Boot-Popcorn Report

Zach Taylor, Michael Seeley, Ben Bushnell

Proof of Concept
CSCI 460
December 12, 2019

Introduction

This is the technical report for the Boot-Popcorn[12] proof of concept project. Our proof of concept consisted of a small boot sector, second-stage bootloader, and kernel. We began by writing the boot sector, which we then tested with QEMU[9]. Once the boot sector had been tested, we wrote a small kernel in C that printed a message to the screen. We compiled the kernel with an elf-i686-gcc cross-compiler[3] and linked the resulting object file with our boot sector binary. Our last step was to write a second-stage bootloader that would lie in between the boot sector and kernel, switching the processor to 32 bit protected mode and reserving room on the stack for the kernel functions. Having linked everything together, we demonstrated our work to the class on the 6th of December 2019.

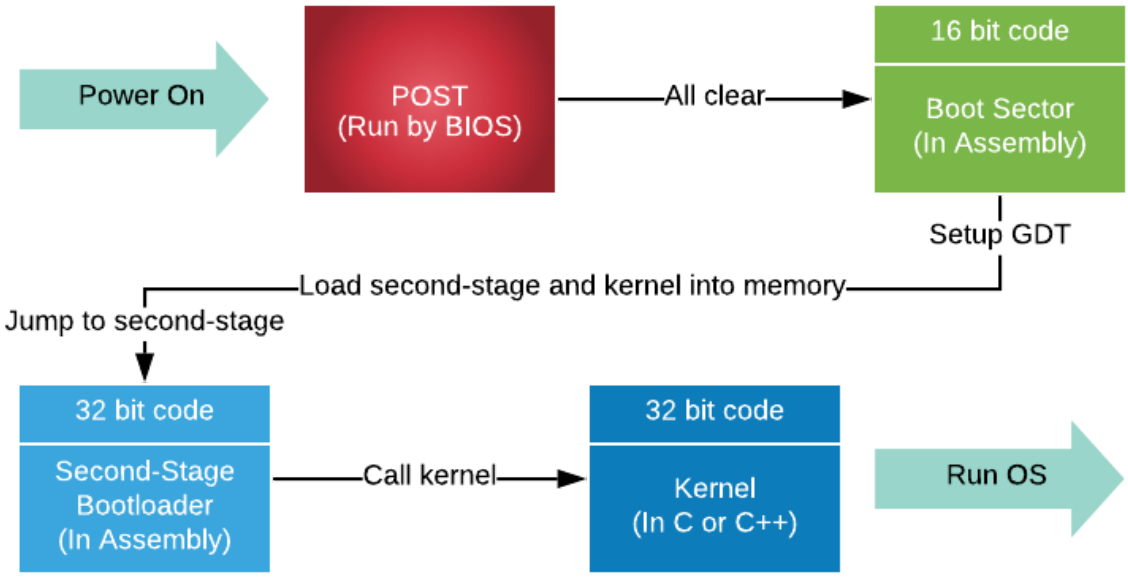
The main thing we learned throughout this project is that operating system development is an enormous undertaking. Some of the guides we followed online recommended a minimum of ten years of software and low level development experience before attempting to follow them. Every step we took proved more difficult than expected, from linking the two stages of the bootloader to making the kernel print to the Video Graphics Array (VGA) buffer correctly. However, as difficult as the project was for us, we only experienced a tiny slice of what it is like to be an operating system developer. It was an exciting, yet exhausting look behind the scenes of the incredible pieces of software that control our computers, which many of us take for granted.

This report will peer into the classic BIOS boot sequence, from the Power-On Self-Test (POST) to the execution of the kernel. A description of how all the pieces of the sequence fit together will proceed the conclusion and parting thoughts.

Boot Process

We were surprised to find that the boot process is not set in stone. The process could be modified for the needs of the kernel, hardware, and operating system. However, for this report we will discuss the typical Intel x86 boot sequence. There are four main steps to the boot process: POST, boot sector, second-stage bootloader, and kernel[11].

The first step is the Power-On Self-Test, which occurs when the hardware is first powered on. The BIOS begins performing diagnostics and searches for bootable devices. It does so by looking for the boot signature in the first sector (first 512 bytes) of a device. The boot signature is located at the end of the first sector of a bootable device, and takes up the last two bytes. The values of the boot signature are `0x55` and `0xAA`. If the BIOS finds the boot signature, it loads that first sector into Random Access Memory (RAM) at the address `0x0000:0x7C00` (segment 0, offset 31744). While in RAM, the boot sector is referred to as the first-stage bootloader.



The first-stage bootloader for the x86 architecture exists for backwards compatibility purposes and runs in 16-bit real mode. In this mode, only the 16-bit registers are available, and all addresses used correspond to real locations in memory. Only about 1MB of memory can be addressed in 16-bit mode. This mode uses BIOS interrupts, which trigger useful BIOS operations, such as printing a character to the screen, getting keyboard input, or reading data from disk. A Global Descriptor Table (GDT) is defined within the boot sector to mark protected sections of memory. This allows the second-stage bootloader to switch over to 32-bit protected mode. All code for the boot sector is written in assembly and is limited to 510 bytes.

