes. You can utilise the timing information from gprof in conjunction with the information from gcov. Gcov only works with GCC-compiled code. Any other profiling or test coverage approach is incompatible with it.

3.7 LCOV

LCOV is an extension of GCOV, a GNU tool that shows which parts of a programme are actually executed (i.e. "covered") during the execution of a test case. The extension consists of a series of Perl scripts that enhance the textual GCOV output by adding the following features:

Output in HTML format: Bar graphs with specific colours are also used to show coverage rates.

- Support for large projects: overview pages allow quick browsing of coverage data by providing three levels of detail: directory view, file view and source code view.

LCOV was created to support Linux kernel coverage measures, but it may also be used to measure coverage in normal user space applications.

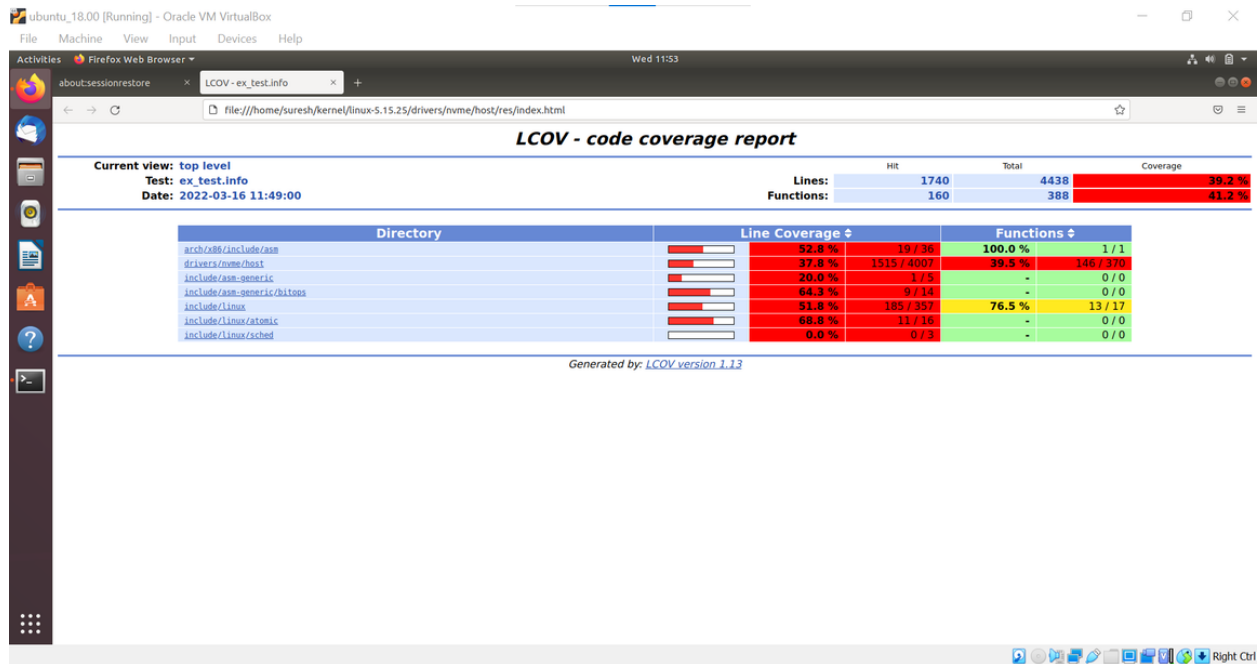


Fig.no.3.1 lcov output

DIFFERENCE BETWEEN LCOV AND GCOV

LCOV	GCOV
• LCOV is a graphical front-end for gcov. It collects gcov data for multiple source files and creates HTML pages containing the source code annotated with coverage information. It also adds overview pages for easy navigation within the file structure. LCOV supports statement, function and branch coverage measurement. • HTML based output: coverage rates are additionally indicate using bar graphs and specific colors. • Support for large projects: overview pages allow quick browsing of coverage data `by providing three levels of detail: directory view,file view and source code view	• gcov is a test coverage program. Use it in concert with GCC to analyze your programs to help create more efficient, faster running code and to discover untested parts of your program. • how often each line of code executes • what lines of code are actually executed • how much computing time each section of code uses

TABLE 3.1 DIFFERENCE BETWEEN LCOV AND GCOV

3.8 KGDB

Kgdb is designed to be used as a Linux kernel source level debugger. To debug a Linux kernel, it is used in conjunction with gdb. Gdb is expected to be able to "break in" to the kernel and inspect memory, variables, and call stack information in the same way that an application developer would use gdb to debug an application. Breakpoints can be set in kernel code, and limited execution stepping can be done.

The use of kgdb necessitates the use of two machines. The development machine is one of them, while the target machine is the other. On the target machine, the kernel to be debugged is running.

Gdb is run on the development machine against the vmlinux file, which contains the symbols (not a boot image such as bzImage, zImage, ulImage...). The developer connects to kgdb using gdb and specifies the connection parameters. The availability of kgdb I/O modules compiled as built-ins or loadable kernel modules in the test machine's kernel determines the type of connection a developer makes with gdb.



FIG.NO.3.2 BLOCK DIAGRAM FOR KGDB

3.9 KDB

Kdb is a simple shell-style interface that you may use with a keyboard or a serial console on a system console. You can use it to look at memory, registers, process lists, and dmesg, as well as set breakpoints to stop at specific moments. Although you can create breakpoints and do some rudimentary kernel run control with Kdb, it is not a source level debugger. Kdb is mostly used to perform analysis in order to aid in the development of kernels or to

diagnose kernel faults. If the code was created with CONFIG KALLSYMS, you can access some symbols in kernel built-ins or kernel modules by name.

