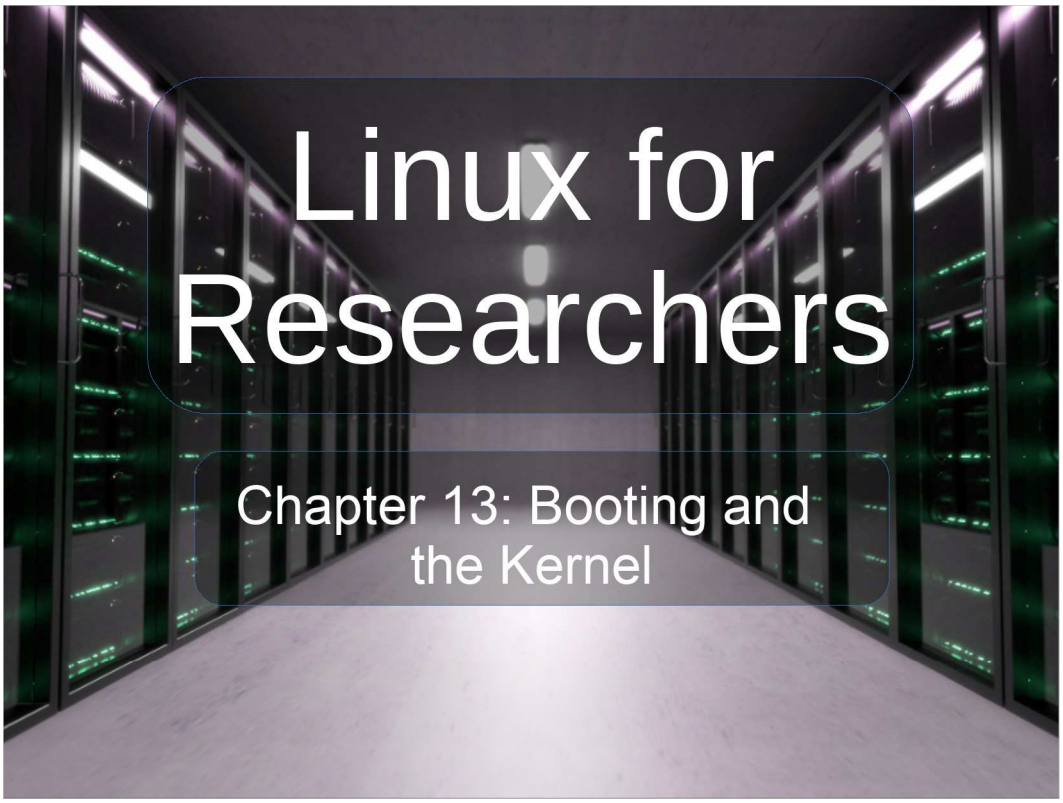


Today we'll finally take a closer look at the boot process and the linux kernel. The boot process is something every Linux administrator needs to be familiar with, because you'll interact with it often and you'll need to modify it from time to time.

You'll also need to be aware, at least, of kernel modules, and how to use them. This knowledge will come in handy when dealing with new hardware, or trying to understand errors.

These days, the process for building the Linux kernel is something that can safely remain a mystery to many system administrators. Kernel updates are usually made available by the vendor (Red Hat, Ubuntu, etc.), and can easily be installed through the regular package management system. Gone are the days when Linux administrators regularly built their own customized kernels.

Still, there will some day come a time when you need to do it. Maybe you'll need a bleeding-edge kernel to make some newfangled device work, or maybe you'll need to turn off some kernel feature that causes problems.

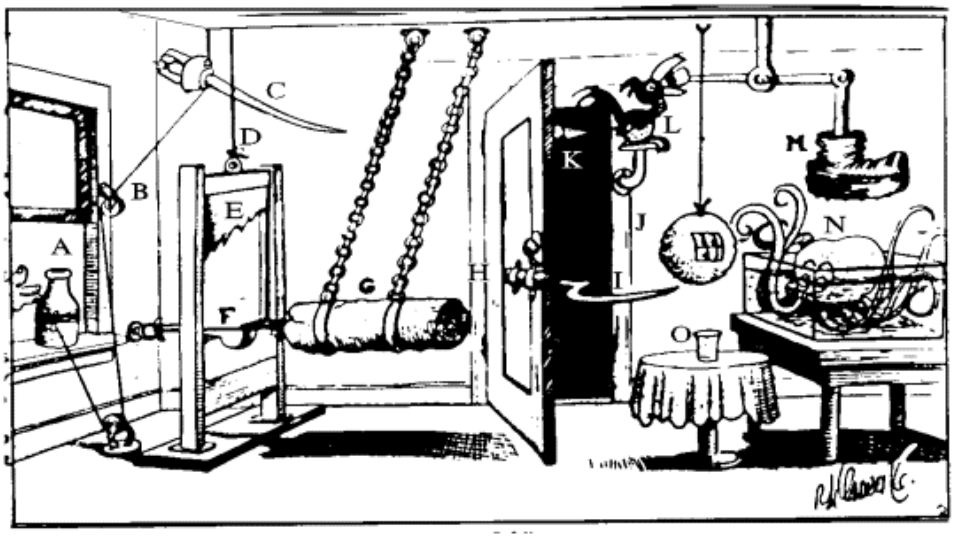
Part 1: The Boot Process

Once upon a time, computers like this PDP-11 were booted by flipping a set of switches, to enter a memory address, and then telling the computer to start executing commands at that address. (http://www.bitsavers.org/pdf/dec/pdp11/dos-batch/DosBatchHandbook_v9_Apr74/G.pdf) The address pointed the computer to a particular operating system on a particular device.

Although we only press one button now, things haven't really changed that much internally. When a modern PC starts, the CPU begins loading instructions from a predefined, standard address in memory (FFFF0, in hexadecimal). This memory location points to the beginning of the BIOS, which then chooses (or lets us choose) a boot device.

“Bootstrapping”:

Bootstrapping (shortened long ago to “booting”) is the process by which a computer is moved from a completely quiescent, “dead”, state to a fully initialized and running state. Usually it begins with a button press that leads to a cascade of successively more complex events that eventually results in a successfully running operating system.



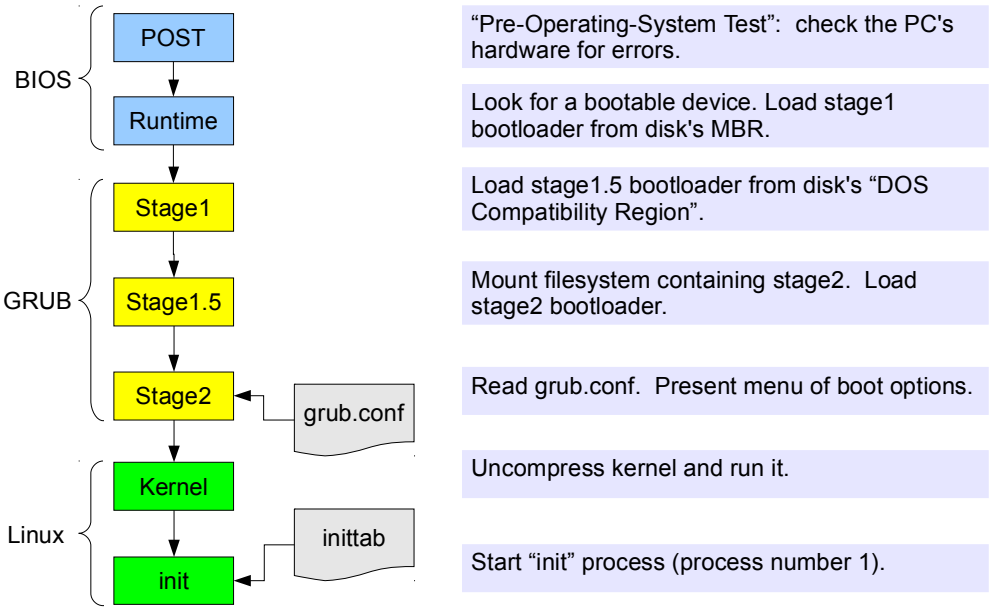
Bootstrapping is hard. Here's an analogy: Imagine that you have all of the parts necessary to build a robot. The parts are scattered all over the floor. You'd like the robot to assemble itself. How would you start?

