RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

Experiential Learning

Kernel Development in C

Submitted By

N Ragavenderan – 1RV22CS122

Ranjana Prabhudas – 1RV22CS158

Pavan Shivakumar – 1RV22CS135

Submitted in

partial fulfilment for the
award of degreeof

BACHELOR OF ENGINEERING
in
Computer Science and Engineering,

Course: Operating Systems – CS235AI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2023-24

CONTENTS:

PROBLEM STATEMENT:

Create a basic operating system kernel with bootloader initialization, display output, and keyboard input handling. Test and validate the kernel using emulation tools like QEMU.

INTRODUCTION:

Operating system (OS) development is a complex yet foundational aspect of computer science. At its core lies the kernel, the engine driving hardware interaction and user experience. This report delves into the process of developing a basic kernel, starting with bootloader creation in 16-bit assembly and extending to keyboard input handling and display functionality.

Beginning with bootloader development, we explore the critical role of initializing the system before OS execution, spotlighting bootloaders like GNU GRUB. Next, we delve into kernel development, emphasizing hardware interaction, memory management, and display output via the Visual Graphics Array (VGA).

The report further examines keyboard input handling, showcasing port I/O operations to capture user keystrokes. Testing and integration methods, including emulation with QEMU, are discussed to validate kernel functionality.

In summary, this report provides a concise overview of kernel development, highlighting its pivotal role in system operation and user interaction.

SYSTEM ARCHITECTURE:

x86 system architecture refers to the hardware and software framework designed around Intel's x86 microprocessor family and its compatible processors, such as those manufactured by AMD. This architecture has evolved over several decades and encompasses a wide range of computing systems, from personal computers to servers and embedded devices. Understanding the x86 system architecture is fundamental for developing software that runs efficiently on x86-based systems. Here's an overview of its key components:

1.Central Processing Unit (CPU): The CPU is the core component of the x86 architecture, responsible for executing instructions and performing calculations. 86 CPUs include a variety of models with different features and capabilities, including various levels of cache, instruction set extensions (such as SSE, AVX), and multiple cores for parallel processing.

2.Memory Management: x86 architecture supports a hierarchical memory model, with physical memory managed by the CPU and virtual memory managed by the operating system. Memory management features include segmentation and paging, which provide mechanisms for memory protection, address translation, and efficient memory allocation.x86 CPUs use a Memory Management Unit (MMU) to translate virtual addresses to physical addresses and enforce memory protection.

3.Instruction Set Architecture (ISA):x86 instruction set architecture defines the set of instructions that the CPU can execute. It includes basic arithmetic and logic operations, control flow instructions, and specialized instructions for tasks such as string manipulation, input/output operations, and system management. The x86 ISA has evolved over time, with new instructions and extensions added to enhance performance and functionality.

4.Interrupts and Exceptions: Interrupts are signals generated by hardware devices or software events that interrupt the CPU's normal execution flow. Exceptions are like interrupts but are triggered by exceptional conditions such as invalid memory access or arithmetic errors.x86 CPUs use interrupt descriptor tables (IDT) to handle interrupts and exceptions, with specific handlers for each type of interrupt or exception.

5.I/O Ports: Input/output (I/O) ports provide a mechanism for the CPU to communicate with peripheral devices such as keyboards, displays, and storage devices.x86 CPUs use special instructions (in and out) to read from and write to I/O ports, which are typically used for low-level device communication.

6.System Management Mode (SMM): SMM is a special operating mode provided by x86 CPUs for system management tasks such as power management, system monitoring, and firmware updates. When the CPU enters SMM, it switches to a separate execution context with its own set of memory and resources, allowing system management functions to run independently of the operating system.

7.Boot Process: The x86 boot process begins with the system firmware (BIOS or UEFI) initializing hardware components and loading the bootloader from storage.The bootloader then loads the operating system kernel into memory and transfers control to the kernel's entry point, starting the operating system initialization process.

