Chapter 2

A practical view
of the Linux system

2.1 Overview

In this chapter, we approach the Linux system from a practical perspective, as experienced by users of the system, in particular administrators and application programmers rather than kernel or driver programmers. We first introduce the essential concepts and techniques that you need to know in order to understand the overall system, and then we discuss the system itself from different angles: what is the OS role in booting and initializing the system; what OS knowledge does a system administrator and systems programmer need. This chapter is *not* a how-to guide, but rather provides you with the background knowledge behind how-to guides. It also serves as a roadmap for the rest of the book.

What you will learn

After you have studied the material in this chapter, you will be able to:

1. Explain basic operating system concepts: processes, users, files, permissions, and credentials.

2. Analyze the chain of events when booting Linux on the Raspberry Pi.

3. Create a Linux kernel module and build a custom Linux kernel.

4. Discuss the administrator and programmers view on the key operating system concepts covered in the further chapters.

2.2 Basic concepts

To understand what happens when the system boots and initializes, as well as how the OS affects the tasks of system administrator and systems programmer, we need to introduce a number of basic operating system concepts. Most of these apply to any operating system, although the discussion here is specific to Linux on Arm-based systems. The in-depth discussion of these concepts forms the subject of the later chapters, so this section serves as a roadmap for the rest of the book as well.

The original Linux announcement on Usenet (1991). Photo by Krd.

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki
Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since april, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).
I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want. Any suggestions
are welcome, but I won't promise I'll implement them :-)
Linus (torvalds@kruuna.helsinki.fi)
PS. Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

2.2.1 Operating system hierarchy

The Linux kernel is only one component of the complete operating system. Figure 2.1 illustrates the complete Linux system hierarchy. Interfacing between the kernel and the user space applications is the system call interface, a mechanism to allow user space applications to interact with the kernel and hardware. This interface is used by system tools and libraries, and finally by the user applications. The kernel provides functionality such as scheduling, memory management, networking and file system support, and support for interacting with system hardware via device drivers.

Interfacing between the kernel and the hardware are device drivers and the firmware. In the Linux system, device drivers interact closely with the kernel, but they are not considered part of the kernel because depending on the hardware different drivers will be needed, and they can be added on the fly.

2.2.2 Processes

A *process* is a running program, i.e., the code for the program and all system resources it uses. The concept of a process is used for the separation of code and resources. The OS kernel allocates memory resources and other resources to a process, and these are private to the process, and protected from all other processes. The scheduler allocates time for a process to execute. We also use the term *task*, which is a bit less strictly defined, and usually relates to scheduling: a task is an amount of work to be done by a program. We will also see the concept of *threads*, which are used to indicate multiple concurrent tasks executing within a single process. In other words, the threads of a process share its resources. For a process with a single thread of execution, the terms task and process are often used interchangeably.

When a process is created, the OS kernel assigns it a unique identifier (called process ID or PID for short) and creates a corresponding data structure called the Process Control Block or Task Control Block (in the Linux kernel this data structure is called `task_struct`). This is the main mechanism the kernel uses to manage processes.

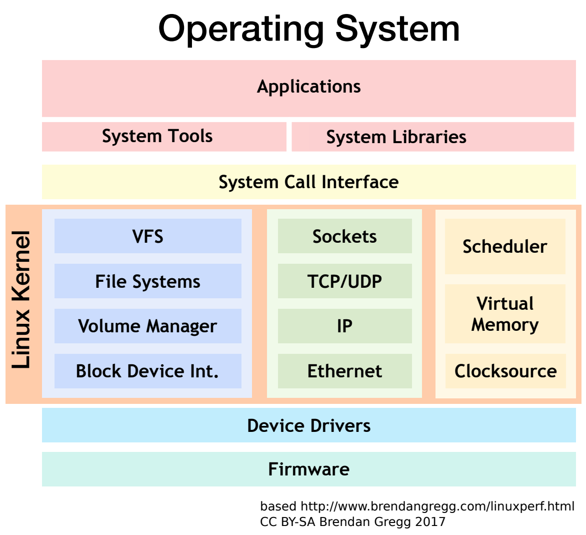


Figure 2.1: Operating System Hierarchy.

2.2.3 User space and kernel space

The terms 'user space' and 'kernel space' are used mainly to indicate process execution with different privileges. As we have seen in Chapter 1, the kernel code can access all hardware and memory in the system, but for user processes, the access is very much restricted. When we use the term 'kernel space,' we mean the memory space accessible by the kernel, which is effectively the complete memory space in the system[1]. By 'user space,' we mean the memory accessible by a user process. Most operating systems support multiple users, and each user can run multiple processes. Typically, each process gets its own memory space, but processes belonging to a single user can share memory (in which case we'll call them threads).

2.2.4 Device tree and ATAGs

The Linux kernel needs information about the system on which it runs. Although a kernel binary must be compiled for a target architecture (e.g., Arm), a kernel binary should be able to run on a wide variety of platforms for this architecture. This means that the kernel has to be provided with information about the hardware at boot time, e.g., number of CPUs, amount of memory, location of memory, devices and their location in the memory map, etc. The traditional way to do this on Arm systems was using a format called ATAGs, which provided a data structure in the kernel that would be populated with information that the bootloader provided. A more modern and flexible approach is called Device Tree[3]. It defines a format and syntax to describe system hardware in a Device Tree Source file. A device tree is a tree data structure with nodes that describe the physical devices in a system. The Device Tree source files can be compiled using a special compiler into a machine-architecture-independent binary format called Device Tree Blob.