DIFFERENCE BETWEEN KDB AND KGDB

Kdb	Kgdb
• kdb can debug on the target machine directly. • The Kernel DeBugger. A simple shell that can do simple peeks/pokes but also has commands that can print kernel state at time of crash.	• kgdb requires two machines that are connected via a serial(or network) connection. • kgdb debug client is gdb, which supports C source code level debugging, and also recognizes the kernel data structure. • The Kernel GDB server. Allows a second computer to run GDB and debug the kernel.

TABLE 3.2 DIFFERENCE BETWEEN KGDB AND KDB

3.10 DEBUGFS

Not to be confused with the debugfs filesystem utility, debugfs is a simple memory-based filesystem developed solely to debug Linux kernel code. Greg Kroah-Hartman introduced debugfs in December 2004 to assist kernel engineers in exporting huge volumes of debug data into user space. This page includes sample code as well as an introduction to debugfs and its application programming interface. Kernel and device driver developers with some experience in Linux kernel development are the target audience. It can also be used to refresh the abilities of the more experienced.

Kernel developers can utilise debugfs to export significant volumes of debug data into user space. Unlike /proc and /sysfs, it has no restrictions on the type of data that can be entered, therefore it's ideal for debugging.

When reading values, `printk` is sufficient; however, it does not allow developers to change the values of variables in user space.

Include `linux/debugfs.h` in your source file to use the API. To use `debugfs`, we must first create a directory in `/sys/kernel/debug`, which is a good place to start. This directory can be used to store the rest of the files.

```
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);
```

The directory name is here, and the parent directory is here (if null, the directory is created in `/sys/kernel/debug`). `ENODEV` is returned if `debugfs` is not enabled in the kernel.

If you need to create a single file within `debugfs`, you can call the following function:

```
struct dentry *debugfs_create_file(const char *name, mode_t mode, struct dentry *parent, void *data, struct file_operations *fops);
```

Name is the name of the file to be created; mode is the file's permissions; parent is the parent directory in which the file should be created (if null, it defaults to the `debugfs` root, `/sys/kernel/debug`); data is the type `inode.i_private`; and fops is the file operations

If you need to write to and read from a single value, you can use this to create an unsigned 8-bit value:

```
struct dentry *debugfs_create_u8(const char *name, mode_t mode, struct dentry *parent, u8 *value);
```

Here, `value` is a pointer to a variable that needs to be read and written to.

A few other helper functions to create files with single integer values are:

- `struct dentry *debugfs_create_u16`
- `struct dentry *debugfs_create_u32`
- `struct dentry *debugfs_create_u64`

(Refer to `fs/debugfs/file.c` for more information.)

Similar functions that give hex output, are:

```
dentry *debugfs_create_x8(const char *name, mode_t mode, struct dentry *parent, u8 *value)
```

```
dentry *debugfs_create_x16
```

```
dentry *debugfs_create_x32
```

```
dentry *debugfs_create_x64
```

Note: `debugfs_create_x64` is the most recent edition of the API; added in May 2010 by Huang Ying, this tells us that `debugfs` is still in active development.

3.11 NVME

Nvme stands for "non-volatile memory - express" and is a sort of software interface for solid state drives (ssd) (non-volatile memory is a common form of flash memory found in ssd). The non-volatile memory host controller interface specification is the official name for nvme. Nvme is modified for use with ssd over a pcie hardware interface, resulting in a substantially quicker disc interface protocol than the previous ahci (advanced host controller interface) used for classic mechanical spinning sata hard discs.

- NVMe (Non-Volatile Memory Express) is a communications interface and driver that defines a command set and feature set for PCIe-based SSDs with the goals of increased and efficient performance and interoperability on a broad range of enterprise and client systems.
- NVMe was designed for SSD. It communicates between the storage interface and the System CPU using high-speed PCIe sockets, independent of storage form factor. Input/Output tasks performed using NVMe drivers begin faster, transfer more data, and finish faster than older storage models using older drivers, such as AHCI (Advanced Host Controller Interface) a feature of SATA SSDs. Because it was designed specifically for SSDs, NVMe is becoming the new industry standard for both servers in the data centre and in client devices like laptop, desktop PCs and even next gen gaming consoles.
- NVMe technology is available in a number of form factors such as the PCIe card slot, M.2, and U.2. While there are SSDs that use the SATA, PCIe slot and M.2 that are AHCI and not NVMe, U.2 is a form factor that exclusively uses the NVMe protocol.

NVME-CLI:

Nvm express (nvme) technology has enabled the development of a robust set of industry-standard storage software, drivers, and management tools. Nvme command line interface is a linux tool for managing nvme ssds (nvme-cli).

Many administration operations are required in data centres to monitor ssd health, endurance, firmware updates, securely wipe storage, and read various logs. Nvme-cli is a feature set that is open source and follows the nvme specification, and it is supported by all major distributions. It supports nvme ssds and nvme over fabrics (nvme-of) architecture, as well as optional vendor plugins that provide additional information beyond the specification. You may discover why ssds fail, as well as why nvme monitoring, management, error reporting, and logging are so important.

CHAPTER 4

KERNEL AND DEVICE DRIVERS

4.1 USER SPACE :

The user space and the kernel space are the two different areas of RAM. User Space is a collection of areas where typical user processes are executed. These processes do not have direct access to kernel space. System calls can access a portion of the kernel space. In kernel space, these system calls serve as software interrupts.

User Mode

The executing code in User mode is unable to access hardware or reference kernel memory directly. To access hardware or kernel memory, user mode code must delegate to system APIs. The majority of the code on your computer will operate in user mode.

