Booting and Installing the Operating System. Boot Loaders

Grado en Informática 2024/2025

Departamento de Computación
Facultad de Informática
Universidad de Coruña

Antonio Yáñez Izquierdo

Contents I

Contents II

Installing an O.S.

Installing an O.S.
→O.S. Installation

Installing an O.S.

- the most common use of O.S.s is having them *"installed"* onto computers, and being run from the computer's storage devices
  - there are also some *"live"* O.S.s that don't require installation but usually have limitations concerning what users can do and what software can be added
- *installing* is the process by which we put the O.S. files in one (or more) of the storage units of the system, thus allowing the system to execute the OS directly

Installing an O.S.

- the process of installing an O.S. usually includes the following steps
  a. booting the new O.S. system from some installation media
  b. writing the O.S. files to some storage media
  c. doing some configuration to allow the O.S. to be booted from the storage media: **Installing the Boot Loader**
  d. rebooting the system

Installing an O.S.

a. to boot the system from some installation media we, obviously, need the installation media
   - we can get the media already prepared
   - we have to prepare them ourselves

b. writing the O.S. files to some storage media usually requires partitioning the drive

c. allowing the O.S. to be booted from the storage media requires installing a *boot loader*

Installing an O.S.
→Installation media

Installation media

- the installation media we use depends on the devices the system is capable of booting from
- nowadays floppy disks and tapes are seldom used, apart from disks, the most common devices used for booting are
  - CD/DVD devices
  - usb devices
  - Network Interface Cards

Booting from the network

- modern systems are capable of booting from the network (usually via the *Netboot* or *PXE* protocol)
- booting from the network requires the existence and configuration of a boot server, that provides both the network configuration and the data necessary to boot
- one of the most usual ways of installing O.S.s is what it's called a *network installation*, which consists of
  - booting from a CD/DVD or usb device
  - doing some basic network configuration
  - retrieve the O.S. files from the network, usually through the `http` or `ftp` protocols

Installing an O.S.
→Preparing the media

Preparing the media

- commercial operating systems usally provide the installation media
- non commercial operating systmes usually provide installation images to be downloaded from the network
  - full sized images: this images may contain all the files necessary to perform the complete installation
  - smaller images to perform a *network installation*

Installation images

- the most common images nowadays are
  - ISO images (to be burnt directly on CD/DVD)
  - special images to be copied to an usb stick
- if we are using some virtualization software we can install directly from the ISO image
- some virtualization software allow booting from usb. Should that not be the case (as in Virtualbox current release), we can create a VirtualHardDisk that is in fact one of the system real usb drives. For example, we can create virtual hard disk in Virtualbox, named *DisguisedUSB.vmdk* which is in fact the `/dev/sdc` drive in the host machine, withe the line

```
# VBoxManage internalcommands  createrawvmdk -filename "DisgusedUSB.vmdk" -rawdisk /dev/sdc
```

ISO images

- ISO images are to be burn directly to the CD/DVD media
  - they contain an image of the filesystem, **they are not a file to be copied onto a CD/DVD file system**
  - most CD/DVD burning software has an option *burn image* or something similar
- booting CD/DVD media can be created with any burning software (`cdrecord`, `k3b`, `nero` ...)
- the images contain the booting code in them (the boot loader program)

usb images

- becoming increasinly common, usb images are supplied
- in order to provide the usb with the adecuate boot code: usb images must be copied directly to the usb device
  - using the `dd` command
  - using the `cat` or `cp` command directly to the device file
- sometimes we are given a boot block to be copied to the usb device using `dd` and a file (or set of files) to be copied to the usb file system

usb images from iso files

- in the case we are given only the iso images but we need to boot from an usb device
  a. some iso files con be copied directly to the usb device
  b. install a boot loader onto the usb and copy the iso image to it
  c. use one of the utilities that does b) in an automated way, for example `unetbootin`

The boot process

The boot process
→Booting

Booting

- booting is the process by which the O.S. is loaded and the system is ready to be used by users
- as the O.S. provides the services necessary for the system to be usable
  - those services would be necessary to load the O.S.
  - the O.S. must be loaded without those services in what we call the *bootstrapping* process
  - usually a loader of the O.S. is loaded and executed and it is this loader that loads the O.S.

automatic booting

- the booting process is very hardware dependent
- we can distinguish between two ways of booting
  - automatic
  - manual
- *automatic booting* is the way the system boots most of the times.
  - it does not requiere human intervention
  - the system boots by it's own and a multiuser environment is available after booting

manual booting

- in *manual booting* the system boots to *single user mode*: only the *root* can login
  - *single user mode* is also called *maintenance mode*
  - usually the system boots to *single user mode* when it encounters some problem during boot, although it can also be told to boot this way
  - System V distinguish several multi-user modes (called *runlevels*), BSD only has *single user mode* and one multi-user mode. *systemd* linux systems also distinguish several multi-user modes (called *targets*)

