INTRODUCTION

1.1 AN OVERVIEW

Every computer system in existence contains a basic kernel whose architecture is dependent upon the system that it is to be run into. Kernel is the core of an operating system. Operating system receives the request from user and processes it on the behalf of user. Requests are received by command shell or some other kind of interfaces and are processed by the kernel. So kernel acts like an engine of the operating system which enables the user to use a computer system. Shell is the outer part of the operating system that provides and interface to the user for communicating with the kernel.

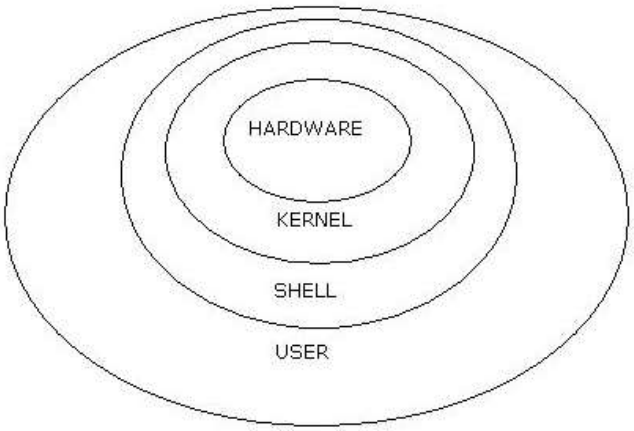

Figure 1.1 – Basic Kernel Structure

Our kernel is very limited in its functioning, as it'll be limited to the educational purpose of understanding kernel internals and to make a basic functional kernel system.

1.2 OBJECTIVES OF THE PROJECT

Every good project must have some goal or objective behind which it was designed. The main motive behind our project was understanding and educational level implementation of the system. The objective behind **Kernel Re-Implementation** is to provide various computer science undergrad students to gain an in-depth knowledge in the system internals and to get a first-hand understanding and implementation of such a system and its working.

So we basically wanted to provide a low level programming based descriptive details to the system internals such that students can easily understand the system and so learn about the Operating Systems in a more effective manner. The main objectives of our projects are as follows:-

Reliability:-The project is reliable in working. If there may be any error in any tool of the project, then it is easily handled and also it is easy to use for the user of it. The errors that would be made by the user are easily handled.

Accuracy: - The project also provides accuracy to the user. The system internals are defined in the most concrete manner and contain the ability to provide the most interfaces to the system.

Flexibility:-The project is flexible in working. It can be run on various systems and can be moved to any other system. It can accept any small changes. If needed a various upgrades regarding functionality can be added into it.

User-friendly:- The interface of the software is not very user friendly as the whole project is system based implementation. All the interfaces are controlled programmatically from inside the system and thus provide the programmed execution.

Easy to use: - The project is easy to use for any user. The project has a user-friendly commented code blocks segments and easy statements to handle all the options.

Thus, the goal of our project is to provide efficient and effective implementation to various system level programming abstracts and thus maintain the nature of the basic operability. This will help in overall understanding of the Kernel internals and various implementations that are done in the systems.

SYSTEM ANALYSIS

2.1 PROPOSED SYSTEM

2.1.1 Problem Definition

The aim is to develop a system that provides a well-documented implementation of a basic Linux based micro-kernel. The project focuses on providing the basic abstraction level understanding while implementing the most complex system features and internals. The main feature of the project is to provide an opportunity to understand the Operating System's working and their interconnections. The system output reports generated are the most effective in providing the insight into the system internals and their working in simplified manner.

Usability of well-structured code segments and commented views in the system are the basic requirements of our system and thus provide an error free and well manageable code data. As the base system is totally non present hence any change done in the code is immediately reflected in the system. The Linux core framework ensures the basic hardware level abstraction as well as the freedom to implement, integrate and to resolve our code for any purpose. Automatic system dump reports provide a detailed description of how the system is functioning and how well is it behaving under various circumstances.

The project code is well documented and is structured in a manner such that the implementation level details are transparent to the users. Along with that the documentation of the system provides a more theoretical overview of the system and insight into the code. Use of the system level commented code structure provides the system to provide a most direct and effective data and application handling approach. Help can be obtained by reading the documentation in the project report.

The system contains the basic register level addressing system and c language data structures. Being a system level implementation the algorithmic and data structure usage is minimalized to the very basics and thus minimalistic and serialized algorithms are used along with array based implementation in the most complex scenarios.

The system we started on was a bare system to which we provided our system data in order to make it accessible to the systems there by performing a system software level details and implementations.

2.2 SYSTEM DESIGN

The system development was done in the incremental phases to ensure the basic development to reach a substantial goal of the system run. The bare bone system is of no use and is totally blank.

