Team members:
Sanket HS - PES1201700158
Sriram S K - PES1201700097
Shreyas Prashanth - PES1201701249
Kevin Arulraj - PES1201700659

Bootloaders: A Case Study

Introduction

A boot loader is a program that loads and starts the boot time tasks and processes of an operating system or the computer system. It enables loading the operating system within the computer memory when a computer is started or booted up. Bootloaders are used to boot other operating systems, usually each operating system has a set of bootloaders specific for it. Since it is usually the first software to run after powerup or reset, it is highly processor and board specific.

The bootloader is typically called from the BIOS (Basic Input/Output System). The BIOS firmware is used for hardware initialization and is the first piece of code that runs when the computer is switched on. It comes pre-installed on the ROM of the motherboard.

One of the BIOS' most important duties is the execution of the Power-on Self-Test(POST) in which it verifies CPU registers, the integrity of the BIOS code, and other basic components like DMA, interrupt vectors etc. After successful execution, it hands over control to the bootloader by reading the MBR (Master Boot Record), a special boot sector,located at the very beginning of the hard disk. This MBR code is usually referred to as a bootloader.

Boot is short for bootstrap or bootstrap load and derives from the phrase 'to pull oneself up by one's bootstraps'. The usage calls attention to the requirement that, if most software is loaded onto a computer by other software already running on the computer, some mechanism must exist to load the initial software onto the computer. Early computers used a variety of ad-hoc methods to get a small program into memory to solve this problem. The invention of read-only memory (ROM) of various types solved this paradox by allowing computers to be shipped with a start-up program that could not

be erased. Growth in the capacity of ROM has allowed ever more elaborate start up procedures to be implemented.

How does it work?

All bootloaders are simply pieces of software, just like the application code they will help you load on to the processor. Generally speaking, bootloaders are designed to be as small and simple as possible so they do not take up space that otherwise could be used for your application.

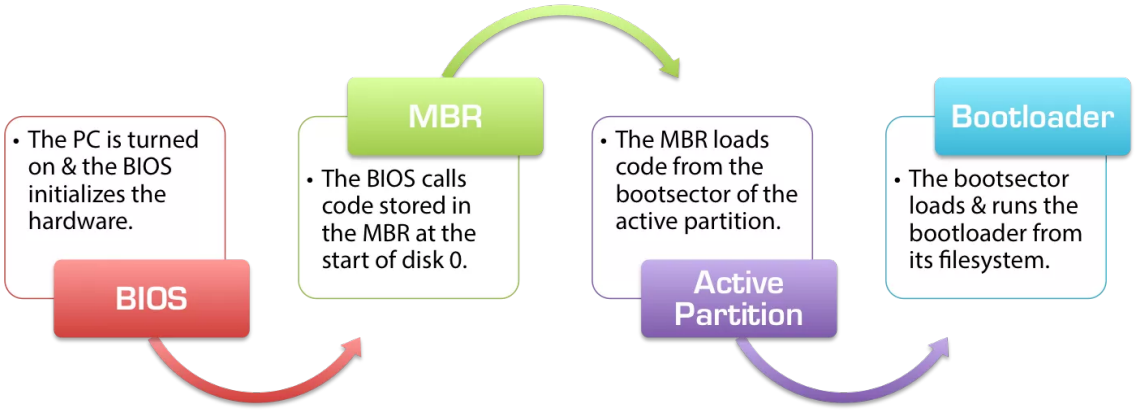
On reset, the processor will always start executing the code located at a particular memory location. Normally in an embedded system, if you don't have a bootloader you would just put your application at this startup memory location. With a bootloader, instead of the application starting at the memory location that the processor first executes after a reset, the bootloader is put in that spot and the application code in another part of memory. This means that the bootloader is the program that the processor will run every time it resets.

The job of the bootloader then, is to do one of two things:

1. Replace the existing application code with a new application, or
2. Start the existing application.

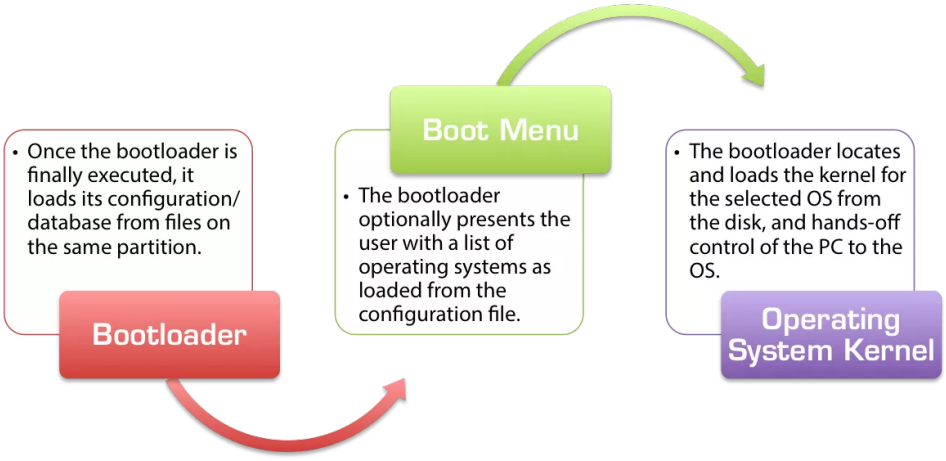
How it determines which of these two things to do varies with implementation, but generally some external signal (such as a command on the serial port or a particular I/O line being pulled low) will tell it to load a new application from the communication port (or some other location) and write it to the application memory.

If the signal to load a new application is not present, the bootloader simply runs the existing application by executing a jump instruction to tell the processor to run the code at the memory location where the application starts.



Execution Steps:

1. Computer is switched on
2. BIOS executes the Power-on Self-Test (POST)
3. If successful, hands over control to the bootloader
4. Bootloader loads basic filesystem drivers
5. Transitions to the second-stage bootloader
6. Displays boot menu
7. Hands over control to the Operating System



Popular bootloaders:

- GNU GRUB Bootloader (GNU GRand Unified Bootloader) - most Linux distributions use this
- NTLDR - used by Windows systems till Windows XP
- BootX - used by Apple's Mac OS
- LILO (Linux Loader)- very basic BIOS Linux bootloader
- iBoot - stage 2 bootloader for all Apple products
- Android bootloader

Windows bootloader

Windows NT (Windows New Technology) is a family of operating systems produced by Windows since 1993. For close to three decades, Microsoft has been a pioneer in innovation and globalisation of technology that aides our daily lives. Over these years they have improved their operating systems making them faster, more reliable and to keep up with the times. And in course with such changes, they have worked on two generations of bootloaders: NTLDR and BOOTMGR. Both NTLDR and BOOTMGR begin with a "real mode stub" (The **real-mode stub** program is an actual program run by MS-DOS when the executable is loaded) which switches the CPU from 16-bit real mode to 32-bit protected mode.

NTLDR

NTLDR (abbreviation of NT loader) is the boot loader for all releases of Windows NT operating system up to and including Windows XP and Windows Server 2003. NTLDR is typically run from the primary hard disk drive, but it can also run from portable storage devices such as a CD-ROM, USB flash drive, or floppy disk. NTLDR can also load a non NT-based operating system given the appropriate boot sector in a file.
NTLDR requires, at the minimum, the following files to be on the system volume:

1. `ntldr`, the main boot loader itself
2. `NTDETECT.COM`, required for booting an NT-based OS, detects basic hardware information needed for a successful boot
3. `BOOT.INI`, an important file that contains boot configuration. If missing, NTLDR will default to *\Windows* on the first partition of the first hard drive

When a PC is powered on its BIOS follows the configured boot order to find a bootable device. This can be a hard disk, floppy, CD/DVD, network connection, USB-device, etc depending on the BIOS. For a hard disk, the code in the Master Boot Record (first sector) determines the active partition. The code in the boot sector of the active partition could then be again a NTLDR boot sector looking for ntldr in the root directory of this active partition. NTLDR first looks for a text file called BOOT.INI in the root directory. If it is not found, it defaults to either C:\WINNT\SYSTEM32 (Windows 2000 and earlier). This implies you need the BOOT.INI file if you have multiple versions of Windows. The following assumes that a Windows NT-style operating system is selected to boot (i.e., not Windows 9x, DOS, Linux, etc). NTLDR reads a file called NTDETECT.COM in the root directory and "calls" it (the code runs and then execution resumes in NTLDR).

Boot Configuration Data (BCD)

The Windows boot loader architecture includes a firmware-independent boot configuration and storage system called *Boot Configuration Data* (BCD) and a boot option editing tool, BCDEdit (BCDEdit.exe).

