INF-2201: Operating system fundamentals

Project 1

Mattias Langseth

UiT id: mla180@uit.no

GitHub users: zexzee

January 29, 2023

1 Introduction

For this project, we were to create the necessary tooling and code for the machines boot-up mechanism, so that we have a foundation to stand on in future projects. To this end, we are to make a bootloader to run at the BIOS' startup address, and an imager that can produce a linked executable containing both this loader, the kernel to be loaded, and any other processes that the kernel will be managing.

2 Methods

2.1 Design and Implementation

The design of this project is twofold, as the programs made are separate and fulfill their own niches with very little direct overlap, and will as such be discussed separately.

2.1.1 Bootloader

The bootloader, or bootable segment, was designed to explicitly fit in the first 512 bytes (or rather, the first sector of the booting drive. Anything larger than this would require a separate loader to read the actual loader. The bootloader goes through a multi-step process, to ready the kernel for execution before passing control to it.

1. Create and load a temporary GDT (Global Descriptor Table)[1]. This is usually something that the kernel itself would set up but some emulation software is set up to panic if one is not provided at launch. The first field of this GDT must be the null-field, and is therefore 0. The following two sections must describe the code and data segments of the GDT, in that order, and are each set to overlapping segments of 4 GB.

2. Next, the loader sets up the segment registers, as the loader itself runs in 16-bit real mode. To allow addressing the boot segment, the base addresses for these segments are set to the boot segment itself (0x07C0). It also sets up the stack at the top of available RAM[2], being careful to allow it to both grow down without overwriting neither the boot segment nor the video memory that is reserved at the upper levels of available 16-bit memory (See Figure 1). To facilitate this, the stack and base pointers are placed at 0x9f000, and the stack segment is placed at 0x80000.

3. Once segments are initialized, the loader starts reading the kernel bits into memory, at position 0x1000 (See figure 1). It does this by utilizing the CHS[3] addressing mode to read segments in order from the booting drive, up to the number of segments that are occupied by the kernel as decided by the provided kernel size written by the imager. In practice, finding a good location to write this size is challenging, and the loader is hard coded to read 16 sectors. This process must be done while in real mode, as it relies on the BIOS interrupts being available, specifically interrupt 0x13. This is the actual operating system code, and the code that will be given control once the loader has finished. If this code happens to be larger than the available 29 KiB of memory that resides between the kernel's base address and the

boot sector, the loader must enable the A20 line[4] and enter unreal mode[5] to move the kernel into memory above the 1 MB wall. This loader does not do so, nor does it check if it has to do so, as the kernel is simply assumed to fit within its predetermined memory.

4. After the kernel has been read into memory, the loader prepares to hand over control of execution to it. It does so, by entering protected mode. At start, it changes to 32-bit mode, then it sets the "Protection Enabled" bit, CR0, to 1. Once this preparation is done, it re-sets the segment registers to ensure that they all contain 32-bit values, moves the base address to 0x1000 to conform with the kernel's location, before performing a long jump to the kernel, setting the normally "immutable" registers (I.e. the code segment and the instruction pointer) to their new and correct values.

5. As a final point, though not currently done in the solution as it may conflict with the intent of the assignment, the loader itself may utilize some assembly compiler hints, to fill itself to 510 bytes exactly, and then writing its magic signature (0xAA55) at its own end. This is preserved in the code for posterity, but is also handled by the disk imager.

2.1.2 Disk Imager

The imager was designed to be as file-count agnostic as possible, to allow for any number of processes to be linked. This trivialization of linker contents poses its own set of challenges, some of which are only theoretically solved in the final solution.

1. The first step of the disk imager, is the creation of its own "final" header. It does this by creating a fitting ELF header for itself, then looping over the files it has to link, fetching the ELF headers from each file, and grabbing the program header tables utilizing the defined sizes and offsets described in the header. Using this information, the imager creates its own array of program header tables, and stores the offsets and pointers of these in its own ELF header, preparing both this header and the array of tables for writing later.

2. Once the header is compiled, the imager will sequentially grab the raw code from each file, again utilizing the header information it was provided. It will then pad these segments, firstly to allow enough space for the file to grow in memory (As file size and memory size are not guaranteed to be equal, or even similar), before padding the segment up to the next sector boundary. Once these are padded correctly (taking extra care to write the magic at the end of the bootloader), they will be stored for the final write. This padding process will be further discussed later in this report.