One of the following factors frequently causes the move from user space to kernel space:

Software interrupt

Fault/Exception in Software Interrupt (e.g. a page fault, divide-by-zero, floating-point exception or some other exception caused during the execution of an instruction)

Interrupt due to hardware

It is raised anytime a piece of hardware requires the CPU's attention (for example, a keyboard interrupt or I/O completion).

system Calls

A context switch from user to kernel space is also caused via system calls (which include accessing devices).

signals

Signals are normally started by a process (kill()) or the operating system kernel (e.g: SIGFPE, SIGSEGV, SIGIO).

The OS kernel manages them and sends them to the target thread/process, invoking either a generic action (ignore, terminate, terminate and dump core) or a signal handler specified by the process.

CPU Instructions

Normally, the system checks the Interrupt Descriptor Table or Interrupt Vector Table whenever one of the aforementioned conditions happens. Every exception (interrupt, fault, etc.) has a unique number that is used to index into this table. The CPU can use this table to determine which interrupt handler to use. This table provides the location of the Interrupt handler routine, which executes all of the necessary procedures to convert the user application to kernel mode and begin executing kernel instructions on the user process's behalf.

INTERRUPT VECTOR TABLE (IVT):

In a table of interrupt vectors, an interrupt vector table is a data structure that associates a list of interrupt handlers with a list of interrupt requests. The address of an interrupt handler is stored in each item of the interrupt vector table, which is referred to as an interrupt vector. The following adjustments are often implemented as part of the transition:

1. Activate the Kernel Stack
2. EFLAGS, Code Segment Selector, and EIP are all saved.
3. Start running the interrupt handler after saving the stack segment selector and stack pointer.
4. The general purpose registers are saved (this is the responsibility of the handler) and the segment selectors are transformed to Kernel selectors (this is the task of the handler).

4.2 KERNEL SPACE

The executing code has complete and unrestricted access to the underlying hardware under Kernel mode. Any CPU instruction can be executed, and any valid memory address can be referenced. Kernel mode is where any operating system function below the system call layer runs.

Difference Between Kernel mode and User mode:

Criteria	Kernel Mode	User Mode
Kernel-mode vs User mode	In kernel mode, the program has direct and unrestricted access to system resources.	In user mode, the application program executes and starts out.
Interruptions	In Kernel mode, the whole operating system might go down if an interrupt occurs	In user mode, a single process fails if an interrupt occurs.
Modes	Kernel mode is also known as the master mode, privileged mode, or system mode.	User mode is also known as the unprivileged mode, restricted mode, or slave mode.
Virtual address space	In kernel mode, all processes share a single virtual address space.	In user mode, all processes get separate virtual address space.
Level of privilege	In kernel mode, the applications have more privileges as compared to user mode.	While in user mode the applications have fewer privileges.
Restrictions	As kernel mode can access both the user programs as well as the kernel programs there are no restrictions.	While user mode needs to access kernel programs as it cannot directly access them.
Mode bit value	The mode bit of kernel-mode is 0.	While; the mode bit of user-mode is 1.

Table 4.1 Difference between kernel mode and user mode

System Calls

System calls serve as a link between a process and the operating system. System calls are often available as assembly language instructions. They are also included in the assembly level programmers' manuals.

When a user mode process requires access to a resource, system calls are often made. Then, via a system call, it asks the kernel to provide the resource.

The Different Types of System Calls

System calls are divided into five categories

Process Control

These system calls handle things like process creation, process termination, and so on.

File Management

File manipulation is handled by these system calls, which include creating, reading, and writing to files.

Device Management

These system calls handle device manipulation like reading from device buffers, writing to device buffers, and so on.

Information Maintenance

These system calls control the flow of data between the operating system and the user programme.

Communication

Interprocess communication is made possible by these system calls. They're also in charge of setting up and disabling communication connections.

The following are some instances of all of the various sorts of system calls in Windows and Unix:

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Table 4.2 System calls

There are many different system calls as shown above. Details of some of those system calls are as follows –

Wait()

In some systems, a process may wait for another process to finish its execution before continuing. When a parent process spawns a child process, the parent process's execution is delayed until the child process completes. The parent process is suspended using the wait() system function. Control is returned to the parent process once the child process has completed its execution.

exec()

This system call executes an executable file in the context of another process that is already running. It takes the place of the preceding executable. This is referred to as an overlay. Although no new process is formed, the existing process's data, heap, stack, and other attributes are replaced by the new process.

fork()

Processes use the fork() system call to generate copies of themselves. This is one of the most common ways to create processes in operating systems. When a parent process spawns a child process, the parent process's execution is halted until the child process completes. Control is returned to the parent process once the child process has completed its execution.

exit()

A program's execution is terminated using the exit() system call. This indicates that the thread execution is complete in a multithreaded context. Following the exit() system function, the operating system reclaims resources that were previously utilised by the process.

kill()

The operating system uses the kill() system function to send a termination signal to a process, urging it to depart. However, a kill system call does not always imply that the process is terminated, and it can have a variety of implications.

4.2.1 PROCESS MANAGEMENT

Process management entails the completion of numerous activities such as process creation, process scheduling, deadlock management, and process termination. It is the role of the operating system to handle all of the system's ongoing processes. Processes are managed by the operating system, which performs duties like as resource allocation and process scheduling. When a process runs on a computer, the machine's RAM and CPU are used. The operating system is also responsible for synchronising the various computer processes.