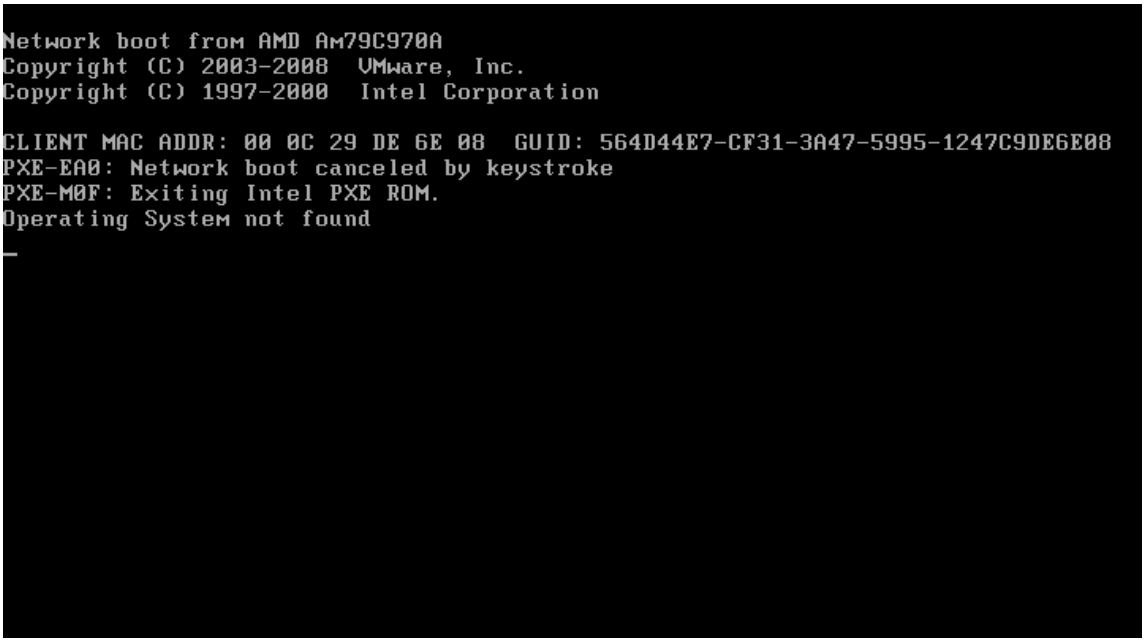

Figure 2.1 – Bare System Response on Boot Up

To provide the system with an operating system the whole system is to be created in the sequential manner. The various phases of the system development are as follows.

2.2.1 Hardware Interfaces

A bare hardware system is not of any use unless it is prepared by the kernel for the tasks. But in that comes various tasks that are required by the system to complete in order to obtain the most prolific abstraction level details into the system. In this part of the system development we laid the system foundation by providing the system a boot loader that could provide the basic startup into the system. The system also required the programming for the output of the system data that we were to provide at the output screen.

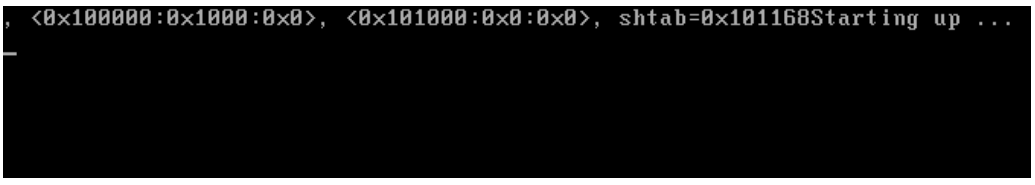

Figure 2.2 – Initializing the system with Boot Loader

Figure 2.3 – First Directed Response from the System

2.2.2 Logical Interfaces

After the basic layout of the system in the hardware level comes the system wide applicability of it on the abstraction level of memory and the logical interfaces that provide the connections among the system and the memory states. At this state the system is provided with the implementation of Global and Interrupt Descriptor Tables.

Figure 2.4 – Interrupting the System Run

2.2.3 Bridges

While dealing with the memory and various system internals we need various mechanisms by which we can interact with those internals and the program level data. For this we use the methodology of using bridged interrupts. Interrupts provide a connection among the normal data flow to the complete system. It also includes the paging mechanism that can be used in the system.

Figure 2.5 – Clocked Interrupt via Interrupt Request Queue

2.2.4 Virtual File System

Now we require the system to provide an abstraction level to the directory structure and the internals of the hard disk data and the capability to access it form the program level structures. This is done by the use of Virtual File System and thus providing a complete storage and directory structure to the system.

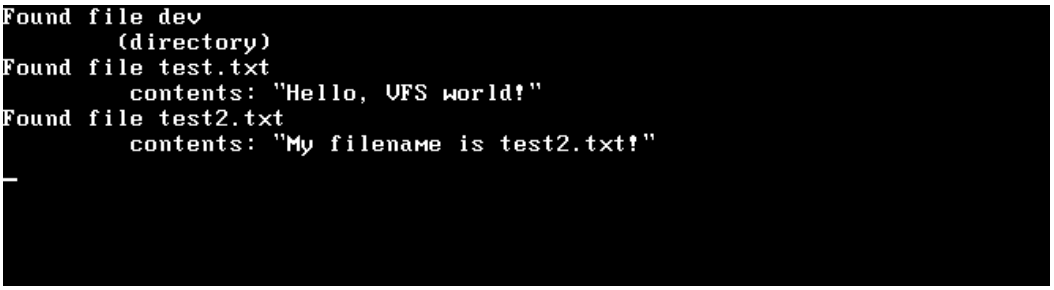

Figure 2.6 – Virtual File System Showing Directories and File Contents

2.2.5 User Mode

Till now our kernel had been running in the “Kernel Mode” or “Supervisor mode”. Though this mode is very powerful but it is not usable by a normal user in order to provide the functionality to the system. For this we make use of the User mode of operations and abstractions in to the system. This way we can restrict the memory access by the system to the internal architecture. This is often used by the system to hide the data from the kernel.

