Linux Essentials

The LPI Introductory Programme

Linup Front GmbH · Postfach 10 01 21 · 64201 Darmstadt · Germany
Telefon +49(0) 6151 9067 0 · Telefax +49(0) 6151 9067 299 · www.linupfront.de

Linup Front GmbH is a leading supplier of high-quality training materials for Linux and Open Source topics, at reasonable prices – for self-study, school, higher and continuing education and professional training.
Please visit http://shop.linupfront.com/ for details and pricing, or contact us with questions or suggestions.

All representations and information contained in this document have been compiled to the best of our knowledge and carefully tested. However, mistakes cannot be ruled out completely. The authors and Linup Front GmbH assume no responsibility or liability resulting in any way from the use of this material or parts of it or from any violation of the rights of third parties.
Reproduction of trade marks, service marks and similar monikers in this document, even if not specially marked, does not imply the stipulation that these may be freely usable according to trade mark protection laws. All trade marks are used without a warranty of free usability and may be registered trade marks of third parties.

This document is published under the “Creative Commons-BY-NC-ND 3.0 Unported” licence. You may copy and distribute it and make it publically available as long as the following conditions are met:

Attribution You must make clear that this document is a product of Linup Front GmbH.

No commercial use You may not use this document for commercial purposes (contact us if you want to use this document commercially).

No derivatives You may not alter, transform, or build upon this document (contact us if necessary).

The full legal license grant can be found at http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode

Authors: Tobias Elsner, Thomas Erker, Anselm Lingnau
Technical Editor: Anselm Lingnau ⟨*anselm.lingnau@linupfront.de*⟩
Typeset in Palatino, Optima and DejaVu Sans Mono

Register this manual on-line via the QR code on the left or http://shop.linupfront.com/register/hnINuUQTazcf7HFRIM80of/ for updates and interesting special offers.

Contents

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

List of Tables

Copyright © 2012 Linup Front GmbH

List of Figures

Copyright © 2012 Linup Front GmbH

Preface

Linux Essentials is a new certification by the *Linux Professional Institute* (LPI) which is aimed especially at schools and universities in order to introduce children and young adults to Linux. The *Linux Essentials* certificate is slated to define the basic knowledge necessary to use a Linux computer productively, and through a corresponding education programme aid young people and adults new to the open source community in understanding Linux and open-source software in the context of the ITC industry. See appendix C for more informaton about the *Linux Essentials* certificate.

With this training manual, Linup Front GmbH introduces the first comprehensive documentation for *Linux Essentials* exam preparation. The manual presents the requisite knowledge extensively and with many practical examples and thus provides candidates, but also Linux newcomers in general, with a solid foundation for using and understanding the free Linux operating system as well as for attaining in-depth knowledge about running and administering Linux. In addition to a detailed introduction to the background of Linux and free/open-source software, we explain the most important Linux concepts and tools such as the shell, how to handle files and scripts, and the file system structure. Insights into system administration, user and permission management and Linux as a networking client round off the presentation.

Based on the content of this training manual, *Linux Essentials* alumni are well-prepared to pursue further certifications including the LPI's LPIC programme as well as vendor-specific certificates like those from Red Hat or Novell/SUSE.

The training manual is particularly suitable for a *Linux Essentials* preparation class at general-education or vocational schools, academies, or universities, but by virtue of its detailed approach and numerous exercises with sample solutions can also be used for self-study.

This courseware package is designed to support the training course as efficiently as possible, by presenting the material in a dense, extensive format for reading along, revision or preparation. The material is divided in self-contained chapters detailing a part of the curriculum; a chapter's goals and prerequisites are summarized clearly at its beginning, while at the end there is a summary and (where appropriate) pointers to additional literature or web pages with further information.

chapters
goals
prerequisites

💡 Additional material or background information is marked by the "lightbulb" icon at the beginning of a paragraph. Occasionally these paragraphs make use of concepts that are really explained only later in the courseware, in order to establish a broader context of the material just introduced; these "lightbulb" paragraphs may be fully understandable only when the courseware package is perused for a second time after the actual course.

⚠ Paragraphs with the "caution sign" direct your attention to possible problems or issues requiring particular care. Watch out for the dangerous bends!

exercises

✎ Most chapters also contain exercises, which are marked with a "pencil" icon at the beginning of each paragraph. The exercises are numbered, and sample solutions for the most important ones are given at the end of the course-

Copyright © 2012 Linup Front GmbH

ware package. Each exercise features a level of difficulty in brackets. Exercises marked with an exclamation point (“!”) are especially recommended.

Excerpts from configuration files, command examples and examples of computer output appear in `typewriter type`. In multiline dialogs between the user and the computer, user input is given in **`bold typewriter type`** in order to avoid misunderstandings. The “◁◁◁◁◁” symbol appears where part of a command’s output had to be omitted. Occasionally, additional line breaks had to be added to make things fit; these appear as “▷ ◁”. When command syntax is discussed, words enclosed in angle brackets (“⟨*Word*⟩”) denote “variables” that can assume different values; material in brackets (“[`-f` ⟨*file*⟩]”) is optional. Alternatives are separated using a vertical bar (“`-a`|`-b`”).

Important concepts
definitions

Important concepts are emphasized using “marginal notes” so they can be easily located; **definitions** of important terms appear in bold type in the text as well as in the margin.

References to the literature and to interesting web pages appear as “[GPL91]” in the text and are cross-referenced in detail at the end of each chapter.

We endeavour to provide courseware that is as up-to-date, complete and error-free as possible. In spite of this, problems or inaccuracies may creep in. If you notice something that you think could be improved, please do let us know, e.g., by sending e-mail to

```
courseware@linupfront.de
```

(For simplicity, please quote the title of the courseware package, the revision ID on the back of the title page and the page number(s) in question.) We also welcome contact by telephone, telefax or “snail mail”. Thank you very much!

1

Computers, Software and Operating Systems

Contents

Goals

- Obtaining basic computer hardware knowledge
- Being aware of different operating systems and assessing their commonalities and differences

Prerequisites

- Basic computing knowledge is useful

1.1 What Is A Computer, Anyway?

Before we get into the details of what a computer is, here are a few quotes from notable people within the computing community:

> “Originally one thought that if there were a half dozen large computers in [the United States], hidden away in research laboratories, this would take care of all requirements we had throughout the country.” *Howard H. Aiken, 1952*

Early computers

Howard Aiken was a computing pioneer and the designer of IBM’s first computer, the “Harvard Mark I”. The first computers in a modern sense were built during World War II to assist with decrypting secret messages or doing difficult calculations, and they were big, complicated and error-prone devices – the electronic components such as transistors or integrated circuits which today’s computers consist of hadn’t been invented yet. What did come to light during this time and the years immediately after the war were a number of basic assumptions that had to hold for a device to be considered a “computer”:

- A computer processes *data* according to a sequence of *automatically* executed instructions, a *program*.
- Programs must allow for *conditional execution* and *loops*.
- It must be possible to change or replace the program that a computer executes.

For example, many technical devices – from television sets and digital cameras to washing machines or cars – today contain programmed control units, almost small computers. Even so, we don’t consider these devices “computers”, because they only execute fixed, unchangeable programs. Conversely, a pocket calculator can be used to “process data”, but – at least as long as it isn’t a more expensive “programmable calculator” – that doesn’t happen automatically; a human being must tap the keys.

In the early 1950s, computers were highly specialised devices that one would – exactly as Aiken stipulated – expect to see mostly within research institutions. Science-fiction films of the time display the halls, replete with rows of cupboards containing mysterious spinning reels. Within the space of not quite 70 years, this image has changed dramatically[1].

> “There is no reason for anyone to have a computer in their home.” *Ken Olsen, 1977*

“Small” computers in the 1970s

Ken Olsen was the CEO of another computer manufacturer, Digital Equipment Corporation (DEC), which spearheaded the development of “small” computers in the 1970s[2] – where “small” at the time was understood as meaning something like “does not need a machine hall with air conditioning and its own power plant and costs less than a million dollars”; advances in hardware technology allowed this to change, towards the end of the 1970s, to something like “can be bodily lifted by two people”.

DEC is important to the Linux community because Unix – the operating system that inspried Linus Torvalds to start Linux some twenty years later – was first developed on DEC PDP-8 and PDP-11 computers.

home computers

The 1970s also saw the advent of the first “home computers”. These cannot be compared with today’s PCs – one had to solder them together on one’s own (which

[1] The classic “quote” in this context is usually ascribed to Thomas J. Watson, the CEO if IBM, who is thought to have said, in 1943, something along the lines of “There is a world market for about five computers”. Unfortunately this has never been verified. And if he did actually claim this in 1943, he would have been right at least for the next ten years or so.

[2] DEC was acquired by Compaq in 1998, and Compaq by Hewlett-Packard in 2002.

Copyright © 2012 Linup Front GmbH

would be physically impossible today), and they rarely featured a reasonable keyboard and seldom if ever a decent display. They were for the most part a tinkerer's pastime, much like an electric train set, because in all honesty they weren't really useful for much at all. Even so, they were "computers" in the sense of our earlier definition, because they were freely programmable – even though the programs had to be laboriously keyed in or (with luck) loaded from audio cassette tape. Still they weren't taken completely seriously, and Ken Olsen's quote has accordingly often been misconstrued: He had nothing whatsoever against small computers (he was in the business of selling them, after all). What he didn't conceive of was the idea of having one's complete household (heating, lights, entertainment and so on) controlled by a computer – an idea that was quite hypothetical at the time but today seems fairly feasible and perhaps no longer as absurd.

IBM PC

Only during the late 1970s and 1980s, "home computers" mutated from kits to ready-to-use devices (names like "Apple II" or "Commodore 64" may still be familiar to the older members of our audience) and started appearing in offices, too. The first IBM PC was introduced in 1981, and Apple marketed the first "Macintosh" in 1984. The rest, as they say, is history – but one should not forget that the world of computers does not consist of PCs and Macs only. The giant, hall-filling computers of yore are still around – even though they tend to get rarer and often really consist of large groups of PCs that are quite closely related to the PCs on our tables and which cooperate. However, the principle hasn't changed from Howard Aiken's time: Computers are still devices that automatically process data according to changeable programs which may contain conditions and loops. And things are likely to stay that way.

Exercises

1.1 [1] What was the first computer you used? What type of processor did it contain, how much RAM, and how big was the hard disk (if there was one – if not, how was data stored permanently)?

1.2 Components Of A Computer

Let's take the opportunity of casting a glance at the "innards" of a computer (or, more precisely, an "IBM-compatible" PC) and the components we are likely to find there:

Processor The processor (or "CPU", for "central processing unit") is the core of the computer: Here is where the automatic program-controlled data processing takes place that actually makes it a computer. Today's processors usually contain several "cores", which means that the major components of the processor exist multiple times and can operate independently, which in principle increases the computer's processing speed and thereby its performance – and particularly fast computers often have more than one processor. PCs normally contain processors by Intel or AMD (which may differ in detail but can execute the same programs). Tablets and smartphones generally use ARM processors, which aren't quite as powerful but much more energy-efficient. Intel and AMD processors cannot directly execute programs prepared for ARM processors and vice-versa.

RAM A computer's working memory is called "RAM" (or "random-access memory", where "random" means "arbitrary" rather than "haphazard"). This stores not only the data being processed, but also the program code being executed.

This is an ingenuous trick going back to the computing pioneer John von Neumann, a contemporary of Howard Aiken. It implies that there is no longer a difference between code and data – this means programs

Copyright © 2012 Linup Front GmbH

can manipulate code just as well as addresses or kitchen recipes. (In the old days, one would "program" by plugging and unplugging leads on the outside of the computer, or programs were punched on paper tape or cards and could not be changed straightforwardly.)

Today's computers normally feature 1 gibibyte of RAM or more. 1 gibibyte is 2^{30}, or $1,073,741,824$ bytes[3] – really an inconceivably large number. By way of comparison: *Harry Potter and the Deathly Hallows* contains approximately 600 pages of up to 1,700 letters, spaces, and punctuation characters – perhaps a million characters. Hence, one gibibyte corresponds to about 1,000 *Harry Potter* tomes, at somewhat more than a pound per book that is already a van full of them, and if you're not just interested in the exploits of the young wizard, 1,000 books is an impressive library.

Graphics card Not so long ago people were happy if their computer could control an electric typewriter to produce its output. The old home computers were connected to television sets, producing images that could often only be called atrocious. Today, on the other hand, even simple "smartphones" feature quite impressive graphics, and common PCs contain graphics hardware that would have cost the equivalent of an expensive sports car or small house in the 1990s[4]. Today's watchword is "3D acceleration", which doesn't mean that the display actually works in 3D (although even that is slowly getting fashionable) but that *processing* the graphics inside the computer does not just involve left, right, top and bottom – the directions visible on a computer monitor – but also front and back, and that in quite a literal sense: For photorealistic games it is quite essential whether a monster lurks in front of or behind a wall, hence whether it is visible or not, and one of the goals of modern graphic cards is to relieve the computer's CPU of such decisions in order to free it up for other things. Contemporary graphics cards contain their own processors, which can often perform calculations much faster than the computer's own CPU but are not as generally useful.

Many computers don't even contain a separate graphics card because their graphics hardware is part of the CPU. This makes the computer smaller, cheaper, quieter and more energy-efficient, but its graphics performance will also take somewhat of a hit – which may not be an actual problem unless you are keen on playing the newest games.

Motherboard The motherboard is the (usually) rectangular, laminated piece of plastic that the computer's CPU, RAM, and graphics card are affixed to – together with many other components that a computer requires, such as connectors for hard disks, printers, a keyboard and mouse, or network cables, and the electronics necessary to control these connectors. Motherboards for computers come in all sorts of sizes and colours[5] – for small, quiet computers that can act as video recorders in the living room or big servers that need a lot of space for RAM and several processors.

Power supply A computer needs electricity to work – how much electricity depends on exactly which components it contains. The power supply is used to convert the 240 V AC mains supply into the various low DC voltages that the electronics inside the computer require. It must be selected such that it can furnish enough power for all the components (fast graphics cards are usually the number-one guzzlers) while not being overdimensioned so that it can still operate efficiently.

[3] People will commonly call this a "gigabyte", but a "gigabyte" is really 10^9 bytes, i. e., nearly 7 percent less.

[4] All thanks, incidentally, to the unflagging popularity of awesome computer games. Whoever believes that video games are of no earthly use should take a minute to consider this.

[5] Really! Even though one shouldn't select one's motherboard according to colour.

Copyright © 2012 Linup Front GmbH

Most of the electricity that the power supply pumps into the computer will sooner or later end up as heat, which is why good cooling is very important. For simplicty, most computers contain one or more fans to blow fresh air onto the expensive electronics, or to remove hot air from the case. With appropriate care it is possible to build computers that do not require fans, which makes them very quiet, but such computers are comparatively expensive and usually not quite as fast (since, with processors and graphics cards, “fast” usually means “hot”).

Hard disks While a computer’s RAM is used for the data currently being processed (documents, spreadsheets, web pages, programs being developed, music, videos, …—and of course the programs working on the data), data not currently in use are stored on a hard disk. The main reason for this is that hard disks can store much more data than common computers’ RAM—the capacity of modern hard disks is measured in tebibytes (1 TiB = 2^{40} Byte), so they exceed typical RAM capacities by a factor of 100–1000.

We pay for this increase in space with a decrease in retrieval times—RAM access times are measured in nanoseconds while those to data on (magnetic) hard disks are measured in milliseconds. This is a mere 6 orders of magnitude—the difference between a metre and 1,000 kilometres.

Traditionally, hard disks consist of rotating platters coated with a magnetisable material. Read/write heads can magnetise this material in different places and re-read the data thus stored later on. The platters rotate at 4,500 to 15,000 RPM, and the difference between the read/write head and the platter is extremely minute (up to 3 nm). This means that hard disks are quite sensitive to disruption and falls, because if the read/write head comes into contact with the platter while the disk is running—the dreaded “head crash”—the disk is destroyed.

Newfangled hard disks for mobile computers have acceleration sensors which can figure out that the computer is falling, to try and shut down the hard disk in order to prevent damage.

The newest fashion is SSDs or “solid-state disks”, which instead of magnetised platters use “flash memory” for storage—a type of RAM which can maintain its content even without electricity. SSDs are faster than magnetic hard disks, but also considerably more expensive per gigabyte of storage. However, they contain no moving parts, are impervious to being shoved or dropped, and save energy compared to conventional hard disks, which makes them interesting for portable computers.

SSDs are also reputed to “wear out” since the flash storage spaces (called “cells”) are only rated for a certain number of write operations. Measurements have shown that this does not lead to problems in practice.

There are various methods of connecting a hard disk (magnetic or SSD) to a computer. Currently most common is “serial ATA” (SATA), older computers use “parallel ATA”, also called “IDE”. Servers also use SCSI or SAS (“serially attached SCSI”) disks. For external disks, one uses USB or eSATA (a variant of SATA with sturdier connectors).

Incidentally: The difference between gigabytes and gibibytes (or terabytes and tebibytes) is most notable with hard disks. For example, you buy a “100 GB drive”, connect it to your computer and, shock horror, realise that your computer only shows you 93 “GB” of free space on the new disk! However, your drive is not damaged (lucky you) –

the disk drive manufacturer only uses (quite correctly) "gigabytes", i. e., billions of bytes, while your computer probably (if inaccurately) calculates the free space in units of "gibibytes" or 2^{30} bytes.

Optical drives Besides hard drives, PCs usually support optical drives that can read, and often also write, media such as CD-ROMs, DVDs or Blu-ray disks. (Mobile devices sometimes have no room physically for an optical drive, which does not mean such drives can't be connected externally.) Optical media—the name derives from the fact that the information on there is accessed by means of a laser—are mostly used for the distribution of software and "content" (music or films), and their importance is waning as more and more companies rely on the Internet as a distribution medium.

In former times one also considered optical media for backup copies, but today this is no longer realistic—a CD-ROM can hold up to approximately 900 MiB of data and a DVD up to 9 GiB or so, thus for a full backup of a 1 TiB hard disk you would require 1000 CD-size or 100 DVD-size media, and constantly swapping them in and out would also be a hassle. (Even Blu-ray discs can only fit 50 GiB or so, and drives that can *write* to Blu-ray discs are still fairly expensive.)

Display You can still see it in old movies: the green sheen of the computer screen. In reality, green displays have all but disappeared, colour is in fashion, and new displays are no longer massive hulks like the CRTs (cathode-ray tubes) we used to have, but are slim, elegant monitors based on liquid crystals (LCD, "liquid-crystal display"). LCDs don't confine themselves to the advantage of taking up less space on a desk, but also neither flicker nor bother the user with possibly harmful radiation—a win-win situation. There are a few disadvantages such as colour changes when you look at the screen at a too-acute angle, and cheaper devices may deliver a blotchy picture because the backlight is not uniform.

With CRTs one used to take care to not let them stand around unused showing the same picture for long periods of time, because the picture could "burn in" and appear as a permanent blurry backdrop. Accordingly one used a "screen saver", which after a certain amount of idle time would replace the content of the screen by a more or less cute animation to avoid burn-in (the classic was an aquarium with fish and other aquatic fauna). LCDs no longer suffer from the burn-in problem, but screen savers are still sticking around for decorative value.

resolution

LCDs are available in all sizes from "smartphone" to wall-size large screens; their most important property is the resolution, which for PC displays usually ranges between 1366 × 768 (horizontally × vertically) and 1920 × 1080 "pixels". (Lower and higher resolutions are possible, but do not necessarily make economic or visual sense.) Many computers support more than one screen in order to enlarge the working space.

Also usual today is an aspect ration of 16 : 9, which corresponds to high-definition television—actually a silly development, since most computers aren't even used for watching television, and a taller but narrower display (such as the formerly-common 4 : 3 format) is better suited most of the more frequently-used applications like word processing or spreadsheet calculations.

Other peripherals Of course you can connect many more devices to a computer besides the ones we mentioned: printers, scanners, cameras, television receivers, modems, robotic arms, small missile launchers to annoy your cubicle neighbours, and so on. The list is virtually endless, and we cannot discuss every class of device separately here. But we can still make a few observations:

Copyright © 2012 Linup Front GmbH

- One commendable trend, for example, is the simplification of connections. While almost every class of device used to have their own interface (parallel interfaces for printers, serial interfaces for modems, "PS/2" interfaces for keyboards and mice, SCSI for scanners, …), today most devices use USB (universal serial bus), a relatively foolproof and reasonably fast method which also supports "hot-plugging" connections while the computer is running.
- Another trend is that towards more "intelligence" in the peripherals themselves: Formerly, even expensive printers were fairly stupid devices at an IQ level of electric typewriters, and programmers had to very carefully send exactly the right control codes to the printer to produce the desired output. Today, printers (at least good printers) are really computers in their own right supporting their own programming languages that make printing much less of a hassle for programmers. The same applies in a similar fashion to many other periperals.

  Of course there are still very stupid printers (especially at lower price points) which leave preparing the output to the computer itself. However, these still make a *programmer's* life as easy as their more expensive relations.

Exercises

1.2 [2] Open your computer (possibly under the direction of your teacher, instructor, or legal guardian—and don't forget switching it off and unplugging it first!) and identify the most important components such as the CPU, RAM, motherboard, graphics card, power supply, and hard disk. Which components of your computer did we not talk about here?

1.3 Software

Just as important as a computer's "hardware", i. e., the technical components it consists of[6], is its "software"—the programs it is running. This can very roughly be divided into three categories:

firmware

- The **firmware** is stored on the computer's motherboard and can only be changed or replaced inconveniently if at all. It is used to put the computer into a defined state after switching it on. Often there is a way of invoking a setup mode that allows you to set the clock and enable or disable certain properties of the motherboard.

  On PCs, the firmware is called "BIOS" (Basic Input/Output System) or, on newer systems, "EFI" (Extensible Firmware Interface).

  Some motherboards include a small Linux system that purportedly boots more quickly than Linux and which is supposed to be used to surf the Internet or watch a DVD without having to boot into Windows. Whether this is actually worth the trouble is up to debate.

operating system

- The **operating system** makes the computer into a usable device: It manages the computer's resources such as the RAM, the hard disks, the processing time on the CPU(s) available to individual programs, and the access to other peripherals. It allows starting and stopping programs and enforces a separation between several users of the computer. Besides, it enables—on an elementary level—the participation of the computer in a local area network or the Internet. The operating system frequently furnishes a graphical user

[6] Definition of hardware: "The parts of a computer that can be kicked" (Jeff Pesis)

interface and thus determines how the computer “looks and feels” to its users.

When you buy a new computer it is usually delivered with a pre-installed operating system: PCs with Microsoft Windows, Macs with OS X, smartphones often with Android (a Linux derivative). The operating system, though, is not tied as closely to a computer as the firmware, but can in many cases be replaced by a different one—for example, you can install Linux on most PCs and Macs.

> 💡 Or you install Linux *in addition to* an existing operating system—usually not a problem either.

User-level programs

applications

utilities

- **User-level programs** allow you to do something useful, such as write documents, draw or manipulate pictures, compose music, play games, surf the Internet or develop new software. Such programs are also called **applications**. Additionally, there are often **utilities** that the operating system provides in order to allow you—or a designated “system administrator”—to make changes to the computer’s configuration and so on. Servers, in turn, often support software that provides services to other computers, such as web, mail or database servers.

1.4 The Most Important Operating Systems

1.4.1 Windows And OS X

When talking about computer operating systems, most people will automatically think of Microsoft Windows[7]. This is due to the fact that nowadays most PCs are sold with Windows preinstalled—really not a bad thing in itself, since their owners can get them up and running without having to take the trouble to install an operating system first, but, on the other hand, a problem because it makes life hard for alternative operating sysetms such as Linux.

> 💡 In fact it is not at all straightforward to buy a computer without a preinstalled copy of Windows—for example, because you want to use it exclusively with Linux—, except when building one from scratch. Theoretically you are supposed to be able to get a refund for an unused preinstalled copy of Windows from the computer’s manufacturer, but we know of nobody who actually managed to obtain any money.

Windows NT

Today’s Windows is a descendant of “Windows NT”, which was Microsoft’s attempt to establish an operating system that was up to the standards of the time in the 1990s (earlier versions such as “Windows 95” were graphical extensions to the then-current Microsoft operating system, MS-DOS, and fairly primitive even by the standards of the day). Decency forbids us a critical appreciation of Windows here; let it suffice to say that it does approximately what one would expect from an operating system, provides a graphical user interface and supports most peripheral devices (support for more is provided by the individual device manufacturers).

Mac OS

Apple’s “Macintosh” was launched in 1984 and has since been using an operating system called “Mac OS”. Over the years, Apple made various changes to the platform (today’s Macs are technically about the same as Windows PCs) and operating system, some of them quite radical. Up to and including version 9, MacOS was a fairly flimsy artefact which, for example, only provided rudimentary support for running several programs at the same time. The current “Mac OS X”—the “X” is a Roman 10, not the letter “X”—is based on an infrastructure related to BSD Unix and is not unlike Linux in many ways.

[7] Many people will not even be aware that there are other operating systems at all.

Since February, 2012, the official name for the Macintosh operating system is "OS X" rather than "Mac OS X". If we let slip a "Mac OS" every so often, you know what we really mean.