2.2.5 Files and persistent storage

The Linux Information Project defines a *file* as:

A file is a named collection of related data that appears to the user as a single, contiguous block of information and that is retained in storage.[2]

In this definition, *storage* refers to computer devices or media which can retain data for relatively long periods (e.g., years or decades), such as solid state drives and other types of non-volatile memory, magnetic hard disk drives (HDDs), CDROMs and magnetic tape, in other words, persistent storage. This is in contrast with RAM memory, the content of which is retained only temporarily (i.e., only while in use or while the power supply remains on).

A persistent storage medium (which I will call 'disk') such as an SD card, USB memory stick or hard disk, stores data in a linear fashion with sequential access. However, in practice, the disk does not contain a single array of bytes. Instead, it is organized using *partitions* and *file systems*. We discuss these in more detail in Chapter 9, but below is a summary of these concepts.

Partition

A disk can be divided into partitions, which means that instead of presenting as a single blob of data, it presents as several different blobs. Partitions are logical rather than physical, and the information about how the disk is partitioned (i.e., the location, size, type, name, and attributes of each partition) is stored in a partition table. There are several standards for the structure of partitions and partition tables, e.g., GUID Partition Table and MBR.

[1] Assuming the system does not run a hypervisor. Otherwise, it is the memory available to the Virtual Machine running the kernel.
[2] http://www.linfo.org/file.html [3] https://www.devicetree.org/specifications

File system
Each partition of a disk contains a further system for logical organization. The purpose of most file systems is to provide the *file* and *directory (folder)* abstractions. There are a great many different file systems (e.g., fat32, ext4, hfs+, ...) and we will cover the most important ones in Chapter 9. For the purpose of this chapter, what you need to know is that a file system not only allows to store information in the form of files organized in directories but also information about the permissions of usages for files and directories, as well as timestamp information (file creation, modification, etc.).

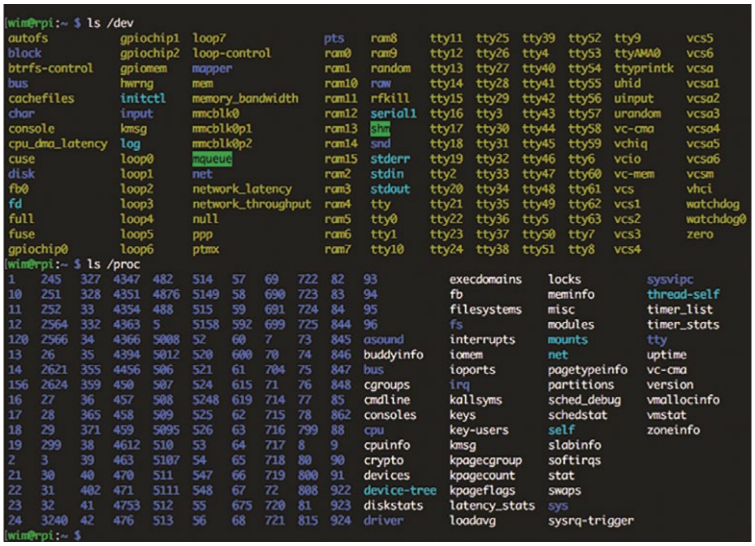
The information in a file system is typically organized as a hierarchical tree of directories, and the directory at the root of the tree is called the root directory. To use a file system, the kernel performs an operation called *mounting*. As long as a file system has not been mounted, the system can't access the data on it.

Mounting a file system attaches that file system to a directory (mount point) and makes it available to the system. The root (/) file system is always mounted. Any other file system can be connected or disconnected from the root file system at any point in the directory tree.

2.2.6 'Everything is a file'

One of the key characteristics of Linux and other UNIX-like operating systems is the often-quoted concept of 'everything is a file.' This does not mean that all objects in Linux are files as defined above, but rather that Linux prefers to treat all objects from which the OS can read data or to which it can write data using a consistent interface. So it might be more accurate to say 'everything is a stream of bytes.' Linux uses the concept of a file descriptor, an abstract handle used to access an input/output resource (of which a file is just one type). So one can also say that in Linux, 'everything is a file descriptor.'

