Drivers and the Kernel

Chapter 11 in Unix and Linux System Administration Handbook

Haakon Andre Reme-Ness

HVL

Haakon.Andre.Reme-Ness@hvl.no

6 February 2025

The Kernel

- Serves as the central government of a Unix/Linux system
  - Enforces rules
  - Shares resources
  - Provides cores services
    - User processes rely on these!
- Hides system hardware details underneath an abstract high-level interface
  - Only device drivers are aware of the specific capabilities and communication protocols of the system's hardware

The Kernel

- The high-level interface provides five basic features:
  - Management and abstraction of hardware services
  - Processes and threads (and ways to communicate between them)
  - Management of memory (virtual memory and memory-space protection)
  - I/O facilities (filesystems, network interfaces, serial interfaces, etc.)
  - Housekeeping functions (startup, shutdown, timers, multitasking, etc.)

Kernel administration

- Most of the kernel is written in C, with a few dabs of assembly language to access CPU features not accessible through C compiler directives
- Much of the kernel behaviour can be controlled by tuning parameters, accessible from user-space
- In some cases, you may need to build a new version of kernel from the source code to add system-specific tweaks
  - Not as frequent as they used to be
- Kernel modules[1] can also be used

[1] More on these later

Kernel administration

- Be aware that kernels can be easily destabilised through minor changes
- Even if the changed kernel boots, performance may have been affected without you knowing it
- So, be conservative with kernel changes, especially on production systems
- Keep a backup of your good configurations

Kernel versions

- The Linux kernel is developed separately from distributions
- Follows its own versioning scheme, based on semantic versioning composed of three numbers:
  - A **major** version
  - A **minor** version
  - A **patch level**
- No relationship between a version number and its intended status as a stable or development kernel
  - Can be checked online instead
- Stable versions are often under long-term maintenance
  - These are often shipped with Linux distributions
- Use **uname -r** to check what kernel a given system is using

Kernel updates

- Newer kernels versions can be installed manually
- Not recommended for production systems
  - Different distributions have different goals, and select their kernels based on these
  - Newer kernels may have intentionally been dropped due to issues or concerns
  - Better to upgrade to a distribution based on a newer kernel instead

Devices and their drivers

A **device driver**...

- Is an abstraction layer that manages the system's interaction with hardware, so that the rest of the kernel doesn't need to know the specifics

Device drivers

- Translate between hardware commands understood by the device, and a stylised programming interface defined and used by the kernel
- The driver layer helps the majority of the kernel to stay device independent
- Drivers are system specific, and often specific to certain kernel revisions as well
- Linux distributions come bundled with drivers for many common hardware devices. Additional drivers must be installed afterwards

Device files and device numbers

- Device drivers are usually part of the kernel, but can be accessed both from within the kernel and user space
- This is done through the use of **device files**
  - These can be found in the /dev directory
- Most (non-network) devices have one or more corresponding files in this directory
- The kernel maps operations on these files into calls to the code of the driver

Device files and device numbers

- Device files each have a minor and major device number associated with them (recall Chapter 5)
  - The major number identifies the driver
  - The minor number identifies the device instance (unit) the driver is associated with
    - It is sometimes called unit number
- The minor number can also be used for other purposes, such as selecting or enabling certain characteristics for a device
  - E.g., two device files for a printer:
    - the minor number of the one file indicates a colour print
    - the minor number of the second file indicates a black-and-white print

Device files and device numbers

Device files can be one of two types:

- Block device files
  - Read or written as a block of bytes at a times (usually multiples of 512)
- Character device files
  - Can be read or written as a single byte at a time
- Traditionally, some devices could have both block device files and character device files
  - Not done anymore

Device files and device numbers

Note that some hardware interactions don't have direct analogues in the file-system (such as ejecting a DVD-drive)

- Handled using the **ioctl** system call instead
- Passes messages directly from user-space to the driver
- **ioctl** requests are handled by a central authority [2]

[2] Similar to the way that network protocol numbers are maintained by IANA (Internet Assigned Numbers Authority)

Challenges of device file management

- Initially, device files were handled manually
  - As there were only a few types of devices available, this was manageable
- Static device files were included in the /**dev** directory for each possible device type
- Quickly became cluttered and unmanageable as new devices became available
  - E.g.: Red Hat Linux 3 shipped with 18 000 device files in /**dev**, one for every possible device that could be attached!
- Device files are handled automatically in modern systems

