Advanced Operating Systems - HamedOS

University of Agder, Spring 2025, IKT218

Introduction

When we turn on our computers, most of us don't think about the complex sequence of events that occurs before we see our desktop. As I've learned through developing HamedOS in this course, there's an intricate boot process happening behind the scenes. This report explores the various stages of this process, from the initial Power-On Self-Test to the loading of the operating system kernel. I'll also compare how this process differs between physical machines and virtual environments, and how modern operating systems have evolved their boot processes. While I'll occasionally reference my own implementation experience with HamedOS for context, this report primarily focuses on the general concepts and principles of operating system boot processes.

Power-On Self-Test (POST)

The first thing that happens when you press the power button on your computer is the Power-On Self-Test, or POST. I find it fascinating that before any operating system code runs, this fundamental hardware check takes place. The POST is essentially a diagnostic testing sequence that the computer's firmware (either BIOS or UEFI) runs to ensure that all the hardware components are functioning correctly.

During the POST, several critical checks are performed:

1. The CPU is tested to ensure it's functioning properly
2. The system memory (RAM) is counted and checked for errors
3. Graphics cards, storage devices, and other essential hardware are detected and initialized
4. Basic input/output systems are verified

What I found particularly interesting is how the POST interacts with the hardware. It doesn't just passively check if components exist - it actively communicates with them using low-level commands. For example, when testing RAM, the POST writes specific patterns to memory and reads them back to verify data integrity. If any critical components fail these tests, the boot process stops, and the system typically emits a series of beep codes or displays error messages to indicate what went wrong.

While working on HamedOS, I never directly interacted with the POST since it happens before my code runs, but understanding it helped me appreciate why certain memory addresses are reserved and why hardware is in specific states when my code begins execution.

Boot Sequence Post-POST

After the POST successfully completes, the system firmware (BIOS or UEFI) takes control and begins the next phase of the boot process. This is when things start to get more interesting from a software perspective.

In a traditional BIOS system, the following sequence typically occurs:

1. The BIOS looks for bootable devices according to the boot order configured in its settings. This could be a hard drive, SSD, DVD drive, USB drive, or network device.
2. When it finds a bootable device, it reads the first sector (usually 512 bytes) from the device into memory at address 0x7C00. This first sector is known as the Master Boot

Record (MBR) and contains both a small program (the primary bootloader) and the partition table.

3. The BIOS then executes this primary bootloader, which has the responsibility of loading a more sophisticated bootloader from the active partition.

In contrast, UEFI (which is newer and more advanced) works quite differently:

1. UEFI can read file systems directly and doesn't rely on the 512-byte limitation of the MBR.
2. It looks for EFI System Partition (ESP) on storage devices, which contains bootloader applications with the .efi extension.
3. UEFI loads and executes the specified bootloader application directly from this partition.

I learned that UEFI offers several advantages over BIOS, including support for drives larger than 2TB, faster boot times, better security features through Secure Boot, and a more modern, programmable environment. When developing HamedOS, I targeted the Multiboot specification, which is predominantly used with BIOS systems, but the principles are similar - my kernel needs to be properly formatted so that a bootloader can recognize and load it correctly.

Bootloaders

Bootloaders fascinate me because they bridge the gap between the hardware-focused firmware and the software-focused operating system. They're small programs with a huge responsibility: loading the operating system kernel into memory and transferring control to it.

Different Types of Bootloaders

Through my research, I've found that there are several popular bootloaders, each with unique features:

1. GRUB (Grand Unified Bootloader): This is perhaps the most widely used bootloader in the Linux world. GRUB has evolved from GRUB Legacy to GRUB 2, which offers a more powerful scripting environment, support for more file systems, and greater flexibility. GRUB can load various operating systems and supports the Multiboot specification.
2. LILO (Linux Loader): An older bootloader for Linux systems. While less common today, it was notable for its simplicity and reliability, though it lacked the flexibility of GRUB.
3. systemd-boot (formerly gummiboot): A lightweight UEFI bootloader focused on simplicity and fast boot times. It only works with UEFI systems and is increasingly popular in modern Linux distributions.
4. Windows Boot Manager: Microsoft's bootloader for Windows systems, which has evolved significantly from its simpler origins to support features like multi-booting, BitLocker encryption, and Secure Boot.
5. U-Boot: Popular in embedded systems and devices like routers and single-board computers. It's highly configurable and supports multiple architectures.