A process is made up of a collection of instructions called process code that must be followed. A procedure is also linked to some data that needs to be analysed. Process components are the resources that a process necessitates for its implementation. There is also a state called process state that is associated with a process at a specific point in time. There are a number of notions related with an operating system's process management function that are similar to these concepts. The following are some of such concepts.

1. Process State

2. Process Control Block (PCB)
3. Process Operations
4. Process Scheduling
5. Process Synchronization
6. Interprocess Communication
7. Deadlock

The status of the process

A process state is defined as the status of a process at a specific point in time. The current state of a process is defined by its state. This allows us to gather more information about our process at a specific point in time. In an operating system, there are seven primary states of process. The seven stages of the process are as follows:

1. New

A new process is established when a programme is called from secondary memory or from the hard disc to primary memory or RAM. In a nutshell, it specifies when a process is started.

2. Ready

The state of the process is loaded into primary memory and ready for execution during this time frame.

3. Waiting

The process is put on pause in this state, and other processes are allowed to begin their execution. In other words, it specifies the time interval during which a process waits for CPU time and other resources to be allocated for its execution.

5. Executing

This is the primary state of a process, and it is in this state that the process is running. In other words, it is the period of time during which a process is run by the CPU.

6. Blocked

It indicates the time interval during which a process is waiting for an event to occur, such as input/output operations.

7. Suspended

It describes the moment when a process is ready to run but has not yet been placed in the operating system's ready queue.

8. Terminated

It indicates the moment when a process is terminated or finished, and all of the resources and memory used by the process are released.

When the process is first created, it is in the new state, as shown in the diagram above. The state of the process is changed to ready when it is formed, and the process is loaded into primary memory or RAM. When the process is waiting for CPU time to be allocated, the state of the process is changed to waiting. The state of the process changes to executing after CPU time and other resources are allotted to it, and the process begins to operate. After the process has successfully ran or executed, the process is ended or terminated, and the process state is changed to terminate.

4.2.2 MEMORY MANAGEMENT

Virtual Memory

One of the most powerful components of memory management is the concept of virtual memory. Since the dawn of computing, there has been a desire for more memory than is currently available in physical memory. Many solutions have been tried throughout the years to solve this problem, the most effective of which has been the concept of virtual memory. Virtual memory gives the impression that your computer has more memory than it actually has. This may sound intriguing, and one might wonder how this is feasible. So, let's get a better understanding of the notion.

To begin, we must first grasp the concept of virtual memory, which is a layer of memory locations that correspond to physical addresses. When a processor executes a programme instruction in the virtual memory model, it reads the instruction from virtual memory and executes it. However, it turns the virtual memory address into a physical address before executing the command. This conversion is dependent on the mapping of virtual to physical addresses, which is done using the mapping information from the page tables (that are maintained by OS).

The virtual and physical memory is separated into pages, which are fixed length blocks of memory.

A virtual address can be broken into two components in this paged model:

1. A virtual page frame number (lowest 12 bits) and an offset (rest of the bits)
2. When a virtual address is encountered, the processor takes the virtual page frame number from it. The offset components allow it travel to the exact address in the physical page by translating the virtual page frame number into a physical page frame number. The page tables are used to accomplish the address translation.

The following information can theoretically be found in a page table:

1. A flag that indicates whether or not the entry is valid.
2. This entry's description of the physical page frame number
3. Obtain information about the page (like read-only, read-write etc)
4. A page table is accessible by utilising the virtual page frame number as an offset for page table entries. A virtual page frame number of '2', for example, links to the page table entry '1' (entry numbers start with '0').
5. VPFN stands for Virtual page frame number, and PFN stands for Physical page frame number in the graphic below. It's possible that a processor will find an invalid item in a processes page table entry with a virtual page frame number. It is the processor's obligation to transmit control to the kernel and ask it to solve the problem in this scenario.

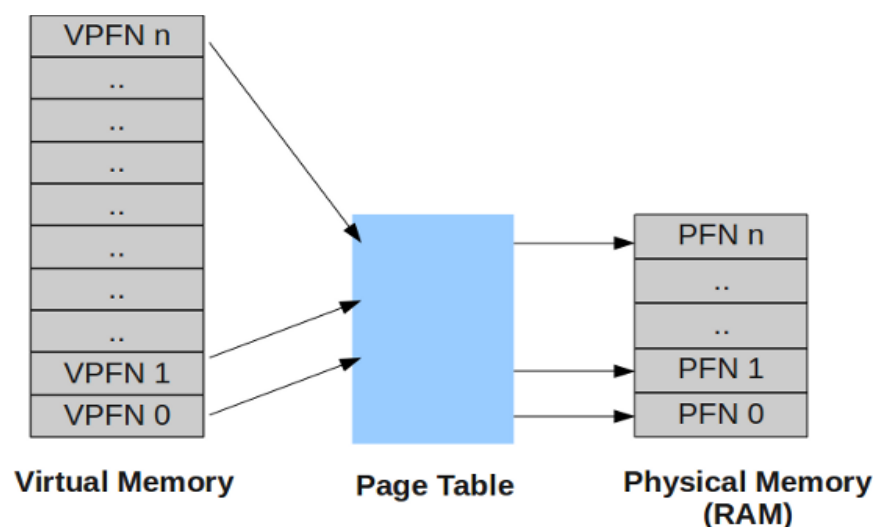


Fig.no.4.1 Mapping of virtual memory

The control is passed in various ways by different processors, but this situation is known as a 'page fault.' If the input was valid, the processor multiplies the physical page frame number by the page size to get the physical page's base address, then adds the offset to get the exact physical address. So now we know that each process believes it has access to the entire range of virtual addresses available to it, and that this concept makes the system appear to have more physical memory than it actually does.