Challenges of device file management

- Some challenges still exist
  - Ideally, a device added as /**dev**/**sda** should be persistent, despite intermittent disconnections and regardless of the activity of other devices and buses.
  - With transient devices such as USB flash drives, phones and other removable media, assigning a persistent identity creates challenges
  - Network interfaces have the same problem:
    - They are devices, but are not represented as devices in /**dev**
    - Solution: the Predictable Network Interface Name system is used instead to assign stable network interface names across reboots and hardware

Modern device file management

- Modern Linux/Unix systems automate the management of device files
- When a new device is detected, the corresponding device files are automatically created
- But there is more to do for newly added devices than just creating their device files, e.g.,
  - if it's a removable storage media, its filesystem should be mounted; or
  - if it's a communication device (like a hub), it should be set up with the appropriate kernel subsystem
- These procedures are handled by a user-space daemon:
  - **udevd** service in Linux
  - **devd** in BSD systems

Linux device management

Three elements are central to device management on Linux:

- **sysfs**
- The **udevadm command**
- The **udevd rules**

Sysfs

- Virtual, in-memory, filesystem
- Implemented by the kernel to provide detailed and well-organised information about
  - The system's available devices
  - Their configurations
  - Their state
- Typically mounted in /**sys**
- Used by **udevd** as a device information repository to get its raw data

Udevadm

- The **udevadm** command can:
  - Query device information
  - Trigger events
  - Control the **udevd** service
  - Monitor **udevd** and kernel events
- It is mainly used to build and test **udevd rules** [3]
- **udevadm** command reads device data from /**sys**

- **udevadm** expects one of six commands :
  - **info**
  - **trigger**
  - **settle**
  - **control**
  - **monitor**
  - **test**

[3] see more later

Udevadm

- The following three are particularly useful for administrators:
  - **info** – Print device-specific information; useful for creating rules
  - **control** – Starts and stops **udevd**, or reloads its rules files
  - **monitor** – Displays events as they occur

Udevadm: Example

```
linux$ udevadm info -a -n sdb
...
looking at device '/devices/pci0000:00/0000:00:11.0/0000:02:03.0/
    usb1/1-1/1-1:1.0/host6/target6:0:0/6:0:0:0/block/sdb':
    KERNEL=="sdb"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{range}=="16"
    ATTR{ext_range}=="256"
    ATTR{removable}=="1"
    ATTR{ro}=="0"
    ATTR{size}=="1974271"
    ATTR{capability}=="53"
    ATTR{stat}==" 71 986 1561 860 1 0 1 12 0 592 872"
...
```

Figure: Showing all **udev** attributes for the device **sdb**. All paths are relative to **/sys**, so **/devices/...** is in reality **/sys/devices/...**

Rules and persistent names

- **udevd** relies on a set of rules to guide its management of devices
- These can be found in the directories:
  - **/lib/udev/rules.d** (default rules), and
  - **/etc/udev/rules.d** (local rules)
- Files from the two rules directories are combined before **udevd** parses them
- Rules in these directories are stored in files, with the name pattern

  *nn-description.**rules***

  where nn is usually a two-digit number
- **.rule** files are processed in a lexical order, i.e., lower numbered rules will be processed first
- The **.rules** suffix is mandatory; files without it are ignored.

Rules and persistent names

Each rule has the following form:

```
match_clause, [match_clause, ...] assign_clause, [ assign_clause, ... ]
```

- The match clause defines the situations where the rule is to be applied
- The assignment clause tells **udevd** what to do when a device is consistent with all the match clauses
- Each clause consists of a key, an operator and a value
- Most match keys refer to device specific properties, which **udevd** reads from /**sys**
  - Examples on the next slide...
- **All** match clauses must hold in order activate a rule

Udevd match keys

Match key	Function
ACTION	Matches the event type, e.g., add or remove
ATTR{*filename*}	Matches a device's sysfs values[a]
DEVPATH	Matches a specific device path
DRIVER	Matches the driver used by a device
ENV{*key*}	Matches the value of an environment variable
KERNEL	Matches the kernel's name for the device
PROGRAM	Runs an external command; matches if the return code is 0
RESULT	Matches the output of the last call through PROGRAM
SUBSYSTEM	Matches a specific subsystem
TEST{*omask*}	Tests whether a file exists; the *omask* is optional

a. The *filename* is a leaf in the sysfs tree that corresponds to a specific attribute.