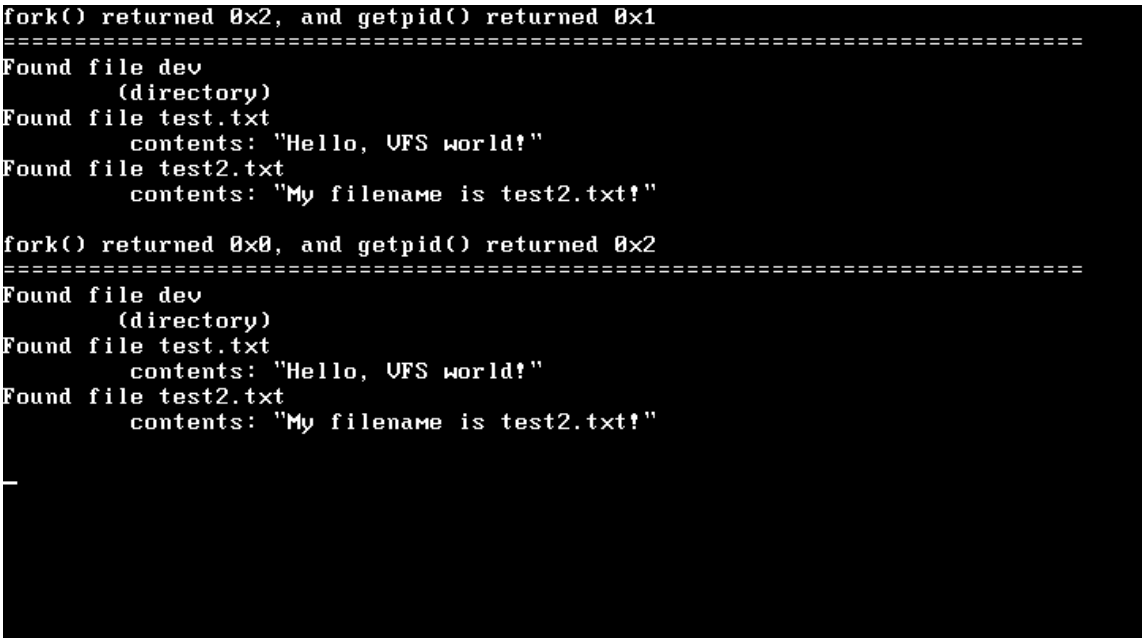

Figure 2.7 – Multiple tasks running in Fork processes

2.3 SYSTEM SPECIFICATION

2.3.1 Kernel Modules Specification

2.3.1.1 Kernel Components

Major components of a kernel are:

Low level drivers: These are the architecture specific drivers and are responsible for the CPU, MMU and on-board devices initialization.

Process Scheduler: Scheduler is responsible for the fair share usage of the CPU time slices and its allocation to different processes.

Memory Manager: Memory management system is responsible for the allocation and sharing memory to different processes.

File System: Linux supports many file systems. i.e. FAT, NTFS, EXT3 etc. The user does not need to worry about the system internals. For this Linux provides the single interface named Virtual File System. By the use of this the complexities of the underlying file system are abstracted from the user.

Device Drivers: These are the higher level abstracted device drivers.

IPC: Inter Process Communication allows the different processes to share data among themselves.

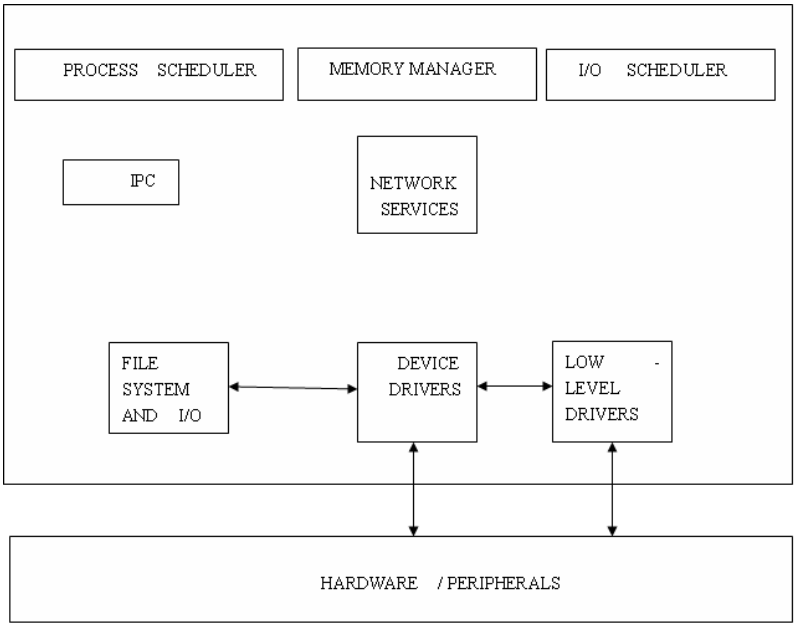

Figure 2.8 – Higher Level Abstraction of a Kernel

2.3.1.2 Integration Design

The integration design tells us how these different components are integrated to create a Kernel's binary image. There are mainly 2 designs used for the Operating Systems design: Monolithic and Micro kernel based.