Differences

The big difference between Windows and OS X is that OS X is sold exclusively with Apple computers and will not run on "normal" PCs. This makes it much more straightforward for Apple to provide a system that is obviously very homogenous. Windows, on the other hand, must run on all sorts of PCs and support a much wider array of hardware components that can occur in completely unforeseen combinations. Hence, Windows users have to contend with incompatibilities that are sometimes difficult or even impossible to sort out. On the other hand, there *is* a much greater selection of hardware for Windows-based computers, and prices are, on the whole, less exorbitant.

Similarities

Windows and OS X are similar in that they are both "*proprietary*" software: Users are forced to accept what Microsoft or Apple put in front of them, and they cannot examine the actual implementation of the system, let alone make changes to it. They are bound to the upgrade schedule of the system, and if the manufacturer removes something or replaces it by something else, they need to adapt to that.

There is one difference here, though: Apple is essentially a hardware manufacturer and only provides OS X to give people an incentive to buy Macs (this is why OS X isn't available for non-Macs). Microsoft, on the other hand, does not build computers, and instead makes its money selling software such as Windows which runs on arbitrary PCs. Therefore, an operating system like Linux is much more of a threat to Microsoft than to Apple—most of the people who buy an Apple computer do this because they want an *Apple* computer (the complete package), not because they are especially interested in OS X. The PC as a platform, however, is being encroached upon by tablets and other new-fangled types of computer that don't run Windows, and that puts Microsoft under extreme pressure. Apple could easily survive selling just iPhones and iPads instead of Macs—Microsoft without Windows would probably go bankrupt fairly soon in spite of having loads of money in their bank account.[8]

1.4.2 Linux

Linux is an operating system that was first started out of curiosity by Linus Torvalds, but then took on a life of its own—in the meantime, hundreds of developers (not just students and hobbyists, but also professionals at companies such as IBM, Red Hat, or Oracle) are developing it further.

Linux was inspired by Unix, an operating system developed in the 1970s at AT&T Bell Laboratories and geared towards "small" computers (see above for the meaning of "small" in this context). Unix soon became the preferred system for research and technology. For the most part, Linux uses the same concepts and basic ideas as Unix, and it is easy to get Unix software to run on Linux, but Linux itself does not contain Unix code, but is an independent project.

Unlike Windows and OS X, Linux isn't backed by an individual company whose economic success hinges on the success of Linux. Linux is "freely available" and can be used by anyone—even commercially—who subscribes to the rules of the game (as outlined in the next chapter). This together with the fact that by now Linux no longer runs just on PCs, but in substantially identical form on platforms ranging from telephones (the most popular smartphone operating

[8] Actually, the real battlefield isn't Windows but Office—most people don't buy Windows because they are big fans of Windows, but because it is the only operating system for arbitrary (i. e., cheap) PCs that runs Office—, but the same consideration applies if you replace "Apple" by "Google". In point of fact, Office and (PC-based) Windows are the only products that actually make Microsoft an appreciable amount of money; everything else that Microsoft does (with the possible exception of the Xbox gaming console) they do at a loss.

system, Android, is a Linux offshoot) to the largest mainframes (the ten fastest computers in the world are all running Linux) makes Linux the most versatile operating system in the history of modern computing.

distributions

Strictly speaking "Linux" is just the operating system *kernel*, i. e., the program that handles the allocation of resources to applications and utilities. Since an operating system without applications isn't all that useful, one usually installs a Linux distribution, which is to say a package consisting of "Linux" proper and a selection of applications, utilities, documentation and other useful stuff. The nice thing is that, like Linux itself, most Linux distributions are "freely available" and hence available free of charge or at very low cost. This makes it possible to equip a computer with software whose equivalents for Windows or OS X would run into thousands of dollars, and you do not run the risk of falling foul of licensing restrictions just because you installed your Linux distribution on all your computers as well as Aunt Millie's and those of your buddies Susan and Bob.

There is more information on Linux and Linux distributions in Chapter 2.

1.4.3 More Differences And Similarities

graphical user interface

Actually, the three big operating systems—Linux, Windows, and OS X—differ only in detail in what they present to the users. All three offer a graphical user interface (GUI) which allows even casual users to manage their files through simple gestures like "drag and drop". Many popular applications are available for all three operating systems, so which one you are using at the end of the day becomes almost immaterial as long as you are spending most of your time inside the web browser, office package, or e-mail program. This is an advantage because it enables a "gradual" migration from one system to the other.

command line

Besides the graphical interface, all three systems also offer a way to use a "command line" to input textual commands which the system then executes. With Windows and OS X, this feature is mostly used by system administrators, while "normal" users tend to shun it—a question of culture. With Linux, on the other hand, the command line is much less ostracised, which may have to do with its descent from the scientific/technical Unix philosophy. As a matter of fact, many tasks are performed more conveniently and efficiently from the command line, especially with the powerful tools that Linux (and really also OS X) provide. As a budding Linux user, you do well to open up to the command line and learn about its strengths and weaknesses, just as you should learn about the strengths and weaknesses of the GUI. A combination of both will give you the greatest versatility.

Exercises

1.3 [1] If you have experience with a proprietary operating system like Windows or OS X: Which applications do you use most frequently? Which of them are "free software"?

1.5 Summary

Today's PCs, whether based on Linux, Windows, or OS X, have more similarities than differences—as far as their hardware, their basic concepts, and their use is concerned. Without doubt you can use any of the three to go about your daily work, and none of them is obviously and uncontestedly "the best".

However, this manual talks mostly about Linux, and we will use the rest of these pages to provide an introduction to that system that is as extensive as possible—explain its use, highlight its strengths and, where necessary, point out its weaknesses. By now, Linux is a serious alternative to the other two systems and surpasses them in various aspects, in some of them widely. We are glad to see that you are prepared to get involved, and we wish you a lot of fun learning, practising, and using Linux and—if you are interested in the LPI's *Linux Essentials* certification—the best of success in the exam!

Summary

- Computers are devices that process data according to an automatically executed, changeable program allowing conditional execution and loops.
- The most important components of a PC include the processor, RAM, graphics card, motherboard, hard disks et cetera.
- The software on a computer can be divided into firmware, the operating system, and user-level programs.
- The most popular PC operating system is Microsoft's Windows. Apple computers use a different operating system called OS X.
- Linux is an alternative operating system for PCs (and many other types of computer) which is not developed by a single company, but by a large number of volunteers.
- Linux distributions extend the Linux operating system kernel with applications and documentation to result in a system that is actually usable.

2

Linux and Free Software

Contents

Goals

- Knowing the basic principles of Linux and free software
- Being able to place the basic FOSS licenses
- Having heard of the most important free applications
- Having heard of the most important Linux distributions

Prerequisites

- Basic knowledge about computers and operating systems (Chapter 1)

2.1 Linux: A Success Story

In the summer of 1991, Linus Torvalds, who was 21 years old at the time, studied computer science at the technical university of Helsinki, Finland[1] At the time he owned a new 386 PC that he wanted to experiment with, and amused himself by writing a terminal emulator which ran on the raw hardware without an operating system, and which allowed him to access the university's Unix system. This program grew into the first Linux operating system kernel.

Unix was already about 20 years old at that point, but was the operating system of choice at universities and wherever research or development were done—the scientific "workstations" of the time almost exclusively ran on various versions of Unix.

Unix origins

Unix itself had started—almost like Linux—as a hobby-type project of Ken Thompson and Dennis Ritchie at Bell Laboratories, the American telecommunications giant AT&T's research arm. It very quickly mutated into quite a useful system, and since it was written, for the most part, in a high-level language (C), it could be ported reasonably quickly to other computing platforms than the original PDP-11. In addition, during the 1970s AT&T was forced by a consent decree to refrain from selling software, so Unix was "given away" at cost, without support—and since the system was small and fairly straightforward, it became a popular case study in the operating system seminars of most universities.

BSD

Towards the end of the 1970s, students of the University of Californa in Berkeley ported Unix to the VAX, the successor of the PDP-11, and introduced various improvements that began to be circulated as "BSD" (short for "Berkeley Software Distribution"). Various offshoots of BSD are still current today.

Minix

To develop the first versions of Linux, Linus made use of "Minix", a Unix-like operating system written for teaching purposes by Andrew S. Tanenbaum of the Free University of Amsterdam. Minix was deliberately kept simple, and it wasn't freely available, so did not represent a serious operating system—help was obviously needed![2]

On 25 August 1991, Linus announced his project to the public and invited the rest of the world to join in. At this point the system functioned as an alternative operating system kernel for Minix.

At that time the system didn't yet have a proper name. Linus called it "Freax" (as a portmanteau word from "freak" and "Unix"); he did briefly consider "Linux" but rejected this as too egotistical. When Linus' system was uploaded to the university's FTP server, Linus' colleague Ari Lemmke, who didn't like the name "Freax", took the liberty of renaming it to "Linux". Linus later approved of the change.

Linux generated considerable interest and many volunteers decided to collaborate. Linux 0.99, the first version licensed under the GPL (Section 2.2.3), appeared in December, 1992, and represented quite a grown-up operating system with complete, if simple, Unix functionality.

Linux 2.0

Linux 2.0 appeared in early 1996 and introduced some important new features such as support for multiprocessors and the ability to load kernel modules at runtime—an important innovation along the road to user-friendly Linux distributions.

Tux

Another new feature in Linux 2.0 was "Tux", the penguin, as the official

[1] Linus Torvalds is an ethnic Finn (he became an American citizen in September, 2010), but a member of the Swedish-speaking minority. This is why he has a reasonably pronounceable name.

[2] BSD wasn't freely available at the time, either; Linus once said that, had BSD been usable at the time, he would never have started Linux.

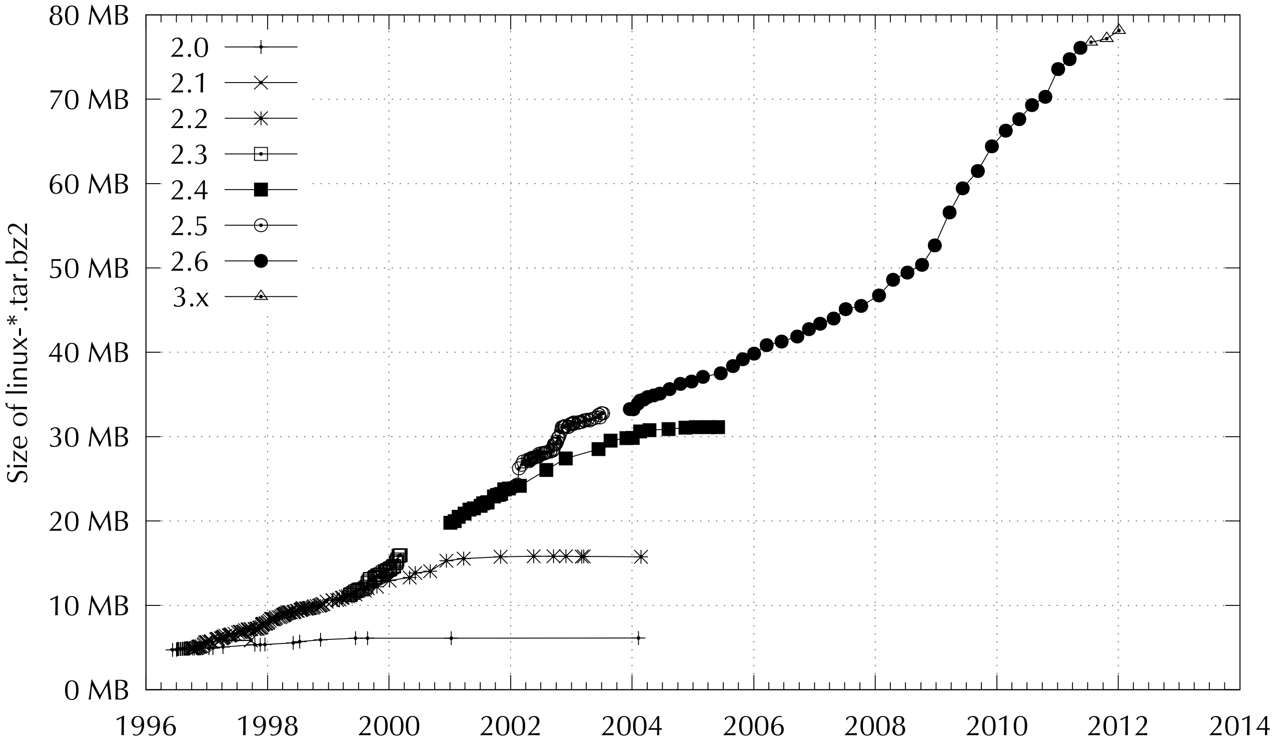


Figure 2.1: The evolution of Linux, measured by the size of `linux-*.tar.bz2`. Each marker corresponds with a Linux version. During the 15 years from Linux 2.0 to Linux 3.2, the size of the compressed Linux source code has increased by a factor of almost 16.

Linux mascot. Linus Torvalds had been set upon by a penguin in Australia, which had greatly impressed him. The iconic sitting penguin with its yellow feet and beak was drawn by Larry Ewing and made available to the community at large.

Linux 2.6 saw a reorganisation of the development process. While, earlier on, version numbers with an odd second component (such as "2.3") had been considered developer versions and those with an even second component (like "2.0") stable versions suitable for end users, the Linux developers resolved to keep developer and stable versions from diverging to the large extent previously seen. Starting with Linux 2.6, there is no longer a separate line of development kernels, but improvements are being introduced into the next version and are tested as extensively as possible before that version is officially released.

development process

This works approximately as follows: After Linux 2.6.37 is released, Linus collects proposed changes and improvements for the next kernel, includes them into his official version and publishes that as Linux 2.6.38-rc1 (for "release candidate 1"). This version is tested by various people, and any fixes or improvements are collected in Linux 2.6.38-rc2 and so on. Eventually the code looks stable enough to be officially released as "Linux 2.6.38", and the process then repeats with version 2.6.39.

In addition to Linus' official version there are Linux versions that are maintained by other developers. For example, there is the "staging tree" where new device drivers can "mature" until (after several rounds of improvements) they are considered good enough to be submitted to Linus for inclusion into his version. Once released, many Linux kernels receive fixes for a certain period of time, so there can be versions like 2.6.38.1, 2.6.38.2, ….

Linux 3.0

In July, 2011, Linus summarily declared the version being prepared, 2.6.40, "Linux 3.0"—purportedly to simplify the numbering, since there were no particularly notable improvements.

Nowadays release candidates are called 3.2-rc1 and so on, and the versions with fixes after the original release are called 3.1.1, 3.1.2, …

The "Linux" project is by no means finished today. Linux is constantly being extended and improved by hundreds of programmers throughout the world, who serve several millions of satisfied private and commercial users. Neither can it be said that the system is "only" being developed by students and other amateurs—many people working on the Linux kernel have important positions within the computer industry and number among the most respected professional developers around. It can be claimed with some justification that Linux is the operating system with the most versatile hardware support in existence, not only judging by the variety of platforms it runs on (including smartphones and giant mainframes), but also by the availability of hardware drivers for, e. g., the Intel PC platform. Linux also serves as a research and development vehicle for new operating system ideas in industry and academia; it is without doubt one of the most innovative operating systems available today.

Virtualisation

cloud computing

The versatility of Linux also makes it the operating system of choice for applications like virtualisation and "cloud computing". Virtualisation makes it possible to simulate, on a single actual ("physical") computer, several to many "virtual" computers which use their own operating system and look like real computers to programs running on them. This leads to a more efficient use of resources and to greater flexibility: The common virtualisation infrastructures make it possible to "migrate" virtual machines very quickly from one physical machine to another, and this lets you as the operator of such an infrastructure react very conveniently to load situations or malfunctions. Based on this, cloud computing is the idea of providing computing power "on demand" in order to allow companies to forego running large computing centres that are only used to a full extent during short periods of peak demand while mostly costing money otherwise. Cloud computing providers allow their customers to use virtual machines through the Internet, charging them based on actual use, and that can lead to sizeable savings compared to maintaining a "real" computing centre, especially when one considers that you as a customer would otherwise have to bear not just the initial construction outlay, but also the personnel, materiel and energy expenses involved in running a computing centre 24/7.

Exercises

2.1 [2] Search the Internet for the notorious discussion between Andrew S. Tanenbaum and Linus Torvalds, in which Tanenbaum says he would have failed Linus for producing something like Linux. What do you think?

2.2 [1] What is the version number of the oldest set of Linux kernel sources that you can locate?

2.2 Free Or Open Source?

2.2.1 Copyright And "Free Software"

During the Middle Ages, duplication of books and other writings was an expensive matter. One needed to find somebody who was able to write and had the required time on their hands—which made larger projects like the copying of bibles the domain of monasteries (the monks there did know how to write and had loads of time available). The invention of the movable-type printing press in the 16th century created a previously-unknown problem: Suddenly copying became considerably simpler and cheaper (at least as long as one had a printing press to hand), and industrious publishers used this to copy everything they considered worth selling. Authors had nearly no rights at the time; they could be glad if publishers paid them anything at all for printing their works. Widespread reprinting also caused the original printers of a work to feel cheated when others took over their publications without compensation. This led many of them to ask the government for special privileges in order to obtain the exclusive right to publish certain works. Government didn't object to this, since the rulers tried to keep some control on what was being printed within their spheres of authority. With time, these "privileges" together with other developments like the (laudable) idea to bestow on *authors* the right for remuneration, mutated into the modern concept of "copyright" (or, more generally, "author's rights").

Copyright essentially means that the creator of a work, i. e., the author of a book, painter of a picture, …, has the right of determining what happens to the work. As the author, they can bestow on (or, in practice, sell for money to) a publisher the right to print a book and put it into circulation; the publisher gets to earn money while the author does not have to bother with printing, publicity, shipment, etc., which serves both sides.

In addition to these "copyrights" there are also "moral rights" like the right to be identified as the creator of a work. It is often impossible to give these moral rights away.

During the 20th century (roughly), the concepts of copyright were put on an internationally accepted basis and extended to other types of work such as sound recordings and motion pictures.

The advent of computers and the Internet towards the end of the 20th century gravely changed the situation once more: While the purpose of copyright used to be to protect publishers from other publishers (private people were seldom in a position to duplicate books, records, or films on a commercially relevant scale), suddenly everybody who owned a computer was able to duplicate digital content (like software or books, music or movies in digital form) as often as desired and without loss of quality—a disaster for publishers, music, film or software companies, since their existing business models, which were based on the sale of physical artefacts such as books or CDs, were endangered. Since then the "content industry" has been lobbying for tougher copyright laws and higher penalties for "pirate copiers", as well as trying to hold copyright infringers liable for their infringements (with varying success).

The current term is "intellectual property", which includes not just copyrights, but also trademarks and patents. Patents are supposed to compensate the inventors of technical processes for documenting and publishing their inventions by giving them a time-limited exclusive right to exploit these inventions (e. g., by allowing others to make use of them for money). Trademarks ensure that nobody gets to exploit, without permission, the popularity of a brand by selling their own products under that name. For example, a trademark on "Coca-Cola" ensures that nobody who is able to concoct a brownish sugar brew gets to sell this as "Coca-Cola". The three types of "intellectual property" are related but distinct—patents are about ideas, copyright is about the concrete expression of ideas in the shape of actual works, and trademarks are about stopping sleazy business practices.

You obtain copyright for a work automatically by producing the work—at least if the work exhibits a certain minimal creativity. Patents must be registered with the patent office and are examined for novelty. Trademarks can also be registered or else obtained by using a mark for a certain time in commerce and becoming identified with it by the public as the provider of a product or service.

Computer software (which can be thought of as a type of written work, and which incorporates possibly considerable creativity) is protected by copyright.

This means that it is, in principle, illegal to copy a program or a complete software package without the explicit consent of the copyright holder (the programmer or their employer).

In the early days of the computer it was not common for software to be sold. You were either given it together with your computer (which used to be expensive enough—millions of dollars) or would write it yourself. On the university scene in the 1960s and 1970s it was utterly normal to swap or copy programs, and in 1976 a certain Bill Gates was horrified to find out that his BASIC interpreter for the MITS Altair 8800 proved very popular indeed and he got lots of kudos for it, but almost nobody found it necessary to pay his asking price! Of course that wasn't proper on the users' part, but even the *idea* that software should be paid for was so outlandish at the time that few people actually entertained the thought.

As computers spread through offices during the late 1970s and 1980s, it became more and more common to sell software instead of giving it away. Not only that—the software companies sold only the executable machine code, not the source code that people would be able to examine to find out how the software worked, or even to modify. At some point, Richard M. Stallman, who used to be an MIT researcher, decided to try and reverse this trend by working on a system that would emphasise the culture of sharing that was the norm during the 1960s and 1970s. This "GNU" system[3] remains unfinished, but many of its components are now in use on Linux systems.

free software

Richard M. Stallman (often called "RMS" for short) can be thought of as the father of the idea of "free software". In this context "free" does not mean "free of charge", but that the user is "free" to do various things they would not be able or allowed to do with proprietary software[4]. RMS calls a software package "free" if four conditions, the "Four Freedoms", are met:

- The freedom to run the program for any purpose (Freedom 0).
- The freedom to study how the program works, and change it to make it do what you wish (Freedom 1).
- The freedom to redistribute copies so you can help your neighbor (Freedom 2).
- The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits (Freedom 3).

Access to the program's source code is a prerequisite for freedoms 1 and 3.

The idea of free software was favourably received in general, but even so the goals of RMS and the *Free Software Foundation* (FSF) were often misunderstood. In particular, companies took exception to the word "free", which in spite of appropriate clarifications was often confused with "cost-free". At the end of the 1990s, Eric S. Raymond, Bruce Perens and Tim O'Reilly created the *Open Source Initiative* (OSI), whose goal it was to provide better and less ideological marketing for free software. The FSF wasn't enthusiastic about this "watering down" of its ideas, and in spite of the very similar goals of the FSF and OSI the controversy has not died down completely even now (which to a certain degree may be due to the considerable egos of some of the main people involved).

FOSS

While the "free" in "free software" encourages the confusion with "no charge", the "open source" in "open-source software" can be interpreted such that the source code may be inspected but not modified or passed on—both of which are basic tenets of the OSI. In this sense, neither of the term is 100% unambiguous. The community often refers to "FOSS" (for *free*

[3] "GNU" is a "recursive acronym" for "GNU's Not Unix".

[4] The "Free Software Foundation", Stallman's organisation, calls this "free as in speech, not as in beer".

Copyright © 2012 Linup Front GmbH

and open-source software) or even "FLOSS" (*free, libre, and open-source software*, where the *libre* is supposed to support the sense of "liberty"). FLOSS

How to make money with free software if anyone is allowed to change and copy the software? A very legitimate question. Here are a few ideas for "open-source business models": business models

- You could provide additional services such as support, documentation, or training for free software and get paid for that (this works very well for your author's company, Linup Front GmbH, and the LPI seems to be able to make a living from selling Linux certification).
- You could create bespoke improvements or extensions for specific customers and be paid for your time (even if the result of your development then becomes part of the generally available version). This works even for free software that you didn't originally write yourself.

  > Within the "traditional" model of proprietary software development, the original manufacturer of the software has a monopoly on changes and further development. As the customer of such a company, you could be in trouble if the manufacturer discontinues the product or disappears outright (by going bankrupt or getting acquired and dissolved by a competitor), because you will have gone to great trouble and expense to adopt a piece of software with no future. With free software, you can always try to find somebody to take on support in place of the original manufacturer—if necessary, you can get together with other users of the software who don't want to be left alone, either.

- If you distribute a software package, you could provide a basic version as FOSS and hope that this will entice enough people to buy the proprietary "full version" to really get things done. (The jargon expression for this is "open core".)

  > This is a two-edged sword: On the one hand, it is of course nice if there is more free software, but on the other hand the result is often that you *need* the proprietary version because important functionality is not available in the free version and it would be too much work to add that functionality independently. In this case, the "free" version mostly serves as a PR engine if the manufacturer wants to appear modern and "open-source friendly"—to "talk the talk" without "walking the walk".

2.2.2 Licences

How does a piece of software become "free" or "open source"? We mentioned that certain rights—for example, the right to copy or modify a work—are reserved for the author of the work, but that these rights can also be passed on to others. This happens (if it happens) by way of a "licence", a legal document which specifies the rights that the receiver of the software obtains by buying, downloading, … it. licence

Copyright as such allows the buyer (or lawful downloader) of a software package to install that software on a computer and to start and run it. This results simply from the fact that the software was made available to them by the author—there is evidently no point in selling somebody a program that the buyer isn't allowed to use[5]. (Conversely, whoever gave the author money for the program has a *right* to use the program in exchange for their payment.) Other actions such as uncontrolled copying and distribution, or modifying the program, are explicitly forbidden by copyright, and if the original author wants people to be allowed to do these things they must be written into the licence.

end-user licence agreement

Proprietary programs often come with an “end-user licence agreement” (EULA) that the buyer must accept before actually being able to use the software. The software vendor uses the EULA to forbid the buyer to do things the buyer would in fact be allowed to do by copyright law—such as selling the software “used” to someone else, or to say bad things (or indeed anything at all) about the software in public. Such an EULA is a contract that needs to be accepted by both sides, and the legal hurdles (at least in Germany) are fairly high. For example, a software buyer must be able to inspect the EULA conditions before buying the software, or it must at least be possible to return the software unused for a full refund if the buyer does not agree with the EULA.

open-source licences

On the other hand, the open-source licences for FLOSS are used to let the receiver of the software perform actions that would otherwise be *forbidden* by copyright. They generally do not try to restrict *use* of the software, but for the most part govern questions of modifying and distributing the software. In that sense they do not leave the software receiver worse off than they would have to expect from the sale of any other good. Hence, unlike EULAs, free-software licenses are usually not contracts that the receiver must explicitly accept, but one-sided declarations on the part of the software author, and the rights conferred by them constitute extra benefits on top of the simple usage rights inherent in the law.