4.2.3 NETWORKING UNIT

We've gone over char and block drivers and are now ready to move on to networking. The third common class of Linux devices is network interfaces, and this chapter discusses how they interact with the rest of the kernel. A network interface has a similar role in the system as a mounted block device. A block device registers its discs and methods with the kernel, and then uses its request function to "transmit" and "receive" blocks on demand. Similarly, in order to be invoked when packets are exchanged with the outside world, a network interface must register itself within appropriate kernel data structures.

Between mounted discs and packet-delivery interfaces, there are a few key distinctions. For starters, a disc is represented by a particular file in the /dev directory, whereas a network interface is not. Because standard file operations (read, write, and so on) make no sense when applied to network interfaces, the Unix "everything is a file" approach cannot be applied to them. As a result, network interfaces have their own namespace and export a unique set of operations. Although you may complain to apps using read and write system calls when using sockets, such operations operate on a separate software entity from the interface. On the same physical interface, many hundred sockets can be multiplexed.

The most significant distinction between the two is that block drivers only respond to kernel requests, whereas network drivers receive packets asynchronously from the outside. As a result, while a block driver is asked to deliver a buffer to the kernel, the network device requests that incoming packets be pushed to the kernel. This alternative manner of operation is integrated into the kernel interface for network drivers. A number of administration functions, such as assigning addresses, altering transmission parameters, and maintaining traffic and error data, must be supported by network drivers. The network

driver API reflects this necessity and, as a result, differs from the other interfaces we've seen thus far.

The Linux kernel's network subsystem has been built to be protocol-agnostic. This holds true for both networking and hardware protocols (Internet protocol [IP] vs. IPX or other protocols) (Ethernet versus token ring, etc.). The interaction between a network driver and the kernel is designed to handle one network packet at a time, allowing protocol concerns to be hidden from the driver while the physical transmission is hidden from the protocol.

This chapter explains how network interfaces interact with the rest of the Linux kernel and includes demonstrations in the form of `snull`, a memory-based modularized network interface. To keep things simple, the interface broadcasts IP packets using the Ethernet hardware protocol. The information you gain from looking at `snull` can be extended to protocols other than IP with ease, and developing a non-Ethernet driver differs only in minor specifics pertaining to the network protocol.

IP numbering schemes, network protocols, and other broad networking concepts are not covered in this chapter. Such issues are not (generally) of interest to the driver author, and a comprehensive review of networking technologies in less than a few hundred pages is unfeasible. Other books on networking issues are recommended for interested readers. Before we go into network devices, a word about terminology is in order. The term `octet` is used in the networking field to refer to a group of eight bits, which is the lowest unit that networking devices and protocols understand. In this context, the term `byte` is nearly never used. When discussing networking equipment, we will use the term `octet` in accordance with industry standards.

The term "header" is also worth mentioning. A header is a set of bytes (err, octets) appended to a packet as it travels through the networking subsystem's various tiers. When an application provides a block of data via a TCP socket, the networking subsystem divides the data into packets and places a TCP header at the start of the stream, defining where each packet belongs. The lower levels then place an IP header in front of the TCP header, which is used to route the packet to its destination. If the packet travels over an Ethernet-like medium, an Ethernet header is placed in front of the rest, which is interpreted by the hardware. Higher-level headers are normally not an issue for network drivers, but they are frequently involved in the construction of the hardware-level header.

4.2.4 VIRTUAL FILE SYSTEM

The Virtual File System (also known as the Virtual Filesystem Switch) is a kernel software layer that provides userspace programmes with a filesystem interface. It also provides a layer of abstraction within the kernel that allows for the coexistence of several filesystem implementations.

Open, stat, read, write, chmod, and other VFS system calls are called from a process context. The document Locking describes how to lock the file system.

Directory Entry Cache (dcache)

The VFS implements system calls such as open, stat, chmod, and others. The VFS searches the directory entry cache using the pathname argument that is supplied to them (also known as the dentry cache or dcache). This gives a very quick way to convert a pathname (filename) into a specified dentry. Dentries exist just to provide performance and are never saved to disc.

The dentry cache is designed to provide you a bird's-eye view of your whole filesystem. Some cache bits are missing since most computers can't fit all dentries in RAM at the same time. The VFS may have to create dentries along the way before loading the inode in order to resolve your pathname into a dentry. This is accomplished by checking up the inode number.

The Inode Object

A pointer to an inode is normally found on each individual dentry. Filesystem objects such as normal files, directories, FIFOs, and other creatures are called inodes. They can be found on the disc (for block device filesystems) or in memory (for memory filesystems) (for pseudo filesystems). When needed, inodes on the disc are copied into memory, and modifications to the inode are written back to the disc. Multiple dentries can point to a single inode (hard links, for example, do this).

The VFS must use the lookup() method of the parent directory inode to look up an inode. The filesystem implementation in which the inode resides installs this technique. We can do all those tedious things like open(2) the file or stat(2) it to peep at the inode data once the VFS has the proper dentry (and hence the inode). The stat(2) operation is straightforward:

once the VFS obtains the dentry, it examines the inode data and returns some of it to userspace.

The File Object

Another operation is required to open a file: the allocation of a file structure (this is the kernel-side implementation of file descriptors). A pointer to the dentry and a set of file operation member functions are initialised in the newly allocated file structure. These are derived from the inode information. The `open()` file method is then invoked, allowing the individual filesystem implementation to do its duties. As you can see, the VFS does yet another switch. For the process, the file structure is stored in the file descriptor table. Reading, writing, and closing files (along with other VFS actions) are accomplished by grabbing the proper file structure using the userspace file descriptor, and then calling the needed file structure method to complete the task. The dentry is kept in use for as long as the file is open, which means the VFS inode is still in use.

