Course: Individual Project 2

Topic: "MyOS"

Student's name: Artjoms
Student's surname: Voroncovs
St. code: st83663

RIGA

TABLE OF CONTENTS

Introduction

The purpose of this project is to create a customized 32-bit Operating System (OS) from nothing using C++ and Assembly for an x86 platform. The project is a good example of applying system programming and OS architecture in action on a code level.

Everything, including memory segmentation, interrupt processing, device drivers, graphics mode initialization, and GUI, was coded by hand from available literature and web tutorials (All references are listed in Appendix A).

A system of such a type would call for a good deal of understanding of computer architecture and how operating systems deal with process management, memory, hardware, and user interface.

Prerequisites

Before starting kernel development, several prerequisites had to be fulfilled to be sure a correct and controlled kernel build environment. The most important prerequisite was the creation of a cross-compilation toolchain which is needed in order to build system independent executable.

Development Platform

Development for the MYOS kernel was performed on Ubuntu Linux distributive, which served as the host operating system. Ubuntu was selected due to its stability and simplicity.

The Ubuntu environment was used for:

- Building the i686-elf cross-compilation toolchain.
- Compiling the kernel and all supporting modules.
- Managing build scripts and Makefiles.
- Running the QEMU emulator for kernel testing.
- Executing Tiny Linux inside QEMU for disk image preparation (Formatting, creation of file system, files, directories).

Cross-Compiler Setup (i686-elf) and why cross-compiler setup is needed ?

OS developer need to use a cross-compiler if development is done on operating system A for some operating system B. The compiler must know the correct target platform (CPU, operating system), otherwise program won't work. OS developer may be able to use the compiler that comes with system A if developer pass a number of options to beat it into submission, but this will create a lot of completely unnecessary problems.

It is possible ask compiler what target platform it is currently using by calling the command:

gcc -dumpmachine

If development is done on 64-bit Linux, then response will be 'x86_64-unknown-linux-gnu'. This means that the compiler thinks it is creating code for Linux. If developer use this GCC to build handwritten kernel, it will use Linux system libraries, headers, the Linux libgcc, and it will make a lot of problematic Linux assumptions. If developer use a cross-compiler such as i686-elf-gcc, then a response back will be - such as 'i686-elf'. That means the compiler knows it is doing something else and it is possible to avoid a lot of problems easily and properly.

The cross-compiler was built manually from source using Binutils and GCC, following established OS development practices:

1. A dedicated installation prefix was defined (e.g. $HOME/opt/cross) to avoid interfering with the system compiler.
2. Binutils (assembler, linker, etc.) were configured and compiled for the i686-elf target with sysroot support.
3. GCC was built with only the C and C++ frontends enabled, without system headers or standard libraries (--without-headers).
4. Only the required components were installed (gcc, libgcc, and libstdc++), sufficient for freestanding kernel development.

The resulting compiler (i686-elf-gcc) is capable of compiling freestanding C/C++ code using the -ffreestanding flag, which is required for kernel development.

This step provided a reliable and reproducible build environment and served as the foundation for all further development stages.

1. System architecture and main classes

This section describes the main elements of the developed operating system, including the boot process, interrupt handling, device drivers, tty, shell, and the graphics subsystem. The OS uses a modular architecture and written in object oriented style, where each component is isolated and described as class. Communication of components defined via public fields and methods of a class.

1.1. Boot Process and Entering Protected Mode

The OS begins execution with a user-defined boot.S code written in x86 Assembly. The GRUB bootloader loads this code first, it performs critical functions such as setting up stack pointer which is needed for high level language as C++ and also calling global constructors (which is needed in order to use some advanced C++ features), after that boot.S calling kernel_main which is entry point to C++ generated code.

Once kernel_main is called, one of first things which is done by C++ code is initialization of GDT as well as IDT, that is mainly needed for transitioning to kernel protected mode.

Transitioning to protected mode is necessary to gain access to 32-bit addressing and modern processor capabilities such as paging.

The initialization kernel stage performs the following operations:

1. Loads the global descriptor table (GDT).

2. Activates the A20 line to access full memory.

3. Switches the processor to protected mode.

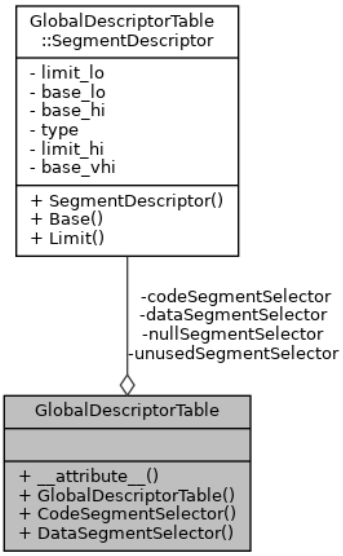
1.2. Global Descriptor Table (GDT)

The GDT defines OS memory segmentation, providing separation between code, data, and other memory areas.

Each GDT entry (segment selectors) contains a base address, a segment boundary, and access flags. The GDT structure follows the generally accepted OSDev schema such as: [GDT Element: Base + Limit + Access Byte].

