RV Educational Institutions ®

RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

Submitted by

Vanshika Khandelwal **1RV22CS224**
Shreya Chakote **1RV22CS189**

Computer Science and Engineering
2023-2024

INDEX

INTRODUCTION

A bootloader is a small program that loads the operating system (OS) into the computer's memory (RAM) and initializes its execution. It resides in the boot sector of a storage device like a hard drive, solid-state drive (SSD), or a floppy disk. When you turn on your computer, the BIOS (Basic Input/Output System) firmware executes the bootloader code to start the boot process. The bootloader then locates the OS kernel, loads it into memory, and hands over control to the operating system.

x86 Architecture:
The x86 architecture is a family of backward-compatible instruction set architectures based on the Intel 8086 CPU. The x86 architecture is characterized by its instruction set, which includes a wide range of operations for arithmetic, logic, control flow, and more. It uses a segmented memory model, which divides memory into segments of various sizes and addresses them using segment selectors and offsets. The x86 architecture also supports different operating modes, such as real mode, protected mode, and long mode (also known as 64-bit mode).

- Power-On Self-Test (POST): Although not strictly part of the bootloader's responsibility, the BIOS firmware, which loads the bootloader, usually performs POST to check the hardware components during system startup.
- Initialization: The bootloader initializes the CPU, memory, and other hardware components necessary for booting the operating system
- Boot Device Detection: It identifies the storage device (e.g., hard drive, SSD, USB drive) containing the OS kernel.
- Loading the Kernel: The bootloader reads the kernel from the boot device into memory.
- Handing Over Control: Once the kernel is loaded into memory, the bootloader transfers control to the kernel code, allowing the operating system to take over.

SYSTEM ARCHITECTURE

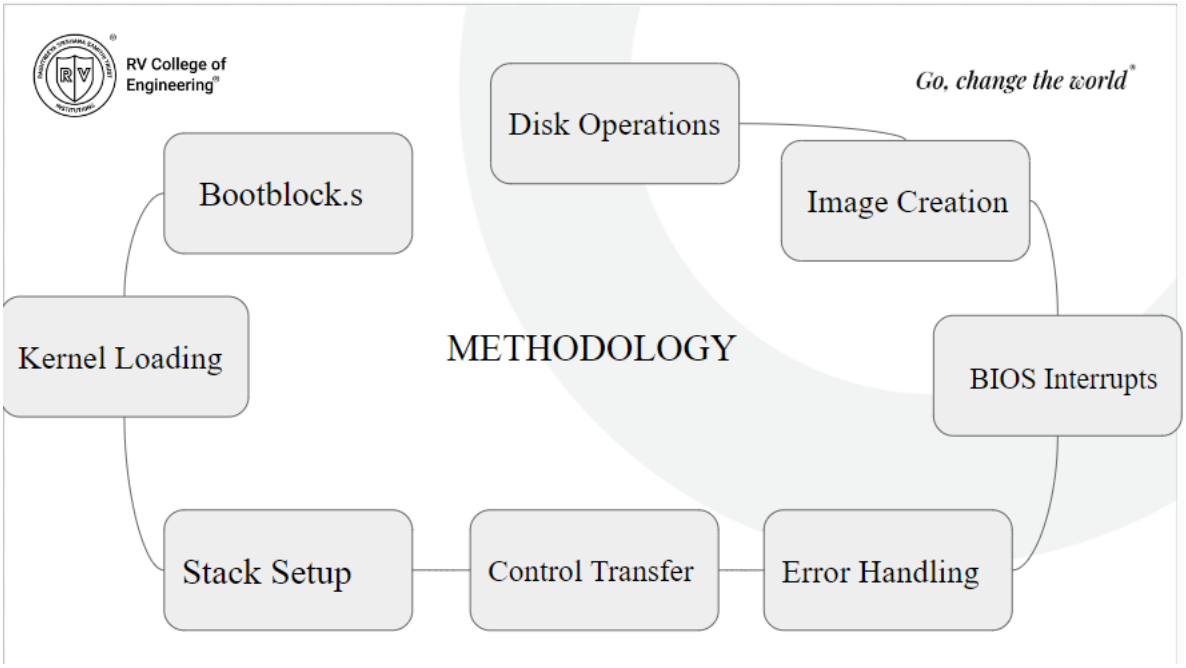
In the realm of x86 architecture, the bootloader serves as the maestro orchestrating the intricate ballet of system initialization and bootstrapping. With a deft touch, the bootloader initializes crucial hardware components, setting the stage for the main event. Memory controllers, timers, and peripheral interfaces awaken under its command, ensuring the system's readiness for the upcoming performance.