What this means in practice is that the interface to, e.g., a network card, keyboard or display is represented as a file in the file system (in the /dev directory); system information about both hardware and software is available under /proc. For example, Figure 2.2 shows the listing of `/dev` and `/proc` on the Raspberry Pi. We can see device files representing memory (ram*), terminals (tty*), modem (ppp),



```
[wim@rpi:~ $ ls /dev
autofs          gpiochip1  loop7                pts     ram8     tty11  tty25  tty39  tty52  tty9       vcs5
block           gpiochip2  loop-control         ram0    ram9     tty12  tty26  tty4   tty53  ttyAMA0    vcs6
btrfs-control   gpiomem    mapper               ram1    random   tty13  tty27  tty40  tty54  ttyprintk  vcsa
bus             hwrng      mem                  ram10   raw      tty14  tty28  tty41  tty55  uhid       vcsa1
cachefiles      initctl    memory_bandwidth     ram11   rfkill   tty15  tty29  tty42  tty56  uinput     vcsa2
char            input      mmcblk0              ram12   serial1  tty16  tty3   tty43  tty57  urandom    vcsa3
console         kmsg       mmcblk0p1            ram13   shm      tty17  tty30  tty44  tty58  vc-cma     vcsa4
cpu_dma_latency log        mmcblk0p2            ram14   snd      tty18  tty31  tty45  tty59  vchiq      vcsa5
cuse            loop0      mqueue               ram15   stderr   tty19  tty32  tty46  tty6   vcio       vcsa6
disk            loop1      net                  ram2    stdin    tty2   tty33  tty47  tty60  vc-mem     vcsm
fb0             loop2      network_latency      ram3    stdout   tty20  tty34  tty48  tty61  vcs        vhci
fd              loop3      network_throughput   ram4    tty      tty21  tty35  tty49  tty62  vcs1       watchdog
full            loop4      null                 ram5    tty0     tty22  tty36  tty5   tty63  vcs2       watchdog0
fuse            loop5      ppp                  ram6    tty1     tty23  tty37  tty50  tty7   vcs3       zero
gpiochip0       loop6      ptmx                 ram7    tty10    tty24  tty38  tty51  tty8   vcs4
[wim@rpi:~ $ ls /proc
1    245   327  4347  482   514   57   69   722  82   93           execdomains   locks          sysvipc
10   251   328  4351  4876  5149  58   690  723  83   94           fb            meminfo        thread-self
11   252   33   4354  488   515   59   691  724  84   95           filesystems   misc           timer_list
12   2564  332  4363  5     5158  592  699  725  844  96           fs            modules        timer_stats
120  2566  34   4366  5008  52    60   7    73   845  asound       interrupts    mounts         tty
13   26    35   4394  5012  520   600  70   74   846  buddyinfo    iomem         net            uptime
14   2621  355  4456  506   521   61   704  75   847  bus          ioports       pagetypeinfo   vc-cma
156  2624  359  450   507   524   615  71   76   848  cgroups      irq           partitions     version
16   27    36   457   508   5248  619  714  77   85   cmdline      kallsyms      sched_debug    vmallocinfo
17   28    365  458   509   525   62   715  78   862  consoles     keys          schedstat      vmstat
18   29    371  459   5095  526   63   716  799  88   cpu          key-users     self           zoneinfo
19   299   38   4612  510   53    64   717  8    9    cpuinfo      kmsg          slabinfo
2    3     39   463   5107  54    65   718  80   90   crypto       kpagecgroup   softirqs
21   30    40   470   511   547   66   719  800  91   devices      kpagecount    stat
22   31    402  471   5111  548   67   72   808  922  device-tree  kpageflags    swaps
23   32    41   4753  512   55    675  720  81   923  diskstats    latency_stats sys
24   3240  42   476   513   56    68   721  815  924  driver       loadavg       sysrq-trigger
[wim@rpi:~ $
```

Figure 2.2: Listing of /dev and /proc on the Raspberry Pi running Raspbian.

and many others. In particular, there is `/dev/null` which is a special device which discards the information written to it, and `/dev/zero` which returns an endless stream of zero bytes (i.e., 0x00, so when you try cat `/dev/zero` you will see nothing. Try `cat/dev/zero | hd` instead.)

2.2.7 Users

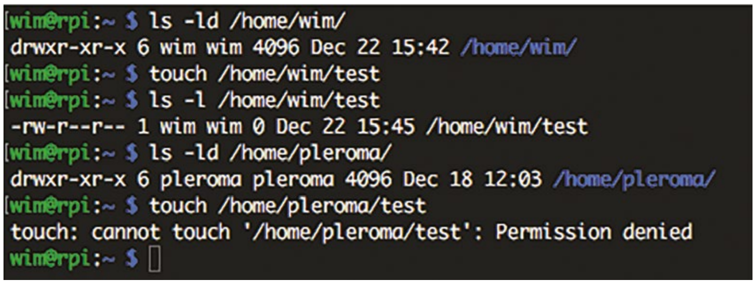
A Linux system is typically a multi-user system. What this means is that it supports another level of separation, permissions, and protection above the level of processes. A user can run and control multiple processes, each in their own memory space, but with shared access to system resources. In particular, the concept of users and permission is tightly connected with the file system. The file system permissions for a given user control the access of that user in terms of reading, writing, and executing files in different parts of the file system hierarchy.

Just as the kernel runs in privileged mode to control the user space processes, there is also a need for a privileged user to control the other users (similar to the 'Administrator' on Windows systems). In Linux, this user is called *root*[3] and when the system boots, the first process (*init*, which has PID=1) is run as the root user. The init process can create new processes. In fact, in Linux, any process can create new processes (as explained in more detail in Chapter 4). However, a process owned by the root user can assign ownership of a created process to another user, whereas processes created by a non-root user process can only be owned by itself.

2.2.8 Credentials

In Linux, *credentials* is the term for the set of privileges and permissions associated with any object. Credentials express, e.g., ownership, capabilities, and security management properties. For example, for files and processes, the key credentials are the user id and group id. To decide what a certain object (e.g., a task) can do to another object (e.g., a file), the Linux kernel performs a security calculation using the credentials and a set of rules. In practice, processes executed as root can access all files and other resources in the system; for a non-root user, file and directory access is determined by a system of permissions on the files and by the membership of groups: a user can belong to one or more groups of users.