In monolithic design all the kernel components are built together into a single binary image. At the boot time the entire kernel gets loaded into the memory and then runs as a single process in a single address space. All the kernel components or services exist in that static kernel image. All the kernel services are running and available at all the times.

Also since inside kernel everything resides in a single address space, so no IPC kind of mechanism is needed for communication amongst different kernel services. For all these reasons monolithic kernels are high performance and most UNIX kernels are monolithic in nature.

The downside of this is that once the static image of kernel is loaded into the memory, we cannot add or remove any of the components of the system. Also it has a high memory footprint. So the resource consumption is high in case of monolithic kernels.

The second type of kernel is microkernel. In such a kernel a static kernel image is not built, instead the kernel image is broken into the different small services. At boot time the core services are loaded into the system and they run in the privileged mode. Whenever any service is required it has to get loaded for running.

Unlike monolithic kernel all services are not up and running at all the times. They run as and when requested. Also unlike monolithic kernels, services in microkernels are in separate address spaces.

So communication between the two different types of services requires the IPC mechanism. Hence they are not high performance but they run on low resources.

Our kernel (based on Linux) takes best of both these designs. Fundamentally it's a monolithic kernel and run as a single process. But it also has the ability to load/unload the services in the form of kernel modules.

2.3.2 Environment Setup

2.3.2.1 Base System

For the development and testing of the system we require a system that is robust in handling various bugs that we might incur in our code. Also since all the coding and implementations are at system level hence it requires the system to provide a better mechanism at handling system level errors and faults. The system is required to be very tolerant to the hardware level faults and crashes that might bring down a normal system. Also it is required to be resilient to the memory dumps and over rights and remain stable and accessible in those states.

Since such a system is not there in existence and using real world hardware for testing hardware critical tasks is not advisable due to high cost of maintenances and system dumps. For this measure we make use of the virtual machine systems and thus provide the system non susceptible to the damage that might occur during the testing and development phases.

We will be using a *nix system, with GNU tool chain. For a windows system we can use Cygwin (*nix emulation environment).

Directory Structure

```
project
 |
 +--                                                    src
 |
 +-- docs
```

All source files are stored in ‘src’ and all the documentation is stored at ‘docs’

2.3.2.2 Compiling

GUN tool chain: gcc, ld, gas

Assembly codes in Intel Syntax (much human readable than AT&T syntax)

NASM: Netwide Assembler

Bootloader: GRUB (Grand Unified Bootloader). Floppy Disk with GRUB preloaded onto it.

2.3.2.3 Useful Scripts

Scripts are useful for automation of various tasks. As we are going to do lots of testing and making (compiling and linking) with the project, so we will be using scripts to automate that task for us.

Makefile

Compiles all the files in ‘src’ and links them together into one ELF binary, ‘kernel’.

link.ld

Linker script, which makes sure that everything, goes in the correct place. It tells LD how to set up the kernel image.

update_image.sh

This script pokes the new kernel binary into the floppy image file.

2.2.3 Hardware Specification

Processor	**:**	32 Bit, x86 Architecture
Hard Disk	**:**	10MB
RAM	**:**	4 MB
Peripherals	**:**	VGA and Floppy Disk Drive

2.2.4 Software Specification

System Architecture	**:**	x86 Intel based
Operating System	**:**	None
POST Script	:	VMware based
Boot loader	:	GRUB Based basic boot loader

TESTING

3.1 PRE-DEVELOPMENT TESTING

3.1.1 Linux Kernel Testing

For the pre-development research and testing purposes we require the system to contain the basic Linux kernel along with its source code. For the educational purposes the best such kernel is Linux Kernel Ver. 2.6.XX. Since this version is the most stable kernel to date and also contains the most basic yet complete structure it's the best for testing of the system.

The directory structure of the Linux system is as follows:

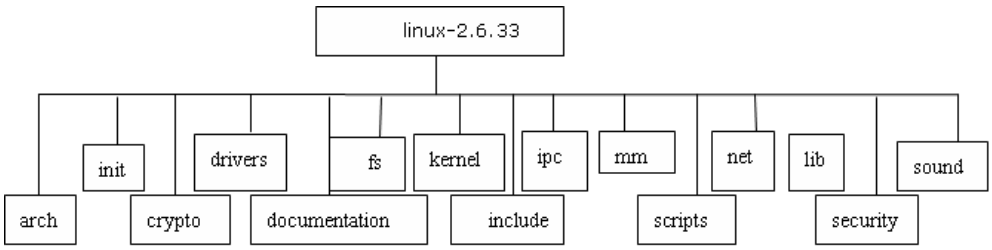

Figure 3.1 – Directory Structure of Linux Kernel

This provides a basic introduction to the kernel structure and the interfaces that it provides along with the code.

The Linux kernels are to be built up by the use of compilation into the system. By descending down into the Linux directory we can compile the source code of these systems. We tested several of our Kernel Concepts based on this directory and code structures.