Choosing a Bootloader

When selecting a bootloader for an operating system project, several considerations come into play:

1. Compatibility: Does it support the hardware architecture and firmware type (BIOS/UEFI) you're targeting?
2. Functionality needed: Do you need support for multiple operating systems, kernel command-line parameters, or loading initial ramdisks?
3. Complexity vs. simplicity: More powerful bootloaders offer greater functionality but may be more complex to configure and maintain.
4. Development activity: Is the bootloader actively maintained and updated for security issues?

For my HamedOS project, I chose to make it compatible with GRUB using the Multiboot specification. This was primarily because GRUB is widely available, well-documented, and simplifies the development process by handling many low-level details.

Challenges of Manual Bootloader Implementation

Implementing a bootloader from scratch would be an incredibly educational but challenging endeavor. From my understanding, the main challenges would include:

1. Size constraints: The first-stage bootloader must fit within tight size limits (often 512 bytes for MBR), requiring extremely efficient assembly code.
2. Limited environment: Initially, there are no libraries, no standard C functions, and often only 16-bit real mode available, with very limited stack space.
3. Hardware interaction: The bootloader must directly interact with hardware using BIOS/UEFI calls or direct port manipulation, which varies across systems.
4. Complex tasks: Despite these limitations, it must initialize hardware, switch CPU modes (from real mode to protected mode or long mode), set up a basic memory map, and load the kernel from storage.
5. Debugging difficulty: Debugging bootloaders is notoriously difficult due to the low-level nature and the fact that traditional debugging tools aren't available.

While I didn't implement a bootloader from scratch for HamedOS, even making my kernel Multiboot-compliant required understanding the constraints and expectations that bootloaders have for the kernels they load.

Memory Layout in the Boot Process

The memory layout during booting is a fascinating aspect of OS development that has historical roots and practical implications. Through my studies and the development of HamedOS, I've learned how this layout influences the entire system.

Standard Memory Layout in i386 Systems

In the i386 architecture, the memory layout during boot has several notable characteristics:

1. First 1MB (0x00000 - 0xFFFFF): This region has special significance due to historical reasons. It contains:
   - The Interrupt Vector Table (0x00000 - 0x003FF)
   - BIOS data area (0x00400 - 0x004FF)
   - The Original bootloader location (0x7C00 - 0x7DFF)
   - Extended BIOS data area (often near 0x9FC00)
   - Video memory (0xA0000 - 0xBFFFF for graphics modes, with text mode at 0xB8000)

- BIOS ROM (0xE0000 - 0xFFFFF)

2. The 0x10000 (64KB) mark: This address is particularly important as it's traditionally where many kernels are loaded. When developing HamedOS, I noticed that most tutorials and references suggested loading the kernel at 0x100000 (1MB) instead, which is just after the reserved first megabyte. This makes sense because:
   - It avoids conflicts with the reserved areas in the first MB
   - It aligns nicely with page boundaries (important for memory management)
   - It provides a clean separation between the bootloader and kernel spaces
3. Higher memory: Above 1MB, memory is generally available for the kernel, modules, and other system components. As physical memory grew beyond what 32-bit addresses could reference directly, various addressing schemes were developed to utilize this extended memory.

Implications for OS Loading

This memory layout has several important implications for the operating system loading process:

1. Relocation needs: If a kernel is compiled to run at a different address than where it's loaded, it must either be relocated by the bootloader or include position-independent code.
2. Memory map importance: The bootloader typically provides the kernel with a memory map indicating which regions are usable, reserved, damaged, or occupied by the ACPI tables. This is crucial for the kernel to set up its memory management system without overwriting important data.
3. Initial paging setup: On modern systems, the kernel often needs to set up paging structures early in the boot process to enable virtual memory addressing. The physical memory layout directly influences how these structures are initialized.
4. Constraints on kernel size: Depending on the memory layout and hardware constraints, there may be limits on how large a kernel can be or where it can be loaded without special handling.

When I implemented HamedOS, I specified in the linker script that the kernel should be loaded at 0x100000 (the 1MB mark). This choice was influenced by these standard practices and helped avoid the complexity of dealing with the reserved areas in lower memory.