File access permissions can be specified for individual users, groups, and everyone. For example, in Figure 2.3, we see that the directory /home/wim can be written to by user wim in group wim. If we try to create an (empty) file using the touch command, this succeeds. However, if we try to do the same in the directory /home/pleroma, owned by user pleroma in group pleroma, we get 'permission denied' because only user pleroma has write access to that directory.



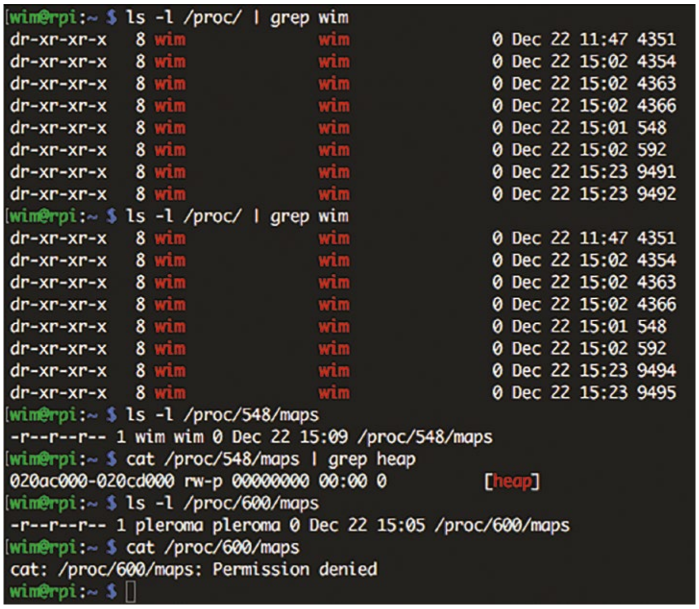
```
wim@rpi:~ $ ls -ld /home/wim/
drwxr-xr-x 6 wim wim 4096 Dec 22 15:42 /home/wim/
wim@rpi:~ $ touch /home/wim/test
wim@rpi:~ $ ls -l /home/wim/test
-rw-r--r-- 1 wim wim 0 Dec 22 15:45 /home/wim/test
wim@rpi:~ $ ls -ld /home/pleroma/
drwxr-xr-x 6 pleroma pleroma 4096 Dec 18 12:03 /home/pleroma/
wim@rpi:~ $ touch /home/pleroma/test
touch: cannot touch '/home/pleroma/test': Permission denied
wim@rpi:~ $
```

Figure 2.3: Example of restrictions on file creation on the Raspberry Pi running Raspbian.

[3] For more info about the origin of the name, root see www.linfo.org/root.html

Note that because of the 'everything is a file' approach, this system of permissions extends in general to devices, system information, etc. However, the actual kernel security policies can restrict access further. For example, in Figure 2.2, the numbers in the `/proc` listing represent currently running processes by their PID.

To illustrate the connection between users, permissions, and processes, Figure 2.4 shows how user `wim` can list processes in `/proc` belonging to two different non-root users, `wim`, and `pleroma`. The command `cat /proc/548/maps` prints out the entire memory map for the process with PID 548. The map is quite large, so for this example, only the heap memory allocation is shown (using `grep heap`).



```
wim@rpi:~ $ ls -l /proc/ | grep wim
dr-xr-xr-x   8 wim            wim            0 Dec 22 11:47 4351
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4354
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4363
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4366
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:01 548
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 592
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:23 9491
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:23 9492
wim@rpi:~ $ ls -l /proc/ | grep wim
dr-xr-xr-x   8 wim            wim            0 Dec 22 11:47 4351
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4354
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4363
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 4366
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:01 548
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:02 592
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:23 9494
dr-xr-xr-x   8 wim            wim            0 Dec 22 15:23 9495
wim@rpi:~ $ ls -l /proc/548/maps
-r--r--r-- 1 wim wim 0 Dec 22 15:09 /proc/548/maps
wim@rpi:~ $ cat /proc/548/maps | grep heap
020ac000-020cd000 rw-p 00000000 00:00 0          [heap]
wim@rpi:~ $ ls -l /proc/600/maps
-r--r--r-- 1 pleroma pleroma 0 Dec 22 15:05 /proc/600/maps
wim@rpi:~ $ cat /proc/600/maps
cat: /proc/600/maps: Permission denied
wim@rpi:~ $ 
```

Figure 2.4: Example of restrictions on process memory access via /proc on the Raspberry Pi running Raspbian.

However, when we try to do the same with `/proc/600/maps`, we get 'Permission denied' because the cat process owned by user `wim` does not have the right to inspect the memory map of a process owned by another user. This is despite the file permissions allowing read access.

2.2.9 Privileges and user administration

The system administrator creates user accounts and decides on access to resources using groups (using tools such as useradd(8), groupadd(8), chgrp(1), etc.). The kernel manages credentials per process using struct cred which is a field of the `task_struct`.

The admin also decides how many resources each user and process gets, e.g., using ulimit.

Resource limits are set in `/etc/security/limits.conf` and can be changed at runtime via the shell command ulimit. Internally, the ulimit implementation uses the getrlimit and setrlimit system calls which modify the kernel struct rlimit in include/uapi/linux/resource.h.