METHODOLOGY:

1.Setting Up the Development Environment: Install the necessary development tools, including a compiler (such as GCC), an assembler (like NASM or GNU as), and a linker (like GNU ld). Set up a virtual machine or an emulator environment for testing the kernel. Tools like QEMU or VirtualBox are commonly used for this purpose.

2.Understanding GRUB Configuration: GRUB (GRand Unified Bootloader) is a commonly used bootloader for x86 systems. Learn how to configure GRUB to boot your kernel. This involves creating or modifying the GRUB configuration file (usually named grub.cfg) to specify the kernel's location and parameters.

3.Bootloader Initialization: GRUB initializes the system and loads the kernel into memory.Understand how GRUB sets up the environment for the kernel, including setting up the Global Descriptor Table (GDT), enabling protected mode, and transitioning control to the kernel's entry point.

4.Kernel Entry Point: Define the entry point of your kernel code. This is typically where execution begins after control is transferred from the bootloader. Initialize essential data structures and set up the environment necessary for kernel execution.

5.Hardware Interaction: Use x86 assembly language or low-level C code to interact with hardware devices. Implement routines to communicate with the keyboard controller and read input from the keyboard buffer. Understand the PS/2 protocol commonly used by keyboards on x86 systems.

6.Interrupt Handling: Configure interrupt handling to respond to keyboard interrupts.Set up the Interrupt Descriptor Table (IDT) to handle keyboard interrupts (IRQ1). Write interrupt service routines (ISRs) to handle keyboard interrupts and process incoming keystrokes.

7.Buffering and Input Processing: Implement a buffer to store keyboard input temporarily.Process incoming keystrokes to convert scan codes into ASCII characters or other representations.

Handle special keys (e.g., function keys, control keys) as needed.

8.Kernel Output: Optionally, implement functionality to display output on the screen, allowing feedback or interaction with the user.

Implement basic text-mode output routines or use BIOS or VESA framebuffer for graphics output.

9.Testing and Debugging: Test the kernel in a virtual machine or emulator environment.Use debugging tools such as GDB, QEMU's built-in debugger, or printf-style debugging to identify and fix issues

<u>SYSTEM CALLS:</u>

The GNU Assembler :often abbreviated as GAS, is the assembler provided as part of the GNU Compiler Collection (GCC). It is a component of the GNU toolchain used for compiling programs written in languages like C, C++, and Fortran, into machine code that can be executed by a computer's processor. The GNU Assembler translates assembly language code, which is a low-level programming language that closely corresponds to the machine code instructions understood by the CPU, into binary machine code that the computer can execute directly. Assembly language provides a human-readable representation of machine instructions, allowing programmers to write code that interacts directly with the hardware of a computer system.

GNU/Linux :is used to refer to the combination of the Linux kernel with the GNU operating system, developed by the Free Software Foundation (FSF) and its founder, Richard Stallman. GNU/Linux distributions come in various flavours, such as Ubuntu, Fedora, Debian, and CentOS, each of which may include different software packages and configurations, but all are based on the Linux kernel and the GNU userland tools. The combination of the Linux kernel with the GNU operating system and various other components has resulted in a powerful, versatile, and widely used operating system that powers everything from servers and desktop computers to embedded systems and mobile devices. Additionally, GNU/Linux is renowned for its stability, security, and the extensive range of free and open-source software available for it.

grub-mkrescue :is a command-line utility used in the GNU GRUB (Grand Unified Bootloader) bootloader system. Its primary function is to create a bootable ISO image that contains the GRUB bootloader along with specified configuration files and bootable kernels. This ISO image can be burned onto a CD/DVD or written to a USB drive to create a bootable media. grub-mkrescue allows users to create rescue discs or installation media for their operating systems, providing a convenient way to boot into a system or perform system recovery tasks. Additionally, it's often used in the process of creating custom Linux distributions or live CDs/DVDs. Overall, grub-mkrescue is a powerful tool for creating bootable media with GRUB bootloader functionality.

