

Advanced Operating System Final Project

International Institute Information Technology, Hyderabad



Project Title : BOOTLOADER

Team Name : boot -y loader

Team Members:

Akhilesh Giriboyina (2022202022)

Abhijeet Renge (2022202002)

Hrishikesh Deshpande (2022201065)

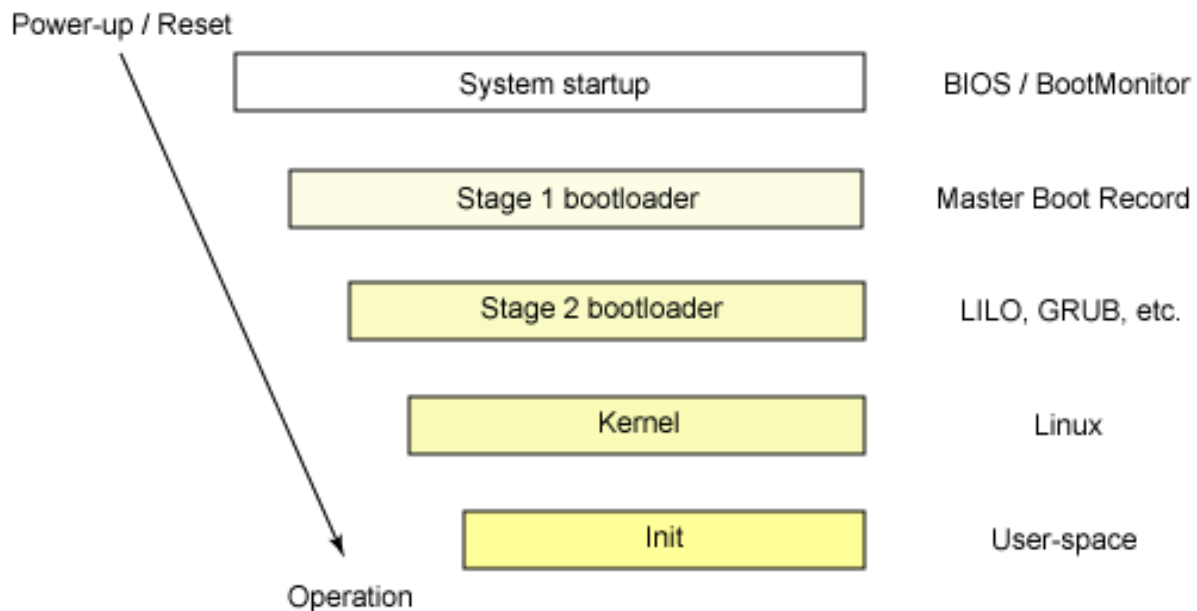
Mayush Kumar (2022201043)

Project Guide : Dr. Manish Srivastava sir

Bootloader

What exactly is a bootloader?

When you start your computer, the kernel gets loaded and after that, you are able to login into the system and use the system, but who loads the kernel into the memory of your computer? It's none other than the bootloader. So, when we start out the system, all the hardware components receive the power on self test (POST) signal and get initialized. Now BIOS which is a basic input-output system will load the Master boot record file and read the instructions and based on the instructions BIOS will look for a bootable device. Once the bootable device is found, the BIOS loads the bootloader. Bootloader will load the kernel into memory. This entire process is called booting. Any kind of hardware device which contains the bootloader is called Bootable Device.



Real Mode and Protected mode:

The systems we have today, all boots into protected mode but before that systems always boots into real mode. Real mode booting generates 16 bit address and codes and initially the system boots into the real mode only. Here BIOS after booting passes the control to CPU and in protected mode, the CPU enforces strict memory and hardware I/O protection as well as restricting the

available instruction set. Therefore we have to switch real mode into protected mode or 16 bit instructions into 32 bit. And our project is all about making a bootloader and switching the kernel into 32 bit i.e. into protected mode. And to ensure we have successfully switched into protected mode, we have created a dummy kernel file where we are checking if graphics and keyboard inputs are enabled and if they are it will show that we have successfully loaded a kernel and switched it into protected mode.

Languages used:

We have used C and assembly programming language.

Below are explanation of all the important files we have in our code:

Master Boot Record (mbr.asm):

The main assembly file for the bootloader is the ``mbr.asm`` file. It contains the definition of the master boot record. It also contains all the necessary file includes for the helper functions. The real mode is 16-bit whereas protected mode is 32-bit. We need to switch between these modes. The directives ``[bits 16]`` and ``[bits 32]`` tell the assembler which format to use when generating the instructions. When the BIOS gives control to the bootloader, we start in 16-bit instruction set as CPU is still in 16-bit mode.