2.3 Booting Linux on Arm-based systems (Raspberry Pi 3)

In this section, we discuss the boot process for Linux on the Raspberry Pi 3. The boot sequence of Linux on Arm-based systems varies significantly from platform to platform. The differences sometimes arise due to the needs of the target market but can also be due to choices made by SoC and platform vendors. The boot sequence discussed here is a specific example to demonstrate what happens on a particular platform.

This Raspberry Pi 3 (Figure 2.5) runs Raspbian Linux on an Arm Cortex-A53 processor which is part of the Broadcom BCM2837 System-on-Chip (SoC). This SoC also contains a GPU (Broadcom VideoCore IV) which shares the RAM with the CPU. The GPU controls the initial stages of the boot process. The SoC also has a small amount of One Time Programmable (OTP) memory which contains information about the boot mode and a boot ROM with the initial boot code.

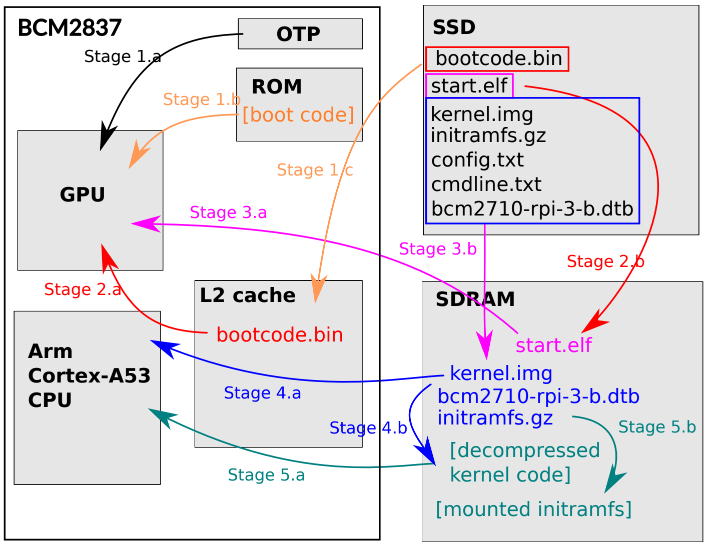


Figure 2.5: Boot Process for Raspbian Linux on the Raspberry Pi 3.

2.3.1 Boot process stage 1: Find the bootloader

Stage 1 of the boot process begins with reading the OTP to check which boot modes are enabled. By default, this is SD card boot, followed by a USB device boot. The code for this stage is stored in the on-chip ROM. The boot code checks each of the boot sources for a file called `bootcode.bin` in the root directory of the first partition on the storage medium (FAT32 formatted); if it is successful, it will load the code into the local 128K (L2) cache and jump to its first instruction to start Stage 2.

Note: The boot ROM supports GUID partitioning and MBR-style partitioning.

2.3.2 Boot process stage 2: Enable the SDRAM

Stage 2 is controlled by `bootcode.bin`, which is closed-source firmware. It enables the SDRAM and loads Stage 3 `(start.elf)` from the storage medium into the SDRAM.

2.3.3 Boot process stage 3: Load the Linux kernel into memory

Stage 3 is controlled by `start.elf`, which is a closed-source ELF-format binary running on the GPU.

`start.elf` loads the compressed Linux kernel binary `kernel.img` and copies it to memory. It reads `config.txt, cmdline.txt` and `bcm2710-rpi-3-b.dtb` (Device Tree Binary).

The file `config.txt` is a text file containing system configuration parameters which would on a conventional PC be edited and stored using a BIOS.

The file `cmdline.txt` contains the command line arguments to be passed on to the Linux kernel (e.g., the file system type and location of the root file system) using ATAGs, and the `.dtb` file contains the Device Tree Blob.

2.3.4 Boot process stage 4: Start the Linux kernel

Stage 4 starts `kernel.img` on the CPU: releasing reset on the CPU causes it to run from the address where the kernel.img data was written. The kernel runs some Arm-specific code to populate CPU registers and turn on the cache, then decompresses itself, and runs the decompressed kernel code. The kernel initializes the MMU using Arm-specific code and then run the rest of the kernel code which is processor-independent.

2.3.5 Boot process stage 5: Run the processor-independent kernel code

Stage 5 is the processor-independent kernel code. This code consists mainly of initialization functions to set up interrupts, perform further memory configuration, and load the initial RAM disk `initramfs`.

This is a complete set of directories that you would find on a normal root file system and was loaded into memory by the Stage 3 boot loader. It is copied into kernel space memory and mounted. This initramfs serves as a temporary root file system in RAM and allows the kernel to fully boot and perform user-space operations without having to mount any physical disks.

A single Linux kernel image can run on multiple platforms with support for a large number of devices/ peripherals. To reduce the overhead of loading and running a kernel binary bloated with features that aren't widely used, Linux supports runtime loading of components (modules) that are not needed during early boot. Since the necessary modules needed to interface with peripherals can be part of the initramfs, the kernel can be very small, but still, support a large number of possible hardware configurations. After the kernel is booted, the initramfs root file system is unmounted, and the real root file system is mounted. Finally, the `init` function is started, which is the first user-space process. After this, the idle task is started, and the scheduler starts operation.