4.2.5 INTERPROCESS COMMUNICATION

In UNIX systems, "Inter-process Communication (IPC)" refers to the technique by which User Mode processes synchronise and communicate data (that includes Linux too). Semaphores, Shared Memory, and Message Queues are "Inter-process Communication Resources" or "Inter-process Communication Facilities," and differ from "Inter-process Communication Mechanisms" like Pipes and FIFOs in the manner they express IPC. System V (AT&T System V.2 release of UNIX) IPC facilities include semaphores, shared memory, and message queues, which are wrapper functions that have been built and incorporated into appropriate libraries to harness the energy and beauty of IPC methods. Data sharing between processes can be achieved by storing data in temporary files that are password-protected. However, because it necessitates access to the disc filesystem, this approach is never implemented. As a result, all UNIX kernels provide a set of system calls that allow processes to communicate without having to deal with the filesystem.

4.2.6 DEVICE DRIVERS

One of the numerous benefits of free operating systems, such as Linux, is that their internals are transparent to anyone. The operating system, which was formerly a dark and mysterious realm whose code was only accessible to a select few programmers, can now be studied, understood, and modified by anybody with the necessary abilities. Linux has contributed to the democratisation of operating systems. However, the Linux kernel is still a vast and complex body of code, and would-be kernel hackers require an entry point that allows them to approach the code without being overwhelmed by it. Device drivers are frequently used as a gateway. In the Linux kernel, device drivers play a unique role. They are unique "black boxes" that enable a piece of hardware to respond to a well-defined internal programming interface while concealing all of the device's internal workings. User activities are carried out using a set of standardised calls that are independent of the individual driver; the device driver's responsibility is then to map those calls to device-specific operations that act on real hardware. Drivers can be developed separately from the rest of the kernel and "plugged in" at runtime as needed using this programming interface. Because of this modularity, writing Linux drivers is simple, and there are currently hundreds of them available. There are several reasons why you might be interested in writing Linux device drivers. Driver writers will be kept busy for the foreseeable future due to the rate at which new hardware becomes available (and obsolete!). Individuals may require knowledge about drivers in order to obtain access to a device that they are interested in. By making a Linux driver available for their goods, hardware makers can expand their potential market to include the enormous and expanding Linux user base. Furthermore, because the Linux system is open source, a driver's source code can be easily spread to millions of users if the driver author so desires. This book teaches you how to develop your own drivers and how to tinker with kernel components. We've taken a device-agnostic approach, which means that the programming concepts and interfaces are taught without being tied to any specific device whenever possible. Each device driver is unique, so as a driver writer, you must be aware of this. Know your device inside and out. However, most of the fundamental principles and strategies apply to all drivers. This book will not teach you how to use your gadget, but it will provide you with the background information you'll need to get started. You learn a lot about the Linux kernel as you learn to create drivers, which can help you understand how your computer works and why things

aren't always as quick as you think or don't perform exactly what you want. We gradually introduce new concepts, beginning with very simple drivers and building on them; each new notion is accompanied by sample code that can be examined without the use of specific hardware. This chapter does not go into how to write code. However, before we get into programming, we'll go over some background information on the Linux kernel that you'll be glad you know.

4.2.6.1 CHARACTER DRIVER

This chapter's purpose is to create a comprehensive char device driver. Because this class is suited for most simple hardware devices, we design a character driver. In addition, char drivers are simpler to comprehend than block or network drivers (which we get to in later chapters). Our ultimate goal is to create a modularized char driver, but we won't get into that in this chapter. We present code fragments retrieved from a real device driver: scull, throughout the chapter (Simple Character Utility for Loading Localities). Scull is a character driver that treats a memory space as a device. Because of scull's uniqueness, we use the terms device and "the memory space used by scull" interchangeably in this chapter. Scull has the advantage of not being hardware dependant. Scull only affects RAM that has been allocated by the kernel. Scull may be compiled and executed by anyone, and it is portable across all of the computer architectures that Linux supports. The device, on the other hand, does nothing "useful" except show the interface between the kernel and the char drivers and allow the user to run some tests.

NVME CONTROLLER

During the host connection process, an NVMe controller is built. It establishes a connection between a host and the storage array's namespaces. Queue for NVMe drives. Over the NVMe interface, commands and messages are passed over a queue.