Xorriso: Is a command-line tool renowned for its versatility in creating, extracting, and modifying ISO 9660 filesystem images commonly utilized for CDs, DVDs, and optical media. Part of the libisoburn library suite, it supports a wide array of features, including the creation of hybrid ISO images compatible with both BIOS and UEFI booting, making it suitable for various system architectures. xorriso is adept at crafting bootable ISOs by incorporating boot records, loaders, and bootable images, and offers extensive support for boot loaders like ISOLINUX, GRUB, and EFI. Its command-line interface empowers users with robust control over image customization, allowing for the addition, modification, and removal of files and directories, as well as the specification of volume attributes and labels. Moreover, xorriso is scriptable and suitable for automation, enhancing its utility for integration into build processes and automated workflows. With support for Rock Ridge extensions, xorriso ensures compatibility with Unix-like operating systems, preserving critical file metadata such as permissions and symbolic links. Overall, xorriso stands as a powerful and flexible toolset indispensable for developers, system administrators, and distributors of optical media.

QEMU (Quick Emulator [3]): is a free and open-source emulator. It emulates a computer's processor through dynamic binary translation and provides a set of different hardware and device models for the machine,

enabling it to run a variety of guest operating systems. It can interoperate with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native

speed. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

QEMU supports the emulation of various architectures, including x86, ARM, PowerPC, RISC-V, and others. QEMU can save and restore the state of the virtual machine with all programs running. Guest operating systems do not need patching in order to run inside QEMU.

DESIGN OF OPERATING SYSTEM KERNEL DEVELOPMENT:

1. Bootloader Development:

- Objective: Initialize the system and load the kernel into memory.

- Implementation:

  - Define necessary information for multiboot compliance: magic number, flags, and checksum.

  - Set up the stack and call the kernel entry point.

- Explanation: Bootloader initializes essential system components and prepares the environment for kernel execution. It sets up parameters required by the multiboot specification and transfers control to the kernel.

2. Kernel Initialization:

- Objective: Initialize the kernel environment and set up display output.

- Implementation:

  - Initialize VGA display buffer and set colors for text output.

  - Define functions for interacting with VGA buffer: print characters, strings, and integers.

  - Implement kernel entry point to initialize VGA display and print initial message.

- Explanation: Kernel initialization is crucial for setting up the operating environment. It prepares the display buffer for output and initializes necessary data structures for further execution.

3. Keyboard Input Handling:

- Objective: Capture user input from the keyboard and process it.

- Implementation:

  - Define constants for keyboard scan codes representing various keys.

  - Implement functions to read input from the keyboard port, convert scan codes to ASCII characters, and handle keyboard interrupts.

- Explanation: Keyboard input handling enables interaction with the user. It allows the kernel to respond to user commands and input in real-time, enhancing the user experience.

4. Integration and Testing:

- Objective: Verify kernel functionality and compatibility using emulation tools.
- Implementation:
  - Compile bootloader and kernel source files into object files.
  - Link object files to create a multiboot-compliant kernel binary.
  - Test kernel using emulation tools like QEMU to ensure proper functionality.
- Explanation: Integration and testing are critical phases to ensure the correctness and robustness of the kernel. Emulation tools like QEMU simulate system environments, allowing developers to assess kernel behaviour without relying on physical hardware.

SOURCE CODE:

1. Boot.S

```
# set magic number to 0x1BADB002 to identified by bootloader
.set MAGIC,    0x1BADB002
 # set flags to 0
.set FLAGS,    0
 # set the checksum
.set CHECKSUM, -(MAGIC + FLAGS)
 # set multiboot enabled
.section .multiboot
 # define type to long for each data defined as above
.long MAGIC
.long FLAGS
.long CHECKSUM
 # set the stack bottom
stackBottom:
 # define the maximum size of stack to 512 bytes
.skip 1024
 # set the stack top which grows from higher to lower
stackTop:
 .section .text
.global _start
.type _start, @function
 _start:
  # assign current stack pointer location to stackTop
```

```
mov $stackTop, %esp
# call the kernel main source
call kernel_entry
 cli
 # put system in infinite loop
hltLoop:
 hlt
jmp hltLoop
 .size _start, . - _start
```