2.3.6 Initialization

After the kernel has booted it launches the first process, called *init*. This process is the parent of all other processes. In the Raspbian Linux distribution that runs on the Raspberry Pi 3, this `init` is actually an alias for `/lib/systemd/systemd` because Raspbian, as a Debian-derived distribution, uses systemd as its init system. Other Linux distributions can have different implementations of init, e.g., SysV init or upstart.

The *systemd* process executes several processes to initialize the system: keyboard, hardware drivers, file systems, network, services. It has a sophisticated system for configuring all the processes under its control as well as for starting and stopping processes, checking their status, logging, changing privileges, etc.

The systemd process performs many tasks, but the principle is always the same: it starts a process under the required user name and monitors its state. If the process exits, systemd takes appropriate action, e.g., restarting the process or reporting the error that caused it to exit.

2.3.7 Login

One of the systemd responsibilities is running the processes that let users log into the system (systemd-logind). To login via a terminal (or virtual console), Linux uses two programs: *getty* and *login* (originally, the tty in getty meant 'teletype,' a precursor to modern terminals). Both run as root.

A basic *getty* program opens the terminal device, initializes it, prints the login prompt, and waits for a user name to be entered. When this happens, *getty* executes the *login* program, passing it the user name to log in as. The *login* program then prompts the user for a password. If the password is wrong, *login* simply exits. The systemd process will notice this and spawn another *getty* process. If the password is correct, *login* executes the user's shell program as that user. From then on, the user can start processes via the shell.

The reason why there are two separate programs is that both *getty* and *login* can be used on their own, for example, a remote login over SSH does not use a terminal but still uses *login*: each new connection is handled by a program called *sshd* that starts a login process.

A graphical login is conceptually not that different from the above description. The difference is that instead of the getty/login programs, a graphical login program called the *display manager* is run, and after authentication, this program launches the graphical shell.

In Raspbian the display manager is LightDM, and the graphical shell is LXDE (Lightweight X11 Desktop Environment). Like most Linux distributions, the graphical desktop environment is based on the X Window System (X11), a project originally started at MIT and now managed by the X.Org Foundation.

2.4 Kernel administration and programming

The administrator of a Linux system does not need to know the inner workings of the Linux kernel but needs to be familiar with tools to configure the operating system, including adding functionality to the kernel through kernel modules, and compilation of a custom kernel.

2.4.1 Loadable kernel modules and device drivers

As explained above, the Linux kernel is modular, and functionality can be loaded at run time using Loadable Kernel Modules (LKM). This feature is used in particular to configure drivers for the system hardware. Therefore the administrator needs to be familiar with the main concepts of the module system and a basic understanding of the role of a device driver.

- To insert a module into the Linux kernel, the command *insmod (8)* can be used. *insmod* makes an *init_module()* system call to load the LKM into kernel memory.

- The *init_module()* system call invokes the LKM's initialization routine immediately after it loads the LKM. *insmod* passes to *init_module()* the address of the initialization subroutine in the LKM using the *macro module_init()*.

- The LKM author sets up the module's init_module to call a kernel function that registers the subroutines that the LKM contains. For example, a character device driver's init_module subroutine might call the *register_chrdev* kernel subroutine, passing the major and minor number of the device it intends to drive and the address of its own *open()* routine as arguments. *register_chrdev* records that when the kernel wants to open that particular device, it should call the *open()* routine in our LKM.

- When an LKM is unloaded (e.g., via the rmmod(8) command), the LKM's cleanup subroutine is called via the macro *module_exit()*.

- In practice, the administrator will want to use the more intelligent modprobe(8) command to handle module dependencies automatically. Finally, to list all loaded kernel modules, the command lsmod(8) can be used.

For the curious, the details of implementation are init_module, load_module, and do_init_module in kernel/module.c.

2.4.2 Anatomy of a Linux kernel module

As an administrator, sometimes you may have to add a new device to your system for which the standard kernel of your system's Linux distro does not provide a driver. That means you will have to add this driver to the kernel.

A trivial kernel module is very simple. The following module will print some information to the kernel log when it is loaded and unloaded.

Listing 2.4.1: A trivial kernel module C

```c
#include <linux/init.h>          // For macros __init __exit
#include <linux/module.h>        // Kernel LKM functionality
#include <linux/kernel.h>        // Kernel types and function definitions

static int __init hello_LKM_init(void){
   printk(KERN_INFO "Hello from our LKM!\n");
   return 0;
}

static void __exit hello_LKM_exit(void){
   printk(KERN_INFO "Goodbye from our LKM!\n");
}

module_init(hello_LKM_init);
module_exit(hello_LKM_exit);
```

However, note that a kernel module is not an application; it is a piece of code to be used by the kernel. As you can see, there is no *main()* function. Furthermore, kernel modules:

- do not execute sequentially: a kernel module registers itself to handle requests using its initialization function, which runs and then terminates. The types of request that it can handle are defined within the module code. This is quite similar to the event-driven programming model that is commonly utilized in graphical-user-interface (GUI) applications.

- do not have automatic resource management (memory, file handles, etc.): any resources that are allocated in the module code must be explicitly deallocated when the module is unloaded.

- do not have access to the common user-space system calls, e.g., *printf()*. However, there is a *printk()* function that can output information to the kernel log, and which can be viewed from user space.

- can be interrupted: kernel modules can be used by several different programs/processes at the same time, as they are part of the kernel. When writing a kernel module you must, therefore, be very careful to ensure that the module behavior is consistent and correct when the module code is interrupted.