The boot process
→Booting steps

booting steps

- although it is very dependent on the hardware, the booting process can be thought of consisting of the following steps
  1. loading and executing the *motherboard firmware* boot program
  2. loading and executing the *boot loader* (how this is done depends on the type of motherboard firmware: BIOS, UEFI, openboot ...). It can consist of several *stages*
  3. loading and executing the *unix* kernel
  4. running the initialization *scripts* and starting the system services

first booting step: motherboard firmware

- the motherboard firmware contains some code to start the booting of the machine
- how this code works depends on the type of firmware. It is a very simple code and it usually involves one of these two alternatives
  a) the first stage of the boot loader is at a predefined block (usually the first) of some device
  b) the first stage of the boot loader resides in some specific file located at some specific directory

first booting step: motherboard firmware

- *Moderboard firmware* con be configured to decide which device or which file use to boot from (depending on the type of firmware)
- the *Moderboard firmware* is dependent on the architecture (sparc, amd64, ia32, arm . . .) and thus the booting code in it. So bootloaders behaviour and installation procedures vary greatly from one architecture to the next
- For the intel/amd x86 platform the two more widespread *moderboard firmware* standards are the BIOS standard (which boots using alternative *a*) and the UEFI standard (which boots using alternative *b*)

second booting step: the boot loader

- the boot loader is (*should be*) a simple program which only has to load the kernel
- its configuration file has only two essential items to define
  - which kernel to load (and where to find it). `bootloaders` capable of loading different O.S.s must also be told what os the kernel is
  - which device to use as `root` file system when that kernel is loaded, so it can pass that information to the kernel
  - in the case of some modular kernels, it might be also needed some form of access to the kernel modules. (For example in linux its usual to have a memory disk image containing modules, called *initrd* or *initramdisk*)

second booting step: the boot loader

- **UNFORTUNATELY** most of the present bootloaders include some *non essential* options such as *splash images, menus . . .* which makes them bigger, slower and more tedious to install and configure.
- some boot loaders understand filesystems, so the kernel location can be specified directly in the boot loader configuration file
- some boot loaders DO NOT UNDERSTAND filesystems, so some additional steps need to be taken in adition to specifying the kernel in the boot loader configuration file
- the boot loader can be unistalled, reinstalled ot have its configuration changed from the O.S.

third booting step: loading and executing the *unix* kernel

- the kernel is loaded in memory and trasfered control
  - in linux the kernel resides in the file `/boot/vmlinuz....`
  - in solaris 10 the kernel resides in the file `/platform/i86pc/multiboot`
  - in solaris 11 the kernel resides in the file `/platform/i86pc/kernel/amd64/unix`
  - in openBSD the kernel resides in the file `/bsd`
  - in FreeBSD the kernel resides in the file `/boot/kernel/kernel`
  - in NetBSD the kernel resides in the file `/netbsd`

third booting step: loading and executing the *unix* kernel

- it creates its data structures, probes for devices and performs initialization routines
- creates some *special system processes* (*sched, paged....*) and *init* (*systemd* in some linux distributions), the first *"user process"* in the system which will initiate the various services

fourth booting step: initialization scripts. starting the system services

- init reads its configuration file (`/etc/inittab`) where it gets the *runlevel to boot to*
  - *systemd* has a default *target* to boot to, that can be changed with *systemctl set-default*.
- if there is some kind of error or the system is configured to boot into single user mode, a root shell is created with only the *root filesystem* mounted
- otherwise the scripts initiating the system services are started (`/etc/rc*` on BSD systems or the *scripts* in directories `/etc/rc?.d` on System V systems)

The boot process
→Booting with BIOS type firmware

booting with BIOS type firmware

- by construction, when the system is powered on (or when a reset is done) the motherboard executes the code at certain memory addresses, there resides the firmware
- this code contains some initialization routines and sometimes access to a system configuration menu
- a device is defined as the first boot device (CD/DVD, disk, tape, usb, floppy . . .). An attempt is made to boot from that device, if unsuccessful, the defined as second boot device is tried and so on

booting with BIOS type firmware

- this firmware **DOES NOT UNDERSTAND** filesystems so: for this type of firmware *booting* from a device means **loading the first block and executing the code in it**
- there's no interface to access this firmware from the O.S.
- changes like which device to boot from can only be made from the firmware configuration program (before booting any O.S.)

booting with BIOS type firmware

- in this type of firmware booting from a device means **LOADING THE CODE AT THE FIRST BLOCK OF THAT DEVICE AND EXECUTING IT**
- when the device is a disk the first block of the disk contains some boot code and the partition table.
- the code at this disk block (usually called MBR, Master Boot Record) is loaded and executed

booting with BIOS type firmware

- the **usual** code to have in this first block of disk is to have very simple program, that reads the partition table, looks which partition has the *active* flag on and then loads that partition's first block and executes it: the boot loader code (at least its first stage) can be copied to that disk block
- there's also the possibility that the boot loader code (at least its first stage) is copied to this block (MBR). Should that be the case, that boot loader gets loaded regardles of what the *active* partition is. We say that the boot loader has been installed onto the MBR

booting with BIOS type firmware

- the partitions scheme for this type of firmware is called MBR partitions
- to install a boot loader in one of this system we can
  - install it at the Master Boot Record (first block on disk): that bootloader will execute upon switching the machine on regardeless of the *active* partition
  - install it at the first block of the partition: that bootloader will execute when the partition is marked active and there's no other bootloader at the MBR
  - for media without partitions we install it at the first block of the media