Boot Process in Modern Operating Systems

Modern operating systems have evolved sophisticated boot processes that go well beyond the basic loading of a kernel. Looking at how commercial systems handle booting has been enlightening during my OS development journey.

Linux Boot Process

In Linux, the boot process typically follows these steps:

1. BIOS/UEFI initialization and bootloader loading
2. Bootloader execution: Usually GRUB, systemd-boot, or another bootloader that presents a menu and loads the selected kernel
3. Kernel loading: The kernel (e.g., vmlinuz) is loaded into memory

4. Initial RAM disk (initrd/initramfs): A temporary root file system is loaded into memory, containing essential drivers and tools needed to mount the actual root filesystem
5. Kernel initialization: The kernel initializes hardware, mounts the real root filesystem, and starts the init process (either SysVinit, Upstart, or more commonly today, systemd)
6. User space initialization: The init system starts system services according to the runlevel or target

Linux has evolved to optimize this process for faster boot times, using techniques like parallel service startup, lazy loading, and boot profiling.

Windows Boot Process

Windows has a significantly different approach:

1. UEFI/BIOS and bootloader: Windows Boot Manager loads
2. Windows OS Loader (winload.exe/winload.efi): Loads the Windows kernel and essential drivers
3. Kernel initialization: The Windows NT kernel (ntoskrnl.exe) initializes and loads the Hardware Abstraction Layer (HAL)
4. Session Manager initialization: smss.exe starts essential subsystems
5. Winlogon and Service initialization: User authentication and services start
6. Explorer shell: The user interface loads

Windows 10 and 11 have implemented fast startup features that use a hybrid shutdown/hibernation approach to accelerate the boot process.

macOS Boot Process

Apple's macOS has its own unique boot sequence:

1. BootROM: Hardware-specific initialization
2. iBoot: Apple's bootloader
3. XNU Kernel loading: The macOS kernel loads
4. launchd initialization: Similar to Linux's init but Apple-specific
5. System services and user interface: Login window and core services start

With Apple Silicon Macs, this process has been further optimized and secured with hardware-verified boot chains.

Evolution of Boot Processes

Having studied these systems, I've noticed several trends in how boot processes have evolved:

1. Security enhancements: Secure Boot, verified boot chains, and code signing have become standard to protect against boot-time malware
2. Speed optimizations: Parallel initialization, delayed loading of non-essential components, and hibernation/resume techniques have dramatically reduced boot times
3. Unified firmware interfaces: The move from BIOS to UEFI has standardized many boot-time operations across platforms

4. Modularity: Modern systems use modular design with dynamic loading of drivers and components as needed

These advancements make modern operating systems much more complex to boot than my simple HamedOS implementation, but the fundamental principles remain similar.

Virtual Machines and Booting

Developing HamedOS gave me firsthand experience with virtual machines, which I found fascinating due to the differences between booting a physical machine and a virtual one.

Main Differences in Boot Process

The primary difference in the boot process for virtual machines compared to physical machines is the layer of abstraction introduced by virtualization. In a physical machine, the firmware directly interacts with the actual hardware. In a virtual machine, a hypervisor mediates this interaction by presenting virtual hardware to the guest operating system.

Specific differences include:

1. Hardware initialization: In physical machines, the BIOS/UEFI performs actual hardware initialization. In VMs, the virtual firmware initializes virtual devices that are already pre-configured by the hypervisor.
2. Boot device selection: Physical machines scan actual storage devices connected to the system. VMs use virtual storage devices (often disk image files) that are configured in the hypervisor settings.
3. Memory detection: Physical machines detect and test actual RAM chips. VMs are allocated a pre-determined amount of the host's memory.
4. Device interaction: Once booted, a physical OS directly accesses hardware through ports and memory-mapped I/O. A VM's guest OS accesses virtual devices that the hypervisor translates into operations on the physical hardware.
5. Boot time: Virtual machines often boot faster than physical machines because the virtual hardware doesn't require the same level of initialization and testing.

Role of a Hypervisor

A hypervisor, I've learned, is the software layer that creates and manages virtual machines. It plays several crucial roles in the context of virtualization:

1. Resource management: It allocates physical system resources (CPU, memory, storage, network) to virtual machines and ensures they don't interfere with each other.
2. Hardware emulation/virtualization: It provides virtual hardware interfaces to the guest operating systems, either by emulating physical devices or by providing paravirtualized interfaces.
3. Isolation: It maintains separation between virtual machines to prevent one VM from accessing or affecting another's data or operations.
4. Execution control: It manages the execution of guest code, using various techniques (binary translation, hardware virtualization extensions, etc.) to run guest instructions safely.