Figure: Match keys understood by udevd

Using rules

- To perform specific actions for identified devices, e.g.,
  - Mounting the filesystem of attached storage devices
  - Call the appropriate kernel subsystems for devices who needs these
  - Create persistent identities for transient devices

Example: A USB flash drive

We would like a flash drive's device name to be persistent across insertions, and we would like the drive to be mounted and unmounted automatically

Example: A USB flash drive

We can run **dmesg**[4] to see how the kernel identifies a USB flash drive when it is inserted

```
Aug  9 19:50:03 ubuntu kernel: [42689.253554] scsi 8:0:0:0:
    Direct-Access     Ut163    USB2FlashStorage 0.00 PQ: 0 ANSI: 2
Aug  9 19:50:03 ubuntu kernel: [42689.292226] sd 8:0:0:0: [sdb]
    1974271 512-byte hardware sectors: (1.01 GB/963 MiB)
...
Aug  9 19:50:03 ubuntu kernel: [42689.304749] sd 8:0:0:0: [sdb]
    1974271 512-byte hardware sectors: (1.01 GB/963 MiB)
Aug  9 19:50:03 ubuntu kernel: [42689.307182]  sdb: sdb1
Aug  9 19:50:03 ubuntu kernel: [42689.427785] sd 8:0:0:0: [sdb]
    Attached SCSI removable disk
Aug  9 19:50:03 ubuntu kernel: [42689.428405] sd 8:0:0:0: Attached
    scsi generic sg3 type 0
```

Note that the log message indicates the drive was recognised as **sdb1**

[4] Alternative commands: **lsusb** or **journalctl**

Example: A USB flash drive

- We now need to find some way to uniquely identify this device at each insertion
- We can examine the device in /**sys** using **udevadm**, to see if we can find some characteristics for the device

Example: A USB flash drive

```
ubuntu$ udevadm info -a -p /block/sdb/sdb1
looking at device '/devices/pci0000:00/0000:00:11.0/0000:02:03.0/
    usb1/1-1/1-1:1.0/host30/target30:0:0/30:0:0:0/block/sdb/sdb1':
    KERNEL=="sdb1"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{partition}=="1"
    ATTR{start}=="63"
    ATTR{size}=="1974208"
    ATTR{stat}==" 71 792 1857 808 0 0 0 0 0 512 808"

  looking at parent device '/devices/pci0000:00/0000:00:11.0/0000:02:03
    .0/usb1/1-1/1-1:1.0/host30/target30:0:0/30:0:0:0/block/sdb':
    KERNELS=="sdb"
    SUBSYSTEMS=="block"
    DRIVERS==""
    ATTRS{scsi_level}=="3"
    ATTRS{vendor}=="Ut163  "
    ATTRS{model}=="USB2FlashStorage"
...
```

Example: A USB flash drive

- The output shows several possibilities for matching
- The size field may appear to be a good identifier, but formatting the drive may change this
- So instead, we can use
  1. the device's model name (`USB2FlashStorage`), combined with
  2. the kernel's naming convention for storage devices (sd plus an additional letter)
- This allows the creation of the following rule:

Example: A USB flash drive

```
ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    SYMLINK+="ate-flash%n"
```

- This rule triggers **udevd** to set up a symbolic link for the device in /**dev**/**ate-flash*N***, where **N** is the next integer in sequence, starting at 0 (in case there are more flash devices already mounted)
- Next, we can use a new rule using the ACTION attribute to create a mount point directory, and mount the device, whenever the device appears on the USB bus

Example: A USB flash drive

```
ACTION=="add", ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    RUN+="/bin/mkdir -p /mnt/ate-flash%n"
ACTION=="add", ATTRS{model}=="USB2FlashStorage", KERNEL=="sd[a-z]1",
    PROGRAM=="/lib/udev/vol_id -t %N", RESULT=="vfat",
    RUN+="/bin/mount vfat /dev/%k /mnt/ate-flash%n"
```

- RUN is an assignment key that is part of the rule's actions once triggered
- PROGRAM is a match key that is active during the rule selection phase

Example: A USB flash drive

- And we can do some cleanup when the device is removed...