3.1.2 Testing Strategies

To begin the testing we have to compile the kernel code, and thus include the required modules by the use of make command. Thus we obtain a custom system that contains the specific modules are required by us. Thus we did the following testing on the various kernel parts:

- Modular Testing: We dived into the individual modules of the kernel code and thus obtained the works set graph of the lowest level functional hierarchy of the data component movement in the registers and through the functions. In this value passing and processing of the system we obtain the hierarchical function and data passing structure of the system internals. This testing methodology proved to be the best possible way to understand the built in features of the micro-kernel.

- Integration Testing: To understand the overall kernel system as humungous monolithic kernels we performed the integration testing on the system. During this phase we did the modular integration by using the self-made and compiled systems. We tested removing various parts of the kernels and thus obtaining the different outcomes in the system stability and run ability.

3.1.3 Testing Methods

The best testing mechanism for a kernel developer is peer to peer review. But being distributed in the geographical region we were unable to perform such a testing more often. For this we used the Inspection testing along with desk testing and auditing.

- Inspection Testing
- Desk Testing
- Auditing

3.2 KERNEL INTERNALS

For the proper understanding of how a kernel internally looks like we need to map the various high level structures and modules of kernel. These structures are categorized on the level of their implementation and the details along the functions that they implement.

Hence the map along these layers is constructed along with the functions at each level. The tree thus obtained is very complex in structure and contains the functional set of the module level hierarchy. It shows the complexity at each implementation layer along with the interconnectivity among the similar layer modules and the interfaced interactions among the different layers.

There are 5 different layers namely: Electronics, Hardware Interfaces, Logical, Bridges, Virtual Subsystems and User Space Interfaces.

They are divided among the system with the functions pertaining to system, networking, storage, memory, processing and human interfaces.

Each of these contains an intermixed module at each layer and there even exists some module that is not bounded by these internal boundaries and semantics of the data and work flow. Thus a very complex structure of kernel is obtained at this level.

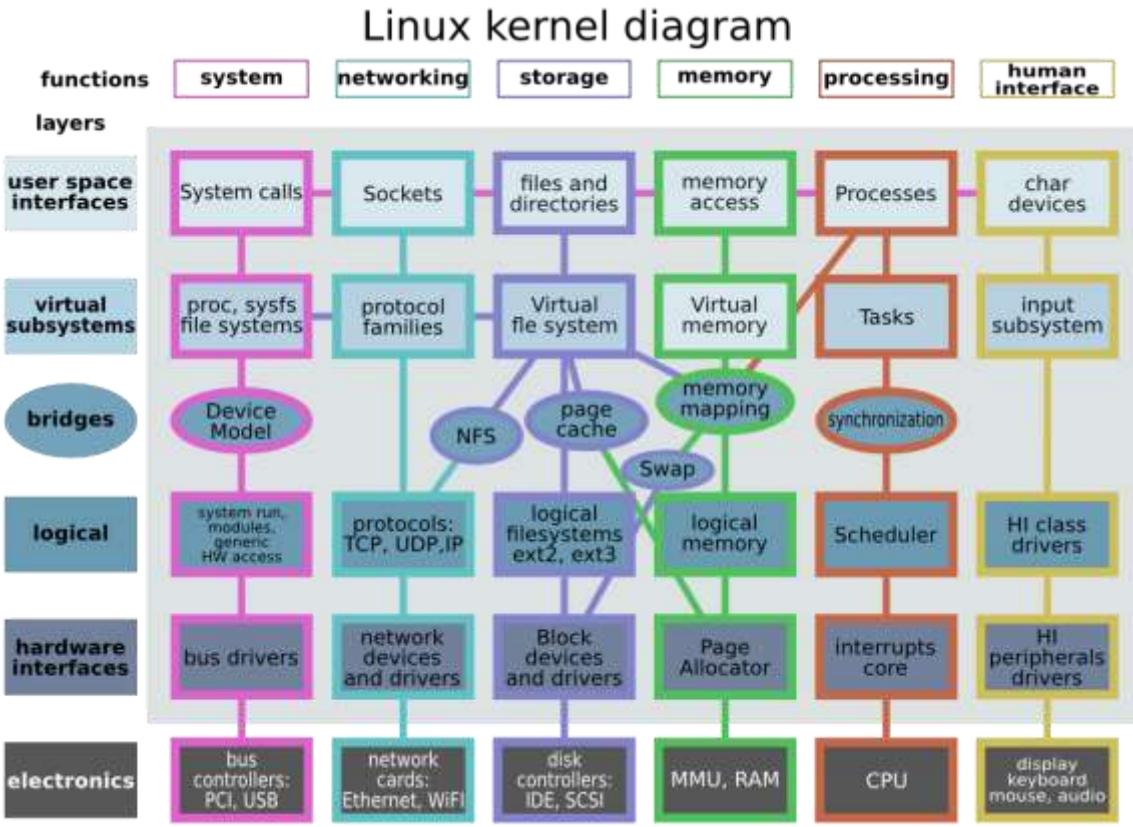

Figure 3.2 – Kernel Diagram on Functional Layer View

This is the basic structural distribution of the kernel internals and is at the top level hierarchy. This can be further broken down into smaller segments and function calls that map to each other and pass data in order to execute in the simplest yet most complex inter network format.