For this OS GDT is initialized with null segment, unused segment, code segment and data segment selectors.

This implementation ensures that kernel will be able to change processor mode from real to protected, which ensures that kernel code will be launched in 32-bit mode.



Pic. 1.1 Global Descriptor Table and Global Descriptor Table Segment descriptor class

1.3. Interrupt Handling and the IDT

The Interrupt Descriptor Table (IDT) contains handlers for processor exceptions and hardware interrupts. The interrupt mechanism is a key part of any OS, as it ensures immediate response to hardware events such as keyboard presses or timer ticks.

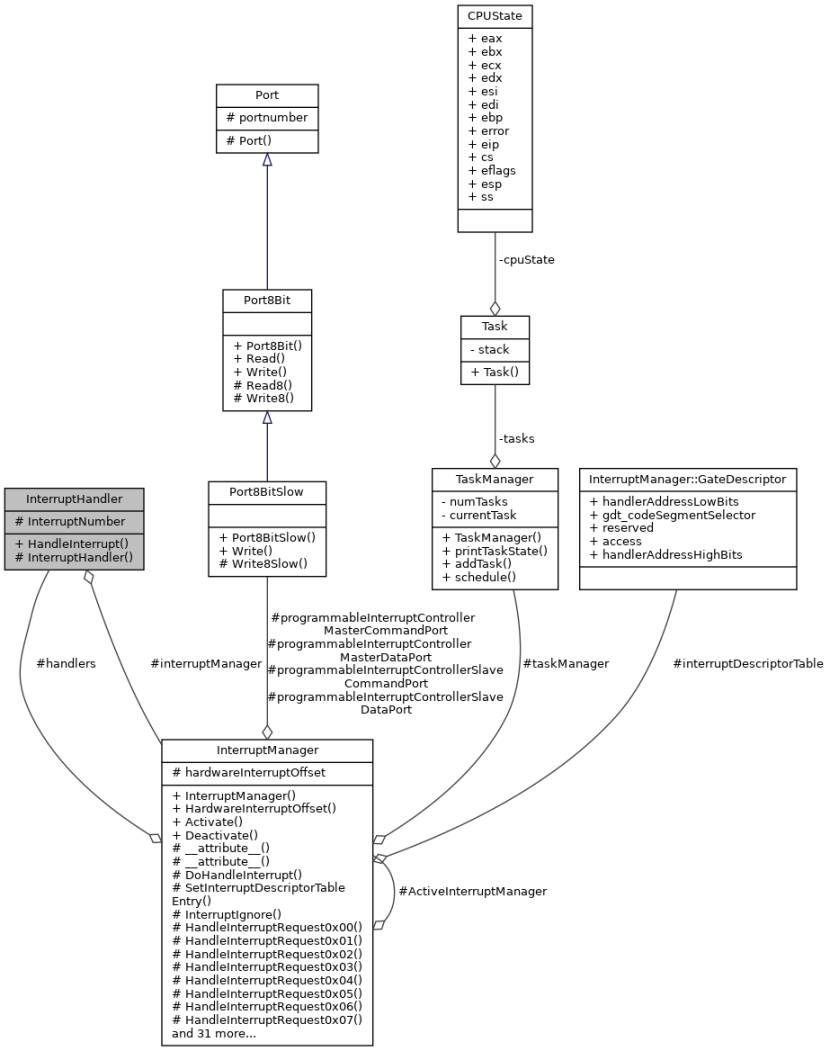
The OS implements:

* CPU exception handlers

* Hardware IRQ handlers

* PIC interrupt controller remapping

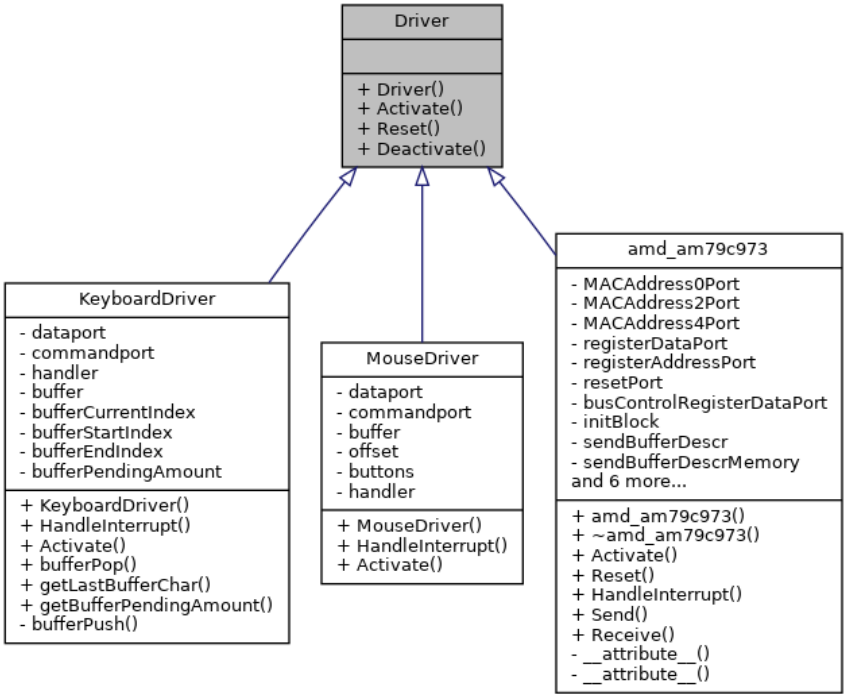
* Common interrupt prologue (Assembly) + ISR dispatcher in C++



Pic. 1.2 Interrupt handling related classes

1.4. Device drivers and driver manager

Drivers provide communication between the hardware and the OS kernel. This system implements several drivers.