Example: A USB flash drive

```
ACTION=="remove", ATTRS{model}=="USB2FlashStorage",
    KERNEL=="sd[a-z]1", RUN+="/bin/umount -l /mnt/ate-flash%n"
ACTION=="remove", ATTRS{model}=="USB2FlashStorage",
    KERNEL=="sd[a-z]1", RUN+="/bin/rmdir /mnt/ate-flash%n"
```

Example: A USB flash drive

- The above rules should be put in the file /**etc**/**udev**/**rules.d**/*rulename*.**rules** [5]
- Then, reload the rules by

  sudo udevadm control --reload-rules
- And that's it!
- When the flash drive is plugged in, it will have a symbolic link in /**dev**/**ate-flash1**, and mount the drive in /**mnt**/**ate-flash1**

[5] Remember *rulename* should be of the form *nn-description*

Linux kernel configuration

- There are three basic ways to configure a Linux kernel
  - Modifying tunable kernel configuration parameters
  - Building a customised kernel using the source code
  - Loading new drivers and modules into an existing kernel on the fly
- These procedures are used in different situations
- Here, modifying tunable parameters is the **easiest** approach, while building a kernel from source is the **hardest**
  - Fortunately, the latter is rarely required

Tuning Linux kernel parameters

- Many modules and drivers in the kernel were designed with the knowledge that one size doesn't fit all
- As a result, special hooks have been added to allow the adjustment of certain parameters on the fly
- These kernel options can be viewed and set using the files in **/proc/sys**
  - The files (not all are writable) are really "back doors" into the kernel

Tuning Linux kernel parameters

As an example, to change the number of files a system can have open at once:

```
linux# cat /proc/sys/fs/file-max
34916
linux# echo 32768 > /proc/sys/fs/file-max
```

Tuning Linux kernel parameters

- Note that the changes applied directly to the files in **/proc/sys** are not remembered across reboots
- A more permanent approach is to use the **sysctl** command
- For instance, to turn off IP forwarding:

```
linux# sysctl net.ipv4.ip_forward=0
```

- Alternatively, one can manually edit **/etc/sysctl.conf**, which is read at boot time to set the initial values of parameters.

Building a custom kernel

- At some point, it may be necessary to perform adjustments beyond the kernel parameters in **/proc/sys**
- An alternative is to compile a custom kernel, tailored for your system
- However, carefully weigh your site's needs against the risks and time to be invested
  - A new release may not be as stable as the current version, even though it could be the latest and greatest

Configuring kernel options

- Most distributions put kernel sources in versioned subdirectories **/user/src/kernels**
- The configuration of these revolve around the **.config** file at the root of the kernel source directory
- Modifying the **.config** file is not trivial, thus **inadvisable**
  - The format can be cryptic
  - Options are frequently interdependent
    E.g.: enabling an option is not always as simple as changing an **n** to a **y**

Configuring kernel options

- To avoid editing the **.config** file directly, there are several **make** targets available to allow kernel configuration through a user interface, e.g.,
  - **make xconfig**
  - **make gconfig**
  - **make menuconfig** (terminal-based alternative)
- Avoid **make config**
- There is also a decoding guide for the kernel options at */kernel-src-dir/***Documentation/Configure.help**

Example: a .config file

```
# Automatically generated make config: don't edit
# Code maturity level options

CONFIG_EXPERIMENTAL=y

# Processor type and features
# CONFIG_M386 is not set
# CONFIG_M486 is not set
# CONFIG_M586 is not set
# CONFIG_M586TSC is not set
CONFIG_M686=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_X86_TSC=y
CONFIG_X86_GOOD_APIC=y
...
```

Building the kernel binary

The entire process of building, configuring and installing a kernel can be outlined as follows:

- Change the directory (**cd**) to the top level of the source directory
- Run **make xconfig** or **make gconfig**
- Run **make clean**
- Run **make**
- Run **make modules_install**
- Run **make install**

Loadable kernel modules

- Loadable kernel modules (LKMs) allow device drivers (or any other kernel components) to be linked into and removed from the kernel while it is is running
- This simplifies the installation of drivers, as it avoids having to update the kernel binary
- It also allows the kernel to be smaller, as modules are not loaded if they are not used
- While LKMs are convenient, they are not 100% safe. Loading and unloading modules has a risk of causing kernel panic
  - So don't attempt loading untested modules on production systems