Maybe you could imagine that there's a button you can press on one of the hands to start the process. The hand then starts crawling around, collecting other parts, until it can assemble the legs and hips. Then these parts can walk around and assemble more of the robot. Eventually the robot is complete.

This is the kind of thing the computer has to do in order to boot itself. It starts with little steps, then progresses through several stages until it's in the final, running state.

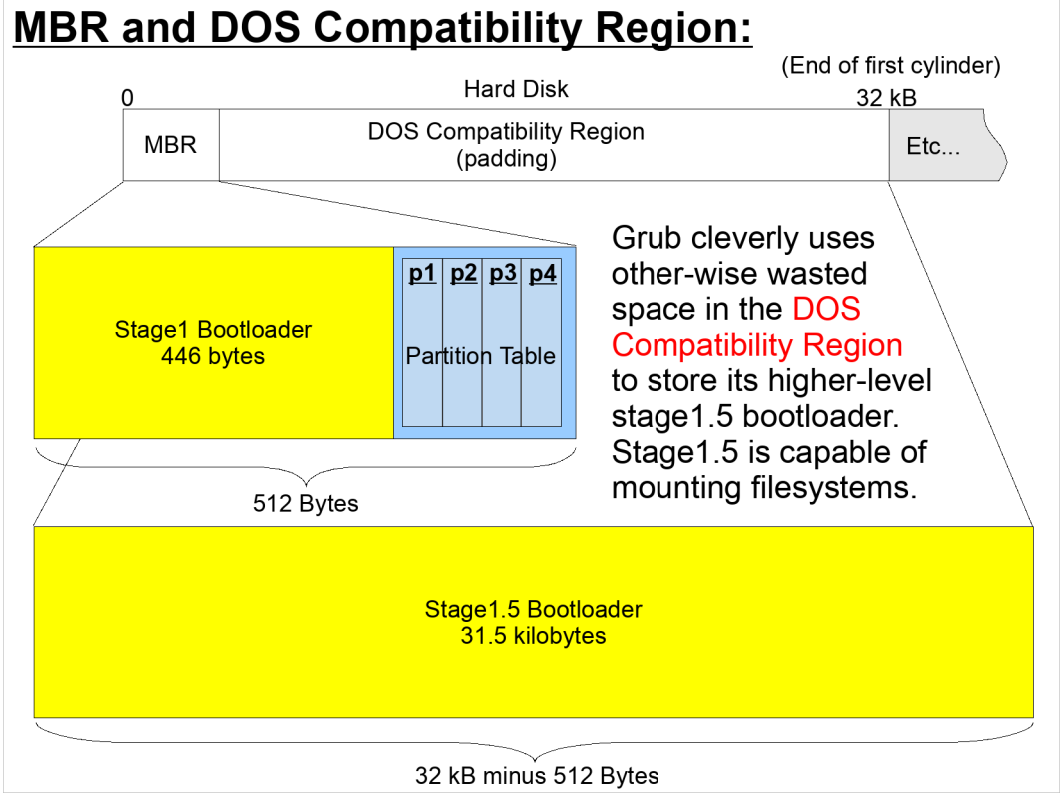
Booting Linux (BIOS/MBR version):

The “Basic I/O System” (BIOS) is a tiny operating system installed on a traditional PC's motherboard. It starts the boot process.



Grub (the “GRand Unified Bootloader”) is the successor to the original Linux boot system, called LILO. With LILO, any time you made a change to the boot configuration, the bootloader needed to be re-installed in the disk's Master Boot Record (MBR). Grub understands filesystems, though, so you an reconfigure it by simply editing a text file that Grub's “stage2” will read when you reboot.

Typically, the stage1 bootloader lives in the MBR's boot code section. As we'll see, the stage1.5 bootloader cleverly makes use of the disk's “DOS Compatibility Region”. The stage2 bootloader is just a file living in a filesystem.



Some operating systems, like DOS, require that partitions start at cylinder boundaries in the C,H,S coordinate system. To accomodate this, most partition managers leave some padding between the MBR and the beginning of the first partition. This padding is called the “DOS Compatibility Region”. On most disks, its size will be 32kB minus the 512 bytes needed for the MBR.

Since only 446 bytes are available in the MBR for boot code, the “stage1” bootloader that lives there needs to be very simple. But the DOS Compatibility Region provides ample space for a more sophisticated bootloader. The stage1 bootloader does some initialization, then invokes the stage1.5 bootloader to continue the boot process.

The stage1.5 bootloader is capable of mounting filesystems, making it easy to reconfigure the boot process by editing normal files with a text editor.

See: http://www.pixelbeat.org/docs/disk/ for more.

Extracting and Saving the MBR:

We can use "dd" ("data dump") to extract the contents of the Master Boot Record and save them in a file, for examination or for later restoring a corrupt MBR:

```
[root@demo ~]# dd if=/dev/sda of=mbr.dat bs=512 count=1
```

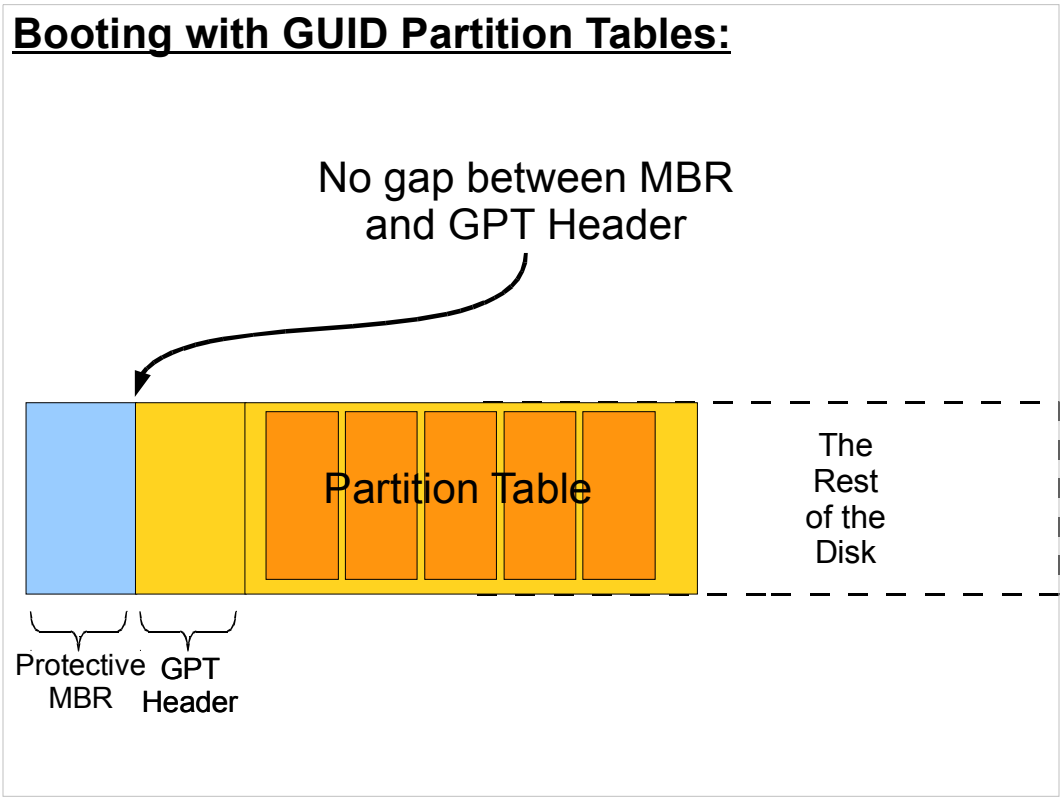
Following the Unix "everything is a file" philosophy, the device special file /dev/sda represents the disk, and lets us treat it like just another file from which we can read data.