Pic. 1.3 Driver classes

Driver Manager Component Architecture and Functionality

The Driver Manager offers a centralized system for registering and activating hardware drivers within the operating system. Its main role is to keep track of driver instances and oversee their lifecycle during system startup.

Drivers are stored in a fixed-size array internally, and new drivers are added via the AddDriver() method. The manager itself does not handle driver discovery; instead, it depends on other subsystems (like the PCI controller) to create and register driver instances.

After all necessary drivers are added, the ActivateAll() method goes through each registered driver and calls its Activate() method. This approach ensures hardware initialization occurs in a predictable and orderly fashion.

Overall, the Driver Manager functions as a lightweight coordinator between hardware discovery components and drivers. Its simple design makes it suitable for kernel development and educational operating systems.

Driver Base Class Architecture and Functionality

The Driver base class provides a standard interface and lifecycle framework for all system drivers. It acts as an abstraction layer, enabling the OS to manage different hardware devices uniformly.

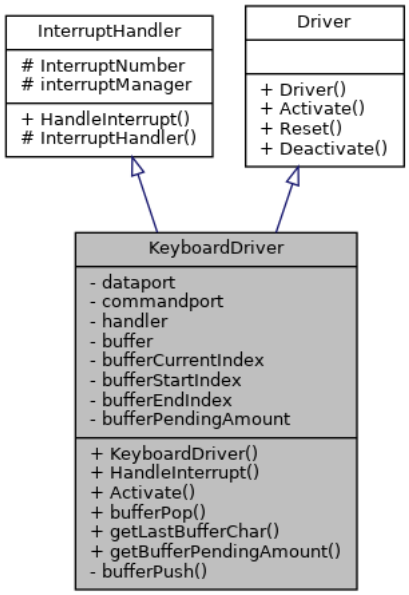
The class includes virtual methods such as Activate(), Reset(), and Deactivate(). Derived classes override these to handle device-specific initialization, resetting, and shutdown procedures.

Using a common base class allows the system to manage diverse drivers through a uniform interface, supporting polymorphism and easing driver management within the kernel.

In summary, the Driver base class sets a standardized contract for device drivers, promoting modularity, extensibility, and clear separation between driver management and hardware-specific code.

1.4.1. Keyboard driver

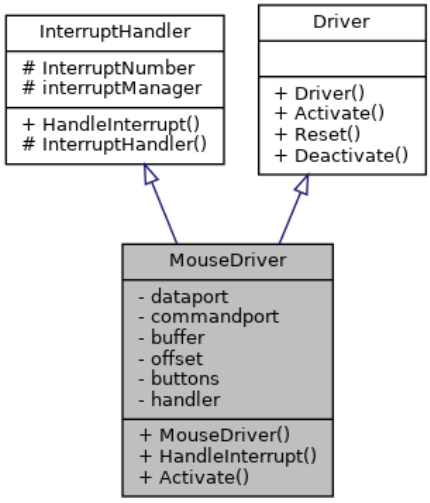
The driver uses PS/2 interrupts. Scan codes are read from port 0x60 and converted to ASCII using a mapping table.



Pic. 1.4 Keyboard driver class

1.4.2. Mouse Driver

The mouse driver initializes the PS/2 controller, updates cursor coordinates, and provides button processing for the GUI.