Loadable kernel modules

- Loadable kernel modules are stored under /**lib**/**modules**/*version*
  - where *version* is the version of the current Linux kernel (**uname -r**)
- Currently loaded modules can be inspected using the **lsmod** command
- Additional kernel modules can be manually loaded using **modprobe**

modprobe

- **modprobe** is a semi-automatic wrapper on the more primitive **insmod** command
- It understands dependencies, options, installation- and removal procedures
- To figure out how to handle each module, **modprobe** consults the file **/etc/modprobe.conf**
- To generate

modprobe

- As an example, a module implementing sound output to USB devices can be loaded using

  sudo modprobe snd-usb-audio

- Additional parameters can also be passed

  sudo modprobe snd-usb-audio nrpacks=8 async_unlink=1

- And the same module can be removed using

  sudo modprobe -r snd-usb-audio

Booting

Having covered some of the basic functions of the Linux Kernel, it is time to learn what happens when the kernel loads and initialises at startup

The examples on the following slides are derived from a **CentOS 7** machine, with the 3.10.0 version of the Linux kernel being in use

Booting

```
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpuset
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpu
Feb 14 17:18:57 localhost kernel: Initializing cgroup subsys cpuacct
Feb 14 17:18:57 localhost kernel: Linux version 3.10.0-327.el7.x86_64
    (builder@kbuilder.dev.centos.org) (gcc version 4.8.3 20140911 (Red
    Hat 4.8.3-9) (GCC) ) #1 SMP Thu Nov 19 22:10:57 UTC 2015
Feb 14 17:18:57 localhost kernel: Command line: BOOT_IMAGE=/
    vmlinuz-3.10.0-327.el7.x86_64 root=/dev/mapper/centos-root ro
    crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb
    quiet LANG=en_US.UTF-8
```

Booting

- The initial messages indicate that the top level control groups (cgroups) are starting on a Linux 3.10.0 kernel.
- There are also info about who built the kernel, where and what compiler was used (**gcc**)
- The messages also indicate that CentOS is a clone of Red Hat
- The parameters after "command line" are passed from the GRUB boot configuration into the kernel

Booting

```
Feb 14 17:18:57 localhost kernel: e820: BIOS-provided physical RAM map:
Feb 14 17:18:57 localhost kernel: BIOS-e820: [mem
    0x0000000000000000-0x000000000009fbff] usable
Feb 14 17:18:57 localhost kernel: BIOS-e820: [mem 0x000000000009fc00-
    0x000000000009ffff] reserved
...
Feb 14 17:18:57 localhost kernel: Hypervisor detected: KVM
Feb 14 17:18:57 localhost kernel: AGP: No AGP bridge found
Feb 14 17:18:57 localhost kernel: x86 PAT enabled: cpu 0, old
    0x7040600070406, new 0x7010600070106
Feb 14 17:18:57 localhost kernel: CPU MTRRs all blank - virtualized
    system.
Feb 14 17:18:57 localhost kernel: e820: last_pfn = 0xdfff0 max_arch_pfn
    = 0x400000000
Feb 14 17:18:57 localhost kernel: found SMP MP-table at [mem
    0x0009fff0-0x0009ffff] mapped at [ffff88000009fff0]
Feb 14 17:18:57 localhost kernel: init_memory_mapping: [mem
    0x00000000-0x000fffff]
...
```

Booting

- These messages describe the detected processor, and how RAM is mapped
- Note that the kernel notices that it is booted in a hypervisor

Booting

```
Feb 14 17:18:57 localhost kernel: ACPI: bus type PCI registered
Feb 14 17:18:57 localhost kernel: acpiphp: ACPI Hot Plug PCI
    Controller Driver version: 0.5
...
Feb 14 17:18:57 localhost kernel: PCI host bridge to bus 0000:00
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [bus 00-ff]
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [io  0x0000-0xffff]
Feb 14 17:18:57 localhost kernel: pci_bus 0000:00: root bus resource
    [mem 0x00000000-0xffffffff]
...
Feb 14 17:18:57 localhost kernel: SCSI subsystem initialized
Feb 14 17:18:57 localhost kernel: ACPI: bus type USB registered
Feb 14 17:18:57 localhost kernel: usbcore: registered new interface
    driver usbfs
Feb 14 17:18:57 localhost kernel: PCI: Using ACPI for IRQ routing
```

Booting

- This part initialises the kernel's various data buses, including the PCI bus and the USB subsystem