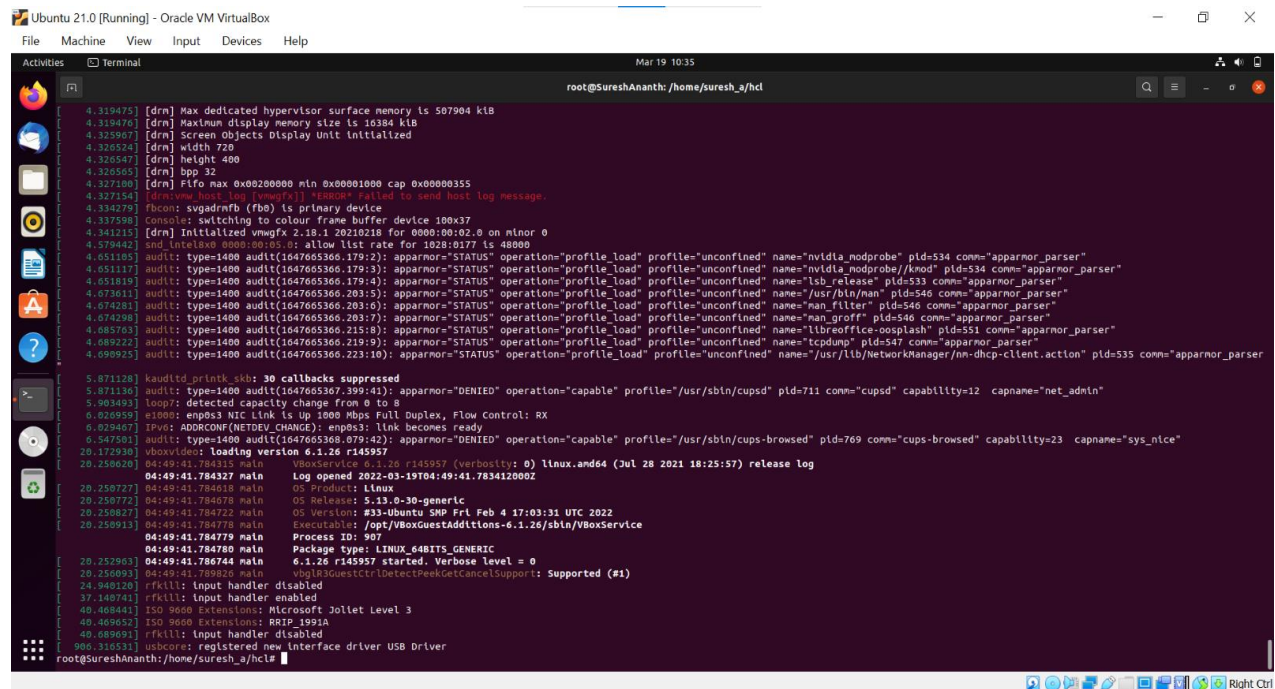
NVM Express (NVMe), also known as the Non-Volatile Memory Host Controller Interface Specification (NVMHCIS), is an open logical-device interface specification for accessing non-volatile storage media connected to a computer via the PCI Express (PCIe) bus.

Multiple namespaces that are accessed using NSID may be supported by an NVMe controller. The Base Specification defines three globally unique namespace identifiers: EUI64 (8 bytes), NGUID (16 bytes), and UUID (128 bytes). The Namespace Management and Namespace Attachment Commands are used to create and delete namespaces.

4.2.6.2 BLOCK DRIVERS

A block driver gives you access to devices that send random data in fixed-size blocks, such as disc drives. Because the Linux kernel considers block devices to be fundamentally different from char devices, block drivers have their own interface and set of issues. Not only for explicit reads and writes in user applications, but also for overall performance, efficient block drivers are essential. Virtual memory in modern systems works by transferring (ideally) unnecessary data to secondary storage, which is commonly a disc drive. Because block drivers serve as a link between core memory and secondary storage, they are considered part of the virtual memory subsystem. While it is possible to construct a block driver without understanding struct page or other fundamental memory concepts, anyone who needs to write a high-performance driver should refer to Chapter 15. The block layer's design is heavily focused on performance. Many char devices can run at a lower pace than their maximum, with little effect on the system's overall performance. However, if the system's block I/O subsystem isn't well-tuned, it won't run well. The Linux block driver interface allows you to get the most out of a block device, but it does so at the expense of some complexity. Fortunately, the 2.6 block The user interface has vastly improved over previous kernels. As one might imagine, the focus of this chapter is on an example driver that implements a block-oriented, memory-based device. In essence, it's a ram disc. Although the kernel already has a far better ram disc implementation, our driver (named s bull) allows us to demonstrate how to create a block driver while limiting unnecessary complexity. , ch16.28124 464th page 9:09 a.m. on Friday, January 21, 2005 e Matter Edition is the title of the book. 2005 O'Reilly & Associates, Inc. Copyright All rights reserved, Inc. 465 | Registration Let's define a few words first before diving into the details. A block is a data chunk of a specific size that is determined by the kernel. Blocks are typically 4096 bytes in size, but this might vary depending on the architecture and file system in use. A