3. The disk imager will then write the header info to a blank file, before dumping the raw code in sequence, making sure that the bootloader and kernel are written first, in order.

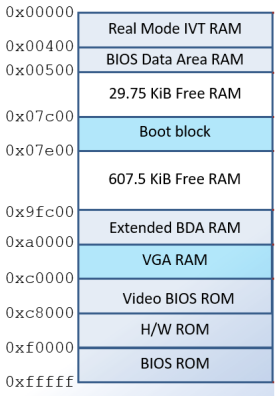

Figure 1: Real mode memory map, p1precept.pptx, UiT, provided by the project pre-files.

3 Discussion

3.1 Bootloader

Mostly, the issues that could be discussed in regards to the bootloader and its implementation carry more relevance to the design decisions around bootloaders as a whole, and what it is intended to load. Having a multi-stage loader would presumably become a requirement eventually, should the loader be general enough to be professionally utilized. Additionally, as multiboot becomes less of a luxury and more of an expectation with newer and more refined bootloaders, assuming that one can stay below 512 bytes and still be competitive is foolish, unless it is a specialized niche (tiny hardware, or as is this case, a miniature project).

3.2 Disk Imager

The more interesting discussion therefore relates to the imager, and the multitude of ways one could realistically compile a disk image, and the multitude of ways where the current design and solution fails. As it currently stands, there are two points of padding that consistently fail: The imager does not currently have any way of making sure that linked files are padded correctly in accordance to their virtual addresses, and, the imager does not currently properly update the offsets of the program header tables that it manages, leading to the headers pointing at wildly incorrect segments. While this does pose a serious issue for the later reading and execution of the file, it was deemed - naively - to be an issue of luxury, as the only real point of importance is the defined entry point matching with the bootloader. However even this pointer is likely offset, due to the inclusion of several additional header tables suddenly interceding between the loader's header and the actual loader segment. This is not irreparable - in fact one could argue that a fix could be achieved with comparatively little effort - but achieving a solution that works effectively on any number of files remains elusive. A brute force approach of simply "keeping track" of how far things are offset sounds simple on paper, but maintaining this in such a way as to retroactively update the offsets of previous headers as new headers are inserted seems to require secondary passes over each and every table, a process that would be exhaustive if given more than a handful of files to process.

3.3 Conclusion

All in all, the code has an inconsistent execution rate, presumably owing to the issues of effectively padding during the imaging process. Regardless, it would seem to me that the code and this report in tandem effectively showcase ability and understanding, even if the images produced currently fail to execute correctly.

4 Sources

References

[1] OS dev contributors. (4 May 2022). Global Descriptor Table. In OSDev wiki. Retrieved 14:26, January 29th, 2023, from `https://wiki.osdev.org/Global_Descriptor_Table`.

[2] OS dev contributors. (17 May 2020). Memory Map (x86). In OSDev wiki. Retrieved 14:26, January 29th, 2023, from `https://wiki.osdev.org/Memory_Map_(x86)`

[3] Wikipedia contributors. (22 November 2022). Cylinder-head-sector. In Wikipedia, The Free Encyclopedia. Retrieved 14:26, January 29th, 2023, from `https://en.wikipedia.org/wiki/Cylinder-head-sector`

[4] OS dev contributors. (3 January 2022). A20 Line. In OSDev wiki. Retrieved 14:26, January 29th, 2023, from `https://wiki.osdev.org/A20_Line`

[5] OS dev contributors. (28 November 2022). Unreal Mode. In OSDev wiki. Retrieved 14:26, January 29th, 2023, from `https://wiki.osdev.org/Unreal_Mode`

[6] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015.

[7] Shanley, Tom. Protected Mode Software Architecture, MindShare Inc., Addison-Wesley Publishing Company. 1996

[8] OS dev contributors. (18 August 2022). Rolling Your Own Bootloader. In OSDev wiki. Retrieved 14:26, January 29th, 2023, from `https://wiki.osdev.org/Rolling_Your_Own_Bootloader`

[9] OS dev contributors. (Posted: Sat Jun 20, 2009 12:36). Forum Topic, Loading Int 13h. Retrieved from `https://forum.osdev.org/viewtopic.php?f=1&t=20353`