Windows includes boot loader components that are designed to load Windows quickly and securely. The previous Windows NT boot loader, *ntldr*, is replaced by three components:

- Windows Boot Manager (Bootmgr.exe) :
  Windows Boot Manager (BOOTMGR) is a small piece of software, called a boot manager, that's loaded from the volume boot code, which is part of the volume boot record. BOOTMGR helps your Windows 10, Windows 8, Windows 7, or Windows Vista operating system start. Configuration data required for BOOTMGR can be found in the Boot Configuration Data (BCD) store, a registry-like database that replaced the *boot.ini* file used in older versions of Windows like Windows XP. The BOOTMGR file itself is both read-only and hidden. It is located in the root directory of the partition marked as *Active* in Disk Management. On most Windows computers, this partition is labeled as *System Reserved* and doesn't have a drive letter. BOOTMGR eventually executes winload.exe, the system loader used to continue the Windows boot process.

- Windows operating system loader (Winload.exe):
  Winload.exe (Windows Boot Loader) is a small piece of software, called a *system loader*, that's started by BOOTMGR. The job of winload.exe is to load essential device drivers, as well as ntoskrnl.exe, a core part of Windows. In older Windows operating systems, like Windows XP, the loading of ntoskrnl.exe is done by NTLDR, which also serves as the boot manager.

- Windows resume loader (Winresume.exe):
  The winresume.exe is a Resume From Hibernate boot application. Winresume.exe is developed by Microsoft Corporation. It’s a system and hidden file. Winresume.exe is usually located in the %SYSTEM% sub-folder and its usual size is 428,112 bytes. The winresume.exe process is safe and disabling it can be dangerous, because the programs on your computer need it to work correctly.

GRUB

GRUB which stands for **Grand Unified Bootloader**, is a boot loader package which provides a user the choice to boot one of multiple operating systems installed on a computer or select specific kernel configuration available on a particular operating system's partitions. It is predominantly used for Unix-like systems. GRUB is independent of any particular operating system and may be thought of as a tiny, function-specific OS. The purpose of the GRUB kernel is to recognise filesystems and load boot images.

GRUB was intended to boot operating systems that conform to the Multi-boot Specification (The **Multi-boot Specification** is an open standard describing how a boot loader can load an x86 operating system kernel.), which was designed to create one booting method that would work on any conforming PC-based operating system. In addition to multi boot-conforming systems, GRUB can boot directly to Linux, FreeBSD, OpenBSD, and NetBSD.

GRUB Interfaces

It provides both menu-driven and command-line interfaces to perform these functions. The command-line interface in particular is quite flexible and powerful. The command line interface is accessible both while the system is booting (the native command environment) and from the command line once os is running.

GRUB Boot Process

The boot process using GRUB requires the GRUB to load itself into memory. This is done in the following stages (executables).

Stage 1 is the piece of GRUB that resides in the MBR(Master Boot Record) or the boot sector of another partition or drive. Since the main portion of GRUB is too large to fit into the 512 bytes of a boot sector, Stage 1 is used to transfer control to the next stage, either Stage 1.5 or Stage 2.

Stage 1.5 is loaded by Stage 1 only if the hardware requires it. Stage 1.5 is file-system specific; that is, there is a different version for each filesystem that GRUB can load. Stage 1.5 image contains file-system drivers, stage 1.5 loads Stage 2.

Stage 2 runs the main body of the GRUB code. It displays the menu, lets you select the operating system to be run, and starts the system you've chosen.

The operating system or kernel is loaded into memory by the boot loader. After that, the control of the machine is transferred to the operating system. The images of the different OS are in the grub.conf file (/boot/grub/grub.conf)

GRUB Naming Conventions

GRUB uses its own naming conventions. Drives are numbered starting from 0; thus, the first hard drive is hd0, the second hard drive is hd1, the first floppy drive is fd0, and so on. Partitions are also numbered from 0, and the entire name is put in parentheses. So the first partition of the first drive, /dev/hda1, is known as (hd0,0) to GRUB. The third partition of the second drive is (hd1,2). GRUB makes no distinction between IDE drives and SCSI drives, so the first drive is hd0 whether it is IDE or SCSI.

GRUB Conf file example

```
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Server (2.6.18-238.el5)
root (hd0,0)
kernel /vmlinuz-2.6.18-238.el5 ro root=/dev/VolGroup00/LogVol00
initrd /initrd-2.6.18-238.el5.img
title Windows XP Pro
rootnoverify (hd0,0)
chainload +1
```

• The default=0 directive points to the first stanza, which is the default Operating System to boot.

• The timeout=5 directive specifies the time, in seconds, for GRUB to automatically boot the default operating system.

• The splashimage directive locates the graphical GRUB screen.

• The hiddenmenu directive means that the GRUB options are hidden.