The "file" command can be used to tell you about a file. In this case, file identifies mbr.dat as a boot sector and tells us about the partition table inside it:

```
[root@demo ~]#  file mbr.dat
mbr.dat: x86 boot sector;
partition 1: ID=0x83, active,
  starthead 1, startsector 63, 208782 sectors;
partition 2: ID=0x8e,
  starthead 0, startsector 208845, 312287535 sectors,
  code offset 0x48
```

Note the odd syntax for the "dd" command. This command has been around for a long time, and predates the conventions now used.

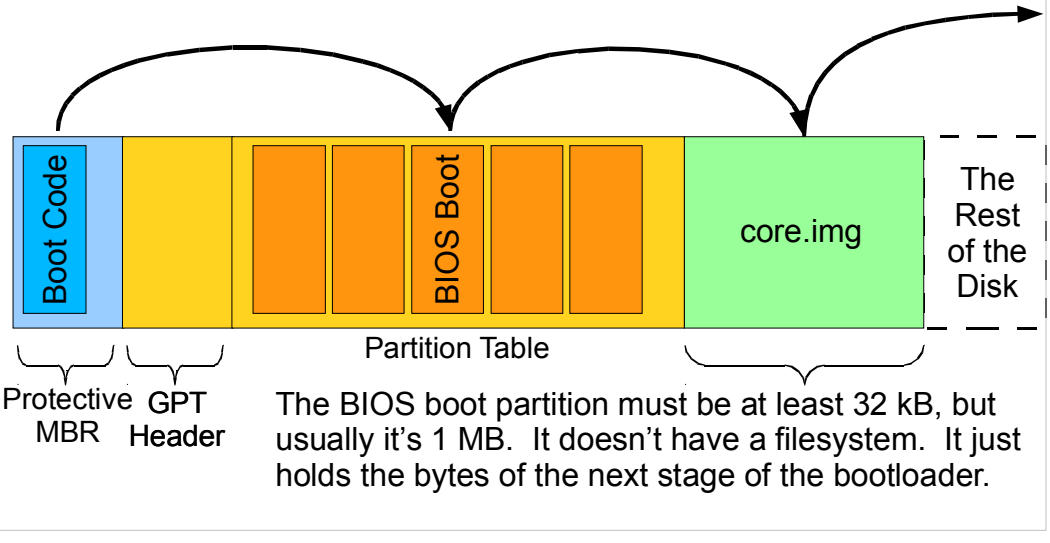
After you've save the MBR in a file, you could later restore it to the disk if you wanted to. To do this, you'd just revers the "if" and "of" arguments in the first command, above. Be careful not to reverse these unintentionally!



With GPT, there's still an MBR at the front of the disk (the MBR's partition table just says there's one big partition on the disk, of type "GPT") , but there's no "Compatibility gap" after this MBR. The GPT header is written right after the MBR. Where can we put the stage 1.5 bootloader now?

Booting Linux (BIOS/GPT version):

To use BIOS (aka “legacy”) booting with GPT, we need to have one “BIOS boot partition”. This is a partition of type EF02 or, equivalently, GUID 21686148-6449-6e6f-744e656564454649. Translated into ASCII, this GUID spells "Hah!IdontNeedEFI".

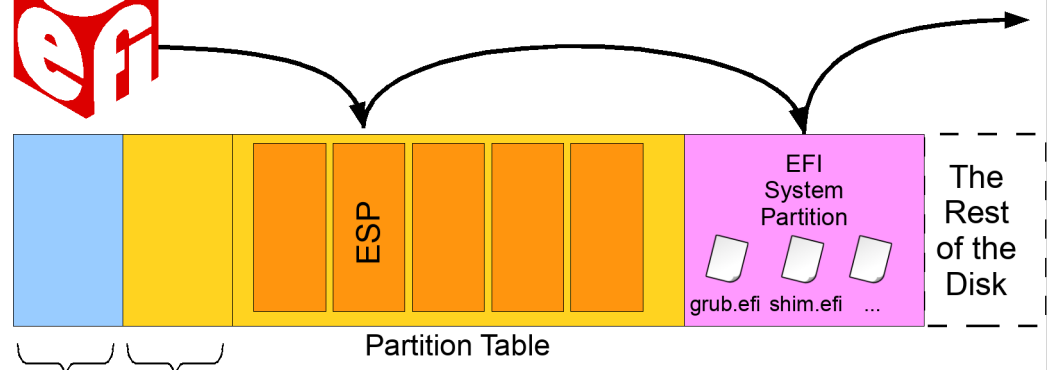


The BIOS boot partition must be at least 32 kB, but usually it’s 1 MB. It doesn’t have a filesystem. It just holds the bytes of the next stage of the bootloader.

Instead of putting the "stage1.5" bootloader into the DOS compatibility region, we now need to make a separate small partition for it. Instead of "stage1.5", grub now calls this "core.img".

Booting Linux (UEFI/GPT version):

To use UEFI with GPT, we need to have one "EFI System Partition" (ESP). This is a partition of type EF00. The ESP contains a VFAT filesystem that holds (among other things) an EFI application named "grub.efi", which is capable of mounting the /boot filesystem and continuing the boot process.



Partition Table

Protective MBR

GPT Header

If "Secure Boot" is enabled, UEFI first invokes "shim.efi" from the ESP, which then starts grub.efi. shim.efi is digitally signed in a way that satisfies the motherboard's "Trusted Platform Module" (TPM).

Finally, when we use UEFI instead of a BIOS, the UEFI processor is smart enough to read the partition table and mount a special filesystem. One or more of the EFI applications (e.g. grub.efi) in this filesystem is then run to continue the boot process.

See also: https://www.gwern.net/Turing-complete#how-many-computers-are-in-your-computer

The UEFI processor, the TPM, and many other devices in your computer are actually turing-complete processors, capable of doing arbitrary calculations. The article linked to above finds that typical computers contain from 15 to several hundred processors or various kinds, including those found in network cards, disks, graphics cards, and even more unlikely places.

The article also notes that Bad Guys have sometimes exploited these processors to do Bad Things.

Managing UEFI Boot Options:

List boot options:

```
[root@localhost ~]# efibootmgr
BootCurrent: 0004
BootNext: 0003
BootOrder: 0004,0000,0001,0002,0003
Timeout: 30 seconds
Boot0000* Diskette Drive(device:0)
Boot0001* CD-ROM Drive(device:FF)
Boot0002* Hard Drive(Device:80)/HD(Part1,Sig00112233)
Boot0003* PXE Boot: MAC(00D0B7C15D91)
Boot0004* Linux
```

Add a boot option:

```
efibootmgr -c -L "Testing" -l '\EFI\centos\shim.efi'
```

`-L` = Label that appears on boot menu
`-l` = Boot loader

Most computers will allow you to configure UEFI through the firmware settings at boot time. You can also use the "efibootmgr" command for this, though.

Part 2: GRUB

Grub (the “GRand Unified Bootloader”) was originally written by Erich Boleyn, who says:

“I'm not a big fan of Microsoft products, but the Microsoft spell-checker says my name must be "Reich Boolean"... the thousand year rule of computers. Scary, huh?”