Booting

```
Feb 14 17:18:57 localhost kernel: Non-volatile memory driver v1.3
Feb 14 17:18:57 localhost kernel: Linux agpgart interface v0.103
Feb 14 17:18:57 localhost kernel: crash memory driver: version 1.1
Feb 14 17:18:57 localhost kernel: rdac: device handler registered
Feb 14 17:18:57 localhost kernel: hp_sw: device handler registered
Feb 14 17:18:57 localhost kernel: emc: device handler registered
Feb 14 17:18:57 localhost kernel: alua: device handler registered
Feb 14 17:18:57 localhost kernel: libphy: Fixed MDIO Bus: probed
...
Feb 14 17:18:57 localhost kernel: usbserial: USB Serial support
    registered for generic
Feb 14 17:18:57 localhost kernel: i8042: PNP: PS/2 Controller
    [PNP0303:PS2K,PNP0f03:PS2M] at 0x60,0x64 irq 1,12
Feb 14 17:18:57 localhost kernel: serio: i8042 KBD port 0x60,0x64 irq 1
Feb 14 17:18:57 localhost kernel: serio: i8042 AUX port 0x60,0x64 irq 12
Feb 14 17:18:57 localhost kernel: mousedev: PS/2 mouse device common
    for all mice
Feb 14 17:18:57 localhost kernel: input: AT Translated Set 2 keyboard as
    /devices/platform/i8042/serio0/input/input2
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: rtc core: registered
    rtc_cmos as rtc0
Feb 14 17:18:57 localhost kernel: cpuidle: using governor menu
Feb 14 17:18:57 localhost kernel: usbhid: USB HID core driver
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: alarms up to one
    day, 114 bytes nvram
```

Booting

- These messages describe the kernel's discovery of various devices, such as the power button, USB hub, mouse, and a real-time clock chip (RTC)

Booting

```
Feb 14 17:18:57 localhost kernel: drop_monitor: Initializing network
    drop monitor service
Feb 14 17:18:57 localhost kernel: TCP: cubic registered
Feb 14 17:18:57 localhost kernel: Initializing XFRM netlink socket
Feb 14 17:18:57 localhost kernel: NET: Registered protocol family 10
Feb 14 17:18:57 localhost kernel: NET: Registered protocol family 17
```

Booting

- This phase initialises a variety of network drivers and facilities

Booting

```
Feb 14 17:18:57 localhost kernel: Loading compiled-in X.509 certificates
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux kpatch
    signing key: ea0413152cde1d98ebdca3fe6f0230904c9ef717'
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux Driver
    update signing key: 7f421ee0ab69461574bb358861dbe77762a4201b'
Feb 14 17:18:57 localhost kernel: Loaded X.509 cert 'CentOS Linux kernel
    signing key: 79ad886a113ca0223526336c0f825b8a94296ab3'
Feb 14 17:18:57 localhost kernel: registered taskstats version 1
Feb 14 17:18:57 localhost kernel: Key type trusted registered
Feb 14 17:18:57 localhost kernel: Key type encrypted registered
```

Booting

- Like other OSes, CentOS provides a way to incorporate and validate updates. The validation portion uses X.509 certificates installed into the kernel

Booting

```
Feb 14 17:18:57 localhost kernel: IMA: No TPM chip found, activating
    TPM-bypass!
Feb 14 17:18:57 localhost kernel: rtc_cmos rtc_cmos: setting system
    clock to 2017-02-14 22:18:57 UTC (1487110737)
```

Booting

- Here, the kernel reports it can not find a Trusted Platform Module (TPM)
- TPM chips are cryptographic hardware devices that provide secure signing operations, making it harder to insert malicious code
- The RTC clock is also set current time of day at this stage

Booting

```
Feb 14 17:18:57 localhost kernel: e1000: Intel(R) PRO/1000 Network
    Driver - version 7.3.21-k8-NAPI
Feb 14 17:18:57 localhost kernel: e1000: Copyright (c) 1999-2006
    Intel Corporation.
Feb 14 17:18:58 localhost kernel: e1000 0000:00:03.0 eth0:
    (PCI:33MHz:32-bit) 08:00:27:d0:ae:6f
Feb 14 17:18:58 localhost kernel: e1000 0000:00:03.0 eth0: Intel(R)
    PRO/1000 Network Connection
```

Booting

- Now, the kernel has found the gigabit ethernet interface and initialised it
- The interface's MAC address is `08:00:27:d0:ae:6f`

