
# OS CASE STUDY

# LINUX KERNAL FROM SCRATCH

## CSE – E

| S.NO | NAME OF THE STUDENT | ROLL NUMBER |
|---|---|---|
| 1 | KSSH HARSHITHA | CB.EN.U4CSE21427 |
| 2 | SHARVESH S | CB.EN.U4CSE21454 |
| 3 | KUNDLA AKHILA PAVANI | CB.EN.U4CSE21433 |
| 4 | VISWAA RAMASUBRAMANIAN | CB.EN.U4CSE21468 |
| 5 | VIGNESH G | CB.EN.U4CSE21466 |

# PROJECT DESCRIPTION

Our project aims to create a Linux kernel from scratch in a Windows environment. To accomplish this, we utilize the x86 assembly language and C language for coding and development.

Working with the Linux kernel requires a test computer as any errors in the code can potentially impact the entire system. Therefore, caution and thorough testing are crucial during the development process.

One of the fundamental concepts in our project is the address space. Address space refers to the set of addresses that a process can use to access memory. In our case, we focus on the 64-bit address space. Within this address space, we allocate portions for RAM, hard disk storage, and other devices.

Another important concept is the process. A process represents a program in execution and includes program instructions, necessary data, heap, and stack within its allocated address space. In the case of multiple processes, each has its own separate address space, while the kernel remains the same for all processes. The operating system kernel contains essential information about each process, such as its state, and user programs are restricted from accessing data in kernel space.

The operating system (OS) runs in kernel mode, which allows it to execute all instructions and access hardware resources. On the other hand, user programs run in user mode, where they have limited access to instructions and cannot directly access kernel space. To interact with the kernel and hardware, user programs make use of system calls, which are special functions acting as an interface between user programs and the operating system. When a program calls a system call, control is transferred to the operating system, which performs the required operation and returns control to the user program.

Moving on to the OS structure, we explore two different approaches. The first is the monolithic system structure, where the entire operating system functions as a single large executable program in kernel mode. In this structure, the modules within the OS communicate directly with each other, making function calls straightforward and efficient. However, the lack of restrictions in calling functions can make the system difficult to maintain, and any errors in a module can potentially affect the entire operating system. For example, encountering an error in the memory management module could lead to a system halt.

The second structure we consider is the microkernel structure. In this approach, the operating system is divided into smaller modules, and the kernel itself remains minimalistic. Errors occurring within these modules only affect themselves and do not crash the entire system. The microkernel structure offers enhanced fault isolation, as issues within one module do not propagate to others. However, implementing a microkernel structure requires more complexity compared to a monolithic kernel.

In summary, our project involves developing a Linux kernel from scratch using x86 assembly language and C language within a Windows environment. We emphasize the importance of addressing space, process management, system calls, and consider different OS structure options such as the monolithic and microkernel structures.

# PRE – REQUISITES

Building a Linux OS from scratch is a complex and challenging task that requires significant knowledge of computer science concepts, operating system development, and low-level programming. Here are some of the steps and skills that you will need to successfully complete this project:

1. Learn the basics of operating system development: Before starting your project, it is important to have a good understanding of the basic principles of operating system development, including process management, memory management, file systems, device drivers, and system calls.
2. Choose a hardware platform and architecture: You will need to select a hardware platform and architecture that you want to target with your operating system. This will determine the specific tools, libraries, and technologies that you will need to use to develop your OS.
3. Build a development environment: You will need to set up a development environment that includes a cross-compiler toolchain, debuggers, emulators, and other development tools that are specific to your chosen hardware platform and architecture.
4. Develop the kernel: You will need to develop the Linux kernel, which is the core component of your operating system. This involves writing device drivers, managing memory, implementing system calls, and managing processes.
5. Develop user space utilities and libraries: Once you have developed the kernel, you will need to build the user space utilities and libraries that are necessary to provide a complete operating system experience. This includes programs like the shell, file system utilities, network utilities, and various libraries.
6. Test and debug your OS: Once you have developed your operating system, you will need to test it thoroughly to ensure that it is stable and reliable. This will involve running tests on a variety of hardware platforms and architectures, as well as debugging any issues that arise.

In terms of the skills and knowledge required to complete this project, you will need to have a strong understanding of low-level programming languages like C and assembly language, as well as knowledge of various development tools and technologies like make, gdb, and git. You should also have a good understanding of computer architecture, hardware design, and computer networking. Additionally, you will need to have good problem-solving skills and be able to work independently to overcome the many challenges that you will face during this project.