By now there is a whole zoo of licences fulfilling the basic requirements for free or open-source software. The best-known free-software licence is the *General Public License* (GPL) promulgated by Richard M. Stallman, but there are several others. The OSI “certifies” licences that according to its opinion embody the spirit of open source, just as the FSF approves licences that safeguard the “four freedoms”. Lists of approved licences are available from those organisations’ web pages.

If you consider starting a free or open-source software project, you are certainly free to come up with your own licence fulfilling the requirements of the FSF or OSI. However, it is usually better to adopt an existing licence that has already been approved. This means that you do not need to obtain approval for your licence from the FSF or OSI, and the existing licences have usually been examined by legal professionals and can be considered reasonably water-tight—as an amateur in contract or intellectual property law you might overlook important details that could get you in trouble later on.

FOSS and copyright

It is important to observe that the proponents of free or open-source software in no way intend to completely abolish copyright for software. In fact, free software as we know it can only work *because* copyright gives software authors the legal right to make the modification and distribution of their software contingent upon conditions such that the receiver of the software must also be given the right to modify and distribute it. Without copyright, everybody could help themselves to any available piece of software, and central tenets such as the “four freedoms” would be endangered because it would be possible for people to hoard without any sharing at all.

2.2.3 The GPL

The Linux kernel and large parts of what one would otherwise consider “Linux” is distributed under the *General Public License* (GPL). The GPL was developed by RMS for the GNU project and is supposed to ensure that software that was originally distributed under the GPL remains under the GPL (this type of licence is also referred to as a *copyleft* licence). This works approximately like this:

[5] In fact, the copyright laws of many countries contain explicit clarifications stating that the process of *copying* a program from disk into RAM in order to run it on the computer is not subject to copyright.

- GPL software must be available in source form and may be used for arbitrary purposes.
- It is expressly allowed to modify the source and to distribute it in unmodified or modified form, as long as the receiver is given the same rights under the GPL.
- It is also expressly allowed to distribute (or even sell) GPL software in executable form. In this case, the source code (including the GPL rights) must be furnished alongside the executables, or must, during a certain period of time, be made available upon request.

  "Source code" in this context means "everything necessary to get the software to run on a computer". What that means in a particular case—for example, if it includes the cryptographic keys necessary to start a modified Linux kernel on an appropriately "locked-down" computer—is the subject of heated discussion.

  If somebody buys GPL software for money, they naturally obtain the right not just to install that software on all their computers, but also to copy and resell it (under the GPL). One consequence of this is that it does not make a lot of sense to sell GPL software "per seat", but one important side benefit is that prices for, e. g., Linux distributions stay reasonable.

- If you write a new program incorporating (parts of) a GPL program, the new program (a "derived work") must also be placed under the GPL.

  Here, too, there is heated debate as to how much and which parts of a GPL program must be incorporated into another program to make that program a "derived work". According to the FSF, using a dynamically loadable GPL library within a program forces that program under the GPL, even if it does not by itself contain any GPL code and therefore cannot be considered a "derived work" in the legal sense. How much of this is wishful thinking and how much is actually legally tenable must, in principle, be determined in a court of law.

The GPL stipulates rules for modifying and distributing software, not its actual use.

GPLv3

Right now two versions of the GPL are in widespread use. The newer version 3 (also called "GPLv3") was released at the end of June, 2007, and differs from the older version 2 (also "GPLv2") by clarifications in areas such as software patents, the compatibility with other free licences, and the introduction of restrictions on trying to make changes to theoretically "free" software within devices impossible through special hardware ("tivoisation", after a Linux-based digital PVR whose kernel cannot be modified or replaced). The GPLv3 allows its users to add further terms.—The GPLv3 did not meet with universal approval within the community, hence many projects (most prominently, the Linux kernel) have deliberately stayed with the simpler GPLv2. In addition, many projects distribute their code under "the GPLv2 or any later version", so you can decide which version of the GPL to follow when distributing or modifying such software.

It is considered good style among the developers of free software to make contributions to a project under the same licence that the project is already using, and most projects insist on this at least for code to be incorporated in the "official" version of the software. Some projects even insist on *copyright assignments*, where the author of code gives their rights to the project (or a suitable organisation). The advantage of this step is that copyright in the code rests solely with the project, and that copyright infringement—which only the copyright holder has legal standing to go after—is easier to address. A side effect that is either wanted or explicitly undesirable is that it becomes easier to change the licence for the whole project, which is also something only the copyright holder is allowed to do.

In the case of the Linux kernel project, which explicitly does not require copyright assignment, a licence change is difficult or impossible, since the code is a patchwork of contributions from more than a thousand authors. The issue was discussed during the introduction of the GPLv3, and the developers agreed that it would be a gigantic project to sort out the legal provenance of every single line of the Linux kernel source code and obtain its authors' consent for a licence change. Some Linux developers would be adamantly opposed, while others cannot be located or may even be deceased, and the corresponding code would have to be replaced by something similar with a clear copyright. However, at least Linus Torvalds remains a supporter of the GPLv2, so the problem does not really arise in practice.

GPL and money

The GPL does not stipulate anything concerning the possible price of the product. It is completely legal to give away copies of GPL software or to ask for money, as long as you furnish source code or make it available on request and as long as the receiver also gets the GPL rights. This means that GPL software is not necessarily "freeware".

You can find out more by studying the GPL [GPL91], which must be distributed with each GPL-ed product (including Linux).

The GPL is considered the most consistent of free licences in the sense that—as we said—it tries to ensure that, once published under the GPL, code must *remain* free. On various occasions companies tried to incorporate GPL code into their own products that they were not about to release under the GPL. However, after being sternly reprimanded by (most often) the FSF as the copyright holder, these companies have come into compliance with the GPL. At least in Germany, the GPL has also been validated in court—a Linux kernel programmer could obtain judgement in the Frankfurt district court against D-Link (a manufacturer of networking components, in this case a Linux-based NAS device), which he had sued for not complying with the GPL when distributing their device [GPL-Urteil06].

Why does the GPL work? Some companies which considered the GPL restrictions onerous have tried to declare, or get it declared, invalid. For example, in the United States it was termed "un-American" or "unconstitutional", and in Germany a company tried to use antitrust law to invalidate the GPL since it supposedly implied illegal price fixing. The idea seems to be that GPL software can be used by anyone for anything if something is demonstrably wrong with the GPL. All of these attacks ignore one important fact: Without the GPL, nobody except the original author would have the right to do anything with the code, since actions such as distributing or, in fact, selling the software are reserved by copyright law. So if the GPL disappears, all other parties interested in the code are a lot worse off than before.

A lawsuit where a software author sues a company that distributes his GPL code without complying with the GPL would approximately look like this:

Judge What seems to be the problem?

Software Author Your Lordship, the defendant has distributed my software without a licence.

Judge (to the defendant's counsel) Is that so?

At this point the defendant can say "yes", and the lawsuit is essentially over (except for the verdict). They can also say "no" but then it is up to them to justify why copyright law does not apply to them. This is an uncomfortable dilemma and the reason why few companies actually do this to themselves—most GPL disagreements are settled out of court.

If a manufacturer of proprietary software violates the GPL (e. g., by including a few hundreds of lines of source code from a GPL project in their product), this does not imply that all of that product's code must now be released under the terms of the GPL. It only implies that they have distributed GPL code without a license. The manufacturer can solve this problem in various ways:

- They can remove the GPL code and replace it by their own code. The GPL then becomes irrelevant for their software.
- They can negotiate with the GPL code's copyright holder (if available and willing to go along) and, for instance, agree to pay a license fee. See also the section on multiple licenses below.
- They can release their entire program under the GPL *voluntarily* and thereby comply with the GPL's conditions (the most unlikely method).

Independently of this there may be damages payable for the prior violations. The copyright status of the proprietary software, however, is not affected in any way.

2.2.4 Other Licences

In addition to the GPL, other licences are popular in the context of FOSS. Here is a brief overview:

BSD licence The BSD licence originated with the University of California in Berkeley's Unix distribution and is intentionally kept very simple: The recipient of the software is basically allowed to do with the software whatever they want as long as they do not create the impression that their use is endorsed by the university (or, by extension, the original software author). Any liability for the program is excluded as far as possible. The licence text must be preserved within the program's source code and—if the program or modified versions are distributed in executable form—its documentation.

If a software package contained BSD-licenced code, the BSD licence used to require that any promotional material for the software or system in question mention this fact and the copyright holder. This "advertising clause" has since been dropped.

Unlike the GPL, the BSD licence does not try to keep the software's source code public. Whoever obtains BSD-licenced software can essentially integrate it into their own software and distribute that in binary form (the GPL would require them to also distribute corresponding source code under the GPL).

Commercial software companies like Microsoft or Apple, who are generally less than enthusiastic about GPL software, usually have no issues with BSD-licenced software. Windows NT, for example, used to contain TCP/IP networking code from BSD (in adapted form), and large parts of the Macintosh's OS X operating system kernel are derived from BSD.

Within the FOSS community, opinions have differed for a long time as to whether the GPL or the BSD licence is "more free". On the one hand it makes sense to state that, as a recipient, one can do more with BSD-licenced software, and that therefore the BSD licence conveys more freedom in absolute terms. The GPL proponents on the other hand say that it is more important for code to stay free for everybody to use rather than disappear within proprietary systems, and that an indication of the greater freedom of the GPL is that those who obtain code from the pool of GPL software are also forced to give something back.

Apache licence The Apache licence is like the BSD licence in that it allows the use and adoption of licenced code without requiring (like the GPL) that modified Apache-licenced code must be made available to the public. It is more complex than the BSD licence, but also contains clauses governing use of patents and trademarks and other details.

Mozilla Public License (MPL) The Mozilla licence (which applies to the Firefox browser, among other software package) is a mixture of the BSD licence and the GPL. It is a "weak copyleft licence", since on the one hand it requires that code obtained under the MPL must be distributed under the MPL if at all (like the GPL), but also allows adding code under other licences which does not need to be distributed under the MPL.

Creative Commons

The success of the FOSS community encouraged the law professor, Lawrence (Larry) Lessig to apply the concept to other works in addition to software. The goal was to increase the pool of cultural assets like books, images, music, films, … that would be available to others for free use, modification and distribution. Since the common FOSS licences are mostly geared towards software, a set of *creative-commons* licences was developed to enable creators to donate their works to the public in a controlled fashion. They can stipulate various restrictions like "verbatim reproduction of the work only", "modifications are allowed but (GPL-like) recipients of the modified work must be allowed to make further modifications" or an exclusion of commercial use of the work.

public domain

The "public domain" applies to cultural works that no longer fall under copyright. While in the Anglo-Saxon legal tradition a creator may explicitly place a work (such as a piece of software) in the public domain by disclaiming all rights to it, this is not possible in other legal environments. For example, in Germany, works enter the public domain automatically 70 years after the last person involved in their creation has died. We shall have to wait a bit for the first computer program to become available to everyone in this way.

There is a fair chance that no copyrighted works created after approximately 1930 will ever enter the public domain. In the United States, Congress extends the copyright term whenever the copyright on a certain cartoon mouse is in danger of expiry, and the rest of the world is generally happy to follow suit. It is not entirely clear why Walt Disney's *great-grandchildren* ought to be able to rake in money like Scrooge McDuck based on the great artist's creativity, but as usual this mostly depends on who has the most influential lobbyists.

Multiple licences

In principle, the copyright holder of a software package can also distribute the package under several licences simultaneously—for example, the GPL for FOSS developers and a proprietary licence for companies which would rather not make their own source code available. Of course this makes most sense for libraries that other programmers can integrate in their own programs. Whoever wants to develop proprietary software can then "buy themselves out of" the GPL's restrictions.

Exercises

2.3 [!2] Which of the following statements concerning the GPL are true and which are false?

1. GPL software may not be sold.

2. GPL software may not be modified by companies in order to base their own products on it.
3. The owner of a GPL software package may distribute the program under a different license as well.
4. The GPL is invalid, because one sees the license only after having obtained the software package in question. For a license to be valid, one must be able to inspect it and accept it before acquiring the software.

2.4 [2] Compare the FSF's "four freedoms" to the *Debian Free Software Guidelines* (`http://www.debian.org/social_contract#guidelines`) of the Debian project (see Section 2.4.4). Which definition of free software do you like better and why?

2.3 Important Free Software

2.3.1 Overview

Linux is a powerful and elegant operating system, but the nicest operating system is worth nothing without programs to run on it. In this section we present a selection of the most important free/open-source programs that can be found on typical Linux PCs.

If a particular program is not included that does not imply that we don't think it's worthwhile. Space is limited, and we try to cover mostly those software packages that the LPI mentions in its exam objectives (just in case).

2.3.2 Office and Productivity Tools

Most computers are probably used for "office applications" like writing letters and memos, seminar papers or Ph. D. theses, the evaluation of data using spreadsheets and graphics packages and similar jobs. Users also spend large parts of their computer time on the Internet or reading or writing e-mail. No wonder that there is a lot of good free software to help with this.

Most of the programs in this section aren't just for Linux, but are also available for Windows, OS X or even other Unix variants. This makes it possible to gradually move users into a FOSS ecosystem by installing, e. g., LibreOffice, Firefox, and Thunderbird on a Windows PC in place of Microsoft Office, Internet Explorer, and Outlook, before replacing the operating system itself with Linux. If you play your cards right[6], the users might not even notice the difference.

OpenOffice.org has been the FOSS community's flagship big office-type application for years. It started many years ago as "StarOffice" and was eventually acquired by Sun and made available as free software (in a slightly slimmed-down package). OpenOffice.org contains everything one would expect from an office package—a word processor, spreadsheet, presentation graphics program, business chart generator, database, …—and can also handle (in a fashion) the file formats of its big competitor from Microsoft.

LibreOffice After Sun was taken over by Oracle and the future of OpenOffice.org was up in the air, some of the main OpenOffice.org developers banded together and published their own version of OpenOffice.org under the name of "LibreOffice". Both packages are currently maintained side by side (Oracle donated OpenOffice.org to the Apache Software Foundation)—not the optimal state, but it is unclear whether and how there will be a "reunification".

[6] For example, you could plant the current Ubuntu on your users as a "Windows 8 beta" …

By now, most major Linux distributions contain LibreOffice, which is more aggressively developed and—more importantly—cleaned up.

Firefox is by now the most popular web browser and is being distributed by the Mozilla Foundation. Firefox is more secure and efficient than the former "top dog" (Mircosoft's Internet Explorer), does more and conforms better to the standards governing the World Wide Web. In addition, there is a large choice of extensions with which you can customise Firefox according to your own requirements.

Chromium is the FOSS variant of the Google browser, Chrome. Chrome has recently started competing with Firefox—Chrom{e,ium} is also a powerful, secure browser with various extensions. Especially Google's web offerings are geared towards the Google browser and support it very well.

Thunderbird is an e-mail program by the Mozilla Foundation. It shares large parts of its underlying infrastructure with the Firefox browser and—like Firefox—offers are large pool of extensions for various purposes.

2.3.3 Graphics and Multimedia Tools

Graphics and multimedia has long been the domain of Macintosh computers (even though the Windows side of things has to offer nice software, too). Admittedly, Linux is still missing true equivalents to programs like Adobe's Photoshop, but the existing software is also nothing to scoff at.

The GIMP is a program for editing photographs and similar images. It is not quite up to par with Photoshop (for example, some pre-press functionality is missing) but is absolutely usable for many purposes and even offers a few tools for jobs such as creating graphics for the World Wide Web that Photoshop does not do as conveniently.

Inkscape If The GIMP is Linux's Photoshop, then Inkscape corresponds to Illustrator—a powerful tool for creating vector-based graphics.

ImageMagick is a software package that allows you to convert almost any graphic format into nearly every other one. It also enables script-controlled manipulation of images in endless different ways. It is wonderful for web servers and other environments where graphics need to be processed without a mouse and monitor.

Audacity serves as a multi-track recorder, mixing desk and editing station for audio data of all kinds and is also popular on Windows and the Mac.

Cinelerra and other programs like KDEnlive or OpenShot are "non-linear video editors" that can edit and dub video from digital camcorders, TV receivers, or webcams, apply various effects and output the result in various formats (from YouTube to DVD).

Blender is not just a powerful video editor, but also allows photorealistic "rendering" of three-dimensional animated scenes and is hence the tool of choice for creating professional-quality animated films.

We might just as well mention here that today no Hollywood blockbuster movie is produced without Linux. The big studios' special effects "render farms" are now all based on Linux.

2.3.4 Internet Services

Without Linux, the Internet would not be recognisable: Google's hundreds of thousands of servers run Linux just as the trading systems of most of the world's big stock exchanges (the German exchange as well as the London and New York stock exchanges), since the required performance can only be made available with Linux. In point of fact, most Internet software is developed on Linux first, and most university research in these areas takes place on the open-source Linux platform.

Apache is by far the most popular web server on the Internet—more than half of all web sites run on an Apache server.

There are of course other good web servers for Linux—for example, Nginx or Lighttpd—, but Apache remains the most common.

MySQL and PostgreSQL are freely available relational database servers. MySQL is best used for web sites, while PostgreSQL is an innovative and high-performance database server for all kinds of purposes.

Postfix is a secure and extremely powerful mail server which is useful for any environment from the "home office" to large ISPs or Fortune 500 enterprises.

2.3.5 Infrastructure Software

A Linux server can prove very useful within a local-area network: It is so reliable, fast, and low-maintenance that it can be installed and forgotten (except for regular backups, of course!).

Samba turns a Linux machine into a server for Windows clients which makes disk space and printers available to all Windows machines on the network (Linux machines, too). With the new Samba 4, a Linux server can even serve as an Active Directory domain controller, making a Windows server extraneous. Reliability, efficiency and saved licence fees are very convincing arguments.

NFS is the Unix environment to Samba and allows other Linux and Unix machines on the network access to a Linux server's disks. Linux supports the modern NFSv4 with enhanced performance and security.

OpenLDAP serves as a directory service for medium and large networks and offers a large degree of redundancy and performance for queries and updates through its powerful features for the distribution and replication of data.

DNS and DHCP form part of the basic network infrastructure. With BIND, Linux supports the reference DNS server, and the ISC DHCP server can cooperate with BIND to provide clients with network parameters such as IP addresses even in very large networks. Dnsmasq is an easy-to-operate DNS and DHCP server for small networks.

2.3.6 Programming Languages and Development

From its beginnings, Linux has always been a system by developers for developers. Accordingly, compilers and interpreters for all important programming languages are available—the GNU compiler family, for example, supports C, C++, Objective C, Java, Fortran and Ada. Of course the popular scripting languages such as Perl, Python, Tcl/Tk, Ruby, Lua, or PHP are supported, and less common languages such as Lisp, Scheme, Haskell, Prolog, or Ocaml, are also part of many Linux distributions.

A very rich set of editors and auxiliary tools makes software development a pleasure. The standard editor, `vi` is available as are professional development environments such as GNU Emacs or Eclipse.

Linux is also useful as a development environment for "embedded systems", namely computers running inside consumer appliances that are either based on Linux itself or use specialised operating systems. On a Linux PC, it is straightforward to install a compiler environment that will generate machine code for, say, ARM processors. Linux also serves to develop software for Android smartphones, and professional tools for this purpose are supplied for free by Google.

Exercises

2.5 [!1] Which FOSS programs have you heard about? Which ones have you used yourself? Do you find them better or worse than proprietary alternatives? If so, why? If not, why not?

2.4 Important Linux Distributions

2.4.1 Overview

If someone says something like "My PC runs Linux", they usually mean not (just) Linux, the operating system kernel, but a complete software environment based on Linux. This normally includes the shell (`bash`) and command-line tools from the GNU project, the X.org graphics server and a graphical desktop environment such as KDE or GNOME, productivity tools like LibreOffice, Firefox or The GIMP and lots of other useful software from the previous section. Of course it is possible to assemble all these tools from their original sources on the Internet, but most Linux users prefer a pre-made software collection or "Linux distribution".

The first Linux distributions appeared in early 1992—however, none of those is still being developed, and they are mostly forgotten. The oldest distribution that is still being worked on is Slackware, which first appeared in July, 1993.

There is a multitude of Linux distributions with different goals and approaches and different organisational structure. Some distributions are published by companies and possibly only sold for money, while other distributions are put together by teams of volunteers or even individuals. In this section we shall be discussing the most important general-purpose distributions.

If we do not mention your favourite distribution here this isn't due to the fact that we can't abide it, but really due to the fact that our space and time is limited and we must (unfortunately!) restrict ourselves to essentials (no pun intended). If a distribution isn't on our list that doesn't imply that it is bad or useless, but only that it isn't on our list.

The "DistroWatch" web site (`http://distrowatch.com/`) lists the most important Linux distributions and serves as a focal point for distribution-oriented news. Right now it contains 317 distributions (that is three hundred and seventeen!), but by the time you're reading this, this number is probably no longer correct.

2.4.2 Red Hat

Red Hat (`http://www.redhat.com/`) was established in 1993 as "ACC Corporation", a distribution company for Linux and Unix accessories. In 1995, the company founder, Bob Young, bought the business of Marc Ewing, who in 1994 had published a Linux distribution called "Red Hat Linux", and changed the name of his corporation to "Red Hat Software". In 1999, Red Hat went public and is by now probably the largest corporation solely based on Linux and open-source software. It is part of the "Standard & Poor's 500", a stock index which serves as an indicator for the US economy.

Red Hat has withdrawn from its original individual-customer business (the last “Red Hat Linux” was published in April, 2004) and now markets a distribution for the professional use by companies under the name of “Red Hat Enterprise Linux” (RHEL). RHEL is licenced per server, although you do not pay for the software—which is furnished under the GPL and similar FOSS licences—but for access to timely updates and support in the case of problems. RHEL is mostly geared towards data centres and, among other things, supports (with appropriate additional tools) the construction of fault-tolerant “clusters”.

“Fedora” (`http://www.fedoraproject.org/`) is a distribution, mostly controlled by Red Hat, which serves as a “test bed” for RHEL. New software and ideas are trialled in Fedora first, and whatever proves useful may show up in RHEL sooner or later. Unlike RHEL, Fedora is not sold but made available for free download instead; the project is governed by a committee whose members are partly elected by the developer community and partly nominated by Red Hat. (The committee chair is nominated by Red Hat and has veto powers.) For many Fedora users, the focus on current software and new ideas is part of the attraction of the distribution, even though this implies frequent updates. Fedora is less suitable for beginners and the use on servers which are supposed to be reliable.

CentOS
Scientific Linux

Since Red Hat distributes its software strictly under FOSS licences like the GPL, it is possible in principle to operate a system that corresponds to the current RHEL without paying licence fees to Red Hat. There are distributions like CentOS (`http://www.centos.org/`) or Scientific Linux (`https://www.scientificlinux.org/`) which are essentially based on RHEL but remove all Red Hat branding. This means that you get essentially the same software but without Red Hat support.

CentOS in particular is so close to RHEL that Red Hat is happy to sell you support for your CentOS machines. You don’t even need to install RHEL first.

2.4.3 SUSE

Foundation

The German company SUSE was first incorporated 1992 as a Unix consultancy under the name of “Gesellschaft für Software- und System-Entwicklung” and accordingly spelled itself “S.u.S.E.”. One of its products was a German version of Patrick Volkerding’s Linux distribution, Slackware, which in turn was derived from the first complete Linux distribution, *Softlanding Linux System* or SLS. S.u.S.E. Linux 1.0 appeared 1994 and slowly diverged from Slackware by taking on features from Red Hat Linux, like RPM package management or the `/etc/sysconfig` file. The first version of S.u.S.E. Linux that no longer looked like Slackware was version 4.2 of 1996. SuSE (the dots had disappeared at some point) soon became the leading German-language Linux distribution and published SuSE Linux as a “boxed set” in two flavours, “Personal” and “Professional”—the latter was noticeably more expensive and contained, among other things, more server-oriented software.

Novell takeover
Attachmate

In November, 2003, the US software company, Novell, announced its takeover of SuSE for 210 million dollars; the deal was concluded in January, 2004. (At this point the “U” was capitalised, too.) In April, 2011, Novell, including SUSE, was acquired by Attachmate, a company selling terminal emulation, system monitoring, and application integration tools and services that so far had not been notable within the Linux and open-source communities. Since then, Novell has continued to operate as two separate business units, one of which is SUSE.