He developed Grub as a more flexible alternative to LILO. His intent was to make a general-purpose bootloader that could boot any operating system.

Grub's Configuration Files:

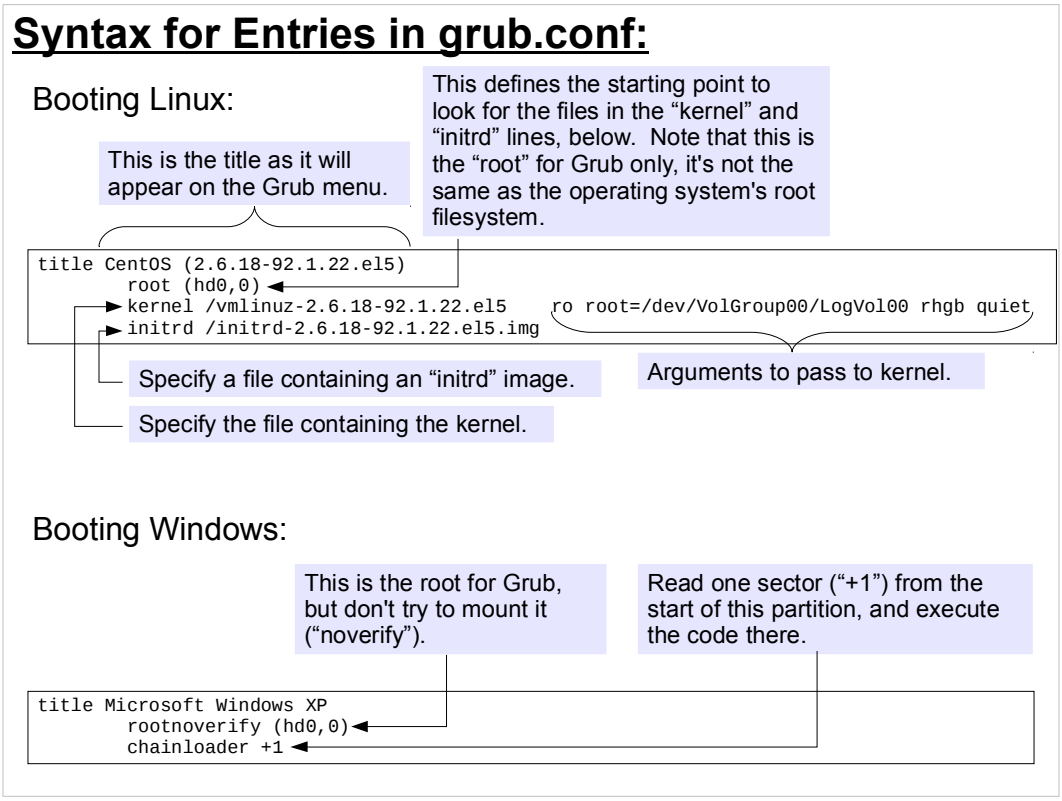
Grub's files usually reside in the directory /boot/grub or /boot/grub2. The most important file is "grub.cfg". You'll notice the following warning at the top of this file:

```
#
# DO NOT EDIT THIS FILE
#
# It is automatically generated by grub2-mkconfig using templates
# from /etc/grub.d and settings from /etc/default/grub
#
```

Earlier versions of grub encouraged you to modify grub's configuration by directly editing grub.cfg. Since version 2, however, grub has provided tools tools that read other configuration files (and examine your computer) and use this information to create grub.cfg.

Still, there's no harm in looking...

In the Olden Days we edited grub.cfg by hand, but nowadays its better to let grub's tools generate this file for you. We'll see how to do that a little later.



The Windows bootloader is embedded in the "partition boot sector" at the beginning of the partition on which Windows is installed. The second example above shows how Grub can just hand off the boot process to the Windows bootloader.

In the "chainloader" command, we could either give a filename (optionally prefixed with a Grub device name) or we could (as we do in the example above) specify a range of raw data on the disk, not inside any filesystem. The latter is specified through "blocklist" notation. The general form of this is offset+length. If the offset is omitted, it's assumed to be zero. We could also specify a device, like "(hd0,0)+1". If the device is omitted, Grub's "root" device is assumed.

Extracting the Windows "Partition Boot Sector":

Just as we extracted the MBR from the first sector of the disk, we can also extract the bootloader that Windows installs at the beginning of its partition. Note that we specify a partition (/dev/sda1) rather than the whole disk (/dev/sda):

```
[root@demo ~]# dd if=/dev/sda1 of=pbr.dat bs=512 count=1
```

Windows Partition

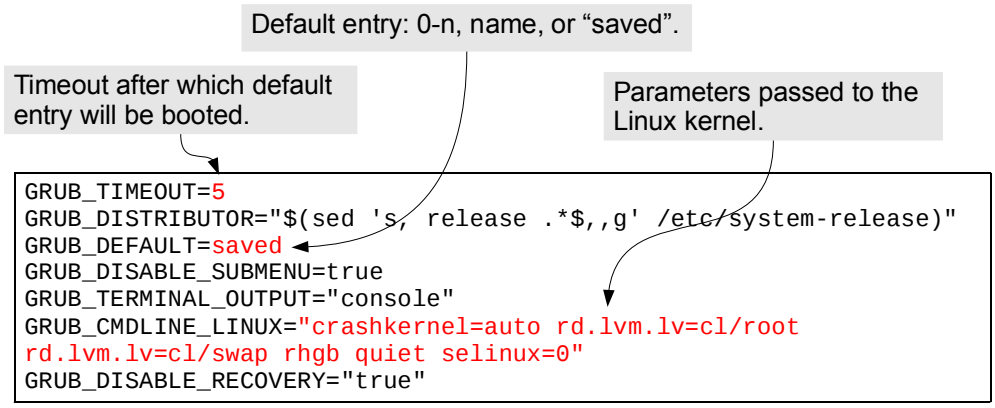
"file" understands the resulting file's format, and can tell us a few things about it:

```
[root@demo ~]# file pbr.dat
pbr.dat: x86 boot sector,
Microsoft Windows XP Bootloader NTLDR,
code offset 0x58, OEM-ID "MSDOS5.0",
sectors/cluster 32, reserved sectors 38,
Media descriptor 0xf8,
heads 255, hidden sectors 63,
sectors 40965687 (volumes > 32 MB) ,
FAT (32 bit), sectors/FAT 9997,
reserved3 0x800000,
serial number 0xa4ea03af, unlabeled
```

As with the master boot record, we could restore the saved "PBR" by swapping "if" and "of" in the dd command. Again, be careful not to do this by accident!

The /etc/default/grub File:

/etc/default/grub is a text file that contains some configuration information used when generating the grub.cfg file. It might look like this:



See:
https://www.gnu.org/software/grub/manual/grub/html_node/Simple-configuration.html
for a list of configuration options.

The /etc/default/grub file lets us choose configuration options for grub. When grub's tools generate grub.cfg for us, they read this file to get our advice about how to do it.

Making grub.cfg:

On Debian/Ubuntu/etc.:

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

On RedHat/CentOS/Fedora/etc.:

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

"grub-mkconfig" reads /etc/defaults/grub, then uses helper scripts located in the directory /etc/grub.d to probe the computer, looking for installed operating systems, etc. It then writes an appropriate grub.cfg file.

grub-mkconfig is the tool you should use to create or update a grub.cfg file. Note that the location of the file needs to be specified on the command line, and that the location will be different depending on which Linux distribution you're using.

Installing the grub Bootloader:

On Debian/Ubuntu/etc.:

```
grub-install –recheck /dev/sda
```

On RedHat/CentOS/Fedora/etc.:

```
grub2-install –recheck /dev/sda
```

“grub-install” reads grub.cfg and writes the appropriate boot code into MBR, BIOS Boot Partition, and/or EFI System Partition, as necessary.

