Operating System Structure and Code Report

Overview

This kernel is a 64-bit operating system kernel designed for x86-64 systems. It is built and tested using **QEMU** as the virtual machine emulator and leverages the **x86-64-elf** toolchain, which includes a cross-compiler specifically tailored for compiling low-level operating system code. The structure of the operating system is modular, with clearly defined bootloader, kernel, and device driver components, making it suitable for further expansion and learning purposes.

Key Components of the Operating System

1. Bootloader

- The bootloader initializes the system, sets up the environment, and transfers control to the kernel.
- It adheres to the **Multiboot Specification**, ensuring compatibility with standard bootloaders like GRUB.
- The assembly code (`boot.s`) defines key functionalities:
    - Multiboot header with magic numbers, flags, and checksum.
    - A stack setup to ensure proper memory management before the kernel executes.
    - Transfers control to the kernel's `kernel_main()` function.
- Utility functions in `boot.s` like `read_port`, `write_port`, and `load_idt` are critical for initializing the hardware and setting up the interrupt descriptor table (IDT).

---

2. Kernel

The kernel forms the core of the operating system and is structured into several subsystems:

Kernel Initialization

- The entry point of the kernel is the `kernel_main()` function.
- Performs key initialization steps:
    - Clears the screen using `clear_screen()`.
    - Prints a welcome message via `print_string()` for basic debugging feedback.
    - Initializes the IDT and keyboard drivers.

Interrupt Handling

- Implements low-level hardware interrupt management.

- **Interrupt Descriptor Table (IDT)**:
    - A key structure that maps hardware interrupts (like keyboard input) to their respective handlers.
    - Populated during initialization with entries like the keyboard interrupt handler (`keyboard_handler`).
- **Programmable Interrupt Controller (PIC)**:
    - Remaps the PIC to avoid conflicts with CPU exceptions.
    - Masks all interrupts initially and selectively enables them, such as IRQ1 for the keyboard.

Device Drivers

1. **Keyboard Driver**:
    - Uses PS/2 keyboard communication.
    - Relies on hardware interrupts via IRQ1 to capture key presses.
    - The `keyboard_handler_main()` function:
        - Reads the keyboard's status and data ports.
        - Uses a keymap (`keyboard_map`) to translate scancodes into readable characters.
        - Prints characters to the screen or handles special keys like `Enter`.
2. **Screen Driver**:
    - Handles direct manipulation of video memory at address `0xb8000`.
    - Functions include:
        - `clear_screen()` to initialize the display.
        - `kernel_print()` to write strings to the screen.
        - `kernel_print_newline()` to move to the next line of output.

---

3. Build and Testing Environment

- **Toolchain**:
    - The kernel is compiled using the **x86-64-elf** cross-compiler.
    - This ensures the output binary is tailored for the target architecture.
- **Emulation**:
    - The OS is tested using **QEMU**, a lightweight and versatile emulator.
    - GRUB is used as the bootloader to load the kernel.

Features and Capabilities

1. **64-Bit Architecture**:

    - Designed specifically for x86-64 systems, utilizing the extended registers and memory capabilities of 64-bit processors.
2. **Interrupt-Driven Design**:
    - Efficient handling of hardware events (e.g., keyboard input) through IRQs and the IDT.
    - Modular interrupt handling with clear separation between hardware setup and driver logic.
3. **Basic Input and Output**:
    - Direct keyboard input handling with real-time display output.
    - Efficient use of hardware ports for communication with the keyboard and screen.
4. **Modular and Expandable Structure**:
    - Subsystems for drivers, interrupt handling, and kernel logic are modular, allowing easy integration of additional features like mouse support or advanced file systems.

Code Highlights

Bootloader (`boot.s`)

- Initializes the system and sets up the stack.
- Defines utility functions like `read_port` and `write_port` for hardware communication.
- Passes control to the kernel while enabling interrupts via the `load_idt` function.

Kernel (`kernel.c`)

- Initializes key structures like the IDT and drivers.
- Provides core functions like `print_string()` and `clear_screen()` for basic debugging and interaction.

Keyboard Driver

- Captures scancodes from the PS/2 keyboard and translates them into readable characters.
- Handles special keys like `Enter` for interaction.
- Demonstrates low-level device driver programming.

Shell currently in the works

The shell implementation is a lightweight command-line interface designed to interact with the system at a basic level. It supports simple commands such as hello and clear, allowing users to interact with the kernel in a straightforward manner. Input is managed using an input_buffer that collects user keystrokes, and commands are processed once the user presses the Enter key. The shell outputs text to the screen using a text buffer mapped to video memory at 0xB8000,

ensuring compatibility with x86-64 architecture's text mode. A prompt (root@RT-DOS:/) is displayed to signal readiness for user input. While functional, the shell remains in its beta stage, requiring enhancements in areas like screen scrolling, cursor handling, and support for richer text formatting to provide a more robust and user-friendly experience.