Like Red Hat, SUSE offers an “enterprise Linux”, the *SUSE Linux Enterprise Server* (SLES, `http://www.suse.com/products/server/`), which resembles RHEL in that it is published fairly infrequently and promises a long life cycle of

Copyright © 2012 Linup Front GmbH

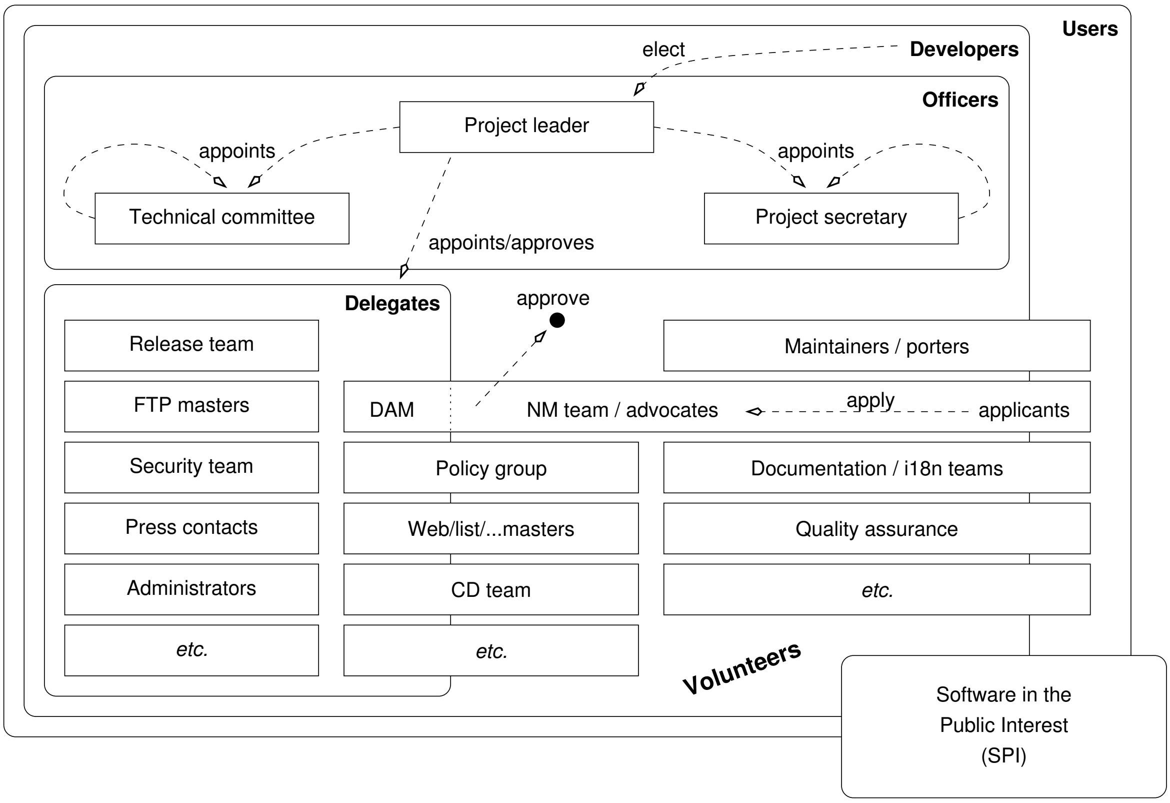


Figure 2.2: Organizational structure of the Debian project. (Graphic by Martin F. Krafft.)

7–10 years. In addition, there is *SUSE Linux Enterprise Desktop* (SLED), a distribution which is intended to be used on desktop workstations. SLES and SLED differ in the choice of packages included; with SLES, the focus is on server software, while SLED is geared more towards interactive software.

SUSE, too, has opened its "private customer" distribution and made it freely available as "openSUSE" (`http://www.opensuse.org/`)—in former times the distribution would only have been made available for download several months after it had been released on optical media. Unlike Red Hat, SUSE still offers a "boxed set" that also contains proprietary software. Unlike Fedora, openSUSE is a serious platform which still uses a fairly brief life cycle.

YaST A notable property of SUSE distributions is "YaST", a comprehensive graphical system administration tool.

2.4.4 Debian

Debian project Unlike the two big Linux distribution companies Red Hat and Novell/SUSE, the **Debian project** (`http://www.debian.org/`) is a collaboration of volunteers whose goal is to make available a high-quality Linux distribution called "Debian GNU/Linux". The Debian project was announced on 16 August 1993 by Ian Murdock; the name is a contraction of his first name with that of his then-girlfriend (now ex-wife) Debra (and is hence pronounced "debb-ian"). By now the project includes more than 1000 volunteers.

Debian is based on three documents:

- The *Debian Free Software Guidelines* (DFSG) define which software the project considers "free". This is important, since only DFSG-free software can be part of the Debian GNU/Linux distribution proper. The project also distributes non-free software, which is strictly separated from the DFSG-free software on the distribution's servers: The latter is in a subdirectory called `main`, the former in `non-free`. (There is an intermediate area called `contrib`; this contains software that by itself would be DFSG-free but does not work without other, non-free, components.)

- The *Social Contract* describes the project's goals.
- The *Debian Constitution* describes the project's organisation (see Figure 2.2.

versions

At any given time there are at least three versions of Debian GNU/Linux: New or corrected versions of packages are put into the `unstable` branch. If, for a certain period of time, no grave errors have appeared in a package, it is copied to the `testing` branch. Every so often the content of `testing` is "frozen", tested very thoroughly, and finally released as `stable`. A frequently-voiced criticism of Debian GNU/Linux is the long timespan between `stable` releases; many, however, consider this an advantage. The Debian project makes Debian GNU/Linux available for download only; media are available from third-party vendors.

derivative projects

By virtue of its organisation, its freedom from commercial interests, and its clean separation of free and non-free software, Debian GNU/Linux is a sound basis for derivative projects. Some of the more popular ones include Knoppix (a "live CD" which makes it possible to test Linux on a PC without having to install it first), SkoleLinux (a version of Linux especially adapted to the requirements of schools), or commercial distributions such as Xandros. Limux, the desktop Linux variant used in the Munich city administration, is also based on Debian GNU/Linux.

2.4.5 Ubuntu

Ubuntu

Ubuntu goals

One of the most popular Debian derivatives is Ubuntu, which is provided by the British company, Canonical Ltd., founded by the South African entrepreneur Mark Shuttleworth. ("Ubuntu" is a word from the Zulu language and roughly means "humanity towards others".) The Ubuntu goals of Ubuntu is to offer, based on Debian GNU/Linux, a current, capable, and easy-to-understand Linux which is updated at regular intervals. This is facilitated, for example, by Ubuntu being offered on only three computer architectures as opposed to Debian's ten or so, and by restricting itself to a subset of the software offered by Debian GNU/Linux.

Ubuntu is based on the `unstable` branch of Debian GNU/Linux and uses, for the most part, the same tools for software distribution, but Debian and Ubuntu software packages are not necessarily mutually compatible. Ubuntu is published on a fairy reliable six-month cycle, and every two years there is an "LTS" or "long-term support" version for which Canonical promises five years' worth of updates.

Ubuntu vs. Debian

Some Ubuntu developers are also active participants in the Debian project, which ensures a certain degree of exchange. On the other hand, not all Debian developers are enthusiastic about the shortcuts Ubuntu takes every so often in the interest of pragmatism, where Debian might look for more comprehensive solutions even if these require more effort. In addition, Ubuntu does not appear to feel as indebted to the idea of free software as does Debian; while all of Debian's infrastructure tools (such as the bug management system) are available as free software, this is not always the case for those of Ubuntu.

Ubuntu vs. SUSE/Red Hat

Ubuntu not only wants to provide an attractive desktop system, but also to take on the more established systems like RHEL or SLES in the server space, by offering stable distributions with a long life cycle and good support. It is unclear how Canonical Ltd. intends to make money in the long run; for the time being the project is mostly supported out of Mark Shuttleworth's private coffers, which are fairly well-filled since he sold his Internet certificate authority, Thawte, to Verisign …

Copyright © 2012 Linup Front GmbH

2.4.6 Others

In addition to the distributions we mentioned there are many more, such as Mandriva Linux (`http://www.mandriva.com/en/linux/`) or Turbolinux (`http://www.turbolinux.com/`) as smaller competitors of Red Hat and SUSE, Gentoo Linux (`http://www.gentoo.org/`) as a distribution focused on source code, various "live systems" for different purposes ranging from firewalls to gaming or multimedia platforms, or very compact systems usable as routers, firewalls or rescue systems.

Android

Also worth mentioning if only because of the number of "installed systems" is Android, which with a grain of salt can be considered a "Linux distribution". Android consists of a Linux operating system kernel with a user space environment maintained by Google and based on Google's version of Java ("Dalvik") instead of the usual environment based on GNU, X, KDE, etc. that forms the basis of most "normal" distributions. An Android smartphone or tablet presents itself to the user completely unlike a typical Linux PC running openSUSE or Debian GNU/Linux, but is still arguably a Linux system.

While most Android users buy their system preinstalled on their phone or tablet and then never change it, most Android-based devices make it possible (sometimes with hacks) to install an alternative Android "distribution", of which there are several. For many devices this is the only way of obtaining up-to-date Android versions, if the device manufacturer and/or telephone service provider do not deem it necessary to publish an official new version.

2.4.7 Differences and Similarities

Similarities

Even though there is a vast number of Linux distributions, at least the major distributions turn out to be fairly similar in daily life. This is partly due to their use of the same basic programs—for example, the command-line interpreter is almost always `bash`. On the other hand there are standards that try to curb rank growth. These include the *Filesystem Hierarchy Standard* (FHS) or the *Linux Standard Base* (LSB), which attempts to codify a unified "base version" of Linux to make it easier for third-party software suppliers to distribute their software for as many Linux distributions as possible.

Unfortunately, LSB did not turn out to be the unmitigated success that it was supposed to be—it was often misunderstood as a method for slowing down or stopping innovation in Linux and reducing diversity (even though most distributions make it possible to provide an LSB environment in parallel to, and independently of, the actual environment for software furnished by the distribution), while the third-party suppliers who were supposed to be targeted in the first place generally preferred to "certify" their software packages for the main "enterprise distributions" like RHEL and SLES, and supporting only these platforms. While it is definitely not unlikely that it will be possible to run (or get to run) SAP or Oracle on, say, Debian GNU/Linux, the cost involved in licencing large commercial software packages like these are such that the licence fees for RHEL and SLES do not make a noticeable difference in the bottom line.

Packaging formats

One noticeable area where distributions differ is the method used to administer (install and remove) software packages, and following on from that the file format of pre-made software packages within the distribution. There are currently two main approaches, namely the one by Debian GNU/Linux ("`deb`") and the one originally developed by Red Hat ("`rpm`"). As usual, neither of the two is clearly superior to the other, but either has enough strong points to keep its proponents from changing over. The `deb` approach is used by Debian GNU/Linux, Ubuntu, and other Debian derivatives, while Red Hat, SUSE, and various distributions derived from those rely on `rpm`.

Table 2.1: Comparison of the most important Linux distributions (as of February, 2012)

	RHEL	Fedora	SLES	openSUSE	Debian	Ubuntu
Supplier	Red Hat	Red Hat + Comm.	SUSE	SUSE/Comm + Comm.	Debian Project	Canonical +Comm.
Target	Enterp.	Geeks	Enterp.	Private	Ent/Priv	Ent/Priv
Fees due?	yes	no	Support	no	no	no
First pub	2003	2003	2000	2006	1993	2004
Rel cycle	3–4 yrs	≈ 6 mth.	3–4 yrs	8 months	≈ 2 yrs	6 mth
Life cyc	10 yrs	≈ 1 yr	7 yrs.	18 months	3–4 yrs	5 yrs (LTS)
Platforms	6	2	5	2	10	3
Pkges (appr.)	3000	26.000	?	14.650	29.050	37.000
Pkg format	`rpm`	`rpm`	`rpm`	`rpm`	`deb`	`deb`
Live media?	?	yes	no	yes	yes	yes

dependencies

Both approaches include the comprehensive management of "dependencies" between software packages, which helps ensure the consistency of the system and to prevent the removal of software packages that other packages in the system rely on (or, conversely, to enforce their installation if another package being installed requires them and they are not installed already).

repositories

While users of Windows and OS X are used to obtaining software from a variety of sources[7], at least the large distributions like Debian, openSUSE, or Ubuntu attempt to provide their users with a very comprehensive selection of software directly via the package administration tools. These allow access to the distributions' "repositories" and offer search functions (of varying quality) for particular software packages which can then be conveniently installed, possibly together with their dependencies, over the network.

It is important to note that it is not necessarily possible to exchange packages between distributions, even if these use the same basic packaging format (`deb` or `rpm`)—the packages contain various assumptions on how a distribution is put together which by far exceed the packaging format. It is not completely infeasible (Debian GNU/Linux and Ubuntu, for example, are similar enough that under the right circumstances it may be quite possible to install a Debian package on an Ubuntu system or vice-versa) but, as a rule of thumb, the deeper a package is rooted inside the system, the larger the chance of trouble occurring—a package that just contains a few executables and their documentation is likely to present fewer problems than a system service that must be integrated into the machine's startup sequence. If in doubt, refrain from experiments with an uncertain outcome.

Table 2.1 shows an overview of the most important Linux distributions. For more information, consult DistroWatch or the individual distributions' web sites.

[7] However, Apple and Microsoft are now eager to establish the centralised "app store" concept known from smartphones for their PC operating systems, too.

Copyright © 2012 Linup Front GmbH

Summary

- The first Linux version was developed by Linus Torvalds and made available as "free software" on the Internet. Today, hundreds of developers collaborate worldwide to update and extend the system.
- Free software allows you to use the software for arbitrary purposes, to inspect and modify the code, and to pass modified or unmodified copies on to others.
- Free-software licences give the receiver of a software package rights that they wouldn't otherwise have, while licence agreements for proprietary software try to get the receiver to waive rights that they would otherwise have.
- The GPL is one of the most popular free-software licences.
- Other common licences for free software include the BSD licence, the Apache licence or the Mozilla Public License. Creative-commons licences are meant for cultural works other than software.
- There is a wide variety of free and open-source software packages for all sorts of purposes.
- There are very many different Linux distributions. The most popular include Red Hat Enterprise Linux and Fedora, SUSE Linux Enterprise Server and openSUSE, Debian GNU/Linux and Ubuntu.

Bibliography

GPL-Urteil06 Landgericht Frankfurt am Main. "Urteil 2-6 0 224/06", July 2006. `http://www.jbb.de/urteil_lg_frankfurt_gpl.pdf`

GPL91 Free Software Foundation, Inc. "GNU General Public License, Version 2", June 1991. `http://www.gnu.org/licenses/gpl.html`

TD01 Linus Torvalds, David Diamond (Eds.) *Just for Fun: Wie ein Freak die Computerwelt revolutionierte*. Hanser Fachbuch, 2001. ISBN 3-446-21684-7.

Copyright © 2012 Linup Front GmbH

3

First Steps with Linux

Contents

Goals

- Trying simple Linux functionality
- Learning to create and modify files using a text editor

Prerequisites

- Basic knowledge of other computer operating systems is useful

3.1 Logging In and Out

access rights

The Linux system distinguishes between different users. As a consequence, you may not be able to start using the computer immediately after it has been switched on. First you must tell the computer who you are—you need to “log in” (or “on”). Based on the information you provide, the system can then decide what you may do (or not do). Of course you need access rights to the system (an “account”)—the system administrator must have entered you as a valid user and assigned you a user name (e. g., `joe`) and a password (e. g., `secret`). The password is supposed to ensure that only you can use your account; you must keep it secret and should not make it known to anybody else. Whoever knows your user name and password can pretend to be you on the system, read (or delete) all your files, send electronic mail in your name and generally get up to all kinds of shenanigans.

Some modern Linux distributions try to make it easy on you and allow you to skip the login process on a computer that only you will be using anyway. If you use such a system, you will not have to log in explicitly, but the computer boots straight into your session. You should of course take advantage of this only if you do not foresee that third parties have access to your computer; refrain from this in particular on laptop computers or other mobile systems that tend to get lost or stolen.

Logging in in a graphical enviroment These days it is common for Linux workstations to present a graphical environment (as they should), and the login process takes place on the graphics screen, too. Your computer displays a form that lets you enter your user name and password.

Don’t wonder if you only see asterisks when you’re entering your password. This does not mean that your computer misunderstands your input, but that it wants to make life more difficult for people who are watching you over your shoulder in order to find out your password.

After you have logged in, the computer starts a graphical session for you, in which you have convenient access to your application programs by means of menus and icons (small pictures on the “desktop” background). Most graphical environments for Linux support “session management” in order to restore your session the way it was when you finished it the time before (as far as possible, anyway). That way you do not need to remember which programs you were running, where their windows were placed on the screen, and which files you had been using.

Logging out in a graphical environment If you are done with your work or want to free the computer for another user, you need to log out. This is also important because the session manager needs to save your current session for the next time. How logging out works in detail depends on your graphical environment, but as a rule there is a menu item somewhere that does everything for you. If in doubt, consult the documentation or ask your system administrator (or knowledgeable buddy).

Logging in on a text console Unlike workstations, server systems often support only a text console or are installed in draughty, noisy machine halls, where you don’t want to spend more time than absolutely necessary. So you will prefer to log into such a computer via the network. In both cases you will not see a graphical login screen, but the computer asks you for your user name and password directly. For example, you might simply see something like

```
computer login: _
```

Copyright © 2012 Linup Front GmbH

(if we stipulate that the computer in question is called “`computer`”). Here you must enter your user name and finish it off with the ↩ key. The computer will continue by asking you for your password:

```
Password: _
```

Enter your password here. (This time you won’t even see asterisks—simply nothing at all.) If you entered both the user name and password correctly, the system will accept your login. It starts the command line interpreter (the *shell*), and you may use the keyboard to enter commands and invoke programs. After logging in, you will be placed in your “home directory”, where you will be able to find your files.

If you use the “secure shell”, for example, to log in to another machine via the network, the user name question is usually skipped, since unless you specify otherwise the system will assume that your user name on the remote computer will be the same as on the computer you are initiating the session from. The details are beyond the scope of this manual; the secure shell is discussed in detail in the Linup Front training manual *Linux Administration II*.

Logging out on a text console On the text console, you can log out using, for example, the `logout` command:

```
$ logout
```

Once you have logged out, on a text console the system once more displays the start message and a login prompt for the next user. With a secure shell session, you simply get another command prompt from your local computer.

Exercises

3.1 [!1] Try logging into the system. After that, log out again. (You will find a user name and password in your system documentation, or—in a training centre/school—your instructor/teacher will tell you what to use.)

3.2 [!2] What happens if you enter (a) a non-existing user name, (b) a wrong password? Do you notice anything unusual? What reasons could there be for the system to behave as it does?

3.2 Desktop Environment and Browser

3.2.1 Graphical Desktop Environments

If you logged into a graphical environment, your Linux computer presents a desktop that does not differ much from what you would get to see on other modern computers.

Unfortunately it is impossible for us to be more specific, since no two “Linuxes” are the same here. Unlike systems like Windows or the Macintosh operating system, which come with an “official” graphical environment, Linux lets you choose—when the system is installed, most major distributions offer you a choice between several graphical environments:

- KDE and GNOME are “desktop environments” which attempt to provide a comprehensive suite of applications with a similar look and feel. The goal of KDE and GNOME is to offer a user experience that is comparable or superior to that of proprietary systems. They try to include innovative features like KDE's "semantic search", which indexes files and documents in the background and is supposed to allow convenient access to "all photographs I took in Spain last month", regardless of where these are stored on disk[1]. Roughly speaking, KDE focuses on comprehensive customisability for sophisticated users, while GNOME, in the interest of simplicity and usability, tends to lean towards providing defaults that are impossible or less straightforward to change.

- LXDE and XFCE are "lightweight" environments. They resemble KDE and GNOME in their basic approach, but are more geared towards economical use of resources and thus dispense with various expensive services like semantic search.
- If you would rather not use a complete desktop environment, you may install any of a number of "window managers". This implies certain tradeoffs regarding the optical consistency and cooperation of programs, which result from the fact that, historically, there used to be few guidelines for the look and feel of graphical programs on Unix and Linux. Formerly—before KDE etc., which did establish a degree of standardisation in this respect—this used to be the usual way of doing things, but today a majority of Linux users relies on one of the preassembled graphical environments.

Even if two distributions use the same graphical environment (say, KDE) this doesn't mean that they will look the same on screen. Usually, the graphical environments allow a large degree of customisation of their "look" based on "themes", and distributions use this to set themselves apart from others. Consider cars; almost all cars have four wheels and a windscreen, but you would still never confuse a BMW with a Citroën or Ferrari.

control bar

In any event, you are likely to find a control bar (dock, panel, what have you) either at the top or at the bottom of the screen, which allows you to access the most important application programs by means of menu entries, or to log out or shut down the computer. KDE relies on a "panel" that roughly resembles that of Windows, where a "start button" (not actually called that) opens a menu of programs, while the rest of the bar shows icons for the currently running applications alongside little useful helpers like a clock, the network state, and so on. GNOME does not use a "start button", but moves the menu bar to the top of the screen; the most important programs are accessible through pull-down menus on the left-hand side of the screen, while the right-hand part is reserved for system status icons and the like.

file manager

The graphical environments usually provide a "file manager", which lets you access directories ("folders") on disk and manipulate the files and subdirectories they contain. The procedures here do not differ a lot from those on other graphical systems: You can copy or move files by dragging them from one directory window into another, and if you use the rightmost mouse button to click on the icon for a file, a "context menu" opens to offer additional actions that you can apply to the file. Do experiment.

dock

Frequently-used files or programs can often be deposited on the screen backdrop or placed in a certain area on screen (a "dock") for quick and convenient access.

virtual desktops

A useful feature of most Linux-based graphical environments which OS X and Windows do not offer (by default, anyway) are "virtual desktops". These multiply the available space on screen by making it convenient to switch back and forth between several simulated "desktops"—each with its own selection of program windows. This allows you to place everything you need to work on a program or document on one desktop, reserve another for your e-mail reader and yet another

[1] Microsoft promised that particular feature a few times, but it always conspicuously disappeared from the novelty list before the next Windows version was released.

for your web browser, and to dash off a quick e-mail message without having to rearrange the windows on your “programming desktop”.

3.2.2 Browsers

One of the most important programs on contemporary computers is the web browser. Fortunately, the most popular browsers are open-source programs, and Firefox or Google Chrome are available for Linux just as well as for Windows or OS X. (Your distribution probably doesn’t offer Google Chrome but the true open-source variant, Chromium, but that isn’t a big difference.) Look in your application menu for an entry like “Internet”, where you ought to find a browser (among other things).

Due to trademark concerns, on Debian GNU/Linux systems and various derivatives the Firefox browser is called “Iceweasel” instead (clever pun, eh?). This is because the Mozilla Foundation, the producer of Firefox, allows the distribution of precompiled versions of the browser under the name of “Firefox” only if the code corresponds to the “official” version. Since the Debian project on the one hand reserves the right to repair security problems on its own authority, and on the other hand takes copyright and trademarks very seriously, the name had to be changed. (Other distributions stay with the official version or don’t look as closely at the naming issue.)

3.2.3 Terminals and Shells

Even within a graphical Linux environment it is often convenient to access a “terminal window” where you can enter textual commands in a “shell” (the remainder of this manual mostly talks about shell commands, so you are likely to need this).

Fortunately, on most Linux desktop environments a terminal window is only a few mouse clicks away. In KDE on Debian GNU/Linux, for example, there is an entry called “Konsole (Terminal)” within the start menu under “System”, which will open a convenient program running a shell that will accept and execute textual commands. Similar methods are available on other desktop environments and distributions.

Exercises

3.3 [!2] Which graphical environment (if any) is installed on your computer? Look around. Open a file manager and figure out what happens if you right-click on a file or directory icon. What happens if you right-click on the empty window background (between icons)? How do you move a file from one directory to another? How do you create a new file or directory? How do you rename a file?

3.4 [2] Which web browser is installed on your computer? Are there several? Try starting the browser (or browsers) and make sure they are working.

3.5 [!2] Open a terminal window and close it again. Does your terminal window program support several sessions within the same window (possibly using subwindows with “tabs”)?

3.3 Creating and Modifying Text Files

No matter whether you are writing scripts or programs, editing configuration files as the system administrator, or simply jot down a shopping list: Linux is at its best when modifying text files. Hence, one of your first acts as a new Linux user ought to be learning how to create and edit text files. The tool of choice for this is a text

text editor

editor.

Text editors for Linux come in all sizes, shapes, and colours. We're taking the easy way out by explaining the most important features of "GNU Nano", a simple, beginner-proof text editor that runs inside a terminal session.

Of course the common graphical interfaces also support graphical text editors with menus, tool bars, and all sorts of useful goodies—comparable to programs like "Notepad" on Windows (or even better). For example, look for "Kate" on KDE or "gedit" on GNOME. We shall not be looking at these editors in detail here, for two reasons:

- They tend to explain themselves for the most part, and we do not want to insult your intelligence more than necessary.
- You will not always be in a position to use a graphical interface. You may be working on a remote computer using the "secure shell", or standing in front of a server console in the basement machine hall, and chances are that you will only have a text screen at your disposal.

In any case, you don't have to decide right now which *one* editor you will be using for the rest of your life. Nobody prevents you from using a graphical editor on your graphical desktop PC and hauling out something like Nano only if there is no other option.

Old-school Linux aficionados will scoff at something like Nano: The editor of choice for the true Linux professional is `vi` (pronounced "vee aye"), which like a living fossil has survived from a time when the greenish light of text terminals filled the machine rooms and one couldn't rely on a keyboard featuring arrow keys (!). If you are about to embark on a career in system administration, you should sooner or later become familiar with `vi` (at least on an elementary level), since `vi` is the only editor worth using that is available in largely identical form on practically every Linux or Unix variant. But that moment isn't now.