Booting

```
Feb 14 17:18:58 localhost kernel: scsi host0: ata_piix
Feb 14 17:18:58 localhost kernel: ata1: PATA max UDMA/33 cmd 0x1f0 ctl
    0x3f6 bmdma 0xd000 irq 14
Feb 14 17:18:58 localhost kernel: ahci 0000:00:0d.0: flags: 64bit ncq
    stag only ccc
Feb 14 17:18:58 localhost kernel: scsi host2: ahci
Feb 14 17:18:58 localhost kernel: ata3: SATA max UDMA/133 abar
    m8192@0xf0806000 port 0xf0806100 irq 21
Feb 14 17:18:58 localhost kernel: ata2.00: ATAPI: VBOX CD-ROM, 1.0,
    max UDMA/133
Feb 14 17:18:58 localhost kernel: ata2.00: configured for UDMA/33
Feb 14 17:18:58 localhost kernel: scsi 1:0:0:0: CD-ROM          VBOX
    CD-ROM          1.0  PQ: 0 ANSI: 5
Feb 14 17:18:58 localhost kernel: tsc: Refined TSC clocksource
    calibration: 3399.654 MHz
Feb 14 17:18:58 localhost kernel: ata3: SATA link up 3.0 Gbps (SStatus
    123 SControl 300)
Feb 14 17:18:58 localhost kernel: ata3.00: ATA-6: VBOX HARDDISK, 1.0,
    max UDMA/133
Feb 14 17:18:58 localhost kernel: ata3.00: 16777216 sectors, multi 128:
    LBA48 NCQ (depth 31/32)
Feb 14 17:18:58 localhost kernel: ata3.00: configured for UDMA/133
Feb 14 17:18:58 localhost kernel: scsi 2:0:0:0: Direct-Access     ATA
    VBOX HARDDISK    1.0  PQ: 0 ANSI: 5
Feb 14 17:18:58 localhost kernel: sr 1:0:0:0: [sr0] scsi3-mmc drive:
    32x/32x xa/form2 tray
Feb 14 17:18:58 localhost kernel: cdrom: Uniform CD-ROM driver Revision:
    3.20
Feb 14 17:18:58 localhost kernel: sd 2:0:0:0: [sda] 16777216 512-byte
    logical blocks: (8.58 GB/8.00 GiB)
Feb 14 17:18:58 localhost kernel: sd 2:0:0:0: [sda] Attached SCSI disk
Feb 14 17:18:58 localhost kernel: SGI XFS with ACLs, security attributes,
    no debug enabled
Feb 14 17:18:58 localhost kernel: XFS (dm-0): Mounting V4 Filesystem
Feb 14 17:18:59 localhost kernel: XFS (dm-0): Ending clean mount
```

Booting

- Finally, the kernel recognises various drives and support devices (harddrives, a virtuel CD-ROM)
- It also maps the XFS filesystem, which is used by the device-mapper subsystem.

- Linux kernel boot messages are verbose almost to a fault
- Though this makes it much easier to debug any potential startup errors!

Read p.349–353 for more detailed explanation.

Kernel errors

- Kernel failures is a reality that can occur even on well-configured systems
- Linux has four varieties of kernel failure:
  - **Soft lockups**
  - **Hard lockups**
  - **Panics** (Kernel crashes)
  - And the infamous "**oops**" (something, somewhere, went wrong)

Lockups

Soft lockups

- Occurs when the system is in kernel mode for more than a few seconds, thus user-level tasks cannot run
  - The interval is configurable, but usually around 10 seconds
- Kernel is the only thing running, and servicing interrupts and data is still flowing in/out of the system
- Can occur on correctly configured systems that are experiencing some kind of extreme condition, like a high CPU load

Hard lockups

- Same as a soft lockups, but most processor interrupts go unserviced
- Usually can be detected relatively quickly

Lockups

When a lockup occurs,

- A stack trace known as a **tombstone**, which is often dumped to the console or a file

Then,

- Either the system stays frozen so that the tombstone remains visible on the console;
- Or, the system panics (craches) and reboot

Oops

- A generalization of the traditional UNIX "panic after any anomaly"
- Oopses can lead to a panic, but not necessarily
  - E.g., killing an individual process
- The kernel generates a tombstone in the kernel message buffer can be read by **dmesg**