A stanza begins with a title, (the text to be displayed in boot menu for selecting the Operating System) and the next three lines specify the location of the /boot directory, the kernel, and the initial RAM disk (The initial RAM disk (initrd) is an initial root file system that is mounted prior to when the real root file system is available), respectively.

• root (hd0,0) - Specifies the boot directory is in first hard disk, first Partition.

• kernel /vmlinuz-2.6.18-8.el5 ro root=LABEL=/ rhgb quiet - Specifies the kernel location which is inside the /boot folder. This location is related to the root(hd0,0) statement. The "ro" option specifies the kernel should be opened as read only to protect it from any accidental writes from the initial RAM disk and "rhgb" enables the RedHat Graphical boot option.

• initrd /initrd-2.6.18-8.el5.img - Initial RAM disk.

Example, when the user selects the first option of Red Hat Enterprise Linux(in this case), following three commands execute:

```
root (hd0,0)
kernel /vmlinuz-2.6.18-238.el5 ro root=/dev/VolGroup00/LogVol00
initrd /initrd-2.6.18-238.el5.img
```

The first command, i.e. "root (hd0,0)" specifies the partition on which a compressed kernel image and initrd file are located.

The second command i.e. "kernel /vmlinuz-2.6.18-238.el5 ro root=/dev/VolGroup00/LogVol00" tells which kernel image to use (in this case vmlinuz-2.6.18-238.el5). The arguments to this command are 'ro' and 'root'. 'root' specifies the device on which the root directory of the filesystem (i.e. / directory) is located; 'ro' means that this partition is to be mounted in read only mode (i.e. the kernel

mounts the root partition in read only mode). The partition for the root filesystem and the partition on which this kernel image resides (i.e. boot partition) are different.

The third command is the location of initrd.

Initrd

The initial RAM disk (initrd) is an initial root file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure.

GRUB Features

- GRUB supports LBA (Logical Block Addressing Mode) which puts the addressing conversion used to find files into the firmware of the hard drive.
- GRUB provides maximum flexibility in loading operating systems with required options using a command based, pre-operating system environment.
- The booting options such as kernel parameters can be modified using the GRUB command line.
- There is no need to specify the physical location of the Linux kernel for GRUB. It only required the hard disk number, the partition number and file name of the kernel.
- GRUB can boot almost any operating system using the direct and chain loading boot methods.

Android Bootloader

A bootloader is a vendor-proprietary image responsible for bringing up the kernel on a device. It guards the device state and is responsible for initializing the Trusted Execution Environment (TEE) and binding its root of trust.

Boot record

Previous releases of Android specified a boot reason format that used no spaces, was all lowercase, included few requirements), and which made allowances for other unique reasons. This loose specification resulted in the proliferation of hundreds of custom (and sometimes meaningless) boot reason strings, which in turn led to an unmanageable situation. As of the current Android release, the sheer momentum of near unparseable or meaningless content filed by the bootloader has created compliance issues for `bootloader_boot_reason_prop`.

Boot Image Header Versioning

Starting in Android 9, the boot image header contains a field to indicate the header version. The bootloader must check this header version field and parse the header accordingly. Versioning the boot image header allows future modifications to the header while maintaining backward compatibility.All devices launching with Android 9 must use boot header version 1.Devices launched before Android 9 using the legacy boot image header are considered as using a boot image header version 0. All devices launching with Android 9 must use the following structure for the boot image header with the header version set to 1.

For all devices launching with Android 9, the Vendor Test Suite (VTS) checks the format of the boot/recovery image to ensure that the boot image header uses version 1.

Android 10 updates the boot image header to version 2, which includes a section to store the device tree blob (DTB) image. Android 10 VTS tests verify that all devices launching with Android 10 use boot image header version 2 and include a valid DTB image as part of the boot/recovery images.

Partition Layout

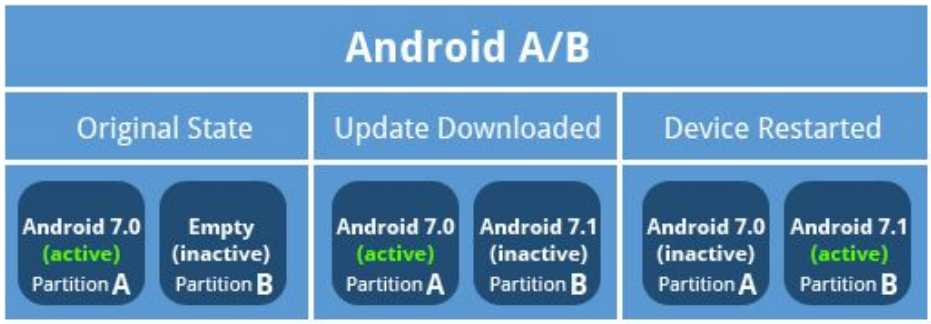
There are two types of partition layouts:

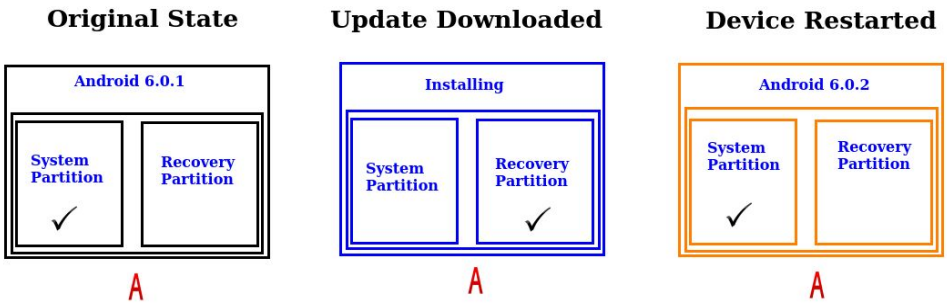
- A/B partition layout
- Non- A/B partition layout

A and B refers to partition names. A is one partition and B is another partition.Devices which do not have A/B partitions have the following partitions:

  - Boot: contains a kernel image
  - System:mainly contains the android framework(Android Framework is the stack of code that makes up the OS which includes native libraries which may or may not be a. It includes tools to design the UI, work with databases and handle user interaction)

- Vendor:applications and libraries that do not have source code on Android Open Source Project
- Userdata:contains user installed applications and data
- Cache:stores temporary data
- Recovery:stores the recovery image
- Misc: used by recovery





Non A/B Partition layout

Here is how the bootloader operates:

1. The bootloader gets loaded first.
2. The bootloader initializes memory.
3. If A/B updates are used, determine the current slot to boot.
4. Determine whether recovery mode should be booted instead as described in Supporting updates.
5. The bootloader loads the image, which contains the kernel.
6. The bootloader starts loading the kernel into memory as a self-executable compressed binary.
7. The kernel decompresses itself and starts executing into memory.
8. From `/system`, `init` launches and starts mounting all the other partitions and then starts executing code to start the device.

Android Verified Boot(AVB) is a program that strives to ensure that all the code comes from a trusted source rather than from an attacker or corruption.

- Recovery Image:
  To prevent Over the Air failures(OTA) failures on non-A/B devices, the recovery partition has to be self sufficient and not depend on other partitions.
- Fastboot is a protocol or a tool that can be used to re-flash partitions on your device
  - Flashing is nothing but re writing the data or files on a system
  - It is this small tool that comes with the Android SDK (Software Developer Kit), which is an alternative to the Recovery Mode for doing installations and updates.
  - While in fastboot mode, the file system images can be modified from a computer over a USB connection.
  - It can start even before Android loads, even under the circumstance when android isn't installed at all. Because of that, fastboot mode is useful for quickly updating the firmware without having to use the recovery mode.

Xv6 Bootloader

The first step after boot up for any system is the execution of BIOS which is stored on motherboard ROM. After BIOS finishes execution, it transfers control to the bootloader which is located in the first boot sector. Traditionally, the bootloader is located at 0x07C and BIOS jumps to that location after execution. On startup, the processor is in Real Mode i.e. it simulates an Intel 8088 which only has 8 16-bit registers. Part of the bootloader's job is to switch to Protected Mode so that the xv6 kernel can be loaded into memory. The xv6 bootloader mainly comprises of two parts, an assembly level bootloader (bootasm.S) and a high level bootloader (bootmain.c)

Assembly Bootloader *bootasm.S*

On beginning execution, it executes the assembly command 'cli' which disables BIOS interrupts as they are no longer required.

We do not know the contents of the segment registers after BIOS terminates so we set them all to 0 as soon as we enter the bootloader.

Since the processor is in Real Mode it generates 16-bit instructions which are then padded with the contents of segment registers to generate 20-bit registers when addressing memory.

The bootloader sets up the segment descriptor table gdt which has three entries for each segment: a null entry, a data entry and a code entry. In the code entry we set a flag for the special register cr0 which enables the system to run in protected mode.