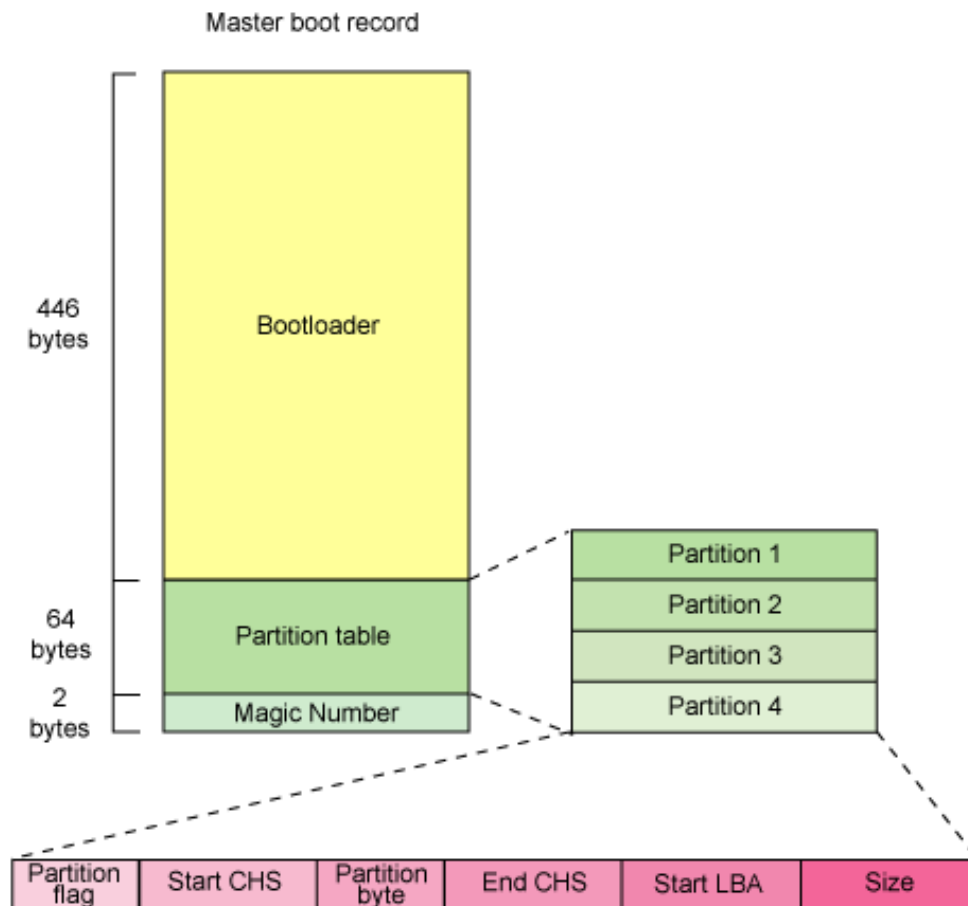
``org 0x7c00`` is NASM directive which sets the location of assembler counter. BIOS stores the bootloader at this location. We then define an assembler constant named ``kernel_base_addr``. Its value is set to 0x1000 at compile time where kernel is to be loaded. Later we will jump to this location to hand over the control to the kernel.

Then we fetch the boot drive from ``dl`` register set by the BIOS into ``org_boot_drive`` variable so that we don't lose it in case ``dl`` is overwritten. Before calling any procedure, we need to set the stack pointers ``sp`` and ``bp`` to some address far enough from our other bootloader memory to avoid collisions. We set them to 0x9000 in our case.

Then we call the ``boot_load_kernel`` procedure that tells BIOS to load the kernel from disk to memory at ``kernel_base_addr`` address, which then calls a helper

procedure called `boot_disk_load` which we will see in `disk.mbr` section. We set `dl`, `dh` and `bx` to boot drive, kernel location and number of sectors respectively to be used by `boot_disk_load`. When the control returns, we call another helper procedure named `protected_mode` which will prepare to be switched to 32bit protected mode, switch to protected mode and then pass the control to the kernel by jumping to the `kernel_base_addr`.

The main bootloading is done. Now we pad the remaining memory with zeroes and declare variables that will be used in our code. The last thing we have to do is set the last word (two bytes) as the magic number which is 0x55aa but when we set it in little endian mode, it becomes 0xaa55. This magic number serves as a validation check of the MBR.



Disk.asm:

Reading from the disk with the help of BIOS as we are using 16 bit mode.

To start reading information from disk using BIOS, we need to store following data in memory.