2. Kernel.c

```
#include "kernel.h"

 uint16 vga_entry(unsigned char ch, uint8 fore_color, uint8 back_color)
{
  uint16 ax = 0;
  uint8 ah = 0, al = 0;

  ah = back_color;
  ah <<= 4;
  ah |= fore_color;
  ax = ah;
  ax <<= 8;
  al = ch;
  ax |= al;
  return ax;
}

//clear video buffer array
void clear_vga_buffer(uint16 **buffer, uint8 fore_color, uint8 back_color)
{
  uint32 i;
  for(i = 0; i < BUFSIZE; i++){
    (*buffer)[i] = vga_entry(NULL, fore_color, back_color);
  }
}
 //initialize vga buffer
void init_vga(uint8 fore_color, uint8 back_color)
{
  vga_buffer = (uint16*)VGA_ADDRESS;  //point vga_buffer pointer to
VGA_ADDRESS
  clear_vga_buffer(&vga_buffer, fore_color, back_color);  //clear buffer
}

void kernel_entry()
{
  //first init vga with fore & back colors
  init_vga(WHITE, BLACK);

  //assign each ASCII character to video buffer
  //you can change colors here
  vga_buffer[0] = vga_entry('H', WHITE, BLACK);
  vga_buffer[1] = vga_entry('e', WHITE, BLACK);
  vga_buffer[2] = vga_entry('l', WHITE, BLACK);
  vga_buffer[3] = vga_entry('l', WHITE, BLACK);
  vga_buffer[4] = vga_entry('o', WHITE, BLACK);
  vga_buffer[5] = vga_entry(' ', WHITE, BLACK);
  vga_buffer[6] = vga_entry('W', WHITE, BLACK);
  vga_buffer[7] = vga_entry('o', WHITE, BLACK);
  vga_buffer[8] = vga_entry('r', WHITE, BLACK);
  vga_buffer[9] = vga_entry('l', WHITE, BLACK);
  vga_buffer[10] = vga_entry('d', WHITE, BLACK);
}
```

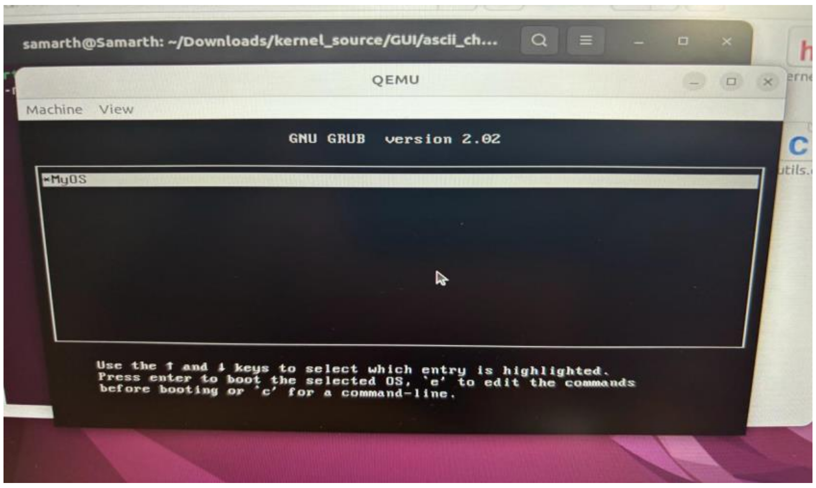
3. run.sh

```
#assemble boot.s file
as --32 boot.s -o boot.o

#compile kernel.c file
gcc -m32 -c kernel.c -o kernel.o -std=gnu99 -ffreestanding -O2 -Wall -Wextra

#linking the kernel with kernel.o and boot.o files
ld -m elf_i386 -T linker.ld kernel.o boot.o -o MyOS.bin -nostdlib

#check MyOS.bin file is x86 multiboot file or not
grub-file --is-x86-multiboot MyOS.bin

#building the iso file
mkdir -p isodir/boot/grub
```

```
cp MyOS.bin isodir/boot/MyOS.bin

cp grub.cfg isodir/boot/grub/grub.cfg

grub-mkrescue -o MyOS.iso isodir


#run it in qemu

qemu-system-x86_64 -cdrom MyOS.iso
```