- Reset Vector: When the system boots up or resets, the processor begins executing instructions from a predefined memory location known as the reset vector. The bootloader code usually resides at this address.

- Initialization: The bootloader initializes the hardware components necessary for the boot process, such as memory controllers, timers, and peripheral interfaces.
- Boot Device Selection: The bootloader determines which device to boot from. This could be a hard drive, solid-state drive (SSD), network interface, or another storage medium. The selection process may involve user input or be based on predefined settings.
- Boot Device Access: Once the boot device is selected, the bootloader accesses the necessary sectors or blocks to load the next stage of the boot process. This might involve reading the Master Boot Record (MBR), Volume Boot Record (VBR), or other boot sectors.

- Loading the Operating System Kernel: The bootloader locates and loads the operating system kernel into memory. This involves reading the kernel image from the boot device and placing it in the appropriate memory location.
- Handoff to the Operating System: After loading the kernel, the bootloader transfers control to the operating system. This typically involves jumping to the entry point of the kernel code and passing any necessary parameters.
- Error Handling: Bootloaders often include error detection and recovery mechanisms. They may perform integrity checks on the boot device or handle errors such as disk read errors or corrupted boot sectors.
- User Interface (Optional): Some bootloaders provide a user interface for configuring boot options, selecting operating systems in a multi-boot environment, or accessing diagnostic tools.
- Security Features (Optional): Depending on the system's requirements, bootloaders may incorporate security features such as secure boot, which ensures that only trusted code is executed during the boot process.
- Customization and Extensibility: Bootloaders are often designed to be customizable and extensible to accommodate different hardware configurations and boot requirements. This may involve configuration files, plugins, or other mechanisms for extending functionality.

METHODOLOGY



To execute the code:

1. Save the bootloader assembly code in a file named bootloader.s.
2. Save the createimage C code in a file named createimage.c.
3. Open a terminal or command prompt.
4. Navigate to the directory containing the files.
5. Compile the bootloader assembly code using an assembler
   nasm -f bin bootloader.s -o bootloader.bin
6. Compile the createimage C code using a C compiler
   gcc createimage.c -o createimage
7. Run the createimage tool to generate the OS image file:
   ./createimage bootloader.bin kernel.bin os_image.img
   Replace kernel.bin with the filename of your kernel binary if it's different.

8. The os_image.img file will be generated, containing the bootloader and kernel binaries.
9. You can then test the image file using an emulator like QEMU or by writing it to a bootable device such as a USB flash drive.

SYSTEMS CALLS USED

Here, there are several system calls used along with file operations.

1.fread():

Purpose: Reads data from a given stream.

Arguments: Pointer to a buffer where the read data will be stored, size of each element to be read, number of elements to read, and the file stream to read from.

Used in: read_elf_headers() function to read ELF headers from bootloader and kernel files.

2.fwrite():

Purpose: Writes data to a given stream.

Arguments: Pointer to the data to be written, size of each element to write, number of elements to write, and the file stream to write to.

Used in: Not used in the provided code, but typically used for writing data to files.

3.fseek():

Purpose: Sets the file position indicator for the given file stream.

Arguments: File stream, offset from the specified position, and the position indicator (e.g., SEEK_SET, SEEK_CUR, SEEK_END).
Used in: write_bootloader_kernel() function to set the position within the image file before writing bootloader and kernel data.

4.fopen():

Purpose: Opens a file and associates a stream with it.
Arguments: Path to the file and the mode in which to open the file ("rb" for read mode, "wb" for write mode).
Used in: main() function to open bootloader, kernel, and image files.