You don't need to do this every time you change grub's configuration. It only needs to be done if grub's boot code has never been installed on this disk.

Part 3: The Kernel

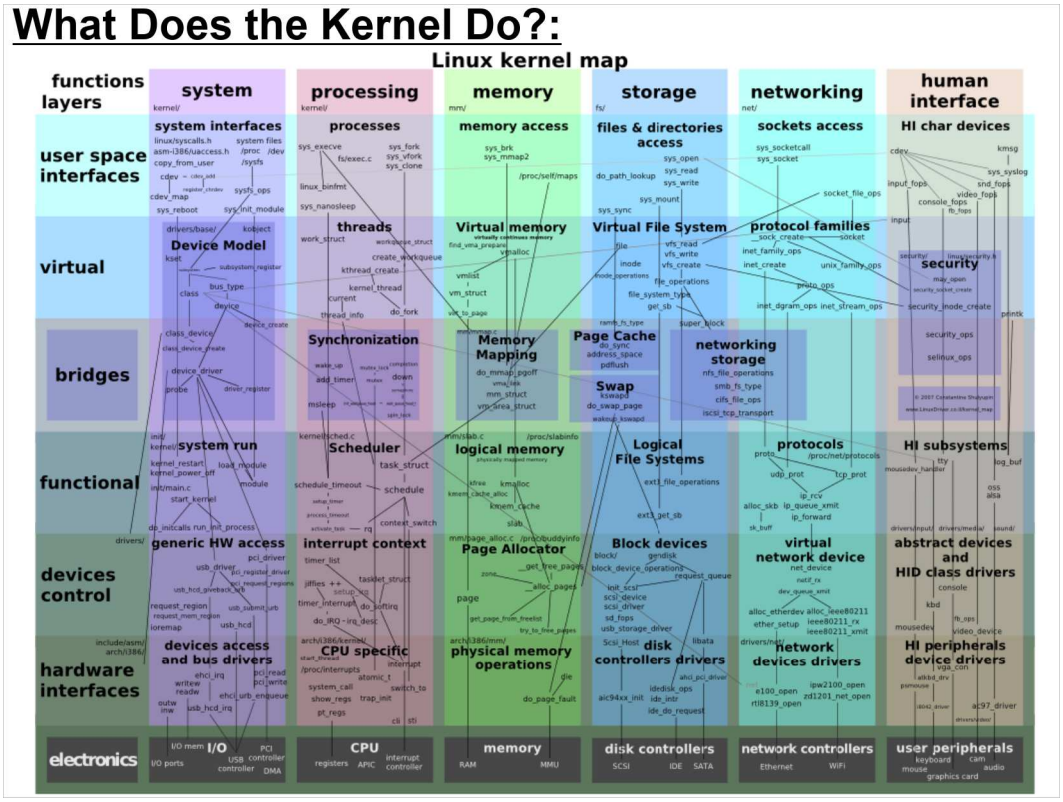
"They know that we know about their leader. We've overheard their whisperings. This "Colonel" guy will be tracked down, and he WILL be brought to justice..."

“...the captured nerds are model prisoners. 'They made a crude but listenable crystal radio out of a light bulb, a crayon, and a square of toilet paper, and a rock. They say they'll have linux on it by next week. ' “

-- From media coverage of the well-known Seattle Linux Riot of April 1, 1999:

http://web.archive.org/web/19990508115937/http://w3.one.net/~sunlion/linuxriot.html

What Does the Kernel Do?:



A beautiful chart showing how some components of the Linux kernel interact with each other. This shows how complex the kernel is, and how many things it does. Just look at the bottom, “electronics”, row. The kernel is the only thing that talks directly to these devices. All communication to and from them goes through the kernel.

Kernel Version History:

- **Version "0" -- April 1991.**
  First public release of Linux.

- **Version 1.0.0 -- March 1994.**
  Supported single-processor computers based on the Intel i386 processor.

- **Version 1.2.0 -- March 1995.**
  Added support for DEC Alpha, Sparc, and MIPS processors, etc.

- **Version 2.0.0 -- June 1996.**
  Added multiprocessor support (SMP) and more processor types, etc.

- **Version 2.2.0 -- January 1999.**
  More filesystem types (including NTFS) and more processor types, etc.

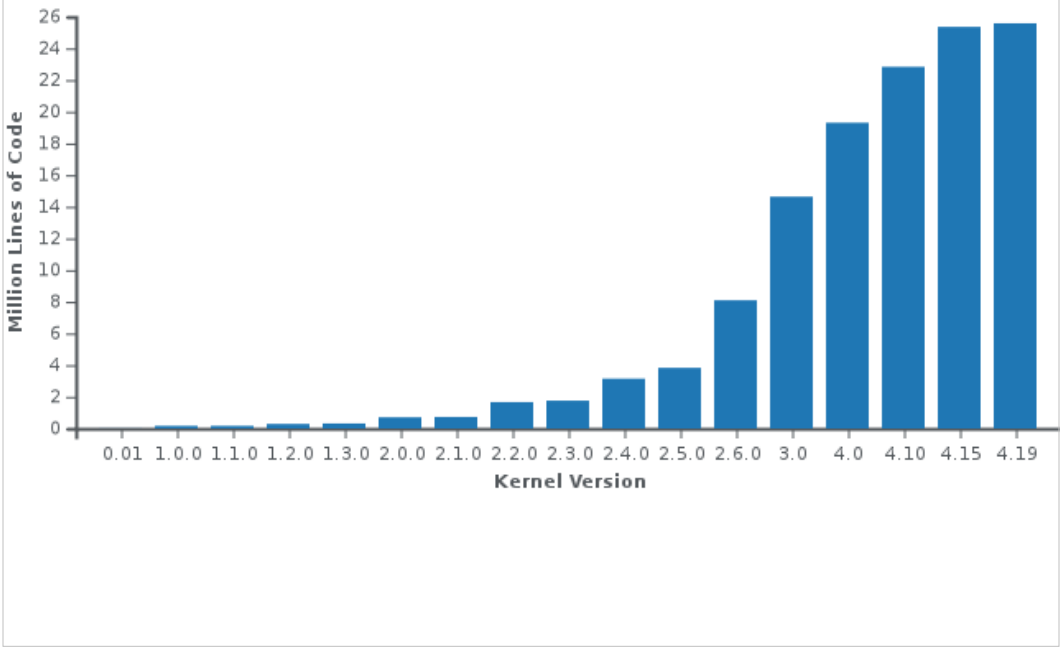
- **Version 2.4.0 -- January 2001.**
  Plug-and-play, USB, PCMCIA, bluetooth, LVM, RAID, etc.

- **Version 2.6.0 -- December 2003.**
  Up to 2^{32} users (was 2^{16}), up to 2^{29} PIDs (was 2^{16}), SELINUX, Infiniband, etc.

- **Version 3.0 – July 2011.**
  No major changes from last 2.6.* release. Just a numbering change.
- ....etc.

The current kernel version is 5. Since around version 3, Linux has emphasized that the version numbers have no meaning. There's no big difference between kernel 3.99 and 4.0. It's just a number to identify the version.

Kernel Source Code:



The Linux kernel is an example of a massive distributed development project. Thousands of people now contribute to the kernel code. This pushes the bounds of project management, and has led to the development of new tools, like git, to make it possible.