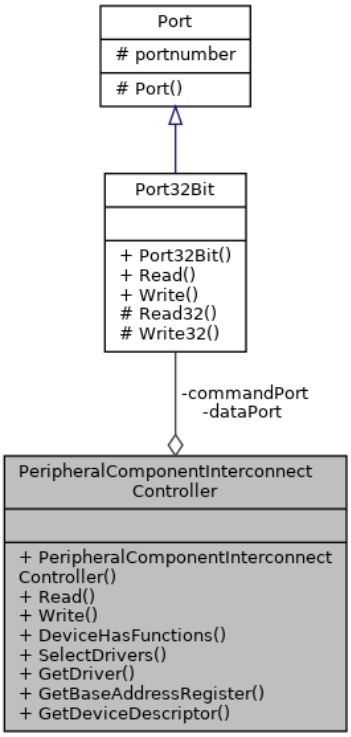


Pic. 1.5 Mouse driver class

1.4.3. PCI Bus Scanning

The PCI controller manages hardware discovery and driver selection for PCI devices within the OS. It scans the PCI bus, finds connected devices, and initializes the correct drivers. It communicates directly with PCI hardware through standard configuration I/O ports (0xCF8 and 0xCFC). It performs low-level read and write operations by creating PCI configuration addresses and retrieving register values. Device enumeration involves going through all supported buses, devices, and functions. It identifies whether a device supports multiple functions and collects device descriptors, which include vendor ID, device ID, class info, revision, and interrupt line. Devices with invalid or unused vendor IDs are filtered out early. For each valid device, the controller reads its Base Address Registers (BARs) to find I/O or memory resources, storing I/O addresses in the device descriptor for driver use. Driver selection matches vendor and device IDs. When a device is detected, the system allocates and creates the appropriate driver and registers it with DriverManager. Interrupt handling supports passing the InterruptManager to driver constructors. The component also has optional logging for debugging and inspection, which provides detailed PCI device information during system startup. Overall, it provides a central, flexible hardware discovery system that separates low-level PCI configuration access

from driver management. This design allows for automatic driver binding and supports modular, scalable kernel-level device management in an educational OS.



Pic. 1.6 PCI driver class

1.5. Graphics Subsystem and GUI (VGA Mode)

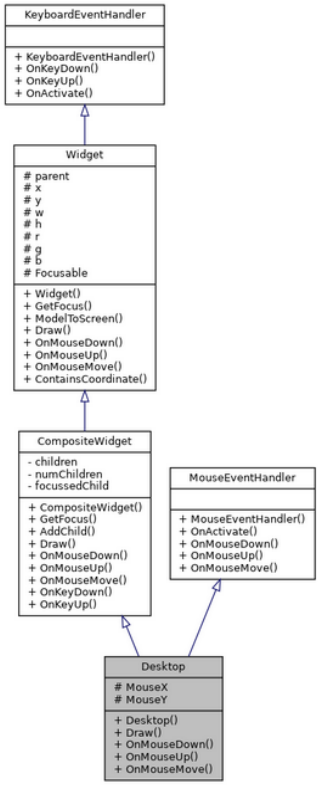
The OS uses VGA Mode 13h (320x200, 256 colors) with direct access to the framebuffer.

Rendering is performed by writing bytes to the memory location 0xA0000.

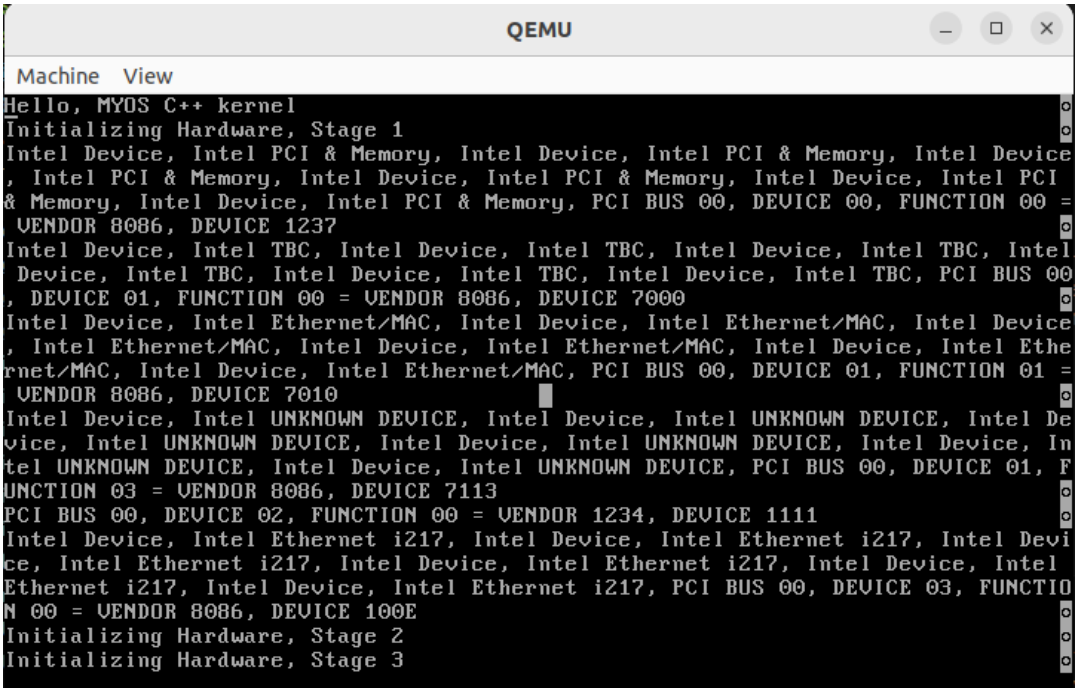
The graphics subsystem supports:

- Window creation
- Window dragging
- Mouse cursor drawing
- Primitive events system
- Text rendering in windows