sector, on the other hand, is a small block of data whose size is normally specified by the hardware. The kernel anticipates encountering devices that use 512-byte sectors. If your device utilises a different size, the kernel adjusts to prevent I/O requests that the hardware can't handle. It's important to remember that whenever the kernel gives you a sector number, it's working in a world of 512-byte sectors. You must scale the kernel's sector numbers if you are utilising a different hardware sector size. In the sbull driver, we can see how this is done.



```

4.319475 [drm] Max dedicated hypervisor surface memory is 507904 KiB
4.319476 [drm] Maximum display memory size is 10384 KiB
4.325967 [drm] Screen Objects Display Unit Initialized
4.326524 [drm] width 720
4.326547 [drm] height 480
4.326565 [drm] bpp 32
4.327100 [drm] Fifo max 0x00200000 min 0x00001000 cap 0x00000355
4.327154 [drm] Host log (poweroff) failed to send host log message.
4.327293 fbcon: vgaconfb (fb0) is primary device
4.337598 Console: switching to colour frame buffer device 100x37
4.341215 [drm] Initialized vmwgfx 2.18.1 20210218 for 0000:00:02:0 on minor 0
4.579442 snd_intel8x0 0000:00:05:0: allow list rate for 1028:0177 is 48000
4.651105 audit: type=1400 audit(1647665366.179:2): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=534 comm="apparmor_parser"
4.651117 audit: type=1400 audit(1647665366.179:3): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=534 comm="apparmor_parser"
4.651119 audit: type=1400 audit(1647665366.179:4): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=533 comm="apparmor_parser"
4.673611 audit: type=1400 audit(1647665366.203:5): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=546 comm="apparmor_parser"
4.674281 audit: type=1400 audit(1647665366.203:6): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=546 comm="apparmor_parser"
4.674283 audit: type=1400 audit(1647665366.203:7): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=546 comm="apparmor_parser"
4.685763 audit: type=1400 audit(1647665366.215:8): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=551 comm="apparmor_parser"
4.689222 audit: type=1400 audit(1647665366.219:9): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/bin/nan" pid=547 comm="apparmor_parser"
4.699925 audit: type=1400 audit(1647665366.223:10): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-client.action" pid=535 comm="apparmor_parser"

5.871128 kauditd: printk skb: 30 callbacks suppressed
5.871136 audit: type=1400 audit(1647665367.399:41): apparmor="DENIED" operation="capable" profile="/usr/sbin/cupsd" pid=711 comm="cupsd" capability=12 capname="net_admin"
5.903493 loop: detected capacity change from 0 to 8
6.025959 e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
6.029467 IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
6.547501 audit: type=1400 audit(1647665368.079:42): apparmor="DENIED" operation="capable" profile="/usr/sbin/cups-browsed" pid=769 comm="cups-browsed" capability=23 capname="sys_nice"
20.172930 vboxvideo: loading version 6.1.26 r145957
20.250620 04:49:41.784315 main VBoxService 6.1.26 r145957 (verbosity: 0) linux.and64 (Jul 20 2021 16:25:57) release log
20.250727 04:49:41.784018 main OS Product: Linux
20.250772 04:49:41.784078 main OS Release: 5.13.0-30-generic
20.250827 04:49:41.784222 main OS Version: #33-Ubuntu SMP Fri Feb 4 17:03:31 UTC 2022
20.250913 04:49:41.784779 main Executable: /opt/VBoxGuestAdditions-6.1.26/sbin/VBoxService
20.250913 04:49:41.784779 main Process ID: 907
20.250913 04:49:41.784780 main Package type: LINUX_64BITS_GENERIC
20.250913 04:49:41.786744 main 6.1.26 r145957 started. Verbose level = 0
20.250913 04:49:41.789120 main VBoxServiceCtrlDetectFreezeCancelSupport: Supported (#1)
24.940120 rfkll: Input handler disabled
37.140741 rfkll: Input handler enabled
40.408441 ISO 9660 Extensions: Microsoft Joliet Level 3
40.409622 ISO 9660 Extensions: RRIP-1991A
40.409691 rfkll: Input handler disabled
906.316531 usbcore: registered new interface driver USB Driver
root@sureshAnanth:/home/suresh_a/hcl#

```

Fig.no.4.2 Output of USB Driver

4.2.6.2.1 NVME NAMESPACE

A namespace is a group of logical block addresses (LBA) accessible to host software in NVME technology. A namespace ID (NSID) is a unique identification assigned to a namespace by a controller. The isolation of logical blocks addressable by the host software, rather than physical isolation of blocks, is what a namespace is. Namespaces appear in Linux devices with their own unique identifiers; /dev/nvme0n1 is looking at controller 0 and namespace 1).

There are a variety of reasons why host software would want to partition an NVMe SSD into multiple namespaces, including logical isolation, multi-tenancy, security isolation (encryption per namespace), write protecting a namespace for recovery, and overprovisioning to improve write performance and endurance.

The size of the namespace on an NVMe SSD is set to the size of the LBAs determined by the manufacturer by default. In the host software, each namespace is shown as a separate target device. Size, Capacity, and Usage of Namespaces. The Namespace Size, Capacity, and Utilization fields in the Identify Namespace data structure are related:

- The Namespace Size (NSZE) column specifies the namespace's overall logical block size (LBA 0 through n-1).
- The NCAP (Namespace Capacity) parameter specifies the maximum number of logical blocks that can be allocated at any one time.
- The Namespace Utilization (NUSE) column specifies how many logical blocks the namespace presently has. This enables the host programme to monitor how a device reacts to the deallocate instruction. The intention is that this will be 0 (no utilisation) after a format, but after writing, the utilisation will increase until a deallocate command is received (this is commonly referred to as "TRIM").

CHAPTER - 5

5.CONCLUSION

Some software was incorporated to keep total expenses low, but the accuracy of the data was degraded as a result. Various debugging tools, including as gcov, lcov, Kgdb, and debugfs, are used to debug the nvme driver.

The kernel in this suggested application is designed to evaluate performance accurately and at a high enough sampling rate to characterise events that occur at fast speeds. The gcov and lcov tools were created to assist users in generating reports on the code generated in the project, allowing developers to determine the percentile of test cases covered and develop software according to a pre-determined design.

REFERENCES

1. <https://www.learnitguide.net/2015/11/linux-boot-process-step-by-step-procedure.html>
2. <https://www.thegeekstuff.com/2011/02/linux-boot-process/>
3. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf
4. <https://nvmexpress.org/wp-content/uploads/NVMe-NVM-Express-2.0a-2021.07.26-Ratified.pdf>
5. <https://github.com/linux-nvme/nvme-cli>
6. <https://www.iitg.ac.in/asahu/cs421/books/LDD3e.pdf>
7. <https://www.youtube.com/watch?v=rOXsKuW5xXw>
8. <https://www.kernel.org/doc/html/v4.14/dev-tools/gcov.html>
9. <https://elixir.bootlin.com/linux/v5.15.25/source/drivers/nvme/host/core.c#L197>
10. <https://www.kernel.org/doc/html/v4.14/dev-tools/kgdb.html>
11. <https://filippo.io/linux-syscall-table/>
12. <https://opensource.com/article/21/7/linux-kernel-ftrace>
13. [ch16.28124 \(timesys.com\)](#)