`pico`

GNU Nano is a "clone" of a simple editor called `pico` which was part of the PINE e-mail package. (PINE wasn't free software according to the generally accepted definitions, so the GNU project wrote the new editor from scratch. In the meantime, PINE's successor is freely available under the name of "alpine", and it contains a free version of `pico`, too.) Most distributions should offer either GNU Nano or `pico`; for simplicity we will spend the rest of this section talking about GNU Nano. Practically anything we say here also applies to `pico`.

Compared to the original `pico`, GNU Nano features some extensions (which shouldn't come as a surprise considering that already according to the name it is three orders of magnitude better), but most of these do not concern us directly. There is really only one extension that is very obvious: GNU Nano is "internationalised", so, for example, on a system that is otherwise set up to use the German language it should delight you with messages and help texts in German.

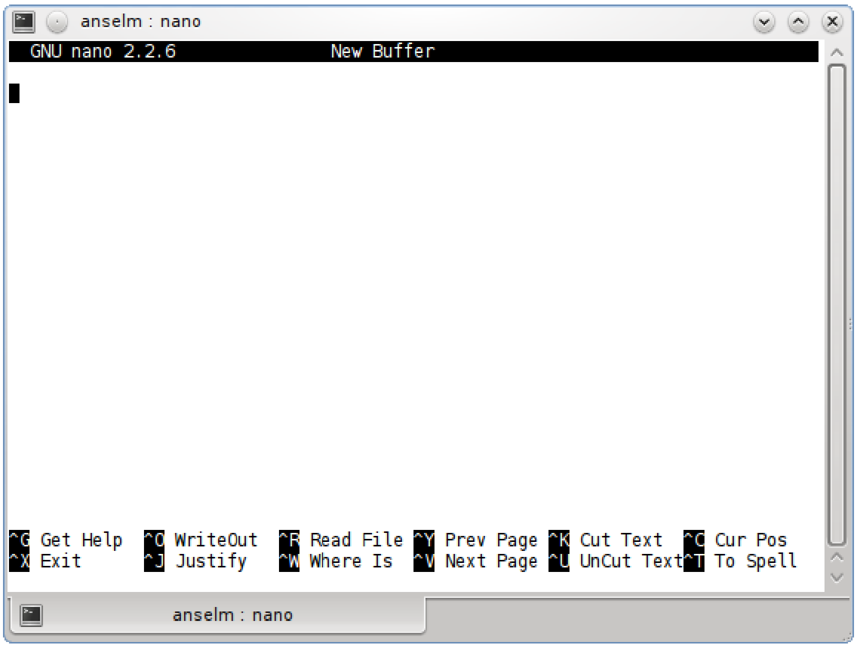
Starting GNU Nano

GNU Nano is most conveniently started inside a terminal window (Section 3.2.3) using a command like

```
$ nano myfile
```

(the "$ " here is just a stylised abbreviation of the command prompt—which may look somewhat more baroque on your system—and you do not need to enter this. Don't forget to finish the command using ↩, though!) Subsequently you sould see something resembling Figure 3.1—that is, a mostly empty window with one highlighted line at the top and two "help lines" at the bottom, which list important commands with brief explanations. The line immediately above the help lines is the "status line", where messages from Nano will appear and where you will be able to enter, e. g., file names when saving data to disk.

Copyright © 2012 Linup Front GmbH



```
  GNU nano 2.2.6                New Buffer




^G Get Help   ^O WriteOut   ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Figure 3.1: The GNU Nano text editor

If you need more useable space on the screen, you can suppress the help lines using Alt+x (press the Alt key—to the left of the space bar on the keyboard—and hold it down while you press x). Another Alt+x displays them again. (If you need even more useable space, you can also suppress the empty line immediately below the top line using Alt+o.)

Entering and changing text To enter new text, simply start to type inside the Nano window. If you make a mistake, the "backspace" key ⇐ will delete the character to the left of the cursor. Use the arrow keys to navigate around the text, for example to change something nearer the beginning. If you type something new, it will appear exactly where the cursor is positioned. The Del key removes the character under the cursor and will cause the remainder of the line (if there is one) to move one position to the left. Everything is really fairly obvious.

Nano and the mouse

Some Nano versions even support a mouse, so—given that you are running Nano on a graphical screen, or your textual environment can handle a mouse—you can click somewhere in your text to place the cursor at that point. You may have to enable the mouse support by pressing Alt+m.

Saving text When you are done entering or editing your text, you can save it using Ctrl+o (hold down Ctrl while pressing o). Nano asks you for a name for the file (on the status line), which you can then enter and finish off with ↩ . (You will find out more about file names in Chapter 6 at the latest.) Nano then stores the text in the named file.

Quitting Nano You can quit Nano using Ctrl+x. If your text contains unsaved modifications, Nano asks you whether the text should be saved; answer y to do that (Nano may ask you for a file name) or n to quit Nano immediately (which will cause your unsaved modifications to be discarded).

Loading files A different (already existing) file can be loaded into your current text using Ctrl+r—it will be inserted at the cursor position. Nano asks you for the name of the file, which you may either enter directly, or alternatively use Ctrl+

t to open the "file browser", which will offer you an interactive choice of existing files. (Incidentally, this also works when saving a file using Ctrl+o.)

Cutting and pasting You may use the Ctrl+k command to remove ("cut") the line containing the cursor and store it in a buffer (*Caution:* Nano will always remove all of the line, no matter where inside the line the cursor is actually positioned!). Ctrl+u will then insert ("paste") the content of the buffer again—either in the same place, if you have pressed Ctrl+k inadvertently or simply wanted to copy the line rather than move it, or elsewhere in your text.

Insertions always happen where the cursor is positioned. Hence if the cursor is in the middle of a line when you hit Ctrl+u, the line from the buffer becomes the right-hand part of that line, and whatever was to the right of the cursor on the original line becomes a new line.

You can move several consecutive lines to the buffer by pressing Ctrl+k a number of times in a row. These lines will then be inserted again *en bloc*.

If you want to cut just part of a line, position the cursor at the corresponding point and press Ctrl+^ (Alt+a on keyboards like some German ones that expect you to type a character to be adorned with a circumflex after you press "^"). Then move the cursor to the end of the material to be cut—Nano helpfully highlights the part of your text that you have selected for cutting—and move the region to the buffer using Ctrl+k. Do note that the character under the cursor itself is *not* cut! Afterwards you can press Ctrl+u as above to insert the buffer content elsewhere.

Searching text If you press Ctrl+w, Nano uses the status line to ask you for a piece of text. The cursor then jumps to the next occurrence of that piece of text in your document, starting at its current position. This makes it convenient to locate specific places in your text.

Online help You can use Ctrl+g to display Nano's internal help screen, which explains the basics of the editor as well as various keyboard commands (there are many more than whe have explained here). Leave the help screen again using Ctrl+x.

These are the most important features of GNU Nano. Practice makes perfect—do feel free to experiment, you will not be able to damage anything.

Back to the topic of `vi` (you may remember—the editor of Linux gurus). If you are game for an adventure, then ensure that the `vim` editor is installed on your system (this is the go-to implementation of `vi` today; hardly anybody uses the original BSD `vi` on Linux), start the `vimtutor` program, and spend an exciting and instructional half hour with the interactive introduction to `vi`. (Depending on your Linux distribution, you may have to install `vimtutor` as a separate package. When in doubt, ask your system administrator or somebody else knowledgeable.)

Exercises

3.6 [!2] Start GNU Nano and enter some simple text—something like

```
Roses are red,
Violets are blue,
Linux is brilliant,
I know this is true.
```

Save this to a file called `roses.txt`

Copyright © 2012 Linup Front GmbH

3.7 [2] Using the text from the previous exercise, cut the line

```
Linux is brilliant,
```

and paste it back three times, so that the text now looks like

```
Roses are red,
Violets are blue,
Linux is brilliant,
Linux is brilliant,
Linux is brilliant,
I know this is true.
```

Then position the cursor on the "i" of "is" in the first of these lines, mark that position, navigate to the "i" of "is" on the third line, and remove the marked region.

Commands in this Chapter

`logout`	Terminates a shell session	`bash`(1)	49
`pico`	Very simple text editor from the PINE/Alpine package	`pico`(1)	52

Summary

- Before using a Linux system, you have to log in giving your user name and password. After using the system, you have to log out again.
- Linux offers various graphical environments, which for the most part work in a similar fashion and fairly intuitively.
- A terminal window allows you to enter textual shell commands within a graphical environment.
- GNU Nano is a simple text editor.

4

Who's Afraid Of The Big Bad Shell?

Contents

Goals

- Appreciating the advantages of a command-line user interface
- Working with Bourne-Again Shell (Bash) commands
- Understanding the structure of Linux commands

Prerequisites

- Basic knowledge of using computers is helpful

4.1 Why?

More so than other modern operating systems, Linux (like Unix) is based on the idea of entering textual commands via the keyboard. This may sound antediluvial to some, especially if one is used to systems like Windows, who have been trying for 15 years or so to brainwash their audience into thinking that graphical user interfaces are the be-all and end-all. For many people who come to Linux from Windows, the comparative prominence of the command line interface is at first a "culture shock" like that suffered by a 21-century person if they suddenly got transported to King Arthur's court – no cellular coverage, bad table manners, and dreadful dentists!

However, things aren't as bad as all that. On the one hand, nowadays there are graphical interfaces even for Linux, which are equal to what Windows or MacOS X have to offer, or in some respects even surpass these as far as convenience and power are concerned. On the other hand, graphical interfaces and the text-oriented command line are not mutually exclusive, but in fact complementary (according to the philosophy "the right tool for every job").

At the end of the day this only means that you as a budding Linux user will do well to *also* get used to the text-oriented user interface, known as the "shell". Of course nobody wants to prevent you from using a graphical desktop for everything you care to do. The shell, however, is a convenient way to perform many extremely powerful operations that are rather difficult to express graphically. To reject the shell is like rejecting all gears except first in your car[1]. Sure, you'll get there eventually even in first gear, but only comparatively slowly and with a horrible amount of noise. So why not learn how to really floor it with Linux? And if you watch closely, we'll be able to show you another trick or two.

4.1.1 What Is The Shell?

Users cannot communicate directly with the operating system kernel. This is only possible through programs accessing it via "system calls". However, you must be able to start such programs in some way. This is the task of the shell, a special user program that (usually) reads commands from the keyboard and interprets them (for example) as commands to be executed. Accordingly, the shell serves as an "interface" to the computer that encloses the actual operating system like a shell (as in "shellfish"—hence the name) and hides it from view. Of course the shell is only one program among many that access the operating system.

> Even today's graphical "desktops" like KDE can be considered "shells". Instead of reading text commands via the keyboard, they read graphical commands via the mouse—but as the text commands follow a certain "grammar", the mouse commands do just the same. For example, you select objects by clicking on them and then determine what to do with them: opening, copying, deleting, …

Bourne shell

C shell

Korn shell

Even the very first Unix—end-1960s vintage—had a shell. The oldest shell to be found outside museums today was developed in the mid-1970s for "Unix version 7" by Stephen L. Bourne. This so-called "Bourne shell" contains most basic functions and was in very wide-spread use, but is very rarely seen in its original form today. Other classic Unix shells include the C shell, created at the University of California in Berkeley and (very vaguely) based on the C programming language, and the largely Bourne-shell compatible, but functionally enhanced, Korn shell (by David Korn, also at AT&T).

Bourne-again shell

Standard on Linux systems is the Bourne-again shell, `bash` for short. It was developed under the auspices of the Free Software Foundation's GNU project by Brian Fox and Chet Ramey and unifies many functions of the Korn and C shells.

[1] This metaphor is for Europeans and other people who can manage a stick shift; our American readers of course all use those wimpy automatic transmissions. It's like they were all running Windows.

Copyright © 2012 Linup Front GmbH

shells: normal programs

Besides the mentioned shells, there are many more. On Unix, a shell is simply an application program like all others, and you need no special privileges to write one—you simply need to adhere to the "rules of the game" that govern how a shell communicates with other programs.

shell scripts

Shells may be invoked interactively to read user commands (normally on a "terminal" of some sort). Most shells can also read commands from files containing pre-cooked command sequences. Such files are called "shell scripts".

A shell performs the following steps:

1. Read a command from the terminal (or the file)
2. Validate the command
3. Run the command directly or start the corresponding program
4. Output the result to the screen (or elsewhere)
5. Continue at step 1.

programming language

In addition to this standard command loop, a shell generally contains further features such as a programming language. This includes complex command structures involving loops, conditions, and variables (usually in shell scripts, less frequently in interactive use). A sophisticated method for recycling recently used commands also makes a user's life easier.

Terminating shell sessions

Shell sessions can generally be terminated using the `exit` command. This also applies to the shell that you obtained immediately after logging in.

Although, as we mentioned, there are several different shells, we shall concentrate here on `bash` as the standard shell on most Linux distributions. The LPI exams also refer to `bash` exclusively.

Exercises

4.1 [2] Log off and on again and check the output of the "`echo $0`" command in the login shell. Start a new shell using the "`bash`" command and enter "`echo $0`" again. Compare the output of the two commands. Do you notice anything unusual?

4.2 Commands

4.2.1 Why Commands?

A computer's operation, no matter which operating system it is running, can be loosely described in three steps:

1. The computer waits for user input
2. The user selects a command and enters it via the keyboard or mouse
3. The computer executes the command

In a Linux system, the shell displays a "prompt", meaning that commands can be entered. This prompt usually consists of a user and host (computer) name, the current directory, and a final character:

```
joe@red:/home > _
```

In this example, user `joe` works on computer `red` in the `/home` directory.

4.2.2 Command Structure

A command is essentially a sequence of characters which is ends with a press of the ↩ key and is subsequently evaluated by the shell. Many commands are vaguely inspired by the English language and form part of a dedicated "command language". Commands in this language must follow certain rules, a "syntax", for the shell to be able to interpret them.

syntax

words
First word: command
parameters

To interpret a command line, the shell first tries to divide the line into words. Just like in real life, words are separated by spaces. The first word on a line is usually the actual command. All other words on the line are parameters that explain what is wanted in more detail.

⚠ DOS and Windows users may be tripped up here by the fact that the shell distinguishes between uppercase and lowercase letters. Linux commands are usually spelled in lowercase letters only (exceptions prove the rule) and not understood otherwise. See also Section 4.2.4.

💡 When dividing a command into words, one space character is as good as many – the difference does not matter to the shell. In fact, the shell does not even insist on spaces; tabulator characters are also allowed, which is however mostly of importance when reading commands from files, since the shell will not let you enter tab character directly (not without jumping through hoops, anyway).

💡 You may even use the line terminator (↩) to distribute a long command across several input lines, but you must put a "Token\" immediately in front of it so the shell will not consider your command finished already.

A command's parameters can be roughly divided into two types:

options

- Parameters starting with a dash ("`-`") are called options. These are usually, er, optional—the details depend on the command in question. Figuratively spoken they are "switches" that allow certain aspects of the command to be switched on or off. If you want to pass several options to a command, they can (often) be accumulated behind a single dash, i. e., the options sequence "`-a -l -F`" corresponds to "`-alF`". Many programs have more options than can be conveniently mapped to single characters, or support "long options" for readability (frequently in addition to equivalent single-character options). Long options most often start with two dashes and cannot be accumulated: "`foo --bar --baz`".

arguments

- Parameters with no leading dash are called arguments. These are often the names of files that the command should process.

command structure

The general command structure can be displayed as follows:

- Command—"What to do?"
- Options—"How to do it?"
- Arguments—"What to do it with?"

Usually the options follow the command and precede the arguments. However, not all commands insist on this—with some, arguments and options can be mixed arbitrarily, and they behave as if all options came immediately after the command. With others, options are taken into account only when they are encountered while the command line is processed in sequence.

⚠ The command structure of current Unix systems (including Linux) has grown organically over a period of almost 40 years and thus exhibits various inconsistencies and small surprises. We too believe that there ought to be a thorough clean-up, but 30 years' worth of shell scripts are difficult to ignore completely … Therefore be prepared to get used to little weirdnesses every so often.

4.2.3 Command Types

In shells, there are essentially two kinds of commands:

Internal commands These commands are made available by the shell itself. The Bourne-again shell contains approximately 30 such commands, which can be executed very quickly. Some commands (such as `exit` or `cd`) alter the state of the shell itself and thus cannot be provided externally.

External commands The shell does not execute these commands by itself but launches executable files, which within the file system are usually found in directories like `/bin` or `/usr/bin`. As a user, you can provide your own programs, which the shell will execute like all other external commands.

External or internal?

You can use the `type` command to find out the type of a command. If you pass a command name as the argument, it outputs the type of command or the corresponding file name, such as

```
$ type echo
echo is a shell builtin
$ type date
date is /bin/date
```

(`echo` is an interesting command which simply outputs its parameters:

```
$ echo Thou hast it now, king, Cawdor, Glamis, all
Thou hast it now, king, Cawdor, Glamis, all
```

`date` displays the current date and time, possibly adjusted to the current time zone and language setup:

```
$ date
Mon May  7 15:32:03 CEST 2012
```

You will find out more about `echo` and `date` in Chapter 9.)

help

You can obtain help for internal Bash commands via the `help` command:

```
$ help type
type: type [-afptP] name [name ...]
  For each NAME, indicate how it would be interpreted if used as a
  command name.

  If the -t option is used, `type' outputs a single word which is one of
  `alias', `keyword', `function', `builtin', `file' or `', if NAME is an
◁◁◁◁◁
```

Exercises

4.2 [2] With bash, which of the following programs are provided externally and which are implemented within the shell itself: `alias`, `echo`, `rm`, `test`?

4.2.4 Even More Rules

As mentioned above, the shell distinguishes between uppercase and lowercase letters when commands are input. This does not apply to commands only, but consequentially to options and parameters (usually file names) as well.

space character

Furthermore, you should be aware that the shell treats certain characters in the input specially. Most importantly, the already-mentioned space character is used to separate words on teh command line. Other characters with a special meaning include

```
$&;(){}[]*?!<>"'
```

“Escaping” characters

If you want to use any of these characters without the shell interpreting according to its the special meaning, you need to “escape” it. You can use the backslash “\” to escape a single special character or else single or double quotes (`'…'`, `"…"`) to excape several special characters. For example:

```
$ touch 'New File'
```

Due to the quotes this command applies to a single file called `New File`. Without quotes, two files called `New` and `File` would have been involved.

We can’t explain all the other special characters here. Most of them will show up elsewhere in this manual – or else check the Bash documentation.

Commands in this Chapter

bash	The “Bourne-Again-Shell”, an interactive command interpreter	bash(1)	58
date	Displays the date and time	date(1)	61
echo	Writes all its parameters to standard output, separated by spaces	bash(1), echo(1)	61
help	Displays on-line help for `bash` commands	bash(1)	61
type	Determines the type of command (internal, external, alias)	bash(1)	61

Summary

- The shell reads user commands and executes them.
- Most shells have programming language features and support shell scripts containing pre-cooked command sequences.
- Commands may have options and arguments. Options determine *how* the command operates, and arguments determine what it operates *on*.
- Shells differentiate between *internal* commands, which are implemented in the shell itself, and *external* commands, which correspond to executable files that are started in separate processes.

5

Getting Help

Contents

Goals

- Being able to handle manual and info pages
- Knowing about and finding HOWTOs
- Being familiar with the most important other information sources

Prerequisites

- Linux Overview
- Basic command-line Linux usage (e. g., from the previous chapters)

5.1 Self-Help

Linux is a powerful and intricate system, and powerful and intricate systems are, as a rule, complex. Documentation is an important tool to manage this complexity, and many (unfortunately not all) aspects of Linux are documented very extensively. This chapter describes some methods to access this documentation.

"Help" on Linux in many cases means "self-help". The culture of free software implies not unnecessarily imposing on the time and goodwill of other people who are spending their free time in the community by asking things that are obviously explained in the first few paragraphs of the manual. As a Linux user, you do well to have at least an overview of the available documentation and the ways of obtaining help in cases of emergency. If you do your homework, you will usually experience that people will help you out of your predicament, but any tolerance towards lazy individuals who expect others to tie themselves in knots on their behalf, on their own time, is not necessarily very pronounced.

If you would like to have somebody listen around the clock, seven days a week, to your not-so-well-researched questions and problems, you will have to take advantage of one of the numerous "commercial" support offerings. These are available for all common distributions and are offered either by the distribution vendor themselves or else by third parties. Compare the different service vendors and pick one whose service level agreements and pricing suit you.

5.2 The `help` Command and the `--help` Option

Internal `bash` commands

In `bash`, internal commands are described in more detail by the `help` command, giving the command name in question as an argument:

```
$ help exit
exit: exit [n]
    Exit the shell with a status of N.
    If N is omitted, the exit status
    is that of the last command executed.
$ _
```

More detailed explanations are available from the shell's manual page and info documentation. These information sources will be covered later in this chapter.

Many external commands (programs) support a `--help` option instead. Most commands display a brief listing of their parameters and syntax.

Not every command reacts to `--help`; frequently the option is called `-h` or `-?`, or help will be output if you specify any invalid option or otherwise illegal command line. Unfortunately there is no universal convention.

5.3 The On-Line Manual

5.3.1 Overview

Nearly every command-line program comes with a "manual page" (or "man page"), as do many configuration files, system calls etc. These texts are generally installed with the software, and can be perused with the "`man` ⟨*name*⟩" command.

Command `man`

Copyright © 2012 Linup Front GmbH

Table 5.1: Manual page sections

Section	Content
NAME	Command name and brief description
SYNOPSIS	Description of the command syntax
DESCRIPTION	Verbose description of the command's effects
OPTIONS	Available options
ARGUMENTS	Available Arguments
FILES	Auxiliary files
EXAMPLES	Sample command lines
SEE ALSO	Cross-references to related topics
DIAGNOSTICS	Error and warning messages
COPYRIGHT	Authors of the command
BUGS	Known limitations of the command

Here, ⟨*name*⟩ is the command or file name that you would like explained. "`man bash`", for example, produces a list of the aforementioned internal shell commands.

However, the manual pages have some disadvantages: Many of them are only available in English; there are sets of translations for different languages which are often incomplete. Besides, the explanations are frequently very complex. Every single word can be important, which does not make the documentation accessible to beginners. In addition, especially with longer documents the structure can be obscure. Even so, the value of this documentation cannot be underestimated. Instead of deluging the user with a large amount of paper, the on-line manual is always available with the system.

Many Linux distributions pursue the philosophy that there should be a manual page for every command that can be invoked on the command line. This does not apply to the same extent to programs belonging to the graphical desktop environments KDE and GNOME, many of which not only do not come with a manual page at all, but which are also very badly documented even inside the graphical environment itself. The fact that many of these programs have been contributed by volunteers is only a weak excuse.

5.3.2 Structure

Man page outline

The structure of the man pages loosely follows the outline given in Table 5.1, even though not every manual page contains every section mentioned there. In particular, the EXAMPLES are frequently given short shrift.

The BUGS heading is often misunderstood: Read *bugs* within the implementation get fixed, of course; what is documented here are usually restrictions which follow from the *approach* the command takes, which are not able to be lifted with reasonable effort, and which you as a user ought to know about. For example, the documentation for the `grep` command points out that various constructs in the regular expression to be located may lead to the `grep` process using very much memory. This is a consequence of the way `grep` implements searching and not a trivial, easily fixed error.

Man pages are written in a special input format which can be processed for text display or printing by a program called `groff`. Source code for the manual pages is stored in the `/usr/share/man` directory in subdirectories called `man`*n*, where *n* is one of the chapter numbers from Table 5.2.

You can integrate man pages from additional directories by setting the `MANPATH` environment variable, which contains the directories which will be searched by `man`, in order. The `manpath` command gives hints for setting up `MANPATH`.

Copyright © 2012 Linup Front GmbH

Table 5.2: Manual Page Topics

No.	Topic
1	User commands
2	System calls
3	C language library functions
4	Device files and drivers
5	Configuration files and file formats
6	Games
7	Miscellaneous (e. g. `groff` macros, ASCII tables, …)
8	Administrator commands
9	Kernel functions
n	"'New"' commands

5.3.3 Chapters

Chapters

Every manual page belongs to a "chapter" of the conceptual "manual" (Table 5.2). Chapters 1, 5 and 8 are most important. You can give a chapter number on the `man` command line to narrow the search. For example, "`man 1 crontab`" displays the man page for the `crontab` command, while "`man 5 crontab`" explains the format of `crontab` files. When referring to man pages, it is customary to append the chapter number in parentheses; we differentiate accordingly between `crontab`(1), the `crontab` command manual, and `crontab`(5), the description of the file format.

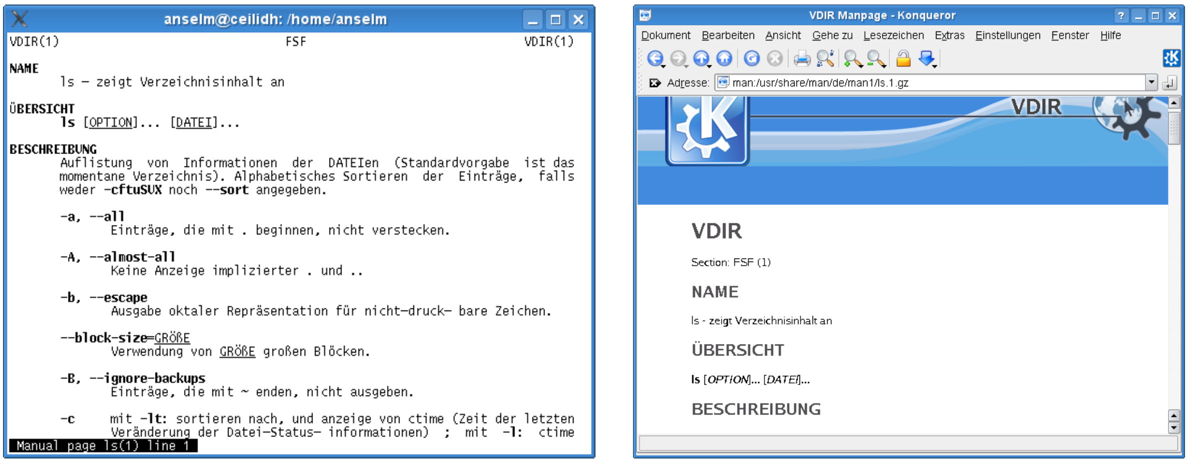
`man -a`

With the `-a` option, `man` displays all man pages matching the given name; without this option, only the first page found (generally from chapter 1) will be displayed.

5.3.4 Displaying Manual Pages

The program actually used to display man pages on a text terminal is usually `less`, which will be discussed in more detail later on. At this stage it is important to know that you can use the cursor keys ↑ and ↓ to navigate within a man page. You can search for keywords inside the text by pressing / —after entering the word and pressing the return key, the cursor jumps to the next occurrence of the word (if it does occur at all). Once you are happy, you can quit the display using q to return to the shell.

Using the KDE web browser, Konqueror, it is convenient to obtain nicely formatted man pages. Simply enter the URL "`man:/`⟨*name*⟩" (or even "`#`⟨*name*⟩")



anselm@ceilidh: /home/anselm

```
VDIR(1)                          FSF                            VDIR(1)

NAME
       ls - zeigt Verzeichnisinhalt an

ÜBERSICHT
       ls [OPTION]... [DATEI]...