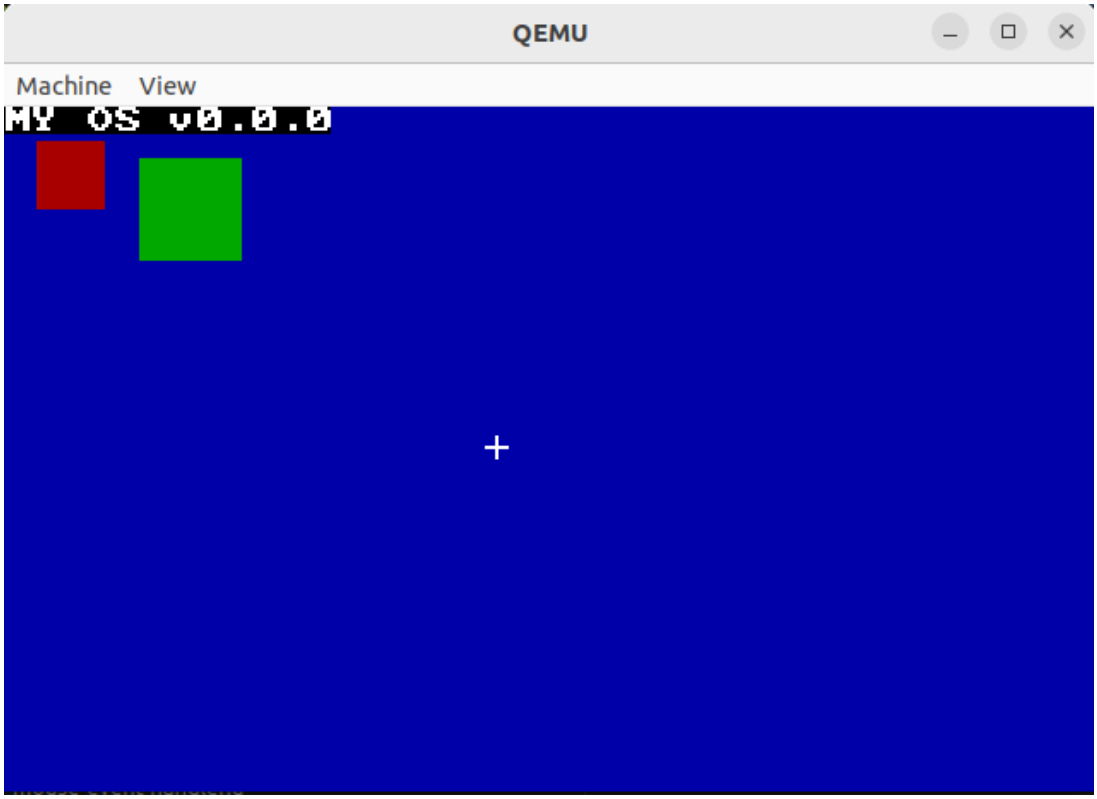
The VGA component offers a low-level graphics output system for the OS, enabling direct access to VGA hardware at 320×200 resolution with 8-bit color (256 colors) for pixel-based graphics. It communicates with VGA hardware via I/O ports like the sequencer, CRT controller, graphics controller, and attribute controller, using dedicated port objects to configure registers and control display modes. During setup, it writes a specific register sequence to set the hardware to 320×200×256 mode. The supportsMode() and setMode() methods ensure only supported resolutions and color depths are used, providing safe mode selection. Pixel rendering is done by writing directly to the framebuffer, with the graphics controller determining the memory segment. The PutPixel() method calculates the address, writes a color index, and performs boundary checks. To ease drawing, the component uses an off-screen buffer (screen); higher-level functions modify this buffer, and RenderScreen() updates the actual VGA memory, reducing direct hardware access. Color is managed through indexed palette, mapping RGB values via a lookup for predefined basic colors. Overall, this VGA component delivers a basic, hardware-oriented graphics abstraction suitable for kernel-level use and educational OS projects, balancing direct register manipulation with simple buffering for effective basic graphics rendering.



1.6 GUI system class relations



Pic. 1.7. Text mode (Basic I/O)



Pic. 1.8. VGA 320×200 graphics mode

1.6 TTY

The Terminal component is a low level text output subsystem program based on VGA text mode. Its purpose is to provide basic screen output to user, cursor control, and character rendering for the kernel and higher level components for example the shell.

The terminal directly write to VGA video memory which location is at physical address 0xB8000, the screen is treated as a fixed-size text grid of 80 columns by 25 rows, where each cell contain a character and its color attribute. Internal state of variables track current cursor position using terminal_row, terminal_column variables.

Initialization is performed by subroutine terminal_initialize(), which resets the cursor, sets a default color, and clears the whole screen by filling the VGA buffer memory with space characters - this ensures a known and clean display state at boot time.

Character output is handled by terminal_putchar() subroutine, terminal supports basic control characters, including newline (\n) and backspace (\b). For printable characters, the terminal writes directly to VGA memory at the current cursor position. After each character, the cursor position is updated , including automatic line wrapping. When the screen limit is reached, the terminal clears the screen and resets the cursor, providing a simple overflow handling mechanism.

The component also controls the hardware cursor by writing to VGA I/O ports (0x3D4 and 0x3D5) using inline assembly. This ensures that the visible cursor position matches the internal software state.

Higher-level output functions such as terminal_write() and terminal_writestring() provide buffered output by repeatedly calling terminal_putchar(). The terminal_clear_screen() function allows the screen to be cleared explicitly, which is useful for shell resets or screen refresh operations.