The boot process
→Booting with UEFI type firmware

booting with UEFI type firmware

- by construction, when the system is powered on (or when a reset is done) the motherboard executes the code at certain memory addresses, there resides the firmware
- this code contains some initialization routines and sometimes access to a system configuration menu
- there's an interface to access the firmware booting configuration from the O.S. Variables of the firmware (efi variables) can be read and changed from the O.S.

booting with UEFI type firmware

- this firmware **UNDERSTANDS THE FAT FILESYSTEM** so a boot loader is just a program in a FAT filesystem
- this firmware is capable of running executables in its own format (.efi). This is used to run the bootloaders
- the boot loader is actually a file in a FAT filesystem that gets executed by the firmware

booting with UEFI type firmware

- disks must be partitioned using a GPT partition table
- there must exist, at least, an EFI System Partition (ESP)
  - this partition must be formatted using either the FAT16 or FAT32 filesystems
  - this partition holds, among other things, the EFI drivers and the EFI bootloaders
  - Operating Systems typically place their bootloaders in a subdirectory of the EFI directory in the ESP
  - booting different operating systems can be done at the firmware level
- **unless otherwise specified**, the firmware will run the program `\EFI\BOOT\BOOTX64.EFI`

booting with UEFI type firmware

- installing a boot loader in a machine with UEFI firmware means
  1) copying the executable file (.efi format) to the EFI System Partition
  2) if we want that boot loader to be run at boot time we must tell the firmware to: we can do so in the firmware setup program or from the O.S.
- should this executables be required to be signed, the booting procedure would be known as *secure boot*
- the EFI variables define which of these .efi files must be loaded when booting. If nothing specific gets defined, the file `\EFI\BOOT\BOOTX64.EFI` on the (first) ESP will get loaded upon booting

Basic disk partitioning

Basic disk partitioning
→disks and partitions

disks

- disks still are the method of choice to run the Operating System from
- nowadays all disks use Logical Block Addressing instead the old CHS interface although they still report a (fake) CHS geometry
- the disks also report a sector size of 512 bytes although internally a 4096 byte sector might be used

disks

- disks are used to create filesystems on them
- several filesystems can exist on a disk device in what we usually call *partitions*
- several *partitions* can be combined into one filesystem via **L**ogical **V**olume **M**anagement software

partitions

- a disk is usually divided into several units called *partitions*
  - BSD systems sometimes refer to *partitions* as *slices*
- filesystems are created in *partitions*, usually one filesystem in each *partition* although several *partitions* can be combined into one filesystem via Logical Volume Management Software
- we even can install different O.S.s in different partitions

partition tables

- each disk has a table, usually located at the begining of the disk (can be one or more blocks), that defines the partitions on that disk
- there are many standard formats to that table
  - MBR partitions
  - BSD disklabel
  - Solaris VTOC label
  - GUID Partition Table (GPT)
  - others...(Amiga Rigid Disk Block, RDB), (Apple Partition Map, APM)

Basic disk partitioning
→MBR partition table

MBR partitions

- the partition is located in the first sector of the disk
- widespread in PC architecture
- used mainly in Windows a linux systems
- up to 4 partitions, called *primary partitions*, can be defined in a disk

MBR table format

offset	**size**	Description
0x000	446	reserved
0x1be	16	partition entry 1
0x1ce	16	partition entry 2
0x1de	16	partition entry 3
0x1ee	16	partition entry 4
0x1fe	2	0xaa55 (little endian)

MBR partitions

- one of the partitions can be defined as *extended partition*
- this partition can be subdivided into what is called *logical partitions*
- the first sector of that partition, called EBR (Extended Boot Record), has the same format as the MBR table except for
  - only the first two entries are used
  - if more partitions are needed, one of these two is defined as *extended partition*, thus allowing for and *"infinite"* number of partitions

EBR format

offset	**size**	Description
0x000	446	reserved
0x1be	16	partition entry 1
0x1ce	16	partition entry 2
0x1de	16	zeroes
0x1ee	16	zeroes
0x1fe	2	0xaa55 (little endian)

format of a partion entry

offset	**size**	Description
0x00	1 byte	80h for active partition, otherwise 00h
0x01	1	head of partition start
0x02	2	cylinder/sector (10/6bits) of partition start
0x04	1	code of partition type
0x05	3	CHS of partition end
0x08	4	LBA partition start
0x0C	4	partition size

creating MBR partitions

- partitions on disks using the MBR partition scheme are limited to 2 Terabytes
- MBR partitions can be created, and manipulated with
  - `fdisk` utility on BSD systems
  - `fdisk` or `cfdisk` utility on linux systems
  - `fdisk` or `format` on solaris/intel systems