# OUR PROJECT SPLIT UP

Our project has been divided into several sub-parts to facilitate the development process and ensure comprehensive documentation. Here is an overview of each section:

1. Installation of Virtual Machine (Bochs) for Windows and Setting up Boot Image File: This section focuses on installing the Bochs virtual machine on a Windows system and configuring the boot image file. It includes step-by-step instructions for setting up the development environment.
2. Printing "Hello World" or "Welcome to Amrita OS" using Assembly Language: In this section, we demonstrate how to write assembly code to print a simple message, such as "Hello World" or "Welcome to Amrita OS," on the screen. We provide the necessary code snippets and explain their functionality.
3. Loading the Loader and Switching to Long Mode: Here, we discuss the process of loading the loader file and switching the system into long mode. We cover the necessary steps, such as setting up the necessary data structures and executing the required instructions.
4. Exceptions and Interrupt Handling: This section focuses on handling exceptions and interrupts in the operating system. We explain the different types of exceptions and interrupts and discuss their handling mechanisms.
5. Working with C: In this section, we explore the integration of C language into our operating system. We discuss how to write C code to perform various tasks and interact with the assembly language components.
6. Memory Management: This section delves into the memory management aspects of our operating system. We discuss concepts such as address space, memory allocation, deallocation, and protection mechanisms.
7. Process: The process section covers the management of processes within the operating system. We explain the process creation, execution, termination, and scheduling mechanisms.

All the relevant documentation, including detailed reports for each section, can be found in our GitHub repository called "OS-LINUX-KERNEL." The repository contains image files, binary files, boot, loader, and kernel assembly files, along with all the necessary resources and code snippets. For further details, please refer to the GitHub repository using the link provided below:

## OS-LINUX-KERNAL GIT HUB LINK

In addition to the individual branches representing each section of the project, there is an additional branch called "reports." This branch serves as a centralized location for all the reports related to the project. It contains a compilation of all the reports from the individual branches, providing a comprehensive overview of the entire project.

By accessing the "reports" branch in the GitHub repository, you will find all the reports conveniently organized in one place. This allows for easy reference and access to the documentation related to each section, facilitating a comprehensive understanding of the project as a whole.

Please navigate to the "reports" branch in the GitHub repository to access the compiled reports for a complete overview of our Linux kernel development project.

# CONCLUSION

In conclusion, our project aimed to create a Linux kernel from scratch using x86 assembly language and C language on a Windows platform. Throughout the project, we focused on various key aspects of operating systems, addressing concepts such as address space, processes, and the role of the kernel.

To begin, we installed the virtual machine Bochs on Windows and set up the boot image file, enabling us to simulate the execution of our kernel. This step provided us with a suitable environment for development and testing.

Next, we successfully printed "Hello World" or "Welcome to Amrita OS" using assembly language. This initial accomplishment demonstrated our ability to interact with the hardware and display information on the screen without relying on high-level functions.

We then proceeded to load the loader and switch to long mode, a critical step for transitioning to 64-bit address space. By checking the CPU's support for long mode and verifying the availability of 1GB page support, we ensured compatibility and prepared the system for extended functionality.

Exception and interrupt handling were essential components of our project. We implemented mechanisms to handle exceptional situations and interrupt requests, guaranteeing proper system behavior and robustness in the face of unexpected events.

Integrating the C language into our project allowed us to leverage its high-level capabilities and libraries. Working with C, we expanded our functionality, enabling more complex operations and enhancing the overall performance and usability of our kernel.

Memory management played a vital role in our project, enabling efficient allocation and deallocation of memory resources. We focused on optimizing memory usage, ensuring reliable memory access, and preventing memory-related issues such as leaks and fragmentation.

Managing processes was another crucial aspect we addressed. By understanding process structures and their relationship with the kernel, we implemented mechanisms for process creation, execution, and termination, enabling multitasking and efficient resource utilization.

In summary, our project involved the step-by-step development of a Linux kernel from scratch. Each section focused on a specific aspect of operating systems, ranging from low-level hardware interactions to high-level process management. Through a combination of assembly language and C language programming, we successfully created a functional kernel capable of executing user programs and handling various system operations.

The detailed reports for each section of the project, along with all the necessary files, can be found in the respective branches of our GitHub repository. This comprehensive documentation provides a deeper understanding of the project's implementation and serves as a valuable resource for further exploration and development in the field of operating systems.