Overall, this terminal component provides a minimal, hardware-oriented text output interface suitable for kernel-level development. It abstracts direct VGA memory and port access behind simple functions while remaining lightweight and deterministic, making it appropriate for educational operating system projects.

1.7 TTY based Shell

The Shell component implements a basic command-line interpreter for a custom operating system. Its primary role is to read user input from the keyboard, parse commands, and execute corresponding filesystem-related operations.

The shell is initialized with two core dependencies: a KeyboardDriver, responsible for buffered keyboard input, and FATVFS, which provides access to the FAT-based virtual file system. These dependencies are injected via the constructor, keeping the shell independent of concrete implementations.

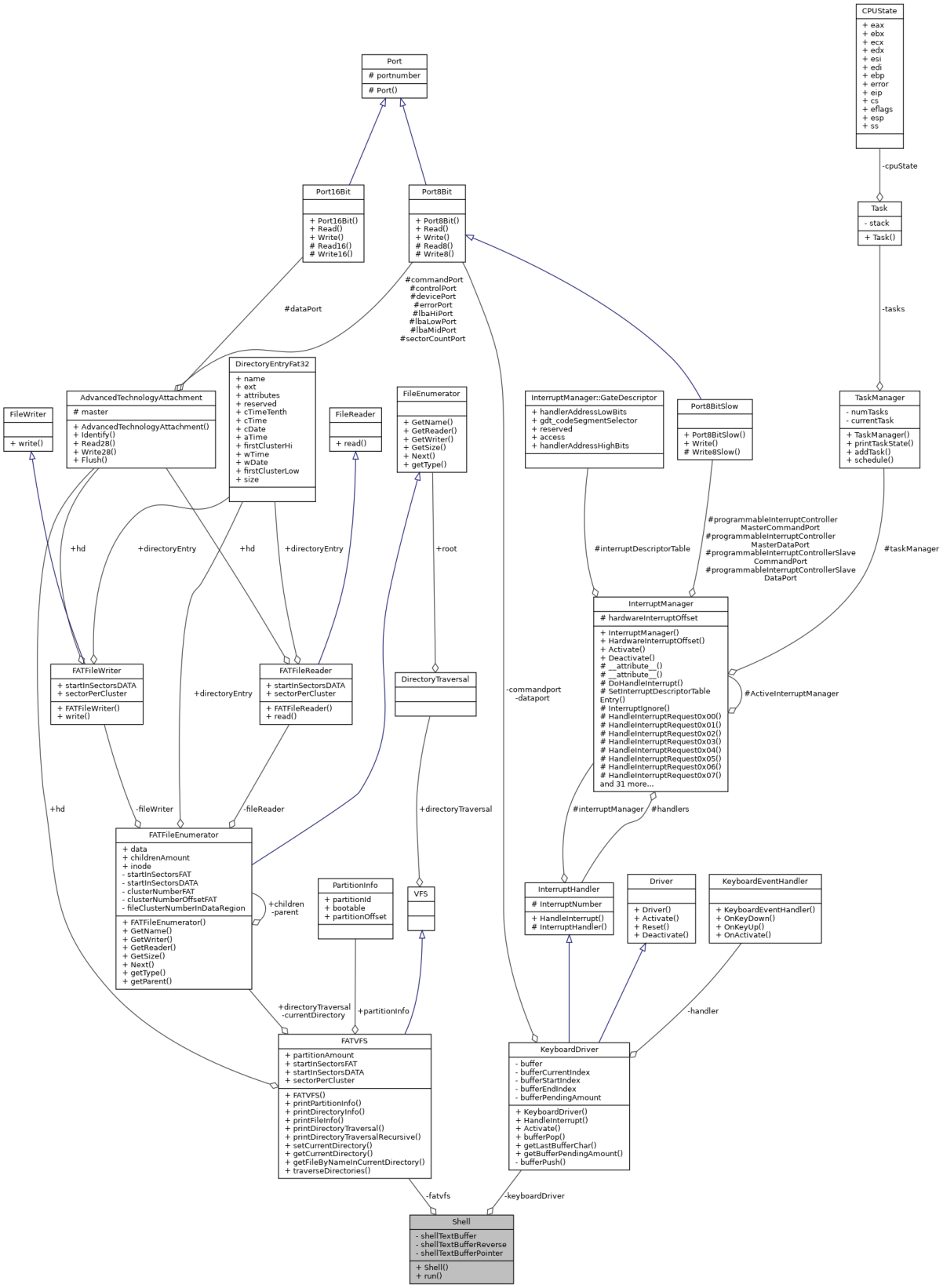
The main logic is executed inside an infinite loop in the run() method. The shell continuously checks the keyboard buffer and processes input only when the ENTER key ('\n') is detected. This design reflects typical OS shell behavior, where the shell is always active and event-driven.

Input is read from the keyboard buffer in reverse order due to the buffer's stack-like nature. The shell reconstructs the command line manually, including handling backspace characters ('\b') to support basic line editing. The final command string is null-terminated and stored in an internal buffer.