There are two main types of hypervisors:

- Type 1 (bare-metal): Runs directly on the host's hardware (e.g., VMware ESXi, Microsoft Hyper-V, Xen)
- Type 2 (hosted): Runs on a conventional operating system (e.g., VMware Workstation, Oracle VirtualBox)

For HamedOS development, I used QEMU, which can operate as either type depending on configuration.

Challenges and Advantages of Virtualized Booting

Through my experience, I've encountered several challenges when booting in virtualized environments:

1. Device differences: Some virtual devices behave differently from their physical counterparts, which can cause compatibility issues.
2. Timing discrepancies: Virtual time may not flow at the same rate as physical time, especially under load, which can affect time-sensitive operations.
3. Performance overhead: There's always some performance penalty due to the additional layer of abstraction.
4. Limited hardware access: Some low-level hardware features may not be accessible or may be virtualized in ways that change their behavior.

On the other hand, virtual environments offer significant advantages for OS development:

1. Safety: Crashes in the guest OS won't affect the host system, making development much safer.
2. Snapshots and rollbacks: The ability to save VM state and roll back to previous points is invaluable for testing.
3. Faster development cycles: Rebooting a VM is much faster than rebooting physical hardware.
4. Environment consistency: Every developer can work with identical virtual hardware, eliminating "it works on my machine" issues.
5. Resource flexibility: Easy adjustment of memory, CPU cores, and other resources for testing different configurations.

Developing HamedOS in a virtual environment allowed me to experiment freely without risk to my physical hardware, which was especially valuable when implementing low-level features like memory management and interrupt handling.

Conclusion

Through my exploration of the boot process for this report and my experiences developing HamedOS, I've gained a deep appreciation for the complexity and elegance of how computers transition from powered-off hardware to functioning operating systems. The journey from the initial POST to a fully loaded operating system involves multiple stages, each building upon the previous one to gradually increase the system's capabilities.

I found it particularly interesting how the memory layout during boot influences OS design decisions, and how different operating systems have evolved their boot processes to meet modern demands for security, speed, and reliability. Additionally, understanding the

differences between booting physical machines and virtual machines has been valuable for my development process.

The bootloader, sitting at the intersection of hardware and software, emerged as a critical component that solves the chicken-and-egg problem of how to load an operating system before one is running. While I relied on existing bootloaders like GRUB for HamedOS, I now understand the considerable challenges that would be involved in creating one from scratch.

This exploration has given me a much more comprehensive understanding of the fundamentals that underpin all operating systems. Even though modern commercial operating systems like Windows, Linux, and macOS have far more sophisticated boot processes than my simple HamedOS implementation, they all build upon these same basic principles.

As computing continues to evolve with technologies like UEFI Secure Boot, Trusted Platform Modules, and increasingly complex virtualization solutions, the boot process will undoubtedly continue to develop as well. Understanding these foundations helps me appreciate both how far operating systems have come and the ingenious solutions that have been developed to overcome the fundamental challenge of bootstrapping a computer system.

References

[1] T. Shanley and D. Anderson, “ISA System Architecture,” 3rd ed. Addison-Wesley Professional, 1995.

[2] R. Hyde, “The Art of Assembly Language,” 2nd ed. No Starch Press, 2010.

[3] A. S. Tanenbaum and H. Bos, “Modern Operating Systems,” 4th ed. Pearson, 2014.

[4] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” 2021.

[5] J. Corbet, A. Rubini, and G. Kroah-Hartman, “Linux Device Drivers,” 3rd ed. O’Reilly Media, 2005.

[6] Unified EFI Forum, “Unified Extensible Firmware Interface Specification,” Version 2.9, 2021.

[7] O. Kiddle, R. Perryman, and S. Langley, “QEMU: A Purely Software Approach to Emulating Hardware,” in Proc. of the Ottawa Linux Symposium, 2006.

[8] B. Smith, “ARM System Developer’s Guide: Designing and Optimizing System Software,” Morgan Kaufmann, 2004.