1. ah : Mode (0x02 => read from disk)
2. al => number of sectors to read
3. ch => low eight bits of Cylinder number
4. cl => Sector number 1-63 (eight bits of cylinder number)
5. dh => Head number
6. dl => Drive number (bit 7 set for hard disk)

Now upon calling 0x13 interrupt signal data from the location specified by above registers will be read from disk and be loaded into memory.

Pushing all general purpose registers on stack before starting execution of our procedure. Push the number of sectors to read onto the stack.

Set the mode, number of sectors to read, cylinder number, sector number, head number.

BIOS interrupt 0x13 is called. It should load kernel into memory. To make sure there were no problems:

1. check carry bit. If carry bit is set then some error occurred.
2. check if number of sectors read is equal to number of sectors attempted to read. If it is unequal then error occurred.

In case of error, execute infinite loop.

If there is not any error return to main function.

Global Descriptor Table (gdt.asm):

Before switching into protected mode we need to define how segmentation is going to work after the mode switching. Every segment has a set of permissions and properties that we define or set the value for in a data structure called Global Descriptor Table (GDT). We need to define descriptors for each segment we are going to use in a protected mode. A descriptor is a list of properties of a segment. GDT contains:

- A null segment descriptor (0x0)
This is required as a safety mechanism to catch errors when our code forgets to select a memory segment.
- Code segment Descriptor
- Data segment Descriptor

Our descriptor table will contains the below information or properties which are defined for both the segments:

- Base address: It is a 32-bit starting memory address of the segment and for both of our segments we have set it to 0x0.
- Segment Limit: It is a 20-bit long word which defines the length of our segment and we have set it to 0xFFFFF for both our segments.
- Present bit: It is set to 1 which defines if the segment is used and all our segments is to be used.
- Privilege: It is a 2 bit sequence whose values lies between 0 to 3(00 | 01 | 10 | 11). It is used to set a segment hierarchy and implement memory protection. Here, 0 is the highest privilege segment.
- Type bit: It is a 1 bit code to define which type of segment it is,as 1 for code segment and 0 for data segment.
- Flags: There are two sets of flag we use:
 - Type flags (4 bits):
 1. First bit is for the code bit: If a segment contains code then it is set to 1. So for the code segment it is 1 and for the data segment it is 0.
 2. Conforming bit: This bit tells us that if this segment can be executed by lower privilege segments. If it can then set it to 1 otherwise 0.
 3. Readable: If this bit is set to 1 then it defines that we can read constants from the segment.
 4. Accessed: This bit is used by the CPU and it is set to 1 when the CPU is using the segment, so we set it to 0.
 - Other flags (4 bits):
 1. Granularity: When it is set to 1, it multiplies the limit by 0x1000
 2. 32 bit flag: We set it to 1 if the segment is going to use 32 bit memory.
 3. We won't be using the last 2 bits so we set it to 00.

Switch to Protected Mode:

To switch to 32 bit protected mode, following steps should be performed:

- Disable interrupts using cli(Clear Interrupt Flag) instruction.
- Load GDT descriptor.
- Enable protected mode in cr0 control register.
- Jump at a location far away from the current location so that the CPU pipeline will be flushed and hence the 16 bit instructions in the pipeline will be discarded.

- Setup all segment registers to point to data segment.
- Setup a new stack by setting a 32 bit bottom pointer and stack pointer.
- Jump back to mbr.asm and handover control to the kernel.

Dummy Kernel - Kernel.c

Till the Bootloader reaches Kernel, it will have changed Real mode to Protected mode i.e., now instructions will be taken in 32 bit format.

Hence we can now use High level language like C for our dummy Kernel.

Aim of Kernel:-

In our Dummy Kernel we will print a welcome string on Monitor.

Kernel also keeps taking Keyboard input from the User and prints the same to the monitor screen.

Among keyboard inputs, the kernel is limited to accepting certain characters like Alphabets, Numbers for demo purpose.

To exit Kernel, we can enter any other character.

Once Kernel is exited, control gets back to mbr.asm which will perform an infinite loop (jmp \$) so that the emulator does not exit.

Note:-

When Kernel is loaded, Instruction set is in protected mode (32 bit), but we do not have loaded any device drivers for the monitor or keyboard. So we will not be able to use scanf/ printf functions from C.

Reading Keyboard strokes -

Keyboard is connected to Port No 0x60. So we use assembly command INB to read each byte (Keystroke) from Port 0x60 i.e., KeyStroke.

Writing output to Monitor -

We can directly use Video memory to print characters to monitor. The Top Left point of Monitor is present in address 0xb8000. So we start printing from this memory address.