- have to be very resource-aware: as a module is kernel code, its execution contributes to the kernel runtime overhead, both in terms of CPU cycles and memory utilization. So you have to be very aware that your module should not harm the overall performance of your system.

The macros *module_init* and *module_exit* are used to identify which subroutines should be run when the module is loaded and unloaded. The rest of the module functionality depends on the purpose of the module, but the general mechanism used in the kernel to connect a specific module to a generic API (e.g., the file system API) is via a struct with function pointers, which functions in the same way as an object interface declaration in Java or C++. For example, the file system API provides a struct file_operations (defined in include/linux/fs.h) which looks as follows:

Listing 2.4.2: file_operations struct from <include/linux/fs.h> C

```c
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
   ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
   int (*iterate) (struct file *, struct dir_context *);
   int (*iterate_shared) (struct file *, struct dir_context *);
   __poll_t (*poll) (struct file *, struct poll_table_struct *);
   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
   long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   unsigned long mmap_supported_flags;
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *, fl_owner_t id);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, loff_t, loff_t, int datasync);
```

```c
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *,
       unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
       size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
       size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
       loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
       loff_t, size_t, unsigned int);
    int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
    u64);
    ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
       u64);
} __randomize_layout;
```

So if you want to implement a module for a custom file system driver, you will have to provide implementations of the calls you want to support with the signatures as provided in this struct. Then in your module code, you can create an instance of this struct and populate it with pointers to the functions you've implemented, for example, assuming you have implemented *my_file_open*, *my_file_read*, *my_file_write, and my_file_close*, you would create the following struct:

Listing 2.4.3: Example file_operations struct *C*

```c
static struct file_operations my_file_ops =
{
   .open = my_file_open,
   .read = my_file_read,
   .write = dmy_file_write,
   .release = dmy_file_close,
};
```

Now all that remains is to make the kernel use this struct, and this is achieved using yet another API call which you call in the initialization subroutine. In the case of a driver for a character file (e.g., a serial port or audio device), this call would be `register_chrdev(0, DEVICE_- NAME, &my_file_ops)`. This API call is also defined in include/linux/fs.h. Other types of devices have similar calls to register new functionality with the kernel.

2.4.3 Building a custom kernel module

If you want to create your own kernel module, you don't need the entire kernel source code, but you do need the kernel header files. On a Raspberry Pi 3 running Raspbian, you can use the following commands to install the kernel headers:

Listing 2.4.4: Installing kernel headers on Raspbian *Bash*

```bash
$ sudo apt-get update
$ sudo apt-get install raspberrypi-kernel-headers
```

The Linux kernel has a dedicated Makefile-based system to build modules (and to build the actual kernel) called kbuild. The kernel documentation provides a good explanation of how to build a kernel module in Documentation/kbuild/modules.txt.

The disadvantage to building a kernel module from source is that you have to rebuild it every time you upgrade the kernel. The Dynamic Kernel Module Support (dkms) framework offers a way to ensure that custom modules are automatically rebuilt whenever the kernel version changes.

2.4.4 Building a custom kernel

In some cases, it might be necessary or desirable for the system administrator to build a custom kernel. Building a custom kernel gives fine-grained control over many of the kernel configurations and can be used to achieve better performance or a smaller footprint.

The process to build a custom kernel is explained on the Raspberry Pi web site. For this, you will need the complete kernel sources. Again the kernel documentation is a great source for additional information, have a look at Documentation/kbuild/kconfig.txt, Documentation/kbuild/kbuild.txt, and Documentation/kbuild/makefiles.txt.

If you compile the Linux kernel on a Raspberry Pi device, it will take several hours—even with parallel compilation threads enabled. On the other hand, cross-compiling the kernel on a modern x86-64 PC only takes a few minutes at most.

2.5 Administrator and programmer view of the key chapters

From a systems programmer or administrator perspective, Linux is a POSIX-compliant system. POSIX (the Portable Operating System Interface) is a family of IEEE standards aimed at maintaining compatibility between operating systems. POSIX defines the application programming interface (API) used by programs to interact with the operating system. In practice, the standards are maintained by The Open Group, the certifying body for the UNIX trademark, which publishes the Single UNIX Specification, an extension of the IEEE POSIX standards (currently at version 4). The key chapters in this book discuss both the general (non-Linux specific) concepts and theory as well as the POSIX-compliant Linux implementations.

2.5.1 Process management

Linux administrators and programmers need to be familiar with processes, what they are, and how they are managed by the kernel. Chapter 4 ‘Process management’ introduces the process abstraction. We outline the state that needs to be encapsulated. We walk through the typical lifecycle of a process from forking to termination. We review the typical operations that will be performed on a process.

2.5.2 Process scheduling

Scheduling of processes and threads has a huge impact on system performance, and therefore Linux administrators and programmers need a good understanding of scheduling in general and the scheduling capabilities of the Linux kernel in particular. It is important to understand how to manage process priorities, per-process and per-user resources, and how to make efficient use of the scheduler. Chapter 5 'Process scheduling,' discusses how the OS schedules processes on a processor. This includes the rationale for scheduling, the concept of context switching, and an overview of scheduling policies (FCFS, priority, ...) and scheduler architectures (FIFO, multilevel feedback queues, priorities, ...). The Linux scheduler is studied in detail, with particular attention to the Completely Fair Scheduler but also discussing soft and hard real-time scheduling in the Linux kernel.

