CS3101: Operating System Lab

5th August, 2025

Lab 1: Booting Unraveled

1 Preliminaries

- **NASM Assembler:** The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32), and 64-bit (x86-64) programs. It is primarily used to compile the assembly programs.
- So, we first need to install the NASM assembler locally on our machine by following the given steps:
  1. First, check whether NASM is previously installed or not using the following command:
     ```
     whereis nasm
     ```
  2. If not previously installed, navigate to the following URL and download the tar file of the latest stable version:
     ```
     https://www.nasm.us/
     ```
  3. From the downloaded folder, extract the archived file and open it in a terminal.
  4. Check whether make is installed or not using the command:
     ```
     make --version
     ```
  5. If make is not previously installed, install it using the command:
     ```
     sudo apt install make
     ```
  6. After this run, the following command before checking whether nasm is installed on your machine or not:
     ```
     sudo make install
     ```

- Once the NASM assembler is installed successfully, we can compile the assembly code using the command:

```
nasm -f bin file_name.asm -o file_name.bin
```

- **QEMU:** QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g., an ARM board) on a different machine (e.g., your x86 PC). By using dynamic translation, it achieves very good performance.

- To execute the compiled binary file and view its result on the emulator, use the command:

```
qemu-system-i386 -drive format=raw,file=file_name.bin
```

- Install qemu on your system using the command:

```
sudo apt-get install qemu-system-x86
```

2 Boot Sector Basics

- At the very core of a computer booting is what we refer to as the boot loader. The boot loader physically reads the first sector, or sector 0, from the system hard disk or other media to ultimately bootstrap an OS.

- When the computer boots, it reads the first sector, which is exactly 0x200 bytes (hex) or 512 bytes in decimal.

- The system that is reading this boot loader is what is referred to as BIOS, which is a basic input output system, and it loads in 16-bit mode. It does this to be compatible with older processors.

- On an x86 machine, the BIOS selects a boot device, then copies the first sector from the device into physical memory at memory address 0x7C00. The boot sector will hold 512 Bytes.

- These 512 bytes contain the boot loader code, a partition table, the disk signature, as well as a `"magic number"` that is checked by the BIOS to avoid accidentally loading something that is not supposed to be a boot sector.

- The BIOS then instructs the CPU to jump to the beginning of the boot loader code, essentially passing control to the boot loader.

3 Basic Components of a Bootloader file

- `[BITS 16]` and `[BITS 32]`: These directives tell the assembler whether to generate 16-bit instructions or 32-bit instructions.
- In NASM, the [org 0x7c00] directive sets the assembler location counter. We specify the memory address where the BIOS is placing the boot loader. This is important when using labels, as they will have to be translated to memory addresses when we generate machine code, and those addresses need to have the correct offset.
- **mov sp, 0xFFFE:** Sets the stack pointer (SP) to 0xFFFE. The stack grows downwards in x86, so 0xFFFE is a high address within the boot sector's memory space.
- **mov si, message:** Loads the effective address of the message string into the SI (Source Index) register. SI will be used as a pointer to iterate through the string.
- **jmp $**: An infinite loop (`jmp $` means "jump to the current instruction"). This halts execution after the message is printed, as there's no further code to execute in this simple bootloader.
- **mov ah, 0x0E:** Sets the AH register to 0x0E, which is the function code for "Teletype Output" (write character to screen) within BIOS interrupt 0x10.
- **lodsb:** Loads a byte from the memory location pointed to by DS:SI into the AL register and then increments SI.
- **cmp al, 0:** Compares the loaded character with 0 (the null terminator). This is required to check the end of the string.
- **int 0x10:** Calls BIOS interrupt 0x10 (Video Services) to display the character in AL.
- **times 510 - ($ - $$) db 0**: Used to pad the remaining bytes of the 512-byte boot sector with zeros.
- **dw 0xAA55**: This is the "boot sector signature" or "magic number." The BIOS checks for this specific two-byte value at the end of the first sector to confirm it's a valid bootable sector. `dw` stands for "define word."

4 Second Stage Bootloader

- The primary reason for the requirement of the second-stage boot loader is the size limit. We cannot store more than 512 bytes of code in the boot sector, so if you want to make a highly efficient boot loader (like GRUB).

- We need to find a way to store all of it somewhere else, but not in the boot sector itself.
- **Note:** For the current context, our task is not switching to 32 or 64-bit modes. All we do here is just print a message, but from the second-stage boot loader.
- So, the goals that must be achieved while designing a second-stage bootloader are:
  1. Procedure that can read from disk, but in the first stage. It will try to read our secondary boot loader and load it into memory.
  2. Procedure that tries to call our secondary boot loader (which is loaded into memory by the step above) and transfers execution to it.
  3. Secondary boot loader itself, that prints a message. This will be an assembly code with minimal function to print a message.
  4. Command to assemble the primary and secondary bootloader files:

     ```
     nasm -f bin bootloader1.asm -o bootloader1.bin

     nasm -f bin bootloader2.asm -o bootloader2.bin
     ```
  5. Command to inspect the sizes of the two bootloader files:

     ```
     ls -l bootloader1.bin bootloader2.bin
     ```

     bootloader1.bin should be exactly 512 bytes
  6. Build a 1.44 MB floppy image
     (a) create a blank 1 144 KiB image (2 880 $\times$ 512 B)

     ```
     dd if=/dev/zero of=bootloader2.img bs=512 count=2880
     ```
     (b) write stage-1 to sector 0

     ```
     dd if=bootloader1.bin of=bootloader2.img conv=notrunc
     ```
     (c) write stage-2 to sector 1 (seek=1 offset 512 B)

     ```
     dd if=bootloader2.bin of=bootloader2.img bs=512
     seek=1 conv=notrunc
     ```