Basic disk partitioning
→BSD disklabel

BSD disklabel

- BSD systems and derivatives use disklabels to partition the disk
- a disklabel can contain up to 8 partitions, designed with letters (a through h)
- openBSD disklabel can hold up to 16 partitions, designed with letters (a through p)
- partition a is the root filesystem (also contains the boot code)
- partition b is the swap space
- partition c (or d) represents the whole disk

BSD disklabel

- when installing a BSD O.S. on a system which has MBR partitions
  - One of the MBR partitions is labeled with code **a6** (openBSD), **a5** (FreeBSD), **a9** (netBSD) ...
  - a disklabel is created in the first sector of that MBR partition
  - the MBR partitions are often refered as *slices*
  - partition c (or d, depending on the BSD flavour) of the disklabel represents the whole disk, not just the *BSD MBR partition

BSD disklabel

- Some systems require for a partition to be mounted **has** to be defined in the disklabel, even if it is a MBR partition
- disklabel partitions are restricted to 2 Terabytes (in Solaris to 1 Terabyte)
- openBSD disklabels do not have the 2 Tb limitation
- Solaris system uses a *variation* of the BSD disklabel called VTOC (Volume Table Of Contents).

BSD disklabel

- to access the disklabel, the following commands can be used
  - `disklabel` on openBSD and netBSD
  - `bsdlabel` on freeBSD
  - `format` on Solaris

Basic disk partitioning
→GPT: GUID Partition Table

GUID Partition Table

- defined as part of the EFI (Extensible Firmware Interface) standard
- sometimes refered to as the *EFI label*, or EFI partition table
- the MBR uses 32 bits for Logical Block Adressing, hence its limitations in size
- GPT uses 64 bits for LBA, this limits the maximun partition size to $2^{64} - 1$ sectors
- most modern O.S. support GPT although some still have some restrictions to boot from such partitions

GUID Partition Table

- two copies of the GPT exist, the *primary GPT* at the beginning of the disk, and the *secondary GPT* at the end
- GBT uses logical block addessing
- the first sector of the disk has a MBR partition table called *protective MBR* that allows the disk to be booted from a system with traditional BIOS
- following sector is the header of the primary GPT
- the GPT partition table consists of 128 bytes entries. The minimun size of the table is 16Kbytes

format of a GPT partion entry

offset	**size**	Description
0x00	16 bytes	GUID partition type
0x10	16 bytes	Partition GUID
0x20	8 bytes	Partition start LBA
0x28	8 bytes	Partition end LBA
0x30	8 bytes	Attribute flags
0x38	72 bytes	Partition name