5.fclose():

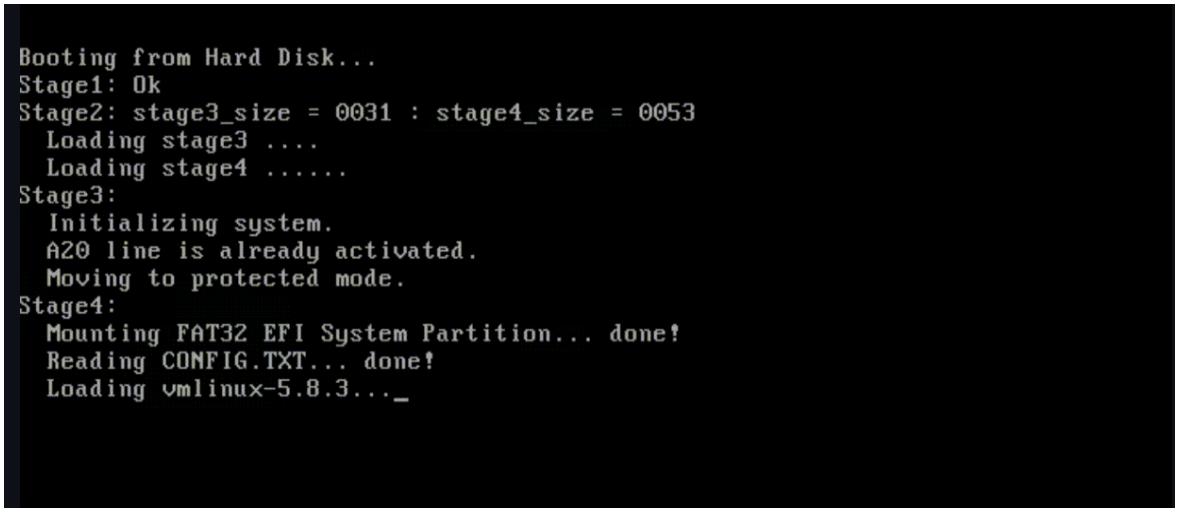
Purpose: Closes the given file stream.
Arguments: File stream to close.
Used in: main() function to close bootloader, kernel, and image files.

6.fprintf():

Purpose: Prints formatted data to a stream (usually stderr or stdout).
Arguments: Output stream (stderr in this case), format string, and additional arguments.
Used in: Error handling to print error messages.

7.exit():
Purpose: Terminates the program immediately.
Arguments: Exit status (usually EXIT_FAILURE or EXIT_SUCCESS).
Used in: Error handling to exit the program when an error occurs.

OUTPUT



CONCLUSION

Creating a bootloader for the x86 architecture is a fundamental aspect of operating system development. It involves writing low-level code to initialize the system, load the operating system kernel into memory, and transfer control to it. Here's a structured overview:

Firstly, understanding the x86 architecture is crucial. This includes knowledge of CPU registers, memory layout, and the instruction set architecture (ISA). With this understanding, developers can write low-level code in assembly language or C to interact directly with hardware components.

Bootloaders typically have a fixed structure. They often consist of a boot sector, which is the initial code loaded by the BIOS, and may include additional stages for more complex bootloading processes.

Loading the kernel is a primary function of the bootloader. This involves reading the kernel image from disk and copying it to a predefined

memory location. For bootloaders responsible for loading ELF (Executable and Linkable Format) kernels, understanding the ELF file structure and parsing it is essential.

System calls and file operations are commonly used in bootloader development to interact with the underlying hardware and load necessary files from disk. These operations enable the bootloader to access storage devices, read data, and perform necessary initialization tasks.

Robust error handling and debugging mechanisms are essential during bootloader development. Errors can lead to system failures, so techniques like printing debug information to the screen or using a serial port for debugging are crucial for diagnosing issues.

Thorough testing and validation are necessary to ensure the bootloader works correctly across different hardware configurations and with various operating systems. Testing should include scenarios such as booting from different storage devices and handling various kernel configurations.

Security considerations are critical in bootloader development. Vulnerabilities in the bootloader can compromise system security, so measures like signature verification and secure boot should be implemented to prevent unauthorized code execution during boot.

Documentation and community resources are valuable assets for bootloader developers. Leveraging tutorials, documentation, and community forums can help navigate challenges and accelerate the development process.

In summary, creating a bootloader for x86 architecture requires a deep understanding of low-level programming, system architecture, and boot process intricacies. With careful attention to detail, thorough testing, and consideration of security measures, developers can create reliable and

efficient bootloaders to initialize the system and boot the operating system kernel.