BESCHREIBUNG
       Auflistung  von  Informationen  der  DATEIen  (Standardvorgabe  ist das
       momentane Verzeichnis). Alphabetisches Sortieren  der  Einträge,  falls
       weder -cftuSUX noch --sort angegeben.

       -a, --all
              Einträge, die mit . beginnen, nicht verstecken.

       -A, --almost-all
              Keine Anzeige implizierter . und ..

       -b, --escape
              Ausgabe oktaler Repräsentation für nicht-druck- bare Zeichen.

       --block-size=GRÖßE
              Verwendung von GRÖßE großen Blöcken.

       -B, --ignore-backups
              Einträge, die mit ~ enden, nicht ausgeben.

       -c     mit -lt: sortieren nach, und anzeige von ctime (Zeit der letzten
              Veränderung der Datei-Status- informationen)  ;  mit  -l:  ctime
 Manual page ls(1) line 1
```

VDIR Manpage - Konqueror

man:/usr/share/man/de/man1/ls.1.gz

VDIR

Section: FSF (1)

NAME

ls - zeigt Verzeichnisinhalt an

ÜBERSICHT

ls [*OPTION*]... [*DATEI*]...

BESCHREIBUNG

Figure 5.1: A manual page in a text terminal (left) and in Konqueror (right)

Copyright © 2012 Linup Front GmbH

in the browser's address line. This also works on the KDE command line

Keyword search

Before rummaging aimlessly through innumerable man pages, it is often sensible to try to access general information about a topic via `apropos`. This command works just like "`man -k`"; both search the "NAME" sections of all man pages for a keyword given on the command line. The output is a list of all manual pages containing the keyword in their name or description.

`whatis`

A related command is `whatis`. This also searches all manual pages, but for a command (file, …) *name* rather than a keyword—in other words, the part of the "NAME" section to the left of the dash. This displays a brief description of the desired command, system call, etc.; in particular the second part of the "NAME" section of the manual page(s) in question. `whatis` is equivalent to "`man -f`".

Exercises

5.1 [!1] View the manual page for the `ls` command. Use the text-based `man` command and—if available—the Konqueror browser.

5.2 [2] Which manual pages on your system deal (at least according to their "NAME" sections) with processes?

5.3 [5] (Advanced.) Use a text editor to write a manual page for a hypothetical command. Read the `man`(7) man page beforehand. Check the appearance of the man page on the screen (using "`groff -Tascii -man` ⟨*file*⟩ `| less`") and as printed output (using something like "`groff -Tps -man` ⟨*file*⟩ `| gv -`").

5.4 Info Pages

hypertext

For some commands—often more complicated ones—there are so-called "info pages" instead of (or in addition to) the more usual man pages. These are usually more extensive and based on the principles of hypertext, similar to the World Wide Web.

The idea of info pages originated with the GNU project; they are therefore most frequently found with software published by the FSF or otherwise belonging to the GNU project. Originally there was supposed to be *only* info documentation for the "GNU system"; however, since GNU also takes on board lots of software not created under the auspices of the FSF, and GNU tools are being used on systems pursuing a more conservative approach, the FSF has relented in many cases.

Analogously to man pages, info pages are displayed using the "`info` ⟨*command*⟩" command (the package containing the `info` program may have to be installed explicitly). Furthermore, info pages can be viewed using the `emacs` editor or displayed in the KDE web browser, Konqueror, via URLs like "`info:/`⟨*command*⟩".

One advantage of info pages is that, like man pages, they are written in a source format which can conveniently be processed either for on-screen display or for printing manuals using PostScript or PDF. Instead of `groff`, the TEX typesetting program is used to prepare output for printing.

Exercises

5.4 [!1] Look at the info page for the `ls` program. Try the text-based `info` browser and, if available, the Konqueror browser.

5.5 [2] Info files use a crude (?) form of hypertext, similar to HTML files on the World Wide Web. Why aren't info files written in HTML to begin with?

Copyright © 2012 Linup Front GmbH

5.5 HOWTOs

Problem-oriented documentation

Both manual and info pages share the problem that the user must basically know the name of the program to use. Even searching with `apropos` is frequently nothing but a game of chance. Besides, not every problem can be solved using one single command. Accordingly, there is "problem-oriented" rather than "command-oriented" documentation is often called for. The HOWTOs are designed to help with this.

HOWTOs are more extensive documents that do not restrict themselves to single commands in isolation, but try to explain complete approaches to solving problems. For example, there is a "DSL HOWTO" detailing ways to connect a Linux system to the Internet via DSL, or an "Astronomy HOWTO" discussing astronomy software for Linux. Many HOWTOs are available in languages other than English, even though the translations often lag behind the English-language originals.

HOWTO packages

HOWTOs on the Web

Most Linux distributions furnish the HOWTOs (or significant subsets) as packages to be installed locally. They end up in a distribution-specific directory—`/usr/share/doc/howto` for SUSE distributions, `/usr/share/doc/HOWTO` for Debian GNU/Linux—, typically either als plain text or else HTML files. Current versions of all HOWTOs and other formats such as PostScript or PDF can be found on the Web on the site of the "Linux Documentation Project" (`http://www.tldp.org`) which also offers other Linux documentation.

5.6 Further Information Sources

Additional information

You will find additional documentation and example files for (nearly) every installed software package under `/usr/share/doc` or `/usr/share/doc/packages` (depending on your distribution). Many GUI applications (such as those from the KDE or GNOME packages) offer "help" menus. Besides, many distributions offer specialized "help centers" that make it convenient to access much of the documentation on the system.

WWW

USENET

Independently of the local system, there is a lot of documentation available on the Internet, among other places on the WWW and in USENET archives.

Some of the more interesting web sites for Linux include:

`http://www.tldp.org/` The "Linux Documentation Project", which is in charge of man pages and HOWTOs (among other things).

`http://www.linux.org/` A general "portal" for Linux enthusiasts.

`http://www.linuxwiki.de/` A "free-form text information database for everything pertaining to Linux" (in German).

`http://lwn.net/` *Linux Weekly News*—probably the best web site for Linux news of all sorts. Besides a daily overview of the newest developments, products, security holes, Linux advocacy in the press, etc., on Thursdays there is an extensive on-line magazine with well-researched background reports about the preceding week's events. The daily news are freely available, while the weekly issues must be paid for (various pricing levels starting at US-$ 5 per month). One week after their first appearance, the weekly issues are made available for free as well.

`http://freecode.com/` This site publishes announcements of new (predominantly free) software packages, which are often available for Linux. In addition to this there is a database allowing queries for interesting projects or software packages.

`http://www.linux-knowledge-portal.de/` A site collecting "headlines" from other interesting Linux sites, including LWN and Freshmeat.

If there is nothing to be found on the Web or in Usenet archives, it is possible to ask questions in mailing lists or Usenet groups. In this case you should note that many users of these forums consider it very bad form to ask questions answered already in the documentation or in a "FAQ" (frequently answered questions) resource. Try to prepare a detailed description of your problem, giving relevant excerpts of log files, since a complex problem like yours is difficult to diagnose at a distance (and you will surely be able to solve non-complex problems by yourself).

A news archive is accessible on `http://groups.google.com/` (formerly DejaNews)

Interesting *news groups* for Linux can be found in the English-language `comp.os.linux.*` or the German-language `de.comp.os.unix.linux.*` hierarchies. Many Unix groups are appropriate for Linux topics; a question about the shell should be asked in a group dedicated to shell programming rather than a Linux group, since shells are usually not specific to Linux.

Linux-oriented mailing lists can be found, for example, at `majordomo@vger.kernel.org`. You should send an e-mail message including "`subscribe LIST`" to this address in order to subscribe to a list called LIST. A commented list of all available mailing lists on the system may be found at `http://vger.kernel.org/vger-lists.html`.

search engine

An established strategy for dealing with seemingly inexplicable problems is to search for the error message in question using Google (or another search engine you trust). If you do not obtain a helpful result outright, leave out those parts of your query that depend on your specific situation (such as domain names that only exist on your system). The advantage is that Google indexes not just the common web pages, but also many mailing list archives, and chances are that you will encounter a dialogue where somebody else had a problem very like yours.

Free documentation

Incidentally, the great advantage of open-source software is not only the large amount of documentation, but also the fact that most documentation is restricted as little as the software itself. This facilitates collaboration between software developers and documentation authors, and the translation of documentation into different languages is easier. In fact, there is ample opportunity for non-programmers to help with free software projects, e. g., by writing good documentation. The free-software scene should try to give the same respect to documentation authors that it does to programmers—a paradigm shift that has begun but is by no means finished yet.

Commands in this Chapter

`apropos`	Shows all manual pages whose NAME sections contain a given keyword	`apropos`(1)	67
`groff`	Sophisticated typesetting program	`groff`(1)	65, 67
`help`	Displays on-line help for `bash` commands	`bash`(1)	64
`info`	Displays GNU Info pages on a character-based terminal	`info`(1)	67
`less`	Displays texts (such as manual pages) by page	`less`(1)	66
`man`	Displays system manual pages	`man`(1)	64
`manpath`	Determines the search path for system manual pages	`manpath`(1)	65
`whatis`	Locates manual pages with a given keyword in its description	`whatis`(1)	67

Summary

- "`help` ⟨*command*⟩" explains internal `bash` commands. Many external commands support a `--help` option.
- Most programs come with manual pages that can be perused using `man`. `apropos` searches all manual pages for keywords, `whatis` looks for manual page names.
- For some programs, info pages are an alternative to manual pages.
- HOWTOs form a problem-oriented kind of documentation.
- There is a multitude of interesting Linux resources on the World Wide Web and USENET.

6

Files: Care and Feeding

Contents

Goals

- Being familiar with Linux conventions concerning file and directory names
- Knowing the most important commands to work with files and directories
- Being able to use shell filename search patterns

Prerequisites

- Using a shell
- Use of a text editor (qv. Chapter 3)

6.1 File and Path Names

6.1.1 File Names

files

One of the most important services of an operating system like Linux consists of storing data on permanent storage media like hard disks or USB keys and retrieving them later. To make this bearable for humans, similar data are usually collected into "files" that are stored on the medium under a name.

> 💡 Even if this seems trivial to you, it is by no means a given. In former times, some operating systems made it necessary to know abominations like track numbers on a disk in order to retrieve one's data.

Thus, before we can explain to you how to handle files, we need to explain to you how Linux *names* files.

Allowed characters

In Linux file names, you are essentially allowed to use any character that your computer can display (and then some). However, since some of the characters have a special meaning, we would recommend against their use in file names. Only two characters are completely disallowed, the slash and the zero byte (the character with ASCII value 0). Other characters like spaces, umlauts, or dollar signs may be used freely, but must usually be escaped on the command line by means of a backslash or quotes in order to avoid misinterpretations by the shell.

Letter case

> ⚠ An easy trap for beginners to fall into is the fact that Linux distinguishes uppercase and lowercase letters in file names. Unlike Windows, where uppercase and lowercase letters in file names are displayed but treated the same, Linux considers `x-files` and `X-Files` two different file names.

Under Linux, file names may be "quite long"—there is no definite upper bound, since the maximum depends on the "file system", which is to say the specific way bytes are arranged on the medium (there are several methods on Linux). A typical upper limit is 255 characters—but since such a name would take somewhat more than three lines on a standard text terminal this shouldn't really cramp your style.

suffixes

A further difference from DOS and Windows computers is that Linux does not use suffixes to characterise a file's "type". Hence, the dot is a completely ordinary character within a file name. You are free to store a text as `mumble.txt`, but `mumble` would be just as acceptable in principle. This should of course not turn you off using suffixes completely—you do after all make it easier to identify the file content.

> 💡 Some programs insist on their input files having specific suffixes. The C compiler, `gcc`, for example, considers files with names ending in "`.c`" C source code, those ending in "`.s`" assembly language source code, and those ending in "`.o`" precompiled object files.

special characters

You may freely use umlauts and other special characters in file names. However, if files are to be used on other systems it is best to stay away from special characters in file names, as it is not guaranteed that they will show up as the same characters elsewhere.

locale settings

> ⚠ What happens to special characters also depends on your locale settings, since there is no general standard for representing characters exceeding the ASCII character set (128 characters covering mostly the English language, digits and the most common special characters). Widely used encodings are, for example, ISO 8859-1 and ISO 8859-15 (popularly know as ISO-Latin-1 and ISO-Latin-9, respectively … don't ask) as well as ISO 10646, casually and not quite correctly called "Unicod" and usually encoded as "UTF-8". File names you created while encoding *X* was active may look completely different when you look at the directory while encoding *Y* is in force. The whole topic is nothing you want to think about during meals.

Copyright © 2012 Linup Front GmbH

convmv

Should you ever find yourself facing a pile of files whose names are encoded according to the wrong character set, the `convmv` program, which can convert file names between various character encodings, may be able to help you. (You will probably have to install it yourself since it is not part of the standard installation of most distributions.) However, you should really get down to this only after working through the rest of this chapter, as we haven't even explained the regular `mv` yet …

Portable file names

All characters from the following set may be used freely in file names:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789+-._
```

However, you should pay attention to the following hints:

- To allow moving files between Linux and older Unix systems, the length of a file name should be at most 14 characters. (Make that "ancient", really.)
- File names should always start with one of the letters or a digit; the other four characters can be used without problems only inside a file name.

These conventions are easiest to understand by looking at some examples. Allowable file names would be, for instance:

```
X-files
foo.txt.bak
50.something
7_of_9
```

On the contrary, problems would be possible (if not likely or even assured) with:

`-10°F`	*Starts with "-", includes special character*
`.profile`	*Will be hidden*
`3/4-metre`	*Contains illegal character*
`Smörrebröd`	*Contains umlauts*

Hidden files

As another peculiarity, file names starting with a dot ("`.`") will be skipped in some places, for example when the files within a directory are listed—files with such names are considered "hidden". This feature is often used for files containing settings for programs and which should not distract users from more important files in directory listings.

For DOS and Windows experts: These systems allow "hiding" files by means of a "file attribute" which can be set independently of the file's name. Linux and Unix do not support such a thing.

6.1.2 Directories

Since potentially many users may work on the same Linux system, it would be problematic if each file name could occur just once. It would be difficult to make clear to user Joe that he cannot create a file called `letter.txt` since user Sue already has a file by that name. In addition, there must be a (convenient) way of ensuring that Joe cannot read all of Sue's files and the other way round.

For this reason, Linux supports the idea of hierarchical "directories" which are used to group files. File names do not need to be unique within the whole system, but only within the same directory. This means in particular that the system can assign different directories to Joe and Sue, and that within those they may call their files whatever they please without having to worry about each other's files.

Copyright © 2012 Linup Front GmbH

In addition, we can forbid Joe from accessing Sue's *directory* (and vice versa) and no longer need to worry about the individual files within them.

slash

On Linux, directories are simply files, even though you cannot access them using the same methods you would use for "plain" files. However, this implies that the rules we discussed for file names (see the previous section) also apply to the names of directories. You merely need to learn that the slash ("`/`") serves to separate file names from directory names and directory names from one another. `joe/letter.txt` would be the file `letter.txt` in the directory `joe`.

directory tree

Directories may contain other directories (this is the term "hierarchical" we mentioned earlier), which results in a tree-like structure (inventively called a "directory tree"). A Linux system has a special directory which forms the root of the tree and is therefore called the "root directory". Its name is "`/`" (slash).

In spite of its name, the root directory has nothing to do with the system administrator, `root`. It's just that their names are similar.

The slash does double duty here—it serves both as the name of the root directory and as the separator between other directory names. We'll come back to this presently.

The basic installation of common Linux distributions usually contains tens of thousands of files in a directory hierarchy that is mostly structured according to certain conventions. We shall tell you more about this directory hierarchy in Chapter 10.

6.1.3 Absolute and Relative Path Names

absolute path name

Every file in a Linux system is described by a name which is constructed by starting at the root directory and mentioning every directory down along the path to the one containing the file, followed by the name of the file itself. For example, `/home/joe/letter.txt` names the file `letter.txt`, which is located within the `joe` directory, which in turn is located within the `home` directory, which in turn is a direct descendant of the root directory. A name that starts with the root directory is called an "absolute path name"—we talk about "path names" since the name describes a "path" through the directory tree, which may contain directory and file names (i. e., it is a collective term).

relative path names

Each process within a Linux system has a "current directory" (often also called "working directory"). File names are searched within this directory; `letter.txt` is thus a convenient abbreviation for "the file called `letter.txt` in the current directory", and `sue/letter.txt` stands for "the file `letter.txt` within the `sue` directory within the current directory". Such names, which start from the current directory, are called "relative path names".

It is trivial to tell absolute from relative path names: A path name starting with a "`/`" is absolute; all others are relative.

The current directory is "inherited" between parent and child processes. So if you start a new shell (or any program) from a shell, that new shell uses the same current directory as the shell you used to start it. In your new shell, you can change into another directory using the `cd` command, but the current directory of the old shell does not change—if you leave the new shell, you are back to the (unchanged) current directory of the old shell.

shortcuts

There are two convenient shortcuts in relative path names (and even absolute ones): The name "`..`" always refers to the directory *above* the directory in question in the directory tree—for example, in the case of `/home/joe`, `/home`. This frequently allows you to refer conveniently to files in a "side branch" of the directory tree as viewed from the current directory, without having to resort to absolute path names. Assume `/home/joe` has the subdirectories `letters` and `novels`. With `letters` as the current directory, you could refer to the `ivanhoe.txt` file within the `novels` directory by means of the relative path name `../novels/ivanhoe.txt`, without having to use the unwieldy absolute path name `/home/joe/novels/ivanhoe.txt`.

The second shortcut does not make sense quite as obviously: the "`.`" name within a directory always stands for the directory itself. It is not immediately clear why one would need a method to refer to a directory which one has already reached, but there are situations where this comes in quite handy. For example, you may know (or could look up in Chapter 9) that the shell searches program files for external commands in the directories listed in the environment variable `PATH`. If you, as a software developer, want to invoke a program, let's call it `prog`, which (a) resides in a file within the current directory, and (b) this directory is not listed in `PATH` (always a good idea for security reasons), you can still get the shell to start your file as a program by saying

```
$ ./prog
```

without having to enter an absolute path name.

As a Linux user you have a "home directory" which you enter immediately after logging in to the system. The system administrator determines that directory's name when they create your user account, but it is usually called the same as your user name and located below `/home`—something like `/home/joe` for the user `joe`.

6.2 Directory Commands

6.2.1 The Current Directory: `cd` & Co.

Changing directory

You can use the `cd` shell command to change the current directory: Simply give the desired directory as a parameter:

```
$ cd letters                          Change to the letters directory
$ cd ..                               Change to the directory above
```

If you do not give a parameter you will end up in your home directory:

```
$ cd
$ pwd
/home/joe
```

current directory

You can output the absolute path name of the current directory using the `pwd` ("print working directory") command.

prompt

Possibly you can also see the current directory as part of your prompt: Depending on your system settings there might be something like

```
joe@red:~/letters> _
```

where `~/letters` is short for `/home/joe/letters`; the tilde ("~") stands for the current user's home directory.

The "`cd -`" command changes to the directory that used to be current before the most recent `cd` command. This makes it convenient to alternate between two directories.

Exercises

6.1 [2] In the shell, is `cd` an internal or an external command? Why?

6.2 [3] Read about the `pushd`, `popd`, and `dirs` commands in the `bash` man page. Convince yourself that these commands work as described there.

Table 6.1: Some file type designations in `ls`

File type	Colour	Suffix (`ls -F`)	Type letter (`ls -l`)
plain file	black	none	`-`
executable file	green	`*`	`-`
directory	blue	`/`	`d`
link	cyan	`@`	`l`

Table 6.2: Some `ls` options

Option	Result
`-a` or `--all`	Displays hidden files as well
`-i` or `--inode`	Displays the unique file number (inode number)
`-l` or `--format=long`	Displays extra information
`-o` or `--no-color`	Omits colour-coding the output
`-p` or `-F`	Marks file type by adding a special character
`-r` or `--reverse`	Reverses sort order
`-R` or `--recursive`	Recurses into subdirectories (DOS: `DIR/S`)
`-S` or `--sort=size`	Sorts files by size (longest first)
`-t` or `--sort=time`	Sorts file by modification time (newest first)
`-X` or `--sort=extension`	Sorts file by extension ("file type")

6.2.2 Listing Files and Directories—`ls`

To find one's way around the directory tree, it is important to be able to find out which files and directories are located within a directory. The `ls` ("list") command does this.

Tabular format

Without options, this information is output as a multi-column table sorted by file name. With colour screens being the norm rather than the exception today, it has become customary to display the names of files of different types in various colours. (We have not talked about file types yet; this topic will be mentioned in Chapter 10.)

Thankfully, by now most distributions have agreed about the colours to use. Table 6.1 shows the most common assignment.

On monochrome monitors—which can still be found—, the options `-F` or `-p` recommend themselves. These will cause special characters to be appended to the file names according to the file's type. A subset of these characters is given in Table 6.1.

Hidden files

Additional information

You can display hidden files (whose names begin with a dot) by giving the `-a` ("all") option. Another very useful option is `-l` (a lowercase "L", for "long", rather than the digit "1"). This displays not only the file names, but also some additional information about each file.

Some Linux distributions pre-set abbreviations for some combinations of helpful options; the SUSE distributions, for example, use a simple `l` as an abbreviation of "`ls -alF`". "`ll`" and "`la`" are also abbreviations for `ls` variants.

Here is an example of `ls` without and with `-l`:

```
$ ls
file.txt
file2.dat
$ ls -l
```

Copyright © 2012 Linup Front GmbH

```
-rw-r--r--  1  joe users   4711  Oct 4 11:11 file.txt
-rw-r--r--  1  joe users    333  Oct 2 13:21 file2.dat
```

In the first case, all visible (non-hidden) files in the directory are listed; the second case adds the extra information.

Long format

The different parts of the long format have the following meanings: The first character gives the file type (see Chapter 10); plain files have "`-`", directories "`d`" and so on ("type character" in Table 6.1).

The next nine characters show the access permissions. Next there are a reference counter, the owner of the file (`joe` here), and the file's group (`users`). After the size of file in bytes, you can see the date and time of the last modification of the file's content. On the very right there is the file's name.

⚠ Depending on the language you are using, the date and time columns in particular may look completely different than the ones in our example (which we generated using the minimal language environment "`C`"). This is usually not a problem in interactive use, but may prove a major nuisance if you try to take the output of "`ls -l`" apart in a shell script. (Without wanting to anticipate the training manual *Advanced Linux*, we recommend setting the language environment to a defined value in shell scripts.)

If you want to see the extra information for a directory (such as `/tmp`), "`ls -l /tmp`" doesn't really help, because `ls` will list the data for all the files within `/tmp`. Use the `-d` option to suppress this and obtain the information about `/tmp` itself.

`ls` supports many more options than the ones mentioned here; a few of the more important ones are shown in Table 6.2.

In the LPI exams, *Linux Essentials* and LPI-101, nobody expects you to know all 57 varieties of `ls` options by heart. However, you may wish to commit the most import half dozen or so—the content of Table 6.2, approximately—to memory.

Exercises

6.3 [1] Which files does the `/boot` directory contain? Does the directory have subdirectories and, if so, which ones?

6.4 [2] Explain the difference between `ls` with a file name argument and `ls` with a directory name argument.

6.5 [2] How do you tell `ls` to display information about a *directory* rather than the *files* in that directory, if a directory name is passed to the program? (*Hint:* Documentation.)

6.2.3 Creating and Deleting Directories: `mkdir` and `rmdir`

To keep your own files in good order, it makes sense to create new directories. You can keep files in these "folders" according to their subject matter (for example). Of course, for further structuring, you can create further directories within such directories—your ambition will not be curbed by arbitrary limits.