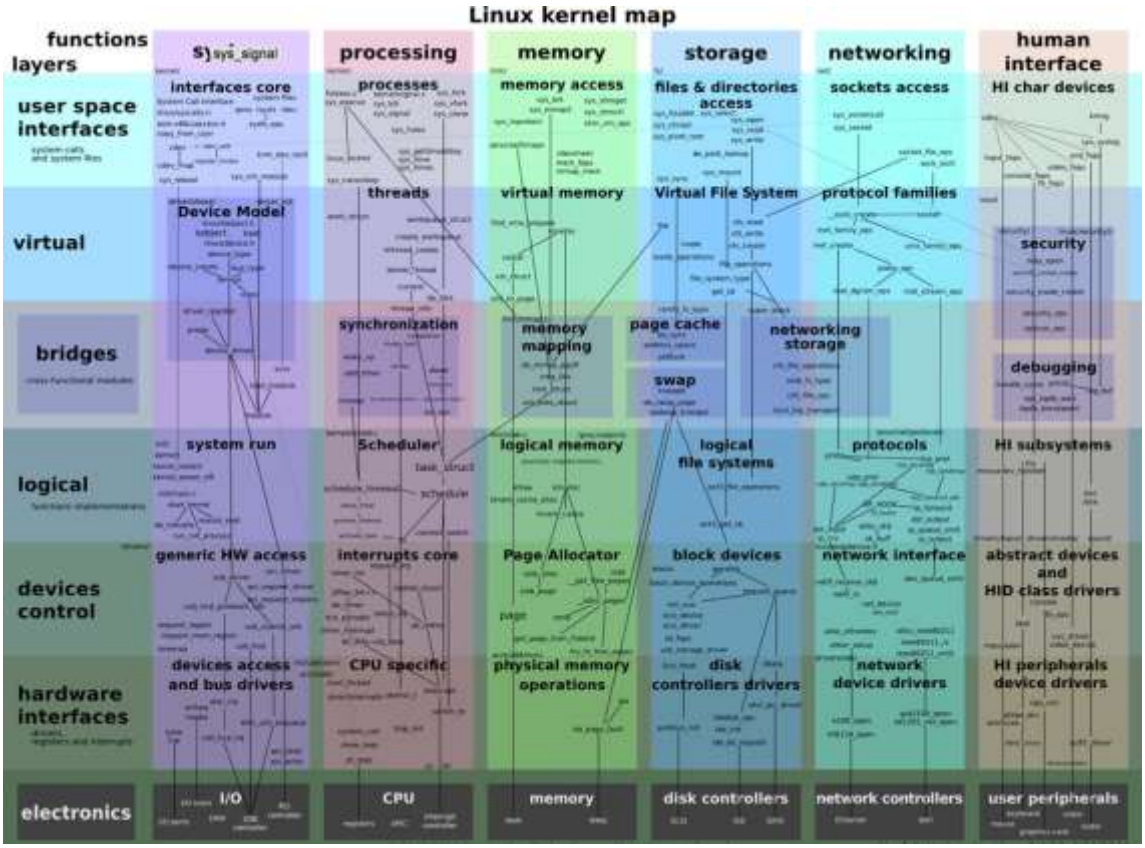

Figure 3.3 – Kernel Diagram Detailing Calling Methods

This concluded the study of the kernel internals and the structural mapping onto the system states.

3.3 CONCLUSIONS

Thus the kernel internal structure is obtained as described above and the data and process flow in the system is mapped for the study. This structure is now used in the design and development of the modules, code and the structural overviews. This also provides an insight into the system mapped data values.

DESIGN AND DEVELOPMENT PROCESS

4.1 FUNDAMENTAL DESIGN CONCEPTS

4.1.1 Fundamental design Concepts

A set of fundamental design concepts are evolved over the past decade for the kernel systems. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the system software designer with a foundation from which more sophisticated design methods can be applied. Fundamental design concepts provide the necessary framework for "getting it right".

4.1.2 Modularity

Modularity is the single attribute software that allows a program to be intellectually manageable. System software architecture embodies modularity, that is, software is divided into named and addressable components, called modules that are integrated to satisfy problem requirements.

4.1.3 Software Architecture

Software Architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". Control hierarchy also called "program structure", represents the organization of control. The tree structure is used to represent the control hierarchy.

4.1.4 Structural Partitioning

The program structure should be partitioned both horizontally and vertically. Horizontal partitioning defines separate branches of the modular hierarchy for each major program function, Vertical partitioning called factoring, suggest that control and work should be distributes top-down in the program architecture. Top level modules should perform control functions and do little actual processing work. Modules reside low in the architecture should be the workers, performing all input, computational and output tasks.

4.1.5 Data Structure

Data Structure is a representation of logical relationship among individual elements of data. Because the structure of information will invariably affects the final procedural design, data structure is very important as the program structure to the representation of the software

architecture. Data structure dictates the organization, methods of access, degree of associatively, and processing alternatives for information. The organization and complexity of a data structure are limited only by the ingenuity of the designer. Scalar item array and linked list are some of the representations of the data structure.

4.1.6 Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. System procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact, decision points, repetitive operations and even data organization / structure. Information hiding suggests that modules be "characterized by design decisions that hide from all others." In other words, modules should be specified and designed so that information contained within module is inaccessible to other module.