Comparision of MBR and GPT

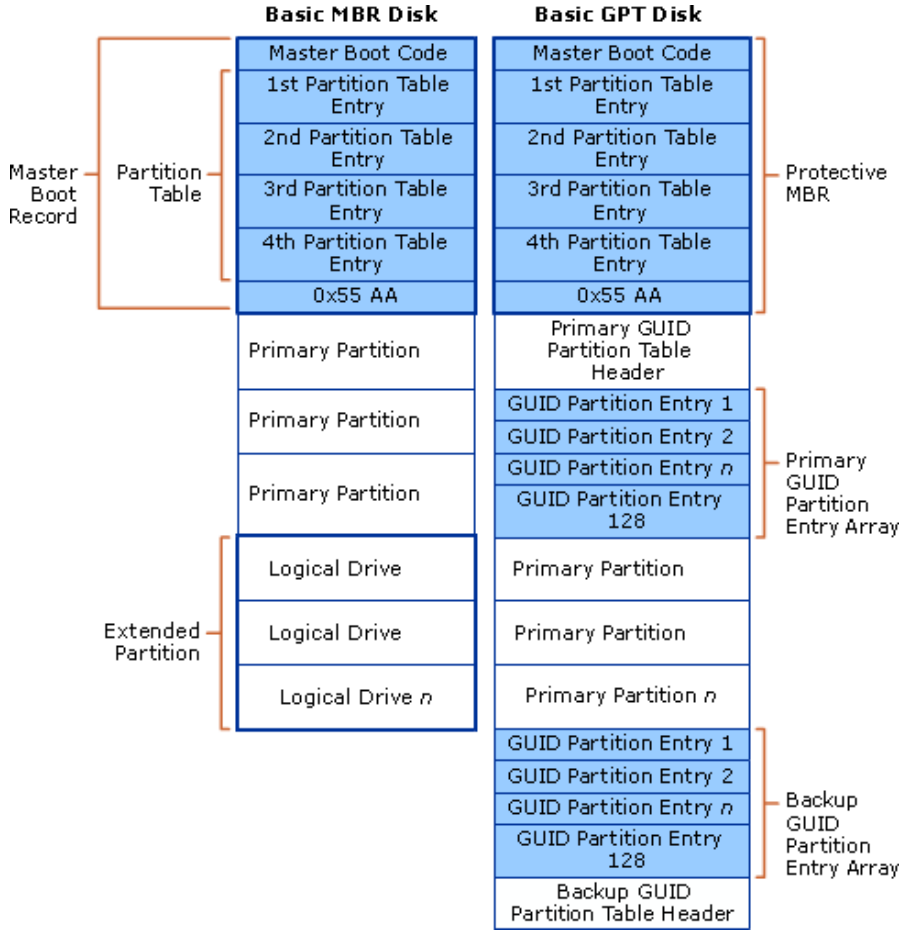

Figure: https://i-technet.sec.s-msft.com/dynimg/IC197579.gif

GPT Partition Table

- to access the GPT the following commands can be used
  - `parted` and `gdisk` on linux (some times `fdisk` will do)
  - `gpart` on freeBSD
  - `gpt` on netBSD
  - `format -e` on Solaris

Basic disk partitioning
→sharing disks among O.S.s

sharing disks between O.S.s

- with the term *sharing disks between O.S.s* we can refer to two diferrent things
  a) have several O.S.s installed on the same disk
  b) have some disk space that can be accesed from different O.S.s
- We'll deal now with a), as item b) usually implies recognizing the partition table format, having support for the filesystem in question and mounting it

sharing disks between O.S.s

- two issues must be considered when having several O.S. on the same disk
  - allocating space to each one of them
  - being able to boot any of them
- allocating space to different O.S.s on the same disk is done through *partitions*

sharing disks between O.S.s

- linux and windows system can share disks via the MBR or GPT partitions
- On BIOS systems: BSD systems require a disklabel. In order to share disk with other systems (BSD or not BSD), a disklabel must be created into one of the MBR partitions for each BSD-type systems
- In UEFI systems: BSD systems can also share disk with other system using GPT. FreeBSD and NetBSD use GPT partitions, OpenBSD will create a *disklabel* inside one of the GPT partitions

sharing disks between O.S.s

- On a BIOS system, Solaris will create its VTOC on a MBR partition when sharing disk with other O.S.
- Solaris 11 will use GPT partitioning when in a UEFI system
- An EFI BIOS is needed to boot from GPT partitions (as a matter of fact, BIOS systems can actually boot from a GPT partioned disk by creating BIOS BOOT partition on the disk. Consider this some form of a *hack*)

booting different O.S.s on the same disk

- as seen in a previous section, an O.S. needs a *boot loader* to be booted
- installing the *boot loader* is part of the installation process of the O.S.
- some boot loaders are only capable of booting one O.S.
- other boot loaders are capable of booting different O.S.s

booting different O.S.s on the same disk

- we have several solutions to booting different O.S.
  - **BIOS firmware**: install boot loaders for each O.S. and change the *active partition* to boot one of them (BIOS firmware)
  - **UEFI firmware**: install boot loaders for each O.S. and configure the firmware to boot one of them
  - install a boot loader capable of booting all of the O.S. and get that boot loader loaded when the system starts
  - install boot loaders for each O.S. and have one of them, capable of loading the other O.S. or chainload their boot loaders, loaded at boot time

Boot loaders

boot loaders

- BSD systems have their own specific boot loaders,
  - freeBSD can change the Master Boot Record for a very simple program (*boot0cfg* that gives the option to chainload to other (primary) partitions
- linux has, among others, `lilo`, `silo` (for Sparc), and `grub` which is becoming the standard today
- Solaris had its own boot loader, but since version 10, it uses a modified version of `grub` to boot

boot loaders that can load several O.S.s

- the `silo` bootloader, which is specific for the Sparc architecture, can load Solaris and linux (it's specific to sparc firmware)
- `lilo` in linux can chainload to other boot loaders (it's specific to BIOS firmware).
- `grub` can chainload to other loaders, and load directly linux, FreeBSD, netBSD, openBSD, Solaris. Grub2 also supports UEFI booting
  - to load Solaris a specially modified version of `grub` is used

Boot loaders
→boot loader installation

boot loader installation

- Boot loader installation depends on the boot loader being installed
- The device where the boot loader is to be installed must be supplied
  - sometimes as a parameter to the bootloader installation program (for example grub's `grub-install` or OpenBSD's `installboot`)
  - sometimes in the bootloader configuration file (for example linux's `lilo`)
- in UEFI systems it's usually enough to copy the bootloader executable file to the ESP (Efi System Partition) and arrange for the firmware to boot it

Boot loaders
→boot loader execution

boot loader execution

- There exist bootloaders that can boot several operating systems or that can *chainload* to other bootloaders
- We can have several boot loaders installed on the same system
  - several operating systems with their own bootloaders
  - several bootloaders to boot the same operating system
  - several O.S. with several bootloaders to boot
- When we have several bootloaders installed on the same system. *How do we decide which one of them gets executed?*

boot loader execution in BIOS firmware

- In BIOS type firmware the first block of the booting device gets loaded and executed upon powering on
- We can use the firmware configuration program (usually interrupting by pressing some key after powering on) to change the booting device (to DVD, usb, second disk . . .)
- Some boot loaders allow us to configure menus to choose which O.S. to load or which other boot loader to chainload. We usually put one of them as the boot loader that gets exxecuted to so thet we can choose choose which O.S. will boot
- That's about all what we can do.