Creating directories

To create new directories, the `mkdir` command is available. It requires one or more directory names as arguments, otherwise you will only obtain an error message instead of a new directory. To create nested directories in a single step, you can use the `-p` option, otherwise the command assumes that all directories in a path name except the last one already exist. For example:

Copyright © 2012 Linup Front GmbH

```
$ mkdir pictures/holiday
mkdir: cannot create directory `pictures/holiday': No such file▷
 ◁ or directory
$ mkdir -p pictures/holiday
$ cd pictures
$ ls -F
holiday/
```

Removing directories

Sometimes a directory is no longer required. To reduce clutter, you can remove it using the rmdir ("remove directory") command.

As with mkdir, at least one path name of a directory to be deleted must be given. In addition, the directories in question must be empty, i. e., they may not contain entries for files, subdirectories, etc. Again, only the last directory in every name will be removed:

```
$ rmdir pictures/holiday
$ ls -F
◁◁◁◁◁
pictures/
◁◁◁◁◁
```

With the -p option, all empty subdirectories mentioned in a name can be removed in one step, beginning with the one on the very right.

```
$ mkdir -p pictures/holiday/summer
$ rmdir pictures/holiday/summer
$ ls -F pictures
pictures/holiday/
$ rmdir -p pictures/holiday
$ ls -F pictures
ls: pictures: No such file or directory
```

Exercises

6.6 [!2] In your home directory, create a directory grd1-test with subdirectories dir1, dir2, and dir3. Change into directory grd1-test/dir1 and create (e. g., using a text editor) a file called hello containing "hello". In grd1-test/dir2, create a file howdy containing "howdy". Check that these files do exist. Delete the subdirectory dir3 using rmdir. Next, attempt to remove the subdirectory dir2 using rmdir. What happens, and why?

6.3 File Search Patterns

6.3.1 Simple Search Patterns

search patterns

You will often want to apply a command to several files at the same time. For example, if you want to copy all files whose names start with "p" and end with ".c" from the prog1 directory to the prog2 directory, it would be quite tedious to have to name every single file explictly—at least if you need to deal with more than a couple of files! It is much more convenient to use the shell's search patterns.

asterisk

If you specify a parameter containing an asterisk on the shell command line—like

```
prog1/p*.c
```

—the shell replaces this parameter in the actual program invocation by a sorted list of all file names that "match" the parameter. "Match" means that in the actual file name there may be an arbitrary-length sequence of arbitrary characters in place of the asterisk. For example, names like

```
prog1/p1.c
prog1/polly.c
prog1/pop-rock.c
prog1/p.c
```

are eligible (note in particular the last name in the example—"arbitrary length" does include "length zero"!). The only character the asterisk will not match is—can you guess it?—the slash; it is usually better to restrict a search pattern like the asterisk to the current directory.

You can test these search patterns conveniently using `echo`. The

```
$ echo prog1/p*.c
```

command will output the matching file names without any obligation or consequence of any kind.

If you really want to apply a command to all files in the *directory tree* starting with a particular directory, there are ways to do that, too. We will discuss this in Section 6.4.4.

All files

The search pattern "*" describes "all files in the current directory"—excepting hidden files whose name starts with a dot. To avert possibly inconvenient surprises, search patterns diligently ignore hidden files unless you explicitly ask for them to be included by means of something like ".*".

You may have encountered the asterisk at the command line of operating systems like DOS or Windows[1] and may be used to specifying the "*.*" pattern to refer to all files in a directory. On Linux, this is not correct—the "*.*" pattern matches "all files whose name contains a dot", but the dot isn't mandatory. The Linux equivalent, as we said, is "*".

question mark

A question mark as a search pattern stands for exactly one arbitrary character (again excluding the slash). A pattern like

```
p?.c
```

thus matches the names

```
p1.c
pa.c
p-.c
p..c
```

(among others). Note that there must be one character—the "nothing" option does not exist here.

You should take particular care to remember a very important fact: *The expansion of search pattern is the responsibility of the shell!* The commands that you execute usually know nothing about search patterns and don't care about them, either. All they get to see are lists of path names, but not where they come from—i. e., whether they have been typed in directly or resulted from the expansion of search patterns.

[1] You're probably too young for CP/M.

Incidentally, nobody says that the results of search patterns always need to be interpreted as path names. For example, if a directory contains a file called "`-l`", a "`ls *`" in that directory will yield an interesting and perhaps surprising result (see Exercise 6.9).

What happens if the shell cannot find a file whose name matches the search pattern? In this case the command in question is passed the search pattern as such; what it makes of that is its own affair. Typically such search patterns are interpreted as file names, but the "file" in question is not found and an error message is issued. However, there are commands that can do useful things with search patterns that you pass them—with them, the challenge is really to ensure that the shell invoking the command does *not* try to cut in with its own expansion. (Cue: quotes)

6.3.2 Character Classes

A somewhat more precise specification of the matching characters in a search pattern is offered by "character classes": In a search pattern of the form

```
prog[123].c
```

the square brackets match exactly those characters that are enumerated within them (no others). The pattern in the example therefore matches

```
prog1.c
prog2.c
prog3.c
```

but not

```
prog.c                  There needs to be exactly one character
prog4.c                                     4 was not enumerated
proga.c                                                a neither
prog12.c                          Exactly one character, please
```

ranges

As a more convenient notation, you may specify ranges as in

```
prog[1-9].c
[A-Z]bracadabra.txt
```

The square brackets in the first line match all digits, the ones in the second all uppercase letters.

Note that in the common character encodings the letters are not contiguous: A pattern like

```
prog[A-z].c
```

not only matches `progQ.c` and `progx.c`, but also `prog_.c`. (Check an ASCII table, e. g. using "`man ascii`".) If you want to match "uppercase and lowercase letters only", you need to use

```
prog[A-Za-z].c
```

A construct like

```
prog[A-Za-z].c
```

does not catch umlauts, even if they look suspiciously like letters.

Copyright © 2012 Linup Front GmbH

negated classes

As a further convenience, you can specify negated character classes, which are interpreted as "all characters except these": Something like

```
prog[!A-Za-z].c
```

matches all names where the character between "`g`" and "`.`" is *not* a letter. As usual, the slash is excepted.

6.3.3 Braces

The expansion of braces in expressions like

```
{red,yellow,blue}.txt
```

is often mentioned in conjunction with shell search patterns, even though it is really just a distant relative. The shell replaces this by

```
red.txt yellow.txt blue.txt
```

In general, a word on the command line that contains several comma-separated pieces of text within braces is replaced by as many words as there are pieces of text between the braces, where in each of these words the whole brace expression is replaced by one of the pieces. *This replacement is purely based on the command line text and is completely independent of the existence or non-existence of any files or directories*—unlike search patterns, which always produce only those names that actually exist as path names on the system.

cartesian product

You can have more than one brace expression in a word, which will result in the cartesian product, in other words all possible combinations:

```
{a,b,c}{1,2,3}.dat
```

produces

```
a1.dat a2.dat a3.dat b1.dat b2.dat b3.dat c1.dat c2.dat c3.dat
```

This is useful, for example, to create new directories systematically; the usual search patterns cannot help there, since they can only find things that already exist:

```
$ mkdir -p revenue/200{8,9}/q{1,2,3,4}
```

Exercises

6.7 [!1] The current directory contains the files

```
prog.c   prog1.c  prog2.c  progabc.c  prog
p.txt    p1.txt   p21.txt  p22.txt    p22.dat
```

Which of these names match the search patterns (a) `prog*.c`, (b) `prog?.c`, (c) `p?*.txt`, (d) `p[12]*`, (e) `p*`, (f) `*.*`?

6.8 [!2] What is the difference between "`ls`" and "`ls *`"? (*Hint:* Try both in a directory containing subdirectories.)

6.9 [2] Explain why the following command leads to the output shown:

Table 6.3: Options for `cp`

Option		Result
`-b`	(*backup*)	Makes backup copies of existing target files by appending a tilde to their names
`-f`	(*force*)	Overwrites existing target files without prompting
`-i`	(engl. *interactive*)	Asks (once per file) whether existing target files should be overwritten
`-p`	(engl. *preserve*)	Tries to preserve all attributes of the source file for the copy
`-R`	(engl. *recursive*)	Copies directories with all their content
`-u`	(engl. *update*)	Copies only if the source file is newer than the target file (or the target file doesn't exist)
`-v`	(engl. *verbose*)	Displays all activity on screen

```
$ ls
-l  file1  file2  file3
$ ls *
-rw-r--r-- 1 joe users 0 Dec 19 11:24 file1
-rw-r--r-- 1 joe users 0 Dec 19 11:24 file2
-rw-r--r-- 1 joe users 0 Dec 19 11:24 file3
```

6.10 [2] Why does it make sense for "`*`" not to match file names starting with a dot?

6.4 Handling Files

6.4.1 Copying, Moving and Deleting—`cp` and Friends

Copying files

You can copy arbitrary files using the `cp` ("copy") command. There are two basic approaches:

1 : 1 copy

If you tell `cp` the source and target file names (two arguments), then a 1 : 1 copy of the content of the source file will be placed in the target file. Normally `cp` does not ask whether it should overwrite the target file if it already exists, but just does it—caution (or the `-i` option) is called for here.

You can also give a target directory name instead of a target file name. The source file will then be copied to that directory, keeping its old name.

```
$ cp list list2
$ cp /etc/passwd .
$ ls -l
-rw-r--r--  1  joe  users   2500  Oct 4 11:11 list
-rw-r--r--  1  joe  users   2500  Oct 4 11:25 list2
-rw-r--r--  1  joe  users   8765  Oct 4 11:26 passwd
```

In this example, we first created an exact copy of file `list` under the name `list2`. After that, we copied the `/etc/passwd` file to the current directory (represented by the dot as a target directory name). The most important `cp` options are listed in Table 6.3.

List of source files

Instead of a single source file, a longer list of source files (or a shell wildcard pattern) is allowed. However, this way it is not possible to copy a file to a different name, but only to a different directory. While in DOS it is possible to use "`COPY *.TXT *.BAK`" to make a backup copy of every `TXT` file to a file with the same name and a `BAK` suffix, the Linux command "`cp *.txt *.bak`" usually fails with an error message.

To understand this, you have to visualise how the shell executes this command. It tries first to replace all wildcard patterns with the corresponding file names, for example `*.txt` by `letter1.txt` and `letter2.txt`. What happens to `*.bak` depends on the expansion of `*.txt` and on whether there are matching file names for `*.bak` in the current directory—but the outcome will almost never be what a DOS user would expect! Usually the shell will pass the `cp` command the unexpanded `*.bak` wildcard pattern as the final argument, which fails from the point of view of `cp` since this is (unlikely to be) an existing directory name.

Move/rename files

While the `cp` command makes an exact copy of a file, physically duplicating the file on the storage medium or creating a new, identical copy on a different storage medium, the `mv` ("move") command serves to move a file to a different place or change its name. This is strictly an operation on directory contents, unless the file is moved to a different file system—for example from a hard disk partition to a USB key. In this case it is necessary to move the file around physically, by copying it to the new place and removing it from the old.

The syntax and rules of `mv` are identical to those of `cp`—you can again specify a list of source files instead of merely one, and in this case the command expects a directory name as the final argument. The main difference is that `mv` lets you rename directories as well as files.

The `-b`, `-f`, `-i`, `-u`, and `-v` options of `mv` correspond to the eponymous ones described with `cp`.

```
$ mv passwd list2
$ ls -l
-rw-r--r--  1  joe  users   2500  Oct 4 11:11 list
-rw-r--r--  1  joe  users   8765  Oct 4 11:26 list2
```

In this example, the original file `list2` is replaced by the renamed file `passwd`. Like `cp`, `mv` does not ask for confirmation if the target file name exists, but overwrites the file mercilessly.

Deleting files

The command to delete files is called `rm` ("remove"). To delete a file, you must have write permission in the corresponding directory. Therefore you are "lord of the manor" in your own home directory, where you can remove even files that do not properly belong to you.

Write permission on a file, on the other hand, is completely irrelevant as far as deleting that file is concerned, as is the question to which user or group the file belongs.

Deleting is forever!

`rm` goes about its work just as ruthlessly as `cp` or `mv`—the files in question are obliterated from the file system without confirmation. You should be especially careful, in particular when shell wildcard patterns are used. Unlike in DOS, the dot in a Linux file name is a character without special significance. For this reason, the "`rm *`" command deletes all non-hidden files from the current directory. Subdirectories will remain unscathed; with "`rm -r *`" they can also be removed.

As the system administrator, you can trash the whole system with a command such as "`rm -rf /`"; utmost care is required! It is easy to type "`rm -rf foo *`" instead of "`rm -rf foo*`".

Where `rm` removes all files whose names are passed to it, "`rm -i`" proceeds a little more carefully:

```
$ rm -i lis*
rm: remove 'list'? n
rm: remove 'list2'? y
$ ls -l
-rw-r--r--  1  joe  users   2500  Oct 4 11:11 list
```

Copyright © 2012 Linup Front GmbH

The example illustrates that, for each file, `rm` asks whether it should be removed ("`y`" for "yes") or not ("`n`" for "no").

Desktop environments such as KDE usually support the notion of a "dustbin" which receives files deleted from within the file manager, and which makes it possible to retrieve files that have been removed inadvertently. There are similar software packages for the command line.

In addition to the `-i` and `-r` options, `rm` allows `cp`'s `-v` and `-f` options, with similar results.

Exercises

6.11 [!2] Create, within your home directory, a copy of the file `/etc/services` called `myservices`. Rename this file to `srv.dat` and copy it to the `/tmp` directory (keeping the new name intact). Remove both copies of the file.

6.12 [1] Why doesn't `mv` have an `-R` option (like `cp` has)?

6.13 [!2] Assume that one of your directories contains a file called "`-file`" (with a dash at the start of the name). How would you go about removing this file?

6.14 [2] If you have a directory where you do not want to inadvertently fall victim to a "`rm *`", you can create a file called "`-i`" there, as in

```
$ > -i
```

(will be explained in more detail in Chapter 8). What happens if you now execute the "`rm *`" command, and why?

6.4.2 Linking Files—`ln` and `ln -s`

Linux allows you to create references to files, so-called "links", and thus to assign several names to the same file. But what purpose does this serve? The applications range from shortcuts for file and directory names to a "safety net" against unwanted file deletions, to convenience for programmers, to space savings for large directory trees that should be available in several versions with only small differences.

The `ln` ("link") command assigns a new name (second argument) to a file in addition to its existing one (first argument):

```
$ ln list list2
$ ls -l
-rw-r--r--  2  joe  users   2500  Oct 4 11:11 list
-rw-r--r--  2  joe  users   2500  Oct 4 11:11 list2
```

A file with multiple names

reference counter

The directory now appears to contain a new file called `list2`. Actually, there are just two references to the same file. This is hinted at by the **reference counter** in the second column of the "`ls -l`" output. Its value is 2, denoting that the file really has two names. Whether the two file names really refer to the same file can only be decided using the "`ls -i`" command. If this is the case, the file number in the first column must be identical for both files. File numbers, also called **inode numbers**, identify files uniquely within their file system:

inode numbers

```
$ ls -i
876543 list  876543 list2
```

Copyright © 2012 Linup Front GmbH

"Inode" is short for "indirection node". Inodes store all the information that the system has about a file, except for the name. There is exactly one inode per file.

If you change the content of one of the files, the other's content changes as well, since in fact there is only one file (with the unique inode number 876543). We only gave that file another name.

Directories are simply tables mapping file names to inode numbers. Obviously there can be several entries in a table that contain different names but the same inode number. A directory entry with a name and inode number is called a "link".

You should realise that, for a file with two links, it is quite impossible to find out which name is "the original", i. e., the first parameter within the `ln` command. From the system's point of view both names are completely equivalent and indistinguishable.

Incidentally, links to directories are not allowed on Linux. The only exceptions are "." and "..", which the system maintains for each directory. Since directories are also files and have their own inode numbers, you can keep track of how the file system fits together internally. (See also Exercise 6.19).

Deleting one of the two files decrements the number of names for file no. 876543 (the reference counter is adjusted accordingly). Not until the reference counter reachers the value of 0 will the file's content actually be removed.

```
$ rm list
$ ls -li
876543 -rw-r--r--  1  joe  users   2500  Oct 4 11:11 list2
```

Since inode numbers are only unique within the same physical file system (disk partition, USB key, …), such links are only possible within the same file system where the file resides.

The explanation about deleting a file's content was not exactly correct: If the last file name is removed, a file can no longer be opened, but if a process is still using the file it can go on to do so until it explicitly closes the file or terminates. In Unix software this is a common idiom for handling temporary files that are supposed to disappear when the program exits: You create them for reading and writing and "delete" them immediately afterwards without closing them within your program. You can then write data to the file and later jump back to the beginning to reread them.

You can invoke `ln` not just with two file name arguments but also with one or with many. In the first case, a link with the same name as the original will be created in the current directory (which should really be different from the one where the file is located), in the second case all named files will be "linked" under their original names into the diréctory given as the last argument (think `mv`).

This is not all, however: There are two different kinds of link in Linux systems. The type explained above is the default case for the `ln` command and is called a "hard link". It always uses a file's inode number for identification. In addition, there are **symbolic links** (also called "soft links" in contrast to "hard links"). Symbolic links are really files containing the name of the link's "target file", together with a flag signifying that the file is a symbolic link and that accesses should be redirected to the target file. Unlike with hard links, the target file does not "know" about the symbolic link. Creating or deleting a symbolic link does not impact the

symbolic links

Copyright © 2012 Linup Front GmbH

target file in any way; when the target file is removed, however, the symbolic link “dangles”, i.e., points nowhere (accesses elicit an error message).

Links to directories

In contrast to hard links, symbolic links allow links to directories as well as files on different physical file systems. In practice, symbolic links are often preferred, since it is easier to keep track of the linkage by means of the path name.

Symbolic links are popular if file or directory names change but a certain backwards compatibility is desired. For example, it was agreed that user mailboxes (that store unread e-mail) should be stored in the `/var/mail` directory. Traditionally, this directory was called `/var/spool/mail`, and many programs hard-code this value internally. To ease a transition to `/var/mail`, a distribution can set up a symbolic link under the name of `/var/spool/mail` which points to `/var/mail`. (This would be impossible using hard links, since hard links to directories are not allowed.)

To create a symbolic link, you must pass the `-s` option to `ln`:

```
$ ln -s /var/log short
$ ls -l
-rw-r--r--  1  joe  users   2500  Oct 4 11:11 liste2
lrwxrwxrwx  1  joe  users     14  Oct 4 11:40 short -> /var/log
$ cd short
$ pwd -P
/var/log
```

Besides the `-s` option to create “soft links”, the `ln` command supports (among others) the `-b`, `-f`, `-i`, and `-v` options discussed earlier on.

To remove symbolic links that are no longer required, delete them using `rm` just like plain files. *This* operation applies to the link rather than the link’s target.

```
$ cd
$ rm short
$ ls
liste2
```

Exercises

6.15 [!2] In your home directory, create a file with arbitrary content (e. g., using “`echo Hello >~/hello`” or a text editor). Create a hard link to that file called `link`. Make sure that the file now has two names. Try changing the file with a text editor. What happens?

6.16 [!2] Create a symbolic link called `~/symlink` to the file in the previous exercise. Check whether accessing the file via the symbolic link works. What happens if you delete the file (name) the symbolic link is pointing to?

6.17 [!2] What directory does the `..` link in the “`/`” directory point to?

6.18 [3] Consider the following command and its output:

```
$ ls -ai /
     2 .      330211 etc         1 proc   4303 var
     2 ..          2 home    65153 root
  4833 bin    244322 lib    313777 sbin
228033 boot   460935 mnt    244321 tmp
330625 dev    460940 opt    390938 usr
```

Obviously, the `/` and `/home` directories have the same inode number. Since the two evidently cannot be the same directory—can you explain this phenomenon?

Copyright © 2012 Linup Front GmbH

Table 6.4: Keyboard commands for `more`

Key	Result
↩	Scrolls up a line
␣	Scrolls up a screenful
b	Scrolls back a screenful
h	Displays help
q	Quits `more`
/ ⟨*word*⟩ ↩	Searches for ⟨*word*⟩
! ⟨*command*⟩ ↩	Executes ⟨*command*⟩ in a subshell
v	Invokes editor (`vi`)
Ctrl + l	Redraws the screen

6.19 [3] We mentioned that hard links to directories are not allowed. What could be a reason for this?

6.20 [3] How can you tell from the output of "`ls -l ~`" that a *subdirectory* of ~ contains no further subdirectories?

6.21 [4] (Brainteaser/research exercise:) What requires more space on disk, a hard link or a symbolic link? Why?

6.4.3 Displaying File Content—`more` and `less`

display of text files

A convenient display of text files on screen is possible using the `more` command, which lets you view long documents page by page. The output is stopped after one screenful, and "`--More--`" appears in the final line (possibly followed by the percentage of the file already displayed). The output is continued after a key press. The meanings of various keys are explained in Table 6.4.

Options

`more` also understands some options. With `-s` ("squeeze"), runs of empty lines are compressed to just one, the `-l` option ignores page ejects (usually represented by "`^L`") which would otherwise stop the output. The `-n` ⟨*number*⟩ option sets the number of screen lines to ⟨*number*⟩, otherwise `more` takes the number from the terminal definition pointed to by `TERM`.

`less`

`more`'s output is still subject to vexing limitations such as the general impossibility of moving back towards the beginning of the output. Therefore, the improved version `less` (a weak pun—think "less is more") is more [sic!] commonly seen today. `less` lets you use the cursor keys to move around the text as usual, the search routines have been extended and allow searching both towards the end as well as towards the beginning of the text. The most common keyboard commands are summarised in Table 6.5.

As mentioned in Chapter 5, `less` usually serves as the display program for manual pages via `man`. All the commands are therefore available when perusing manual pages.

6.4.4 Searching Files—`find`

Who does not know the following feeling: "There used to be a file `foobar` … but where did I put it?" Of course you can tediously sift through all your directories by hand. But Linux would not be Linux if it did not have something handy to help you.

The `find` command searches the directory tree recursively for files matching a set of criteria. "Recursively" means that it considers subdirectories, their subdirectories and so on. `find`'s result consists of the path names of matching files, which can then be passed on to other programs. The following example introduces the command structure:

Table 6.5: Keyboard commands for `less`

Key	Result
↓ or e or j or ↩	Scrolls up one line
f or ␣	Scrolls up one screenful
↑ or y or k	Scrolls back one line
b	Scrolls back one screenful
Home or g	Jumps to the beginning of the text
End or Shift ⇑ + g	Jumps to the end of the text
p ⟨*percent*⟩ ↩	Jumps to position ⟨*percent*⟩ (in %) of the text
h	Displays help
q	Quits `less`
/ ⟨*word*⟩ ↩	Searches for ⟨*word*⟩ towards the end
n	Continues search towards the end
? ⟨*word*⟩ ↩	Searches for ⟨*word*⟩ towards the beginning
Shift ⇑ + n	Continues search towards the beginning
! ⟨*command*⟩ ↩	Executes ⟨*command*⟩ in subshell
v	Invokes editor (`vi`)
r or Ctrl + l	Redraws screen

```
$ find . -user joe -print
./list
```

This searches the current directory including all subdirectories for files belonging to the user `joe`. The `-print` command displays the result (a single file in our case) on the terminal. For convenience, if you do not specify what to do with matching files, `-print` will be assumed.

Note that `find` needs some arguments to go about its task.

Starting Directory The starting directory should be selected with care. If you pick the root directory, the required file(s)—if they exist—will surely be found, but the search may take a long time. Of course you only get to search those files where you have appropriate privileges.

Absolute or relative path names?

An absolute path name for the start directory causes the file names in the output to be absolute, a relative path name for the start directory accordingly produces relative path names.

Directory list

Instead of a single start directory, you can specify a list of directories that will be searched in turn.

Test Conditions These options describe the requirements on the files in detail. Table 6.6 shows the most important tests. The `find` documentation explains many more.

Table 6.6: Test conditions for `find`

Test	Description
`-name`	Specifies a file name pattern. All shell search pattern characters are allowed. The `-iname` option ignores case differences.

Copyright © 2012 Linup Front GmbH

Table 6.7: Logical operators for `find`

Option	Operator	Meaning
`!`	Not	The following test must not match
`-a`	And	Both tests to the left and right of `-a` must match
`-o`	Or	At least one of the tests to the left and right of `-o` must match

Table 6.6: Test conditions for `find`

Test	Description
`-type`	Specifies a file type (see Section 10.2). This includes: `b` block device file `c` character device file `d` directory `f` plain file `l` symbolic link `p` FIFO (named pipe) `s` Unix domain socket
`-user`	Specifies a user that the file must belong to. User names as well as numeric UIDs can be given.
`-group`	Specifies a group that the file must belong to. As with `-user`, a numeric GID can be specified as well as a group name.
`-size`	Specifies a particular file size. Plain numbers signify 512-byte blocks; bytes or kibibytes can be given by appending `c` or `k`, respectively. A preceding plus or minus sign stands for a lower or upper limit; `-size +10k`, for example, matches all files bigger than 10 KiB.
`-atime`	(engl. *access*) allows searching for files based on the time of last access (reading or writing). This and the next two tests take their argument in days; …`min` instead of …`time` produces 1-minute accuracy.
`-mtime`	(engl. *modification*) selects according to the time of modification.
`-ctime`	(engl. *change*) selects according to the time of the last inode change (including access to content, permission change, renaming, etc.)
`-perm`	Specifies a set of permissions that a file must match. The permissions are given as an octal number (see the `chmod` command. To search for a permission in particular, the octal number must be preceded by a minus sign, e.g., `-perm -20` matches all files with group write permission, regardless of their other permissions.
`-links`	Specifies a reference count value that eligible files must match.
`-inum`	Finds links to a file with a given inode number.