Kernel Files in /boot:

The kernel itself usually lives in the /boot directory. In the example below, there are two different kernels, “vmlinuz-*”:

```
[root@demo ~]# ls -l /boot
-rw-r--r-- 1 root root   65411 Nov 12 09:54 config-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root   65411 Dec 16 12:28 config-2.6.18-92.1.22.el5
drwxr-xr-x 2 root root    1024 Dec 17 01:38 grub
-rw------- 1 root root 3133534 Nov 13 03:18 initrd-2.6.18-92.1.18.el5.img
-rw------- 1 root root 3133544 Dec 17 01:36 initrd-2.6.18-92.1.22.el5.img
-rw-r--r-- 1 root root   91738 Nov 12 09:54 symvers-2.6.18-92.1.18.el5.gz
-rw-r--r-- 1 root root   91760 Dec 16 12:28 symvers-2.6.18-92.1.22.el5.gz
-rw-r--r-- 1 root root  912912 Nov 12 09:54 System.map-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root  913350 Dec 16 12:28 System.map-2.6.18-92.1.22.el5
-rw-r--r-- 1 root root 1806900 Nov 12 09:54 vmlinuz-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root 1805940 Dec 16 12:28 vmlinuz-2.6.18-92.1.22.el5
```

There are other files in /boot, too:

config-* These are copies of the kernel configuration file used when compiling each of the kernels. They're kept here for your reference.

System.map-* These files list the memory location of all of the symbols in the given kernel. This is useful for debugging.

symvers-* Each symbol in a Linux kernel module has a version number identifying a particular version of the kernel. This version information is stored in the symvers files, and is used when compiling 3rd-party kernel modules.

the config-*, System.map-* and symvers-* files are optional. They aren't necessary if you only need to run the kernel, and Grub (for example) doesn't use them.

Notice that there are also “initrd=*” files here. We'll talk about those in a few minutes.

What Kernels Do I Have?:

```
dpkg -l | grep linux-image
rc  linux-image-4.15.0-45-generic 4.15.0-
45.48 amd64 Signed kernel image generic
ii  linux-image-4.15.0-46-generic 4.15.0-
46.49 amd64 Signed kernel image generic
ii  linux-image-4.15.0-47-generic 4.15.0-
47.50 amd64 Signed kernel image generic
```

```
rpm -q kernel
kernel-3.10.0-957.1.3.el7.x86_64
kernel-3.10.0-957.5.1.el7.x86_64
kernel-3.10.0-957.10.1.el7.x86_64
```

You can limit the number of installed kernels by adding this parameter to your /etc/yum.conf file. It won't remove kernels that are in use!

yum.conf

```
[main]
...
installonly_limit=3
...
```

Installed kernels show up automatically in the boot menus generated by grub-mkconfig. The "installonly_limit" option available in the RedHat/CentOS/Fedora world helps keep these menus uncluttered.

What Kernel Am I Running?:

The "uname" command will tell you about the kernel you're currently running, among other things:

```
uname -a
Linux mypc.example.com 3.10.0-957.1.3.el7.x86_64
#1 SMP Thu Nov 29 14:49:43 UTC 2018 x86_64 x86_64
x86_64 GNU/Linux
```

"cat /proc/cmdline" will tell you this too, as well as giving you the parameters that were passed to the kernel at boot time:

```
cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-3.10.0-957.1.3.el7.x86_64
root=/dev/mapper/cl-root ro crashkernel=auto
rd.lvm.lv=cl/root rd.lvm.lv=cl/swap rhgb quiet
selinux=0 net.ifnames=0 biosdevname=0
LANG=en_US.UTF-8
```

Installing Bleeding-Edge Kernels:

www.elrepo.org

rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org

yum install
https://www.elrepo.org/elrepo-release-7.0-3.el7.elrepo.noarch.rpm

yum install kernel-lt ← Install the current "long-term support" kernel.

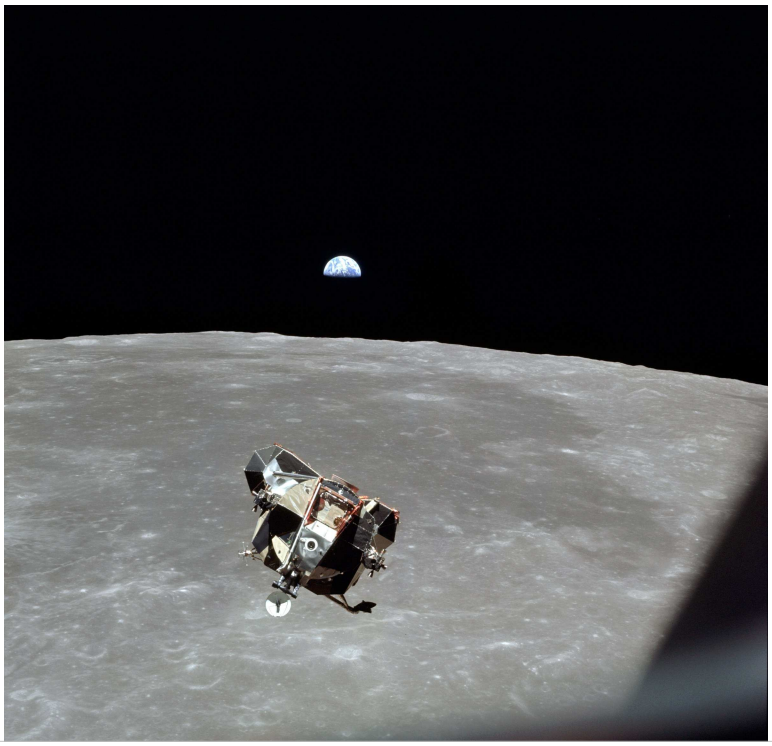
yum install kernel-ml ← Install the current "mainline" kernel.

Linux distributions usually ship well-tested, stable kernels, with the intent of supporting them for several years. Although the distribution will provide security updates for these kernels, the kernels may lack features of newer kernels released later.

If you find that the distribution's current kernel doesn't support a device in your computer, or provide a feature you need, you might want to try a kernel from the elrepo repository.

Kernels installed this way will receive updates from the elrepo repository whenever you run "yum update".

Part 4: Kernel Modules



Early on, Linux kernels were “monolithic”. Every feature of the kernel was compiled into a single file. The problem with this was that, whenever you needed to add a driver for a new device or support for a new type of filesystem, you needed to rebuild the kernel, re-install it and reboot the computer.

To make things easier, many components of the kernel can now be separated out into “kernel modules”. Modules can be loaded or unloaded on-the-fly, as needed. (Sometimes literally: http://www.conceptlab.com/fly/)

Kernel Modules:

Many parts of the Linux kernel can be built as "modules". These kernel modules can be dynamically loaded or unloaded, without rebooting the computer. Most of them are drivers for particular devices, filesystems, etc. The modules for a particular kernel usually live under /lib/modules, in a directory specific to that kernel version. For example:

```
[root@demo ~]# ls /lib/modules/2.6.18-92.1.22.el5/drivers/net
3c59x.ko     dummy.ko      natsemi.ko      ppp_synctty.ko  sunhme.ko
8139cp.ko    e1000         ne2k-pci.ko     qla3xxx.ko      tg3.ko
8139too.ko   e1000e        netconsole.ko   r8169.ko        tlan.ko
8390.ko      e100.ko       netxen          s2io.ko         tokenring
acenic.ko    epic100.ko    ns83820.ko      sis190.ko       tulip
amd8111e.ko  fealnx.ko     pcmcia          sis900.ko       tun.ko
b44.ko       forcedeth.ko  pcnet32.ko      skge.ko         typhoon.ko
bnx2.ko      ifb.ko        phy             sky2.ko         via-rhine.ko
bnx2x.ko     igb           ppp_async.ko    slhc.ko         via-vel.ko
bonding      ixgb          ppp_deflate.ko  slip.ko         wireless
cassini.ko   ixgbe         ppp_generic.ko  starfire.ko
chelsio      mii.ko        ppp_mppe.ko     sundance.ko
cxgb3        mlx4          pppoe.ko        sungem.ko
dl2k.ko      myri10ge      pppox.ko        sungem_phy.ko
```

The files with names ending in ".ko" are kernel modules. Other files are directories containing more modules.