boot loader execution in UEFI firmware

- In this type of firmware, the default is to load the bootloader at `\EFI\BOOT\BOOTX64.EFI` (note that FAT filesystems are not case sensitive
- As with the previous case:
  - We can have several boot loaders installed.
  - We can use the firmware configuration program (usually interrupting by pressing some key after powering on) to change which bootloader gets executed
  - Some boot loaders allow us to configure menus to choose which O.S. to load or which other boot loader to chainload. We usually put one of them as the boot loader that gets executed to so thet we can choose choose which O.S. will boot

boot loader execution in UEFI firmware

- In adition using the firmware configuration program, we can change the bootloader to run by default (instead of `\EFI\BOOT\BOOTX64.EFI` , from the Operating System
- The program `efibootmgr` is becoming the standard to do that
- This program also allows us to change the bootloader for the next boot only, to add new entries to the firmware booting ...

the Grub boot loader

The Grub boot loader

- **Gr**and **U**nified **B**oot Loader is the boot loader of choice in linux since several years ago
- It can boot directly linux and other O.S.: freeBSD, OpenBSD, Solaris . . . or chainload to other bootloaders
- Highly configurable, can display splash images, a menu . . .
- It understands different file systems (ext2fs, ext4fs, ntfs, fat, ufs . . .) and different partition types (MBR, gpt . . .) through loadable modules
- The boot menu can be edited at boot time and it has a rescue mode command line interepreter capable o accesing filesystems

The Grub boot loader. Versions

- There are two versions
- Grub version 1, also refered to as Grub legacy
  - Configuration file editable by hand, tipycally `/boot/grub/menu.lst`
  - Can only boot systems with BIOS type firmware
- Grub version 2, the one that is being installed at present by mostly every linux distro
  - Script generated configuration file (non editable by hand). Typically `/boot/grub/grub.cfg`
  - Can boot both BIOS type and UEFI type firmware
- Both versions provide a boottime-editable boot menu and a rescue mode command line interepreter capable o accesing filesystems

The Grub boot loader. Basic usage

- to install the grub with BIOS type firmware: `grub-install` *`device`*. Example to install it in partition 2

  ```
  # grub-install /dev/sda2
  ```

  should we want it installed onto the MBR

  ```
  # grub-install /dev/sda
  ```

- to install the grub with UEFI type : `grub-install --efi-directory` *`dir_with_ESP`*: Example

  ```
  # grub-install --efi-directory=/boot/efi
  ```

the Grub boot loader
→Grub legacy

The Grub Legacy boot loader. Basic usage

- Only capable of booting in BIOS firmare platforms
- To change de grub legacy configuration
  - edit the file `/boot/grub/menu.lst`
- *grub* reads it's configuration file when it loads and understands several filesystem structures so it can load kernels directly
- For BIOS firmware it can be installed either in the MBR or in the superblock of the partition.
- To install it we can use the `grub-install` or `install-grub` commands, depending on the system where we are
- To unistall it we overwrite it with another bottloader (or with a standard MBR code, if its installed onto the MBR)
- Can chainload to other bootloaders

grub legacy examples

- The/boot/grub/menu.lst
- an example of configuration file booting linux and freeBSD

```
title linux with kwenel 2.4.7-10
        root (hd0,4)
        kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda5
        initrd /boot/initrd-2.4.7-10.img
title FreeBSD
        root (hd0,2,a)
        kernel /boot/loader
```

grub legacy examples

- an example of configuration file booting opensolaris and chainloading linux and openBSD

```
title Opensolaris-snv130
findroot (pool_rpool,0,a)
bootfs rpool/ROOT/Opensolaris-snv130
splashimage /boot/solaris.xpm
foreground d25f00
background 115d93
kernel$ /platform/i86pc/kernel/$ISADIR/unix -B $ZFS-BOOTFS,console=graphics
module$ /platform/i86pc/$ISADIR/boot_archive
#============ End of LIBBE entry =============
title OpenBSD
        rootnoverify (hd0,2)
        chainloader +1
title linux 64 bits
        rootnoverify (hd0,5)
        chainloader +1
```

the Grub boot loader
→GRUB 2: Grand Unified Boot Loader

grub version2

- Capable of booting both BIOS and UEFI systems
- It's the bootloader of choice for most linux distros
- A slightly modified version is supplied with de Solaris 11 O.S.
- Can boot linux, solaris, most BSD's and chainload to other bootloaders

grub version2: Installation

- in linux or BSD platforms we can install with the commands `grub-install` or `grub2-install`
- we supply the device where we want it installed or, in the case of uefi systems, the directory where we have mounted the ESP partition
- In solaris 11 we install it with the command `bootadm install-bootloader`
- For EFI firmware, it is typically installed in one subdirectory of the EFI directory in the ESP as `grubx64.efi`