Command parsing is implemented using a simple whitespace-based approach. The shell extracts the command name and up to two arguments using fixed-size character arrays and state flags. This parser is intentionally minimal and does not support advanced features such as quoting, piping, or variable-length arguments.

Command execution is performed through string comparison against known commands (ls, cat, cd, write, pwd). Each command is implemented as a separate class and exposes a run() method. The shell creates the appropriate command object and delegates execution to it, passing the required arguments and the filesystem reference.

Overall, this shell provides a minimal, modular, and low-level command execution environment suitable for educational operating system development. Its design emphasizes simplicity, deterministic behavior, and clear separation between input handling, parsing, and command logic, while leaving room for future extensions such as scripting or advanced parsing.



Pic. 1.9 Shell class relations

2. OPERATING SYSTEM FEATURES

This section lists the implemented operating system features.

2.1. Main implemented features

The system currently provides:

- Protected mode transition
- GDT and IDT initialization
- Keyboard and mouse drivers
- PCI scanning
- Graphics rendering to VGA
- GUI framework with support of windows, simple text and multiple colors.
- Shell
- Software for shell such as pwd, ls, cat, write, cd
- ATA device support
- Simple FAT32 filesystem implementation

This functionality demonstrates the full path from low-level boot to the graphical interface/shell.

2.2 Keyboard and mouse drivers

The operating system includes low-level keyboard and mouse drivers that provide basic input handling. The keyboard driver captures key press and release events, translates scan codes into characters, and buffers input for higher-level components such as the shell. The mouse driver processes movement and button events, enabling basic pointer interaction required by graphical user interface components. Both drivers operate through hardware interrupts to ensure responsive and efficient input processing

2.3 PCI scanning

The system implements PCI bus scanning to automatically detect connected hardware devices. The PCI controller enumerates available buses, devices, and functions, reads configuration space registers, and identifies devices using vendor and device identifiers. Based on this

information, suitable drivers are selected and registered with the driver manager. This mechanism enables modular hardware support and automatic driver initialization at boot time.

2.4 Graphics rendering to VGA

Graphics output is supported through direct interaction with VGA hardware. The system configures the VGA controller into a 320×200 resolution with 256 colors and provides pixel-level rendering functionality. Drawing operations write directly to the VGA framebuffer, enabling basic graphical output without relying on external libraries. This feature forms the foundation for graphical user interface components.

2.5 GUI framework with support of windows, simple text and multiple colors.

A simple graphical user interface framework is implemented on top of the VGA graphics subsystem. It supports multiple windows, basic text rendering, and the use of different colors. The framework manages screen regions and rendering order, allowing graphical elements to coexist on the screen. While minimal, it provides essential functionality for visual interaction and demonstrates the integration of graphics rendering with higher-level UI abstractions.

2.6 ATA device support

The operating system includes basic support for ATA storage devices. The ATA driver enables communication with disk hardware using programmed I/O operations, allowing sectors to be read from and written to storage media. This functionality serves as the foundation for persistent data storage and higher-level filesystem support.

2.7 Simple FAT32 filesystem implementation

A simplified FAT32 filesystem implementation is provided to enable file and directory management on ATA storage devices. The filesystem supports basic operations such as directory traversal, file reading, and file writing. It abstracts low-level disk access and exposes a virtual filesystem interface to higher-level components such as the shell, enabling user-level interaction with persistent data.

2.8 Shell

Shell is interface layer for OS which allow user to communicate with operating system via command line interface or CMD in short.

MYOS Shell support several build in software solutions in order to communicate with filesystem of OS.

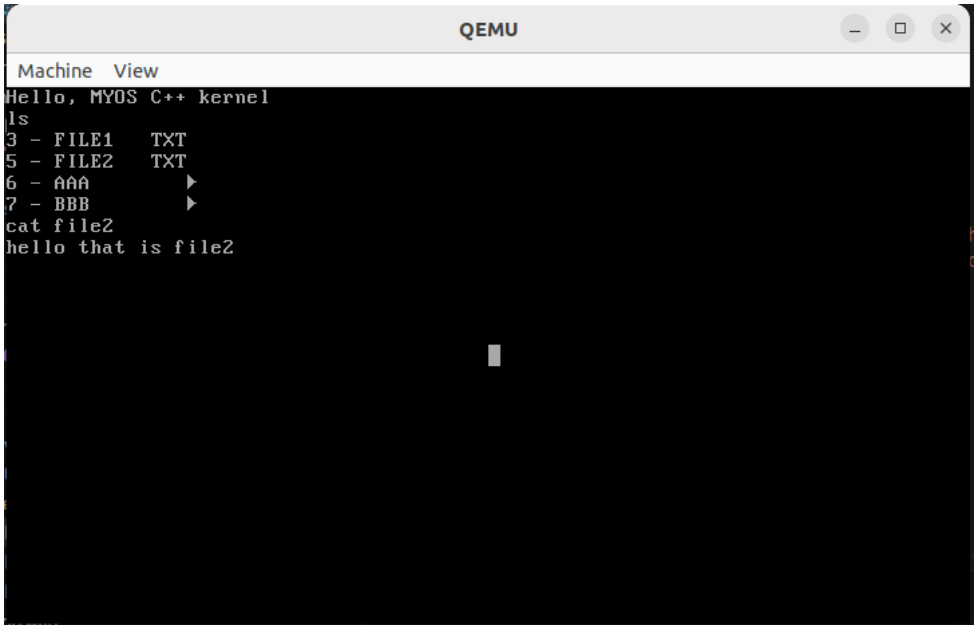
2.9 Software for shell such as pwd, ls, cat, write, cd