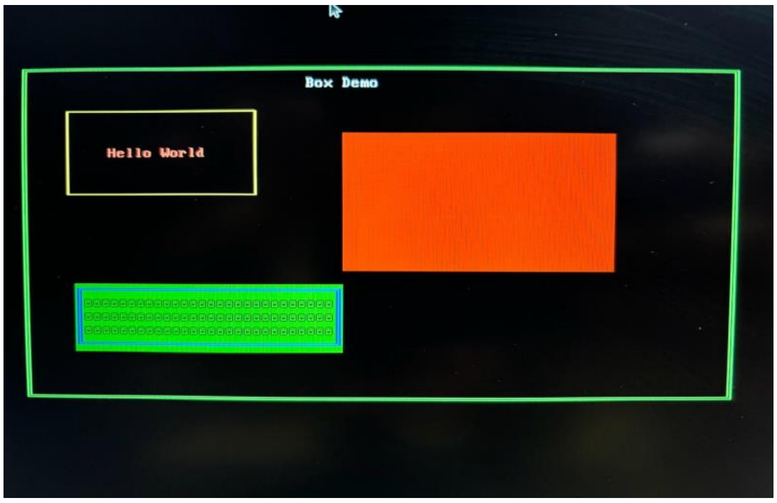
OUTPUT:

1.GNU Grub

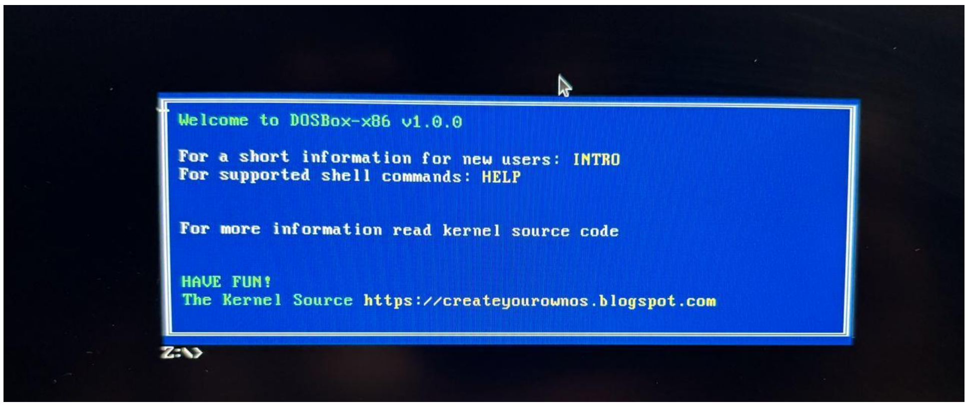


```
GNU GRUB  version 2.02

*MyOS

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands
before booting or `c' for a command-line.
```

2.Box Drawing GUI

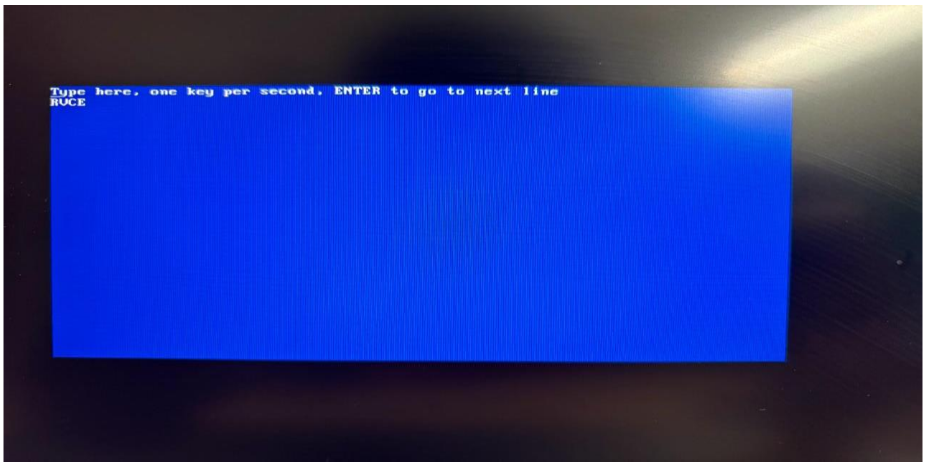


3.DOSBox



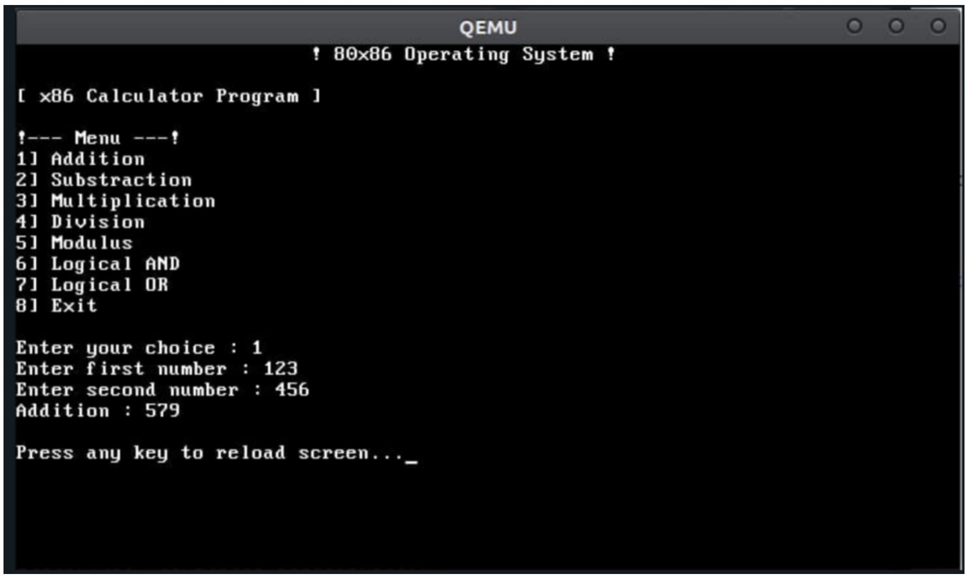
```
Welcome to DOSBox-x86 v1.0.0

For a short information for new users: INTRO
For supported shell commands: HELP


For more information read kernel source code


HAVE FUN!
The Kernel Source https://createyourownos.blogspot.com
Z:\>
```

4.Keyword



```
Type here, one key per second, ENTER to go to next line
RVCE
```

5.Calculator:



```
QEMU
                 ! 80x86 Operating System !

[ x86 Calculator Program ]

!--- Menu ---!
1] Addition
2] Substraction
3] Multiplication
4] Division
5] Modulus
6] Logical AND
7] Logical OR
8] Exit

Enter your choice : 1
Enter first number : 123
Enter second number : 456
Addition : 579

Press any key to reload screen..._
```

Conclusion:

In conclusion, developing a simple kernel that incorporates keyboard input, implements a basic calculator, and displays a "Hello, World!" message involves a series of fundamental steps in kernel development. Beginning with setting up the development environment and understanding x86 system architecture, the process extends to configuring the bootloader, handling interrupts for keyboard input, and designing input processing mechanisms. Integration of a calculator functionality requires implementing arithmetic operations and user interaction logic within the kernel. Additionally, displaying the "Hello, World!" message involves outputting text to the screen, which necessitates interfacing with the display hardware. Throughout this endeavor, meticulous attention to detail, thorough testing, and proficiency in low-level programming techniques are essential. Ultimately, this project offers a valuable learning experience in kernel development, providing insights into hardware interaction, system architecture, and software design principles.