grub version2: Configuration

- Its configuration resides in the file `grub.cfg` (typically in `/boot/grub` or `/rpool/boot/grub`)
- In the case of EFI firmware it can also reside in the ESP (typically mounted under `/boot/efi`).
  - For example, in fedora we can find it in `/boot/efi/EFI/fedora/grub.cfg`

grub version2: Configuration

- This file is not intended to be edited directly, instead we must
  - edit the corresponding file in `/etc/grub.d` (the contents of the files `40_custom` and `41_custom` will get copied into `/boot/grub/grub.cfg` by the program `update-grub`)
  - update the grub configuration file (`/boot/grub/grub.cfg` with the commands `update-grub` or `grub-mkconfig`

```
# grub-mkconfig -o /boot/grub/grub.cfg
```

  - in solaris 11 we'll use the command `bootadm` to change its configuration
  - some versions allow for some customization to the configuration file at boot time using `/boot/grub/custom.cfg`

grub2

- examples of the *menuentries* to load freeBSD and netBSD

```
menuentry "FreeBSD" {
  insmod ufs2
  set root='(/dev/ad4,msdos1)'
  search --no-floppy --fs-uuid --set 4c0029f407b3cd1d
  kfreebsd /boot/kernel/kernel
  kfreebsd_loadenv /boot/device.hints
  kfreebsd_module /boot/splash.bmp type=splash_image_data
  set kFreeBSD.vfs.root.mountfrom=ufs:ad4s1a
}

menuentry "NetBSD on sda1" {
  insmod ufs2
  set root=(hd0,msdos1)
  knetbsd /netbsd --root=wd0a
}
```

grub2

- examples of the 40_custom file to chainload Solaris and openBSD

```
#!/bin/sh
exec tail -n +3 $0
# This file provides an easy way to add custom menu entries.  Simply type the
# menu entries you want to add after this comment.  Be careful not to change
# the 'exec tail' line above.

menuentry "Solaris grub" {
        set root=(hd0,msdos2)
        chainloader +1
}
menuentry "openBSD" {
        set root=(hd0,msdos3)
        chainloader +1
}
```

grub2

- example of grub in EFI firmware chainloading to load Solaris' grub

a) Solaris and fedora use the same ESP

```
menuentry Solaris {
  chainloader /EFI/ORACLE/grubx64.efi
  boot
}
```

b) Solaris and fedora use different ESP

```
menuentry Solaris {
   insmod part_gpt
   insmod fat
   set root=(hd0,gpt1)
   chainloader (${root})/EFI/ORACLE/grubx64.efi
   boot
}
```

other boot loaders

other boot loaders
→lilo and elilo bootloaders

lilo

- it **was** the boot loader of choice in linux for BIOS type firmware
- can chainload other O.S. bootloaders
- it is configured through the file `/etc/lilo.conf`
- it can't read any configuration file when booting so after making any change to its configuration file, `/sbin/lilo` must be run to create the executable and put it in the right place
- it is now being deprecated

example lilo configuration

- example of lilo configuration file chainloading another O.S.. The bootloader will be installed to de masterboot (/dev/sda) and the root linux partition is on /dev/sda3

```
...
boot=/dev/sda
root=/dev/sda3
image=/boot/vmlinuz-2.6.38-2-amd64
        label="linux 2.6.38"
        initrd=/boot/initrd.img-2.6.38-2-amd64
        read-only
other=/dev/sda4
   label="openBSD"
```

elilo

- a very simple UEFI boot loader for linux.
- Consists of just two files
  - **elilo.efi**: the loader itself, can be reanamed to bootx64.efi in the EFI/BOOT directory (to be loaded by default), chainloaded from other boot loader, or loaded directly by the UEFI firmware
  - **elilo.conf**: configuration file. Must reside in the same directory of the ESP where elilo.efi is

elilo

- Sample of **elilo.conf** (paths are always in the ESP)

```
default=linux
prompt
timeout=50
image=/EFI/debian/vmlinuz-4.6.0-1-amd64
        label=linux
        initrd=/EFI/debian/initrd.img-4.6.0-1-amd64
        read-only
        root=/dev/disk/by-uuid/2d473ffb-3253-4c32-a68c-3015d4b439c9
        append=""
image=/EFI/debian/memtest86.bin
        label=test
```

other boot loaders
→syslinux

syslinux and syslinux-efi

- boot loader that loads linux from FAT filesystems in machines with BIOS firmware
- To install it we issue the command `syslinux --install device`. This installs the loader and its neccesary files (`ldlinux.c32, ldlinux.sys`) on the media
- Its behaviour is controlled via the `syslinux.cfg` file on the root directory of the filesystem.
- Sample `syslinux.cfg`

```
DEFAULT linux
  SAY arrancando con SYSLINUX..............
LABEL linux
  KERNEL vmlinuz-3.16.0-4-amd64
  APPEND ro root=/dev/sda5 initrd=initrd.img-3.16.0-4-amd64
```