Listing Loaded Modules:

The “lsmod” command will show you a list of the currently loaded kernel modules:

```
[root@demo ~]# lsmod
Module                  Size  Used by
vfat                   15809  0
fat                    51165  1 vfat
usb_storage            76705  0
netloop                10945  0
ipt_MASQUERADE          7617  1
iptable_nat            11205  1
ip_nat                 21101  2 ipt_MASQUERADE,iptable_nat
ext3                  123593  2
tg3                   109380  0
bluetooth              53797  5 hidp,rfcomm,l2cap
sunrpc                144893  1
ipt_REJECT              9537  3
nfnetlink              10713  2 ip_nat,ip_conntrack
iptable_filter          7105  1
ip_tables              17029  2 iptable_nat,iptable_filter
video                  21193  0
sbs                    18533  0
backlight              10049  1 video
etc...
```

Each kernel module provides a set of “symbols”, representing functions or variables. These functions and variables can be used by other modules. Thus, some modules will require that other modules be installed in order to function properly.

Module Information and Loading Modules:

The “modinfo” command will tell you things about a particular module:

```
[root@demo ~]# modinfo ext3
filename:    /lib/modules/2.6.18-92.1.22.el5xen/kernel/fs/ext3/ext3.ko
license:     GPL
description: Second Extended Filesystem with journaling extensions
author:      Remy Card, Stephen Tweedie, Andrew Morton,
             Andreas Dilger, Theodore Ts'o and others
srcversion:  D01BE9DB9B4D2A251EC9ACA
depends:     jbd
vermagic:    2.6.18-92.1.22.el5 SMP mod_unload 686 REGPARM
             4KSTACKS gcc-4.1
module_sig:
883f3504947f4fbc6715965c252fdcb112aba90a08be4317172196118b79dff75c55eb
d8013d7c109e20208068e86646726d4b95ea4bb7a53b9ffa8e9
```

You can load a module with the “modprobe” command:

```
[root@demo ~]# modprobe joydev
```

and you can use “modprobe -r” to remove (unload) it:

```
[root@demo ~]# modprobe -r joydev
```

modprobe looks at the modules.dep file to determine whether other modules need to be loaded at the same time, and loads them automatically.

Identifying Modules used by a Device:

Most of the interesting devices in your computer are connected through the PCI bus. You can find information about devices on the PCI bus by using the “lspci” command. Among the things it can tell you are the names of any kernel modules used by each PCI device. To find this information, use the “-k” qualifier:

```
[root@demo ~]# lspci -k
...
04:00.0 VGA compatible controller: nVidia Corporation Device 0a76 (rev
a2)
    Subsystem: ASUSTeK Computer Inc. Device 8446
    Kernel driver in use: nouveau
    Kernel modules: nouveau, nvidiafb
```

Type “man lspci” to see more of lspci's many capabilities.

Module Parameters:

Some modules allow you to control their behavior by setting parameters when the module is loaded.

```
[root@demo ~]# modinfo e1000
...
description: Intel(R) PRO/1000 Network Driver
author:      Intel Corporation, <linux.nics@intel.com>
...
depends:
vermagic:       2.6.18-92.1.22.el5 SMP mod_unload
                686 REGPARM 4KSTACKS gcc-4.1
parm: Speed:Speed setting (array of int)
parm: Duplex:Duplex setting (array of int)
parm: AutoNeg:Advertised auto-negotiation setting (array of int)
parm: FlowControl:Flow Control setting (array of int)
parm: InterruptThrottleRate:Interrupt Throttling Rate (array of int)
parm: SmartPowerDownEnable:Enable PHY smart power down (array of int)
parm: debug:Debug level (0=none,...,16=all) (int)
...
```

We could load the e1000 ethernet controller driver and tell it to run at 1Gbps and full duplex:

```
[root@demo ~]# modprobe e1000 Speed=1000 Duplex=2
```

Documentation is left up to the author of the module, and may sometimes be cryptic.

The modprobe ".conf" Files:

The modprobe command looks at files named *.conf in the /etc/modprobe.d directory for configuration information:

```
[root@demo ~]# cat /etc/modprobe.d/my.conf
alias eth0 tg3
alias eth1 e1000
alias scsi_hostadapter ata_piix
alias snd-card-0 snd-hda-intel
options snd-card-0 index=0
options snd-hda-intel index=0
```

The example above shows two types of configuration lines that can be present in modprobe.conf:

- **alias**
  These lines define aliases that can be used for module names in modprobe commands. These aliases are also used, indirectly, by the kernel. When the kernel encounters a new device, like "eth0", it calls "modprobe eth0" to load the appropriate module.
- **options**
  These are parameters that modprobe will pass to the given module, by default, when it is loaded.

Note that, in addition to any aliases defined in the *.conf files, modules may have other aliases compiled into them. These can be seen with the "modinfo" command. When looking up a name, modprobe first looks for an alias in the *.conf files, then it looks for a module that actually has that name, then it looks through all modules for one that has the given name as a compiled-in alias.

Part 5: The Initial RAM Disk (initrd)

Although there were problems with a monolithic kernel (needing to recompile and reboot to add drivers), this type of kernel was a little easier to boot. Now that we have modules, we may need not just the kernel image, but also a few modules in order to get the root filesystem mounted and start the init process.

For example, if our root filesystem is an ext3 filesystem, we'll need to have the ext3.ko kernel module available to the kernel before the root filesystem can be mounted. Unfortunately, this kernel module normally lives inside that filesystem, itself!

The initial RAM disk provides a mechanism for providing necessary modules early in the boot process.

The initrd File:

The kernel may need some modules early on in the boot process. How can the kernel start the “init” program, for example, if it doesn't have the modules necessary to mount the filesystem on which “init” lives?

Linux bootloaders like Grub allow you to specify an initrd file associated with a kernel. The initrd file is a compressed archive containing a few kernel modules, utilities and scripts that help the kernel out during the early part of the boot process. For example, the initrd might contain the ext3.ko module needed for mounting an ext3 filesystem.

```
[root@demo ~]# ls -l /boot
-rw-r--r-- 1 root root   65411 Nov 12 09:54 config-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root   65411 Dec 16 12:28 config-2.6.18-92.1.22.el5
drwxr-xr-x 2 root root    1024 Dec 17 01:38 grub
-rw------- 1 root root 3133534 Nov 13 03:18 initrd-2.6.18-92.1.18.el5.img
-rw------- 1 root root 3133544 Dec 17 01:36 initrd-2.6.18-92.1.22.el5.img
-rw-r--r-- 1 root root   91738 Nov 12 09:54 symvers-2.6.18-92.1.18.el5.gz
-rw-r--r-- 1 root root   91760 Dec 16 12:28 symvers-2.6.18-92.1.22.el5.gz
-rw-r--r-- 1 root root  912912 Nov 12 09:54 System.map-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root  913350 Dec 16 12:28 System.map-2.6.18-92.1.22.el5
-rw-r--r-- 1 root root 1806900 Nov 12 09:54 vmlinuz-2.6.18-92.1.18.el5
-rw-r--r-- 1 root root 1805940 Dec 16 12:28 vmlinuz-2.6.18-92.1.22.el5
```

```
title CentOS (2.6.18-92.1.22.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-92.1.22.el5    ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        initrd /initrd-2.6.18-92.1.22.el5.img
```

Creating an initrd Image with “dracut”:

The best way to make an initrd image for the current kernel on RHEL-based systems is by using the “dracut” utility:

```
[root@demo ~]# dracut initrd-new.img
```

initrd File

To make an initrd for a different kernel:

```
[root@demo ~]# dracut initrd-new.img 2.6.18-92.1.22
```

initrd File Kernel Version

Normally, dracut determines which modules to include by looking at the computer on which it's running. If you want to force mkinitrd to include some modules, you can use the “--add-drivers” switch:

```
dracut --add-drivers=mpt2sas initrd-new.img
```

On Debian-based systems, you can use a similar tool called “mkinitramfs”. Consult the man page for details.