The second-stage bootloader is loaded from the first-stage with BIOS interrupts. It then runs in 32 bit protected mode, which gives the bootloader access to 4GB of addressable memory with virtual memory and paging and lets us use 32-bit registers. At this point, BIOS interrupts can no longer be accessed. The second-stage bootloader can be written in C or C++ but requires some assembly code to start the C executable. For our project, we did not do much with the second-stage besides print a message using the VGA buffer, reserve memory for the kernel stack, and call the kernel.

Finally, the kernel can execute. As its name implies, the kernel is the core of an operating system. For our project, we wrote some simple C code, which executed some print statements that displayed a fun message to the VGA buffer. If we had wanted more functionality in our C code, we would have had to link freestanding and hosted libraries to the kernel, which fell outside the scope of our project.

Putting It All Together

Our project focused on booting our small kernel in a processor emulator called QEMU[9] from a flat binary file. Operating systems today are distributed as images, which contain helpful metadata for the processor to use while booting, but we decided to boot from a flat binary to highlight important parts of the boot process. For simplicity's sake, we combined our boot sector and second-stage bootloader into one file. The boot sector is written in 16 bit code, and any empty space of its first 512 bytes is filled with zeros, ending with the boot signature `0xAA55` (little endian). The second-stage bootloader is written in 32 bit code, which sets up the kernel stack and calls the kernel entry function. To prepare the boot code to be linked with the kernel code, we compiled the assembly file into a flat binary file using an assembler called NASM[5].

Our kernel consists of a C file that prints a welcome message to the terminal by using the processor's VGA buffer, through which the processor accesses the video output of the computer. If we were to add more functionality to the kernel or allow C functions like 'printf' to be used in our code, we would have to write our own freestanding libraries or host third-party libraries. Though a time-consuming task, writing and hosting such C libraries eases development and testing of the kernel. Since we had just one file for the kernel, we were able to both compile it in 32 bit and link the compiled object file to the boot binary with one gcc command. This command outputs a single flat binary file that can be run on any x86 processor. When developing operating systems, one must use a gcc cross-compiler[3] to ensure that the output binary can run on the target processor.

When the binary file is run, three messages will print to the screen. The first is a message from the boot sector, which is sent to the terminal screen by BIOS interrupts. The second-stage bootloader does not have access to these interrupts since the processor is no longer running in 16 bit real mode. Instead, the bootloader writes to the VGA buffer to display its message in the terminal. The kernel does the same but with more colors and in C rather than assembly. Once all messages have been printed, the kernel hangs since execution never returns from the kernel entry point. Were execution to somehow return to the bootloader from the kernel, the bootloader would clear all interrupts and halt, ending execution of the binary file.

Conclusion

The biggest takeaway from this project is that operating systems development is a gargantuan undertaking. As previously mentioned, many guides and tutorials we followed recommended years of experience in both software development and assembly programming before even considering creating an operating system. These recommendations are not without merit; developing an operating systems comes with challenges most programmers never have to face. Some of the more notable challenges that we encountered were the need to intimately understand the memory layout of the computer and the lack of libraries that many software developers may take for granted, such as standard C libraries that include basic functions like

‘printf’. Herein lies the reason that only a select few operating systems have gained popularity and survived to the present day.

Regardless of how insurmountable and discouraging these guides and tutorials may make operating systems development feel, it is an interesting and rewarding field. We have learned much about what an operating system needs to run properly and have gained a great appreciation for the libraries and Application Programming Interfaces we take for granted in high-level programming languages. If we were to move forward with this project, our next steps would be to separate the second-stage bootloader into its own file, implement a file system, and write some basic libraries for use in the kernel.

Resources

[1] Brunmar, Gregor. “The World of Protected Mode.” *Bona Fide OS Development*, www.osdever.net/tutorials/view/the-world-of-protected-mode.

[2] Fenollosa, Carlos. “os-tutorial.” *GitHub*, last commit 2 Nov. 2018, github.com/cfenollosa/os-tutorial.

[3] “GCC Cross-Compiler.” *OSDev.org*, 23 Dec. 2018, wiki.osdev.org/GCC_Cross-Compiler.

[4] “Meaty Skeleton.” *OSDev.org*, 9 Sept. 2018, wiki.osdev.org/Meaty_Skeleton.

[5] NASM. *The NASM development team*, 2015, nasm.us/.

[6] “Operating System Development Series.” *BrokenThorn Entertainment*, 12 Sept. 2008, brokenthorn.com/Resources/OSDevIndex.html.

[7] Parker, Alex. “Writing a Bootloader Part 1.” *Alex Parker’s Website*, 13 Oct. 2017, 3zanders.co.uk/2017/10/13/writing-a-bootloader/.

[8] Parker, Alex. “Writing a Bootloader Part 2.” *Alex Parker’s Website*, 16 Oct. 2017, 3zanders.co.uk/2017/10/16/writing-a-bootloader2/.

[9] QEMU the FAST! processor emulator. *QEMU*, www.qemu.org/.

[10] “User:Zester/Bare Bones.” *OSDev.org*, 4 Aug. 2017, wiki.osdev.org/User:Zesterer/Bare_Bones.

[11] “Operating System Development First And Second Stage Bootloaders” Independent-software.com, Oct 25, 2013, http://www.independent-software.com/operating-system-development-first-and-second-stage-bootloaders.html

[12] Seeley, Bushnell, and Taylor. “boot-popcorn.” *GitHub*, last commit 5 Dec. 2019, github.com/2altoids/boot-popcorn.