syslinux-efi

- boot loader that loads linux in machines with UEFI type firmware (it is the efi version of *syslinux*)
- to install it we copy the efi executable (`syslinux.efi`), the modules and its configuration file onto a directory in the ESP
- its behaviour is controlled via the `syslinux.cfg` file in the same directory as the efi executable file
- this loader only understands the FAT filesystem and cannot chainload to other loaders, so the kernels to load must reside in the ESP

sample syslinux-efi configuration file

```
DEFAULT ubuntu
TIMEOUT 10

UI vesamenu.c32

LABEL ubuntu
  MENU LABEL ubuntu
  LINUX /EFI/KERNELS/kernel-u
  INITRD /EFI/KERNELS/init-u
  APPEND root=/dev/sda2 ro

LABEL fedora
  MENU LABEL fedora
  LINUX /EFI/KERNELS/kernel-f
  INITRD /EFI/KERNELS/init-f
  APPEND root=/dev/sda6 ro
```

other boot loaders
→systemd-boot

systemd-boot

- Not even the booting of the O.S. is safe from systemd's crawling for new funcionality
- The package name is `systemd-boot`
- Once the package is installed the bootloader can be installed with the command *bootctl install*
  - It copies systemd-boot to the EFI partition
  - it creates an entry in th efi variables to boot that loader

systemd-boot

- systemd-boot is able to chainload.
- its configuration (by default) is stored in `/loader/loader.conf` (in the EFI partition) (default and timeout options)
- one `.conf` file in `/loader/entries` (in the EFI partition) for each kernel (or efi loader) to load
- paths included there are relative to the EFI partition

systemd-boot examples

- example of `loader.conf`

```
default  mi_bsd.conf
timeout 3
```

- example of `mi_linux.conf`

```
title   Arrancar Mi Distro de Linux
linux   /KERNELS/kernel6
initrd  /KERNELS/initrd
options root=UUID=c84ec471-5958-4d98-b437-3ea2d6f73cde
```

- example of `mi_bsd.conf`

```
title   Arrancar Mi Distro de BSD
efi    /EFI/BSD/BSDloader.efi
```

other boot loaders
→rEFInd

rEFInd

- loader and bootmanager for UEFI type firmware
- evolution of the *rEFIt* boot manager.
- although it has a configuration file (`refind.conf`) in the same directory as the refind efi executable, it scans most of the subdirectories of the EFI directory, as well as every filesystem it can access for files with names that end in .efi or that begin with vmlinuz, bzImage, or kernel
- the scanning con be prevented with the `dont_scan_volumes`, `dont_scan_dirs`, `...` options in its configuration file.
- we can also create specific menuentries (even with submenus) in its configuraion file
- it can load other loaders

rEFInd configuration file

- here we have an example of some parts of an actual configuration file

```
...
dont_scan_dirs ESP:/EFI/BOOT,/EFI/ubuntu,/EFI/fedora,/EFI/KERNELS,/EFI/syslinux
...
dont_scan_volumes "a91b2fee-4466-49ce-8b2c-7c6724af5ca4"
...
menuentry ubuntu  {
    icon EFI/refind/icons/os_ubuntu.png
    loader /EFI/ubuntu/grubx64.efi
    submenuentry "KernelDirecto" {
        loader /EFI/KERNELS/kernel-u
        initrd /EFI/KERNELS/init-u
        options "ro root=UUID=a304886e-a9b6-4804-9678-a56524475c83"
    }
}
....
```

other boot loaders
→BSD bootloaders

BSD bootloaders

- In adition to GRUB (which can also boot linux and solaris) BSD system usually come with their specific bootloader.
- It is a simple bootloader that only loads BSD.
- In NetBSD and OpenBSD can be installed with the `installboot` command
- in FreeBSD it can be installed with the `gpart` utility
- The boot code inside the MBR can be replaced with the `fdisk` utility. In FreeBSD we can also use `gpart` or `boot0cfg`
- The bootcodes can be found in `/boot/`(FreeBSD) or `/usr/mdec/` (NetBSD and OpenBSD)

BSD bootloaders for UEFI firmware

- Again this bootloaders are really simple to install: jus copy the appropiate file to the ESP, and have the firmware or another bootloader execute it
- Thsi loader can be found
- **FreeBSD:** files `boot1.efi` and `loader.efi` in `/boot`. (`boot1.efi` does not load the kernel directly from the BSD partition but instead anothe loader)
- **NetBSD:** file `/usr/mdec/bootx64.efi`
- **OpenBSD:** file `/usr/mdec/BOOTX64.EFI`

other boot loaders
→using removable media

Using removable media

- Boot loaders can be installed into removable media, even to boot an O.S. which is installed in a non-removable media
- To boot in machines with BIOS firmware, boot code must exist in the first block of the device (when creating CDs or DVDs, we should provide an image of a boot device to the CD/DVD mastering software)
- To boot in machines with UEFI firmware, the removable media must be FAT formatted and contain a directory named EFI in its root directory. The firmware will boot the file **EFI****BOOT****bootx64.efi** from the removable media