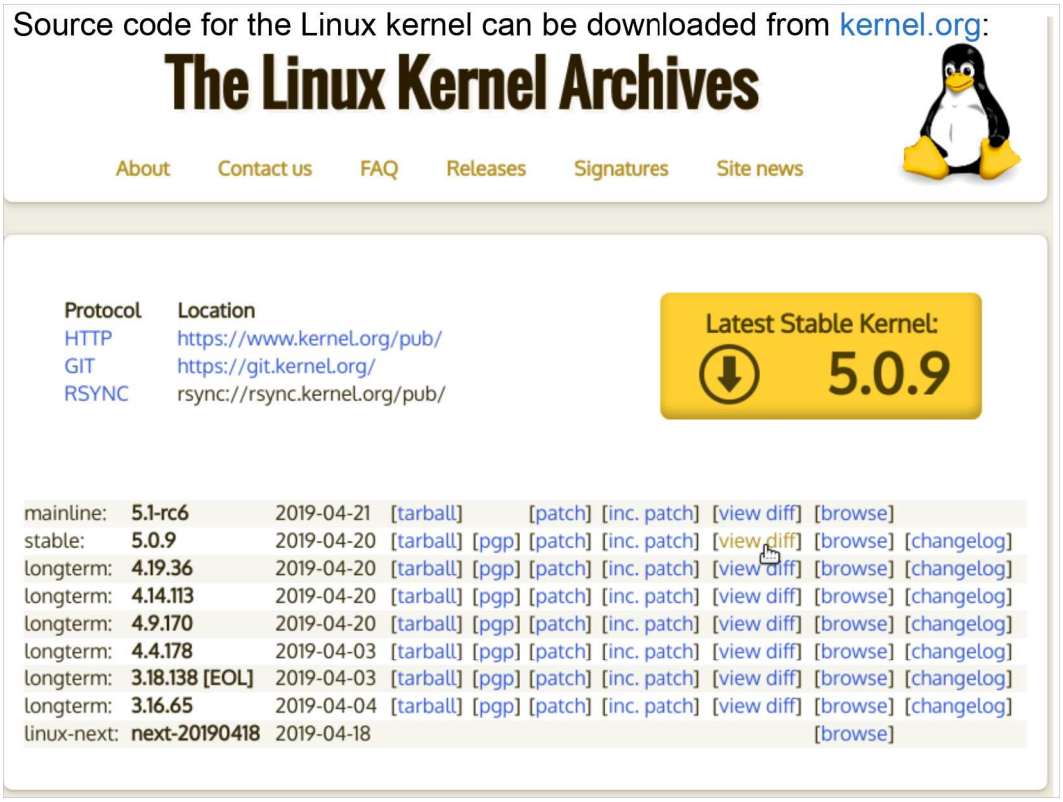
A given initrd will only work with one version of the kernel, because kernel modules are generally tied to a particular kernel version. dracut needs to know the intended kernel version so it can pack up appropriate modules.

Part 6: Building the Kernel

In the 1990s, you absolutely needed to know how to configure and build the Linux kernel if you were managing Linux computers. You'd need to add a new driver, or turn off a troublesome option, or you'd need to apply a patch to fix a bug.

These days vendors provide tons of pre-built kernel modules. Kernels are also tested much more thoroughly than in earlier years, so it's much less likely that you'll need to change any of the compiled-in options in the kernel provided by your vendor. When a bug fix needs to be made, the vendor will usually promptly provide an updated package that can easily be installed to fix it.

So, it's possible you'll never need the information that follows. But let's look at it anyway, since I think it'll give you a little more insight into how things work.



The “master repository” for the source code for the Linux kernel is kernel.org. You can always find the latest version there. If you want to download the complete source code for the current version, click on “tarball" beside that version.

Much of the work on the kernel is done on the Linux Kernel Mailing List (LKML). You can watch patches flying back and forth here:
http://patchwork.kernel.org/project/LKML/list/

Unpacking the Kernel Source:

The kernel source you download will probably be in the form of a bzip2-compressed tar file. The second command below will unpack it:

```
[root@demo ~]# cd /usr/src
[root@demo ~]# tar xJvf linux-5.0.9.tar.xz
[root@demo ~]# cd linux-5.0.9
[root@demo ~]# make mrproper
```

The “make mrproper” command will make sure the source tree is in a pristine state, ready to build a new kernel.

Useful documentation in this tree includes:

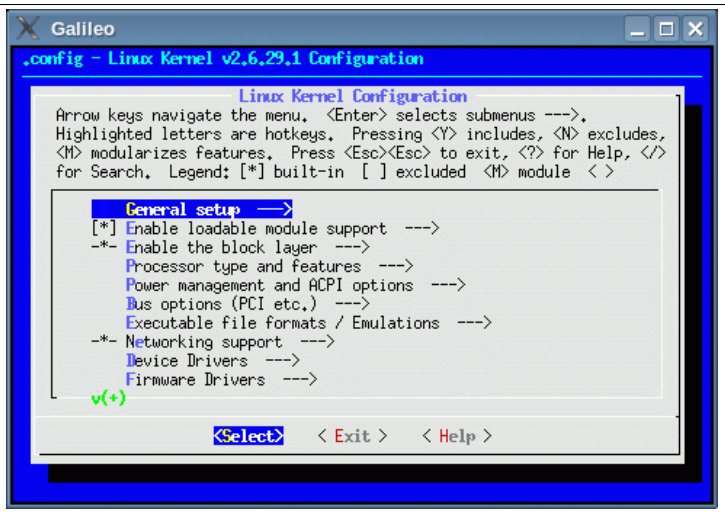
- **README**
  This file in the top directory contains general information about Linux, and some fairly detailed instructions for building the Linux kernel from source code.
- **Documentation/kernel-parameters.txt**
  This file contains much useful information about the boot-time parameters that can be passed to the Linux kernel.

Much other information is arranged by category under the “Documentation” directory.

Configuring the Kernel:

Before compiling the kernel, you'll need a kernel configuration file called ".config" in the kernel source directory. The easiest way to create one is to use the "make menuconfig" command:

```
[root@demo ~]# make menuconfig
```



This will give you a text-based menu interface for creating a .config file.

If you want to re-use or build upon an old configuration, just copy it into the kernel source directory before you invoke "make menuconfig". These settings will then become the default values in the menu interface.

As we noted before, you may find the configuration file for your current kernel in /boot, with a name like "config-2.6.18-92.1.22.el5". Just copy this into a file with the name ".config" in the kernel source directory.

Compiling the Kernel and Modules:

To compile the kernel and associated modules, type the following commands:

```
[root@demo ~]# make clean
[root@demo ~]# make bzImage
[root@demo ~]# make modules
[root@demo ~]# make modules-install
```

The “make bzImage” command will build a compressed kernel image. “make modules” builds the modules (this may take a very long time) and “make modules-install” puts the compiled modules into the appropriate directory under /lib/modules.

The resulting kernel image will be in the “arch/x86/boot” directory. You'll probably want to copy it into your /boot directory and rename it:

```
cp arch/x86/boot/bzImage /boot/vmlinuz-5.0.9
```

Then you'll need to make a matching initrd file:

```
dracut /boot/initrd-5.0.9.img 5.0.9
```

Finally, you'll need to edit your grub.conf to tell it about the new kernel.

Thanks!