Here is a list of build in software with short explanation of each one:

- Pwd - Stands from “print working directory”, and that prints path where user located relative to root directory.
- Ls - Stands from “list”, that command lists current directory file and directory names.
- Cat <arg> - In short from “concatenate”, this command needs argument <arg1> which should be file name. It reads a file and prints contents of a file.
- Write <arg1> <arg2> - Writes text to a file which is passed as <arg1>, text is passed as <arg2>/
- Cd <arg1> - Stands from “change directory”, <arg1> is accepted which should be directory name to which user want to enter.

Pic 2.1 Shell startup

Pic 2.2 Shell ls command



Pic 2.3 Shell cat command

Pic 2.4 Shell ls command

Pic 2.5 Shell cd command

Pic 2.6 Shell pwd command

Pic 2.7 Shell cd + pwd command

Pic 2.2 Shell write + cat commands

2.3. Testing and Verification

The OS was tested on:

- QEMU - the main debugging environment
- Bare metal - Lenovo Thinkpad T440 (with legacy boot enabled)

Tests performed:

- Interrupt validation testing
- Memory access robustness testing
- Driver operation testing
- Graphics operation testing
- Window operation and mouse interaction
- Shell commands testing

3. Key Algorithms Implemented in the FAT32 Module

3.1 Filesystem Layout Derivation from the BIOS Parameter Block (BPB)

The module derives the complete physical layout of the FAT32 filesystem using metadata stored in the BIOS Parameter Block. From this structure, it calculates the start of the FAT area, the data region, and the root directory location. This algorithm is essential because it shows how FAT32 volumes are self-describing and how logical filesystem structures are mapped to physical disk sectors.

3.2 Directory Traversal with Metadata Filtering

The root directory is scanned linearly by reading directory entries from disk. During traversal, invalid entries and long file name records are skipped, while file and directory entries are identified using attribute flags. This algorithm demonstrates how FAT32 directories are parsed and how filesystem metadata is interpreted at a low level.

3.3 FAT Cluster Chain Traversal for File Reading

File contents are accessed by following a chain of clusters stored in the FAT table. Starting from the first cluster, the algorithm reads data cluster by cluster and uses FAT entries to locate the next cluster in the chain. This process continues until the entire file is read, illustrating the core FAT32 mechanism of linked disk allocation.

4. Development prospects

The OS was created as an extensible educational platform. The following extensions are planned:

4.1. Multitasking

Planned implementation of:

- Task scheduler with context switching

4.2. Network stack (basic)

Planned:

- Ethernet driver
- Frame forwarding
- Partial implementation of POSIX-compatible system calls

4.3. Full-fledged network stack

At an advanced stage:

- Ethernet
- ARP
- IPv4
- ICMP
- UDP
- TCP
- A simple HTTP server

4.4. File Systems

Next Tasks:

- Create file FAT32
- Delete file FAT32
- Create file VFS
- Create file VFS
- Write large amount of text to files (cluster spread files)
- Polishing VFS
- Implementation of some other more advanced filesystem
- Analogue of mkfs, fdisk

Conclusion

This project developed a functional operating system, from level 0 Boot.S assembly program to the graphical desktop environment. Memory segmentation mechanisms, interrupt handling, device drivers, a graphics subsystem, and a window manager were implemented.

The project demonstrates a deep understanding of systems programming and OS architecture and provides a foundation for further research in the areas of multitasking, networking, and file systems.

References

- Intel 64 and IA-32 Architectures Software Developer's Manual - https://cdrdv2-public.intel.com/825743/325462-sdm-vol-1-2abcd-3abcd-4.pdf
- OSDev Wiki (2023) - https://wiki.osdev.org/Expanded_Main_Page
- OS Development Series - https://www.youtube.com/watch?v=1rnA6wpF0o4&list=PLHh55M_Kq4OApWScZyPl5HhgsTJS9MZ6M
- Operating Systems: Three Easy Pieces - https://pages.cs.wisc.edu/~remzi/OSTEP/
- Structured Computer Organization - https://www.amazon.com/Structured-Computer-Organization-Andrew-Tanenbaum/dp/0132916525

Abbreviated statements

OS – Operating System

GDT – Global Descriptor Table

IDT – Interrupt Descriptor Table

ISR – Interrupt Handler

IRQ – Interrupt Request

PCI – Peripheral Component Interconnect

PS/2 – PS/2 Hardware Interface

MBR – Master Boot Record

VGA – Video Graphics Array

CPU – Central Processing Unit

API – Application Programming Interface