Once we switch into Protected Mode, we can use advanced memory management schemes like paging. Until then the xv6 virtual address is the same as the x86 logical address which is the same as the linear address generated by the segment table.

After switching to protected mode, we initialize the data segment registers with SEG_KDATA i.e. segments for kernel data. The last step before switching to bootmain.c is to allocate memory for a stack. The stack starts at 0x07c (bootloader) and grows downward towards 0x0000 away from the bootloader.

C Bootloader *bootmain.c*

The C bootloader bootmain.c only returns if there is an error. Otherwise it passes control to the kernel. The job of bootmain.c is to set up an environment for the kernel, which is stored in the second sector.

The kernel is an .elf binary file. Executable and Linkable Format is a commonly used file format for executable files. It is flexible and cross-platform which is why it is the primary choice for operating system binaries, for whom the capability to execute on many different architectures is essential.

bootmain.c loads 4096 bytes of the ELF binary into memory at 0x1000 to gain access to the ELF headers. Next it checks if the loaded file is actually an ELF binary, checking if the magic number matches. If the magic number doesn't match, the kernel might be corrupt or unreadable and bootmain.c returns with an error.

The kernel is compiled and linked and is stored starting at virtual address 0x8010 (KERNBASE) which is mapped to physical address 0x1000. The virtual address is halfway through the 32-bit address space so as to make space for user space programs. The physical memory space between 0x0000 and 0x10000 is likewise occupied with I/O device drivers and the bootloader.

However, one major assumption being made is that the kernel is stored contiguously in memory. In the real world, this is rarely the case. They are stored as part of file systems over different sectors. Handling all these possibilities in the basic bootloader with the 512 byte limit is almost impossible. Thus we end up with a 2-step bootloader. The first bootloader runs within the 512 byte constraints in Real Mode and loads the secondary bootloader in protected mode which is far more powerful and can implement the complexity required to load the kernel. More commonly, UEFI (Unified Extensible Firmware Interface) is used which allows the PC to read and start bootloader in protected 32-bit mode.

Basic Bootloader

We have implemented a basic bootloader for a 32-bit x86 operating system which is capable of switching to Protected Mode and printing to the monitor.

Requirements:

Nasm:
The **Netwide Assembler** (**NASM**) is an assembler and disassembler for the x86 architecture. It can be used to write 16-bit, 32-bit and 64-bit programs. NASM is one of the most popular assemblers for Linux.

QEMU:
QEMU (Quick EMUlator) is an open-source emulator that can perform hardware virtualization. It emulates the machine's processor through binary translation and provides a set of different models for the machine, enabling it to run a variety of operating systems and architectures.

For this basic bootloader, we use a Floppy Disk Bootloader as it is simple and doesn't require a file system.

Real mode, also called **real address mode**, is an operating mode of all x86 CPUs. It is called Real Mode because addresses in real mode always correspond to real locations in memory. It is characterized by a 20-bit segmented memory address space. Real mode provides no support for memory protection, multitasking, code privilege levels or memory management schemes like paging.

Alternatively, **Protected Mode** is another operational mode of x86 CPUs. It allows system software to use features such as virtual memory and paging designed to increase an operating system's control over application software.

When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors.

CPU begins execution in Real mode, where it can access 8 16-bit registers, call BIOS functions using interrupts and address upto 512 bytes. We switch to Protected mode - where we can access 16 32-bit registers and implement memory management schemes like paging.
We do this switch using an interrupt to activate the A20 gate. This is disabled by default to keep backward compatibility with 16-bit computers. We move value into the required register and call the interrupt.

```
mov ax, 0x2401
int 0x15 ; enable A20 bit

mov ax, 0x3
int 0x10 ; set vga text mode 3
```

Next we enable 32-bit instructions and access to all 32 registers using the Global Descriptor Table which defines a 32-bit code segment. We define the structure of the GDT table according to standard by initializing the constants in memory using assembly instructions. The lgdt instruction loads the GDT from memory location [gdt_pointer], and then we proceed by setting the protected mode bit in the special control register cr0 and jump to the boot segment.

The last step is writing to the monitor. Video memory is mapped to the location 0xb8000/ We store this memory location in a register. The VGA text buffer also has a specific format which we can use to adjust foreground colour, background colour etc. In another register we store the memory location of the text to be printed. Finally, we iterate in a loop, check if there's still a character to print, print it and increment the register holding the memory location of the text to be printed.

Execution Steps:

```
nasm -f bin boot2.asm -o boot2.bin
qemu-system-x86_64 -fda boot2.bin
```

Output: