RV College of Engineering

Create your own Kernel Implementation

Go, change the world

Course: Operating Systems – CS235AI DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
RV College of Engineering, Bangalore – 560059, INDIA

PROBLEM STATEMENT:

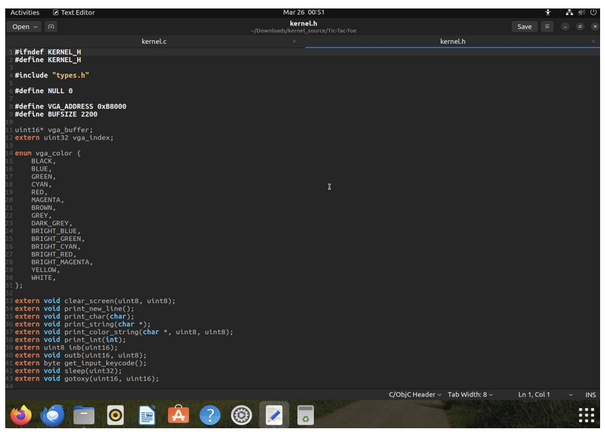
Create a basic operating system kernel with bootloader initialization, display output, and keyboard input handling. Test and validate the kernel using emulation tools like QEMU.

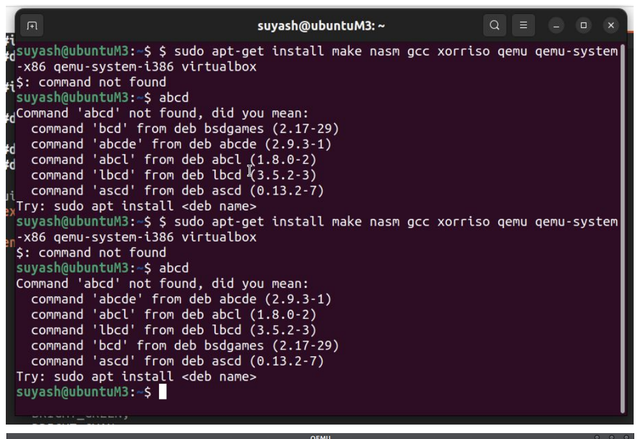
INTRODUCTION:

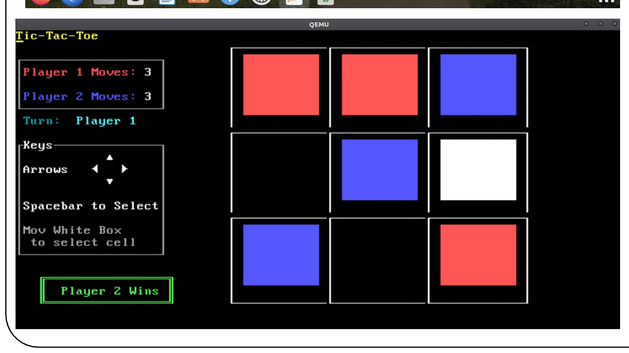
Operating system (OS) development is a complex yet foundational aspect of computer science. At its core lies the kernel, the engine driving hardware interaction and user experience. This report delves into the process of developing a basic kernel, starting with bootloader creation in 16-bit assembly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

RELEVANT OPERATING SYSTEM CONCEPTS:

1. **GNU Assembler (GAS):** it is part of the GNU Compiler Collection (GCC) toolchain, used to convert programs from languages like C, C++, and Fortran into machine code for direct execution by a computer's processor. It translates low-level assembly code closely related to machine code guidelines into executable machine code, providing programmers with a readable representation of machine instructions for hardware interaction.
2. **GNU/Linux:** it refers to the combination of the Linux kernel with the GNU operating system, championed by the Free Software Foundation (FSF) and Richard Stallman. It comes in various distributions like Ubuntu, Fedora, Debian, and CentOS, all relying on the Linux kernel and GNU tools. This powerful and widely used operating system is known for its stability, safety, and extensive collection of free and open-source software, supporting a wide range of devices from servers to smartphones.
3. **GRUB-MKRESCUE:** it is a command-line utility within the GNU GRUB bootloader system that creates bootable ISO images containing the GRUB bootloader, configuration files, and bootable components. These ISO images can be copied to discs/DVDs or written to USB drives for bootable media creation, useful for system recovery or installation. It's commonly used for custom Linux distributions or live CDs/DVDs, offering a convenient way to boot into systems or perform recovery tasks with GRUB bootloader functionality.
4. **QEMU (Quick Emulator):** it is a free and open-source emulator that mimics computer processors through dynamic binary translation, supporting various architectures like x86, ARM, PowerPC, RISC-V, etc. It runs guest operating systems without patching, and it can save/restore VM states, allowing applications from one architecture to run on another. QEMU can interoperate with KVM for close-to-native speed virtual machines.
5. **Bootloader:** it is a little program or piece of code that is executed when a PC framework is turned on or restarted. Its basic role is to introduce the equipment parts of the PC and burden the working framework (operating system) or other fundamental projects into the PC's memory (Smash) so the framework can fire up and become functional. During the boot process, the bootloader basically serves as a link between a computer system's software and hardware layers.







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

SYSTEM ARCHITECTURE:

x86 architecture refers to a family of instruction set architectures (ISAs) developed by Intel and later adopted by other processor manufacturers such as AMD. It has been the dominant architecture for personal computers and servers since the 1980s. Here's a brief overview:

History: The x86 architecture traces its roots back to Intel's 8086 microprocessor, released in 1978. It was a 16-bit processor designed as an improvement over Intel's earlier 8080 and 8085 processors Subsequent iterations, such as the 80286, 80386, and 80486, introduced various enhancements, including wider data paths and support for virtual memory.

Key Features:

Complex Instruction Set Computer (CISC): x86 architecture is known for its complex instruction set, which includes a wide range of instructions for performing various tasks. This richness in instructions allows for more operations to be performed directly by the hardware, reducing the need for software emulation.

Segmented Memory Model: In the original x86 architecture, memory addressing was based on a segmented memory model, dividing memory into segments of fixed size. This model was later extended to support flat memory addressing in protected mode.

Protected Mode and Virtual Memory: x86 processors support protected mode, which provides features like memory protection, multitasking, and virtual memory. These features enable modern operating systems to provide robust memory management and process isolation.

SMP and Multicore Support: x86 architecture supports symmetric multiprocessing (SMP) and multicore processors, allowing multiple processor cores to execute instructions concurrently. This capability enhances system performance and scalability.

Continued Evolution: The x86 architecture continues to evolve with advancements in processor technology. Modern x86 processors feature multiple cores, advanced vector processing units, hardware-level security features, and power-efficient designs.

Overall, the x86 architecture has played a significant role in the evolution of computing, powering a wide range of devices from desktops and laptops to servers and data centers. Its compatibility, performance, and versatility have made it a dominant force in the computing industry for decades.

Guide Information:
Dr. JYOTI SHETTY
Assistant Professor, Department of Computer Science and Engineering

TEAM: **Vansh Goel (1RV22CS223)**
Suyash Alva (1RV22CS212)