2.5.3 Memory management

While memory itself is remarkably straightforward, OS architects have built lots of abstraction layers on top. Principally, these abstractions serve to improve performance and/or programmability. For both the administrator and the programmer, it is important to have a good understanding of how the memory system works and what its performance trade-offs are. This is tightly connected with concepts such as virtual memory, paging, swap space, etc. The programmer also needs to understand how memory is allocated and what the memory protection mechanisms are. All this is covered in Chapter 6, 'Memory Management.' We briefly review caches (in hardware and software) to improve access speed. We go into detail about virtual memory to improve the management of physical memory resource. We will provide highly graphical descriptions of address translation, paging, page tables, page faults, swapping, etc. We explore standard schemes for page replacement, copy-on-write, etc. We will examine concrete examples in Arm architecture and Linux OS.

2.5.4 Concurrency and parallelism

Concurrency and parallelism are more important for the programmer than the administrator, as concurrency is needed for responsive, interactive applications and parallelism for performance. From an administrator perspective, it is important to understand the impact of the use of multiple hardware threads by a single application. In Chapter 7, 'Concurrency and parallelism,' we discuss how the OS supports concurrency and how the OS can assist in exploiting hardware parallelism. We define concurrency and parallelism and discuss how they relate to threads and processes. We discuss the key issue of resource sharing, covering locking, semaphores, deadlock, and livelock. We look at OS support for concurrent and parallel programming via POSIX threads and present an overview of practical parallel programming techniques such as OpenMP, MPI, and OpenCL.

2.5.5 Input/output

Chapter 8 'Input/output,' presents the OS abstraction of an I/O device. We review device interfacing, covering topics like polling, interrupts, and DMA, and we discuss memory-mapped I/O. We investigate a range of device types, to highlight their diverse features and behavior. We cover hardware registers, memory mapping, and coprocessors. Furthermore, we examine the ways in which devices are exposed to programmers, and we review the structure of a typical device driver.

2.5.6 Persistent storage

Because Linux, as a Unix-like operating system, is designed around the file system abstraction, a good understanding files and file systems is important for the administrator, in particular of concepts such as mounting, formatting, checking, permissions and links. Chapter 9 ‘Persistent storage’ focuses on file systems. We discuss the use cases and explain how the raw hardware (block- and sector-based storage, etc.) is abstracted at the OS level. We talk about mapping high-level concepts like files, directories, permissions, etc. down to physical entities. We review allocation, space management, and recovery from failure. We present a case study of a Linux file system. We also discuss Windows-style FAT, since this is how USB bulk storage operates.

2.5.7 Networking

Networking is important at many levels: when booting, the firmware deals with the MAC layer, the kernel starts the networking subsystem (arp, dhcp and init starts daemons; then user processes start clients and/or daemons. The administrator may need to tune the TCP/IP stack and configure the kernel firewall. Most applications today require network access. As the Linux networking stack is handled by the kernel, the programmer needs to understand how Linux manages networking as well as the basic APIs.

Chapter 10 ‘Networking’ introduces networking from a Linux kernel perspective: why is networking treated differently from other types of IO, what are the OS requirements to support the network stack, etc.. We introduce socket programming with a focus of the role the OS plays (e.g.~buffering, file abstraction, supporting multiple clients, ...).

2.6 Summary

In this chapter, we have introduced several basic operating system concepts and illustrated how they relate to Linux. We have discussed what happens when a Linux system (in particular on the Raspberry Pi) boots and initializes. We have introduced kernel modules and kernel compilation. Finally, we have presented a roadmap of the key chapters in the book, highlighting their relevance to Linux system administrators and programmers.

2.7 Exercises and questions

2.7.1 Installing Raspbian on the Raspberry Pi 3

1. Following the instructions on raspberrypi.org, download the latest Raspbian disk image and install it either as a Virtual Machine using qemu or on an actual Raspberry Pi 3 device.
2. Boot the device or VM and ping it (as explained on the Raspberry Pi web site).

2.7.2 Setting up SSH under Raspbian

1. Configure your Raspberry Pi to start an ssh server when it boots (this is not discussed in the text).
2. Log in via ssh and create a dedicated user account.
3. Forbid access via ssh to any account except this dedicated one.

2.7.3 Writing a kernel module

1. Write a simple kernel module that prints some information to the kernel log file when loaded, as explained in the text.
2. Write a more involved kernel module that creates a character device in /dev.

2.7.4 Booting Linux on the Raspberry Pi

1. Describe the stages of the Linux boot process for the Raspberry Pi.
2. Explain the purpose of the `initramfs` RAM disk.

2.7.5 Initialization

1. After the kernel has booted it launches the first process, called *init*. What does this process do?
2. Are there specific requirements on the *init* process?

2.7.6 Login

1. Which are the programs involved in logging in to the system via a terminal?
2. Explain the login process and how the kernel is involved.

2.7.7 Administration

1. Explain the role of the /dev and /proc file systems in system administration.
2. Explain the Linux approach to permissions: who are the participants, what are the restrictions, what is the role of the kernel?
3. As a system administrator, which tools do you have at your disposal to control and limit the behavior of your user processes in terms of CPU and memory utilization.