Design is defining a model of the new system and continues by converting this model to a new system. The method is used to convert the model of the proposed system into computer specification. Data models are converted to a database and processes and flows to user procedures and computer programs. Design proposes the new system that meets these requirements .This new system may be built by a fresh or by changing the existing system. The detailed design starts with three activities, database design, user design and program design. Database design uses conceptual data model to produce a database design.

4.2 SYSTEM DESIGN

4.2.1 Data Flow Diagram

The data flow diagram (DFD) is one of the most important tools used by system analysts. Data flow diagrams are made up of a number symbols, which represent system components. Most data flow modeling methods use four kinds of symbols. These symbols are used to represent four kinds of system components. Processes, data stores, data flows and external entities. Processes are represented by circles in DFD. Data Flow represented by a thin line in the DFD and each data store has a unique name and square or rectangle represents external entities.

Unlike detailed flowchart, Data Flow Diagrams do not supply detailed description of the modules but graphically describes a system's data and how the data interact with the system.

To construct a Data Flow Diagram, we use,

- Arrow
- Circles
- Open End Box
- Squares

An arrow identifies the data flow in motion. It is a pipeline through which information is flown like the rectangle in the flowchart. A circle stands for process that converts data into information. An open-ended box represents a data store, data at rest or a temporary repository of data. A square defines a source or destination of system data.

Rules for constructing a Data Flow Diagram:

- Arrows should not cross each other.
- Squares, circles and files must bear names.
- Decomposed data flow squares and circles can have same names.
- Choose meaningful names for data flow
- Draw all data flows around the outside of the diagram.

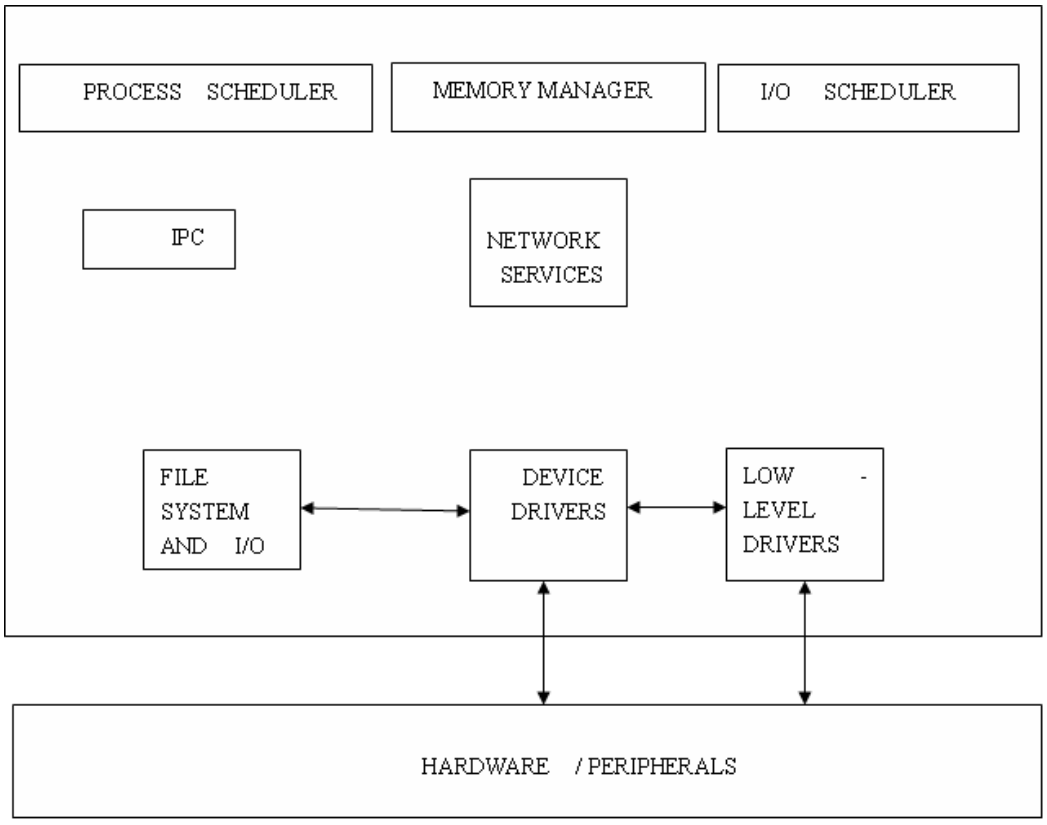

Figure 4.1 – High level Kernel Modules Distribution

4.2.2 ER Diagram

A conceptual model describes the essential features of system data. This conceptual model is described by modeling method known as Entity Relationship analysis. Entity relationship analysis uses three major abstractions to describe data. These are entities- which are distinct things in the enterprise. Relationship-which are meaningful interactions between the objects and the attributes-which are properties of entities and relationship