For printing each character, two bytes will be needed - One for ASCII of character and other for colour of character

Ex:-

To print A to Video Memory

0xb8000 - A (Char)

0xb8001 - White (Colour)

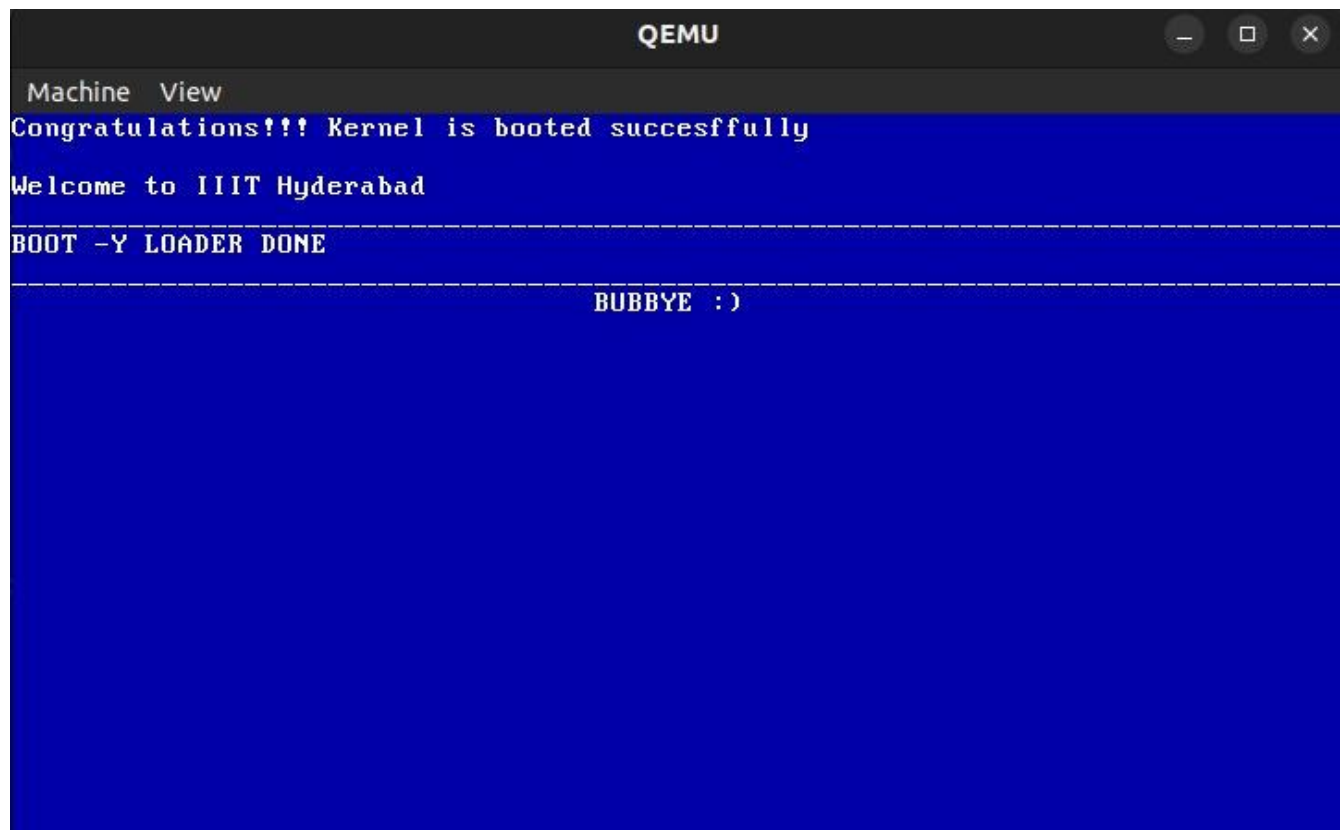
Instructions:

To run our project code go into the project folder and enter command:

\$ make run

We have built a Makefile which contains all the necessary instructions to run our project. After running our kernel will load into QEMU emulator and since we have coded into assembly language it is system specific so it will only run onto x86 architecture. On the Qemu emulator you can check by entering anything on keyboard and it will be printed on screen ensuring switching into 32 bit is also done successfully.

End Result:



```
QEMU
Machine View
Congratulations!!! Kernel is booted succesffully
Welcome to IIIT Hyderabad
-----
BOOT -Y LOADER DONE
-----
BUBBYE :)
```


References:

- ☐ <https://developer.ibm.com/articles/l-linuxboot/#:~:text=Stage%201%20boot%20loader&text=The%20MBR%20ends%20with%20two,validation%20check%20of%20the%20MBR>
- ☐ https://wiki.osdev.org/Rolling_Your_Own_Bootloader
- ☐ <https://wiki.osdev.org/Bootloaders>
- ☐ https://wiki.osdev.org/Bare_bones
- ☐ <https://docs.yoctoproject.org/dev-manual/qemu.html>