Multiple tests

If multiple tests are given at the same time, they are implicitly ANDed together—all of them must match. `find` does support additional logical operators (see Table 6.7).

In order to avoid mistakes when evaluating logical operators, the tests are best enclosed in parentheses. The parentheses must of course be escaped from the shell:

```
$ find . \( -type d -o -name "A*" \) -print
./.
./..
./bilder
./Attic
```

```
$ _
```

This example lists all names that either refer to directories or that begin with “A” or both.

Actions As mentioned before, the search results can be displayed on the screen using the -print option. In addition to this, there are two options, -exec and -ok, which execute commands incorporating the file names. The single difference between -ok and -exec is that -ok asks the user for confirmation before actually executing the command; with -exec, this is tacitly assumed. We will restrict ourselves to discussing -exec.

Executing commands

There are some general rules governing the -exec option:

- The command following -exec must be terminated with a semicolon (“;”). Since the semicolon is a special character in most shells, it must be escaped (e.g., as “\\;” or using quotes) in order to make it visible to find.
- Two braces (“{}”) within the command are replaced by the file name that was found. It is best to enclose the braces in quotes to avoid problems with spaces in file names.

For example:

```
$ find . -user joe -exec ls -l '{}' \;
-rw-r--r--  1  joe  users   4711  Oct 4 11:11 file.txt
$ _
```

This example searches for all files within the current directory (and below) belonging to user test, and executes the “ls -l” command for each of them. The following makes more sense:

```
$ find . -atime +13 -exec rm -i '{}' \;
```

This interactively deletes all files within the current directory (and below) that have not been accessed for two weeks.

Sometimes—say, in the last example above—it is very inefficient to use -exec to start a new process for every single file name found. In this case, the xargs command, which collects as many file names as possible before actually executing a command, can come in useful:

```
$ find . -atime +13 | xargs -r rm -i
```

xargs reads its standard input up to a (configurable) maximum of characters or lines and uses this material as arguments for the specified command (here rm). On input, arguments are separated by space characters (which can be escaped using quotes or “\”) or newlines. The command is invoked as often as necessary to exhaust the input.—The -r option ensures that rm is executed only if find actually sends a file name; otherwise it would be executed at least once.

Weird filenames can get the find/xargs combination in trouble, for example ones that contain spaces or, indeed, newlines which may be mistaken as separators. The silver bullet consists of using the “-print0” option to find, which outputs the file names just as “-print” does, but uses null bytes to separate them instead of newlines. Since the null byte is not a valid character in path names, confusion is no longer possible. xargs must be invoked using the “-0” option to understand this kind of input:

```
$ find . -atime +13 -print0 | xargs -0r rm -i
```

Copyright © 2012 Linup Front GmbH

Exercises

6.22 [!2] Find all files on your system which are longer than 1 MiB, and output their names.

6.23 [2] How could you use `find` to delete a file with an unusual name (e. g., containing invisible control characters or umlauts that older shells cannot deal with)?

6.24 [3] (Second time through the book.) How would you ensure that files in `/tmp` which belong to you are deleted once you log out?

6.4.5 Finding Files Quickly—`locate` and `slocate`

The `find` command searches files according to many different criteria but needs to walk the complete directory tree below the starting directory. Depending on the tree size, this may take considerable time. For the typical application—searching files with particular names—there is an accelerated method.

The `locate` command lists all files whose names match a given shell wildcard pattern. In the most trivial case, this is a simple string of characters:

```
$ locate letter.txt
/home/joe/Letters/letter.txt
/home/joe/Letters/grannyletter.txt
/home/joe/Letters/grannyletter.txt~
◁◁◁◁◁
```

Although `locate` is a fairly important service (as emphasised by the fact that it is part of the LPIC1 curriculum), not all Linux distributions include it as part of the default installation.

For example, if you are using a SUSE distribution, you must explicitly install the `findutils-locate` package before being able to use `locate`.

The ”‘`*`”’, ”‘`?`”’, and ”‘`[…]`”’ characters mean the same thing to `locate` as they do to the shell. But while a query *without* wildcard characters locates all file names that contain the pattern anywhere, a query *with* wildcard characters returns only those names which the pattern describes *completely*—from beginning to end. Therefore pattern queries to `locate` usually start with “`*`”:

```
$ locate "*/letter.t*"
/home/joe/Letters/letter.txt
/home/joe/Letters/letter.tab
◁◁◁◁◁
```

Be sure to put quotes around `locate` queries including shell wildcard characters, to keep the shell from trying to expand them.

The slash (“`/`”) is not handled specially:

```
$ locate Letters/granny
/home/joe/Letters/grannyletter.txt
/home/joe/Letters/grannyletter.txt~
```

`locate` is so fast because it does not walk the file system tree, but checks a “database” of file names that must have been previously created using the `updatedb` program. This means that `locate` does not catch files that have been added to the system since the last database update, and conversely may output the names of files that have been deleted in the meantime.

Copyright © 2012 Linup Front GmbH

You can get `locate` to return existing files only by using the "`-e`" option, but this negates `locate`'s speed advantage.

The `updatedb` program constructs the database for `locate`. Since this may take considerable time, your system administrator usually sets this up to run when the system does not have a lot to do, anyway, presumably late at night.

The `cron` service which is necessary for this will be explained in detail in *Advanced Linux*. For now, remember that most Linux distributions come with a mechanism which causes `updatedb` to be run every so often.

As the system administrator, you can tell `updatedb` which files to consider when setting up the database. How that happens in detail depends on your distribution: `updatedb` itself does not read a configuration file, but takes its settings from the command line and (partly) environment variables. Even so, most distributions call `updatedb` from a shell script which usually reads a file like `/etc/updatedb.conf` or `/etc/sysconfig/locate`, where appropriate environment variables can be set up.

You may find such a file, e.g., in `/etc/cron.daily` (details may vary according to your distribution).

You can, for instance, cause `updatedb` to search certain directories and omit others; the program also lets you specify "network file systems" that are used by several computers and that should have their own database in their root directories, such that only one computer needs to construct the database.

An important configuration setting is the identity of the user that runs `updatedb`. There are essentially two possibilities:

- `updatedb` runs as `root` and can thus enter every file in its database. This also means that users can ferret out file names in directories that they would not otherwise be able to look into, for example, other users' home directories.
- `updatedb` runs with restricted privileges, such as those of user `nobody`. In this case, only names within directories readable by `nobody` end up in the database.

The `slocate` program—an alternative to the usual `locate`—circumvents this problem by storing a file's owner, group and permissions in the database in addition to the file's name. It outputs a file name only if the user who runs `slocate` can, in fact, access the file in question. `slocate` comes with an `updatedb` program, too, but this is merely another name for `slocate` itself.

In many cases, `slocate` is installed such that it can also be invoked using the `locate` command.

Exercises

6.25 [!1] `README` is a very popular file name. Give the absolute path names of all files on your system called `README`.

6.26 [2] Create a new file in your home directory and convince yourself by calling `locate` that this file is not listed (use an appropriately outlandish file name to make sure). Call `updatedb` (possibly with administrator privileges). Does `locate` find your file afterwards? Delete the file and repeat these steps.

6.27 [1] Convince yourself that the `slocate` program works, by searching for files like `/etc/shadow` as normal user.

Copyright © 2012 Linup Front GmbH

Commands in this Chapter

`cd`	Changes a shell's current working directory	`bash`(1)	75
`convmv`	Converts file names between character encodings	`convmv`(1)	72
`cp`	Copies files	`cp`(1)	82
`find`	Searches files matching certain given criteria	`find`(1), Info: `find`	87
`less`	Displays texts (such as manual pages) by page	`less`(1)	87
`ln`	Creates ("hard" or symbolic) links	`ln`(1)	84
`locate`	Finds files by name in a file name database	`locate`(1)	91
`ls`	Lists file information or directory contents	`ls`(1)	75
`mkdir`	Creates new directories	`mkdir`(1)	77
`more`	Displays text data by page	`more`(1)	87
`mv`	Moves files to different directories or renames them	`mv`(1)	83
`pwd`	Displays the name of the current working directory	`pwd`(1), `bash`(1)	75
`rm`	Removes files or directories	`rm`(1)	83
`rmdir`	Removes (empty) directories	`rmdir`(1)	78
`slocate`	Searches file by name in a file name database, taking file permissions into account	`slocate`(1)	92
`updatedb`	Creates the file name database for `locate`	`updatedb`(1)	91
`xargs`	Constructs command lines from its standard input	`xargs`(1), Info: `find`	90

Summary

- Nearly all possible characters may occur in file names. For portability's sake, however, you should restrict yourself to letters, digits, and some special characters.
- Linux distinguishes between uppercase and lowercase letters in file names.
- Absolute path names always start with a slash and mention all directories from the root of the directory tree to the directory or file in question. Relative path names start from the "current directory".
- You can change the current directory of the shell using the `cd` command. You can display its name using `pwd`.
- `ls` displays information about files and directories.
- You can create or remove directories using `mkdir` and `rmdir`.
- The `cp`, `mv` and `rm` commands copy, move, and delete files and directories.
- The `ln` command allows you to create "hard" and "symbolic" links.
- `more` and `less` display files (and command output) by pages on the terminal.
- `find` searches for files or directories matching certain criteria.

Copyright © 2012 Linup Front GmbH

7

Regular Expressions

Contents

Goals

- Understanding and being able to formulate simple and extended regular expressions
- Knowing the `grep` program and its variants, `fgrep` and `egrep`

Prerequisites

- Basic knowledge of Linux, the shell, and Linux commands (e. g., from the preceding chapters)
- Handling of files and directories (Chapter 6)
- Use of a text editor (Chapter 3)

7.1 Regular Expressions: The Basics

Many Linux commands are used for text processing—patterns of the form “do *xyz* for all lines that look like this” appear over and over again. A very powerful tool to describe bits of text, most commonly lines of files, is called “regular expressions”[1]. At first glance, regular expressions resemble the shell’s file name search patterns (Section 6.3), but they work differently and offer more possibilities.

characters

Character classes

complement

Regular expressions are often constructed “recursively” from primitives that are themselves considered regular expressions. The simplest regular expressions are letters, digits and many other characters from the usual character set, which stand for themselves. “a”, for example, is a regular expression matching the “a” character; the regular expression “abc” matches the string “abc”. Character classes can be defined in a manner similar to shell search patterns; therefore, the regular expression “[a-e]” matches exactly one character out of “a” to “e”, and “a[xy]b” matches either “axb” or “ayb”. As in the shell, ranges can be concatenated—“[A-Za-z]” matches all uppercase and lowercase letters—but the complement of a range is constructed slightly differently: “[^abc]” matches all characters *except* “a”, “b”, and “c”. (In the shell, that was “[!abc]”.) The dot, “.”, corresponds to the question mark in shell search patterns, in that it will match a single arbitrary character—the only exception is the newline character, “\n”. Thus, “a.c” matches “abc”, “a/c” and so on, but not the multi-line construction

```
a
c
```

This is due to the fact that most programs operate on a per-line basis, and multi-line constructions would be more difficult to process. (Which is not to say that it wouldn’t sometimes be nice to be able to do it.)

Line start

Line end

While shell search patterns must always match beginning at the start of a file name, in programs selecting lines based on regular expressions it usually suffices if the regular expression matches anywhere in a line. You can restrict this, however: A regular expression starting with a caret (“^”) matches only at the beginning of a line, and a regular expression finishing with a dollar sign (“$”) matches only at the end. The newline character at the end of each line is ignored, so you can use “xyz$” to select all lines ending in “xyz”, instead of having to write “xyz\n$”.

Strictly speaking, “^” and “$” match conceptual “invisible” characters at the beginning of a line and immediately to the left of the newline character at the end of a line, respectively.

Repetition

precedence

Finally, you can use the asterisk (“*”) to denote that the preceding regular expression may be repeated arbitrarily many times (including not at all). The asterisk itself does not stand for any characters in the input, but only modifies the preceding expression—consequently, the shell search pattern “a*.txt” corresponds to the regular expression “^a.*\\.txt” (remember the “anchoring” of the expression to the beginning and end of the input line and that an unescaped dot matches any character). Repetition has precedence over concatenation; “ab*” is a single “a” followed by arbitrarily many “b” (including none at all), not an arbitrary number of repetitions of “ab”.

7.1.1 Regular Expressions: Extras

extensions

The previous section’s explanations apply to nearly all Linux programs that deal with regular expressions. Various programs support different extensions provid-

[1] This is originally a term from computer science and describes a method of characterization of sets of strings that result from the concatenation of “letters”, choices from a set of letters, and their potentially unbounded repetition. Routines to recognize regular expressions are elementary building blocks of many programs such as programming language compilers. Regular expressions appeared very early in the history of Unix; most of the early Unix developers had a computer science background, so the idea was well-known to them.

Copyright © 2012 Linup Front GmbH

ing either notational convenience or additional functionality. The most advanced implementations today are found in modern scripting languages like Tcl, Perl or Python, whose implementations by now far exceed the power of regular expressions in their original computer science sense.

Some common extensions are:

Word brackets The "\<" matches the beginning of a word (a place where a non-letter precedes a letter). Analogously, "\>" matches the end of a word (where a letter is followed by a non-letter).

Grouping Parentheses ("(...)") allow for the repetition of concatenations of regular expressions: "a(bc)*" matches a "a" followed by arbitrarily many repetitions of "bc".

Alternative With the vertical bar ("|") you can select between several regular expressions. The expression "motor (bike | cycle | boat)" matches "motor bike", "motor cycle", and "motor boat" but nothing else.

Optional Expression The question mark ("?") makes the preceding regular expression optional, i. e., it must occur either once or not at all. "ferry(man)?" matches either "ferry" or "ferryman".

At-Least-Once Repetition The plus sign ("+") corresponds to the repetition operator "*", except that the preceding regular expression must occur at least once.

Given Number of Repetitions You can specify a minimum and maximum number of repetitions in braces: "ab{2,4}" matches "abb", "abbb", and "abbbb", but not "ab" or "abbbbb". You may omit the minimum as well as the maximum number; if there is no minimum number, 0 is assumed, if there is no maximum number, "infinity" is assumed.

Back-Reference With an expression like "\\n" you may call for a repetition of that part of the input that matched the parenthetical expression no. *n* in the regular expression. "(ab)\\1", for example, matches "abab", and if, when processing "(ab*a)x\1", the parentheses matched abba, then the whole expression matches abbaxabba (and nothing else). More detail is available in the documentation of GNU grep.

Non-Greedy Matching The "*", "+", and "?" operators are usually "greedy", i. e., they try to match as much of the input as possible: "^a.*a" applied to the input string "abacada" matches "abacada", not "aba" or "abaca". However, there are corresponding "non-greedy" versions "*?", "+?", and "??" which try to match as little of the input as possible. In our example, "^a.*?a" would match "aba". The braces operator may also offer a non-greedy version.

Not every program supports every extension. Table 7.1 shows an overview of the most important programs. Emacs, Perl and Tcl in particular support lots of extensions that have not been discussed here.

7.2 Searching Files for Text—grep

Possibly one of the most important Linux programs using regular expressions is grep. It searches one or more files for lines matching a given regular expression. Matching lines are output, non-matching lines are discarded.

Varieties

There are two varieties of grep: Traditionally, the stripped-down fgrep ("fixed") does not allow regular expressions—it is restricted to character strings—but is very fast. egrep ("extended") offers additional regular expression operators, but is a bit slower and needs more memory.

Copyright © 2012 Linup Front GmbH

Table 7.1: Regular expression support

Extension	GNU `grep`	GNU `egrep`	trad `egrep`	`vim`	`emacs`	Perl	Tcl
Word brackets	●	●	●	●1	●1	●4	●4
Grouping	●1	●	●	●1	●1	●	●
Alternative	●1	●	●	●2	●1	●	●
Option	●1	●	●	●3	●	●	●
At-least-once	●1	●	●	●1	●	●	●
Limits	●1	●	○	●1	●1	●	●
Back-Reference	○	●	●	○	●	●	●
Non-Greedy	○	○	○	●4	●	●	●

●: supported; ○: not supported
Notes: 1. Requires a preceding backslash ("\"), e. g. "`ab\+`" instead of "`ab+`". 2. Needs no parentheses; alternatives always refer to the complete expression. 3. Uses "`\=`" instead of "`?`". 4. Completely different syntax (see documentation).

Table 7.2: Options for `grep` (selected)

Option		Result
`-c`	(*count*)	Outputs just the number of matching lines
`-i`	(*ignore*)	Uppercase and lowercase letters are equivalent
`-l`	(*list*)	Outputs just the names of matching files, no actual matches
`-n`	(*number*)	Includes line numbers of matching lines in the output
`-r`	(*recursive*)	Searches files in subdirectories as well
`-v`	(*invert*)	Outputs only lines that do *not* match the regular expression

These observations used to be true to some extent. In particular, `grep` and `egrep` used completely different algorithms for regular expression evaluation, which could lead to wildly diverging performance results depending on the size and structure of the regular expressions as well as the size of the input. With the common Linux implementation of `grep`, all three variants are, in fact, the same program; they differ mostly in the allowable syntax for their search patterns.

syntax

`grep`'s syntax requires at least a regular expression to search for. This is followed by the name of a text file (or files) to be searched. If no file name is specified, `grep` refers to standard input (see Chapter 8).

regular expression

The regular expression to search in the input may contain, besides the basic regular expressions from Section 7.1, most of the extensions from Section 7.1.1. With `grep`, however, the operators "`\+`", "`\?`", and "`\{`" must be preceded by a backslash. (For `egrep`, this is not necessary.) There are unfortunately no "non-greedy" operators.

You should put the regular expression in single quotes to prevent the shell from trying to expand it, especially if it is more complicated than a simple character string, and definitely if it resembles a shell search pattern.

In addition to the regular expression and file names, various options can be passed on the command line (see Table 7.2).

Search pattern in file

With the `-f` ("file") option, the search pattern can be read from a file. If that file contains several lines, the content of every line will be considered a search pattern in its own right, to be searched simultaneously. This can simplify things considerably especially for frequently used search patterns.

As mentioned above, `fgrep` does not allow regular expressions as search patterns. `egrep`, on the other hand, makes most extensions for regular expressions more conveniently available (Table 7.1).

Copyright © 2012 Linup Front GmbH

Finally some examples for `grep`. The `frog.txt` file contains the Brothers Grimm fairytale of the Frog King (see appendix B). All lines containing the character sequence `frog` can be easily found as follows:

```
$ grep frog frog.txt
frog stretching forth its big, ugly head from the water. »Ah, old
»Be quiet, and do not weep,« answered the frog, »I can help you, but
»Whatever you will have, dear frog,« said she, »My clothes, my pearls
◁◁◁◁◁
```

To find all lines containing exactly the word "`frog`" (and not combinations like "`bullfrog`" or "`frogspawn`"), you need the word bracket extension:

```
$ grep \<frog\> frog.txt
frog stretching forth its big, ugly head from the water. »Ah, old
◁◁◁◁◁
```

(it turns out that this does not in fact make a difference in the English translation). It is as simple to find all lines *beginning* with "`frog`":

```
$ grep ^frog frog.txt
frog stretching forth its big, ugly head from the water. »Ah, old
frog, that he had caused three iron bands to be laid round his heart,
```

A different example: The file `/usr/share/dict/words` contains a list of English words (frequently called the "dictionary")[2]. We're interested in all words containing three or more "`a`":

```
$ grep -n 'a.*a.*a' /usr/share/dict/words
8:aardvark
21:abaca
22:abacate
◁◁◁◁◁                                              … 7030 more words …
234831:zygomaticoauricularis
234832:zygomaticofacial
234834:zygomaticomaxillary
```

(in order: an African animal (*Orycteropus afer*), a banana plant used for fibre (*Musa textilis*), the Brazilian name for the avocado (*Persea sp.*), a facial muscle and two adjectives from the same—medical—area of interest.)

With more complicated regular expressions, it can quickly become unclear why `grep` outputs one line but not another. This can be mitigated to a certain extent by using the `--color` option, which displays the matching part(s) in a file in a particular colour:

```
$ grep --color root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

A command like `export GREP_OPTIONS='--color=auto'` (for example, in `~/.profile`) enables this option on a permanent basis; the `auto` argument suppresses colour output if the output is sent to a pipe or file.

Exercises

7.1 [2] Are the `?` and `+` regular expressions operators really necessary?

[2] The size of the dictionary may vary wildly depending on the distribution.

7.2 [!1] In `frog.txt`, find all lines containing the words "king" or "king's daughter".

7.3 [!2] In `/etc/passwd` there is a list of users on the system (most of the time, anyway). Every line of the file consists of a sequence of fields separated by colons. The last field in each line gives the login shell for that user. Give a `grep` command line to find all users that use `bash` as their login shell.

7.4 [3] Search `/usr/share/dict/words` for all words containing exactly the five vowels "`a`", "`e`", "`i`", "`o`", and "`u`", in that order (possibly with consonants in front, in between, and at the end).

7.5 [4] Give a command to locate and output all lines from the "Frog King" in which a word of at least four letters occurs twice.

Commands in this Chapter

`egrep`	Searches files for lines matching specific regular expressions; extended regular expressions are allowed	`grep`(1)	97
`fgrep`	Searches files for lines with specific content; no regular expressions allowed	`fgrep`(1)	97
`grep`	Searches files for lines matching a given regular expression	`grep`(1)	97

Summary

- Regular expressions are a powerful method for describing sets of character strings.
- `grep` and its relations search a file's content for lines matching regular expressions.

8

Standard I/O and Filter Commands

Contents

Goals

- Mastering shell I/O redirection
- Knowing the most important filter commands

Prerequisites

- Shell operation
- Use of a text editor (see Chapter 3)
- File and directory handling (see Chapter 6)

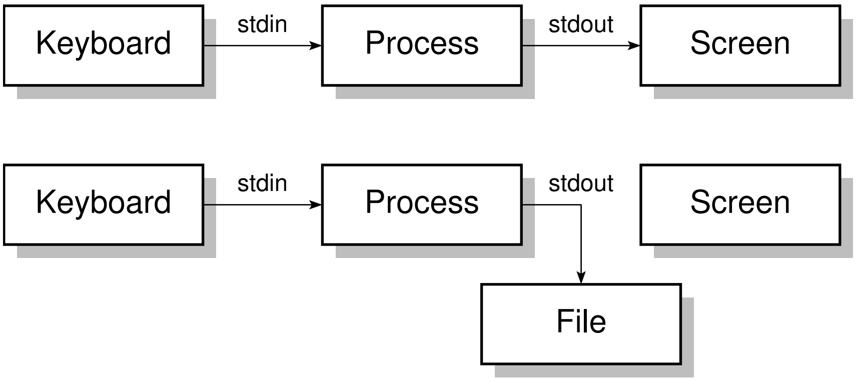


Figure 8.1: Standard channels on Linux

8.1 I/O Redirection and Command Pipelines

8.1.1 Standard Channels

Many Linux commands—like `grep` and friends from Chapter 7—are designed to read input data, manipulate it in some way, and output the result of these manipulations. For example, if you enter

```
$ grep xyz
```

you can type lines of text on the keyboard, and `grep` will only let those pass that contain the character sequence, “xyz”:

```
$ grep xyz
abc def
xyz 123
xyz 123
aaa bbb
YYYxyzZZZ
YYYxyzZZZ
Ctrl+d
```

(The key combination at the end lets `grep` know that the input is at an end.)

standard input
standard output
standard error output

We say that `grep` reads data from “standard input”—in this case, the keyboard—and writes to “standard output”—in this case, the console screen or, more likely, a terminal program in a graphical desktop environment. The third of these “standard channels” is “standard error output”; while the “payload data” `grep` produces are written to standard output, standard error output takes any error messages (e. g., about a non-existent input file or a syntax error in the regular expression).

In this chapter you will learn how to redirect a program’s standard output to a file or take a program’s standard input from a file. Even more importantly, you will learn how to feed one program’s output directly (without the detour via a file) into another program as that program’s input. This opens the door to using the Linux commands, which taken on their own are all fairly simple, as building blocks to construct very complex applications. (Think of a Lego set.)

We will not be able to exhaust this topic in this chapter. Do look forward to the manual, *Advanced Linux*, where constructing shell scripts with the commands from the Unix “toolchest” plays a very important rôle! Here is where you learn the very important fundamentals of cleverly combining Linux commands even on the command line.

Copyright © 2012 Linup Front GmbH

Table 8.1: Standard channels on Linux

Channel	Name	Abbreviation	Device	Use
0	standard input	`stdin`	keyboard	Input for programs
1	standard output	`stdout`	screen	Output of programs
2	standard error output	`stderr`	screen	Programs' error messages

standard channels

The **standard channels** are summarised once more in Table 8.1. In the patois, they are normally referred to using their abbreviated names—`stdin`, `stdout` and `stderr` for standard input, standard output, and standard error output. These channels are respectively assigned the numbers 0, 1, and 2, which we are going to