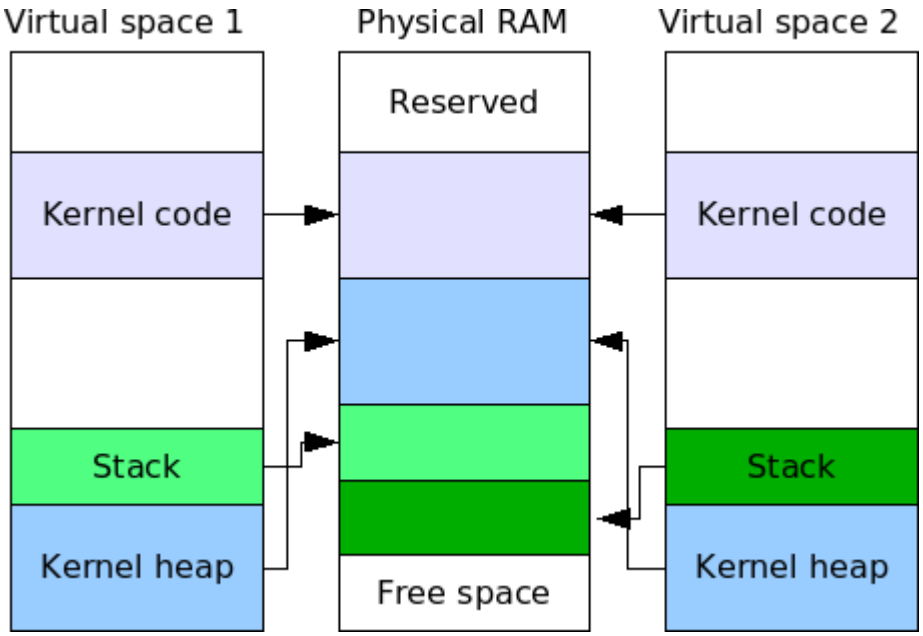

Figure 4.2 – Memory and Virtual Space Allocation

4.3 DESIGN PROCESS

4.3.1 Boot Code

boot.s

It's the kernel start location. It defines multi boot header. Multi boot is a standard to which GRUB expects a kernel to comply. It's a way for the boot loader to

Know exactly what environment the kernel wants/needs when it boots

Allow the kernel to query the environment it is in.

To make kernel multi boot compatible, we need to add the header structure in our kernel. This is done in the initial 4 Mb memory of the Kernel.

dd – The command that lets us embed specific constants in our code.

MBOOT_HEADER_MAGIC

A Magic Number (0x1BADB002). This identifies the kernel as multi boot compatible.

MBOOT_HEADER_FLAGS

A field of flags, we ask for GRUB to page-align all the kernel sections.

MBOOT_CHECKSUM

Magic number + Flags + Checksum = 0 (ZERO). It is used for error checking.

mboot

The address of the structure that we are currently writing. GRUB uses this to tell if we are expected to be relocated.

code, bss, end, start

These symbols are all defined by the linker. We use them to tell GRUB where the different section of our kernel can be located. On boot up, GRUB will load a pointer to another information structure into the EBX register. This can be used to query the environment GRUB set up for us.

So, immediately on boot up, the asm snippet tells the CPU to push the contents of EBX onto the stack (EBX now contains a pointer to the multi boot information structure), disable interrupts (CLI), call our 'main' C function, then enter an infinite loop.

4.3.2 Input Design

Input design is the link between the information system and the users and those steps that are necessary to put transaction data in to a usable form for processing data entry. The activity of putting data into the computer for processing can be activated by instructing the computer to read data from a written printed document or it can occur by keying data directly into the system. The designs of input focusing on controlling the amount of input required controlling the errors, avoid delay extra steps, and keeping the process simple. System analyst decides the following input design details

- What data to input?
- What medium to use?
- How the data is arranged and coded?
- The dialogue to guide the users in providing input.
- Data items and transaction needing validation to detect errors.
- Methods for performing input validation

4.3.3 Compiling, linking and running

CLFLAGS: Stop GCC trying to link to Linux C library with our kernel.

Set CLFLAGS with

-nostdlib: No standard library

-nostdinc: No standard include

-fno-builtin: No inbuilt function

-fno-stack-protector: No stack protector

Now make file and update image using the scripts. The resultant image is available for run. Boot into the test-bed using this disk and the system will freeze while saying 'starting up…'.

Use log file to check for the EAX register value. It'll be set at 'deadbaba' – the return value of main ().

4.3.4 Output Design

Designing computer should proceed in well thought out manner. The term output means any information produced by the information system whether printed or displayed. When analyst design computer out put they identified the specific output that is needed to meet the requirement. Computer is the most important source of information to the users. Output design is a process that involves designing necessary outputs that have to be used by various users according to requirements. Efficient intelligent output design should improve the system relationship with the user and help in decision making. Since the reports are directly required by the management for taking decision and to draw the conclusion must be simple, descriptive and clear to the user. Options for outputs and forms are given in the system menus.

When designing the output, system analyst must accomplish the following:

- Determine the information to present.
- Decide whether to display, print, speak the information and select the output medium
- Arrange the information in acceptable format.
- Decide how to distribute the output to intended receipt.

5 SCOPES FOR FURTHER ENHANCEMENTS

1. Boot loader Development

2. Networking Implementation

3. SWAP Memory Implementation

4. GUI and Human Interface

5. Implementing Cryptology

6 REFERENCES

1. Linux Kernel Mailing List (LKML)

2. www.kernel.org

3. Pune Linux User Group (PLUG) mailing lists

4. Kernel-newbies mailing list

5. www.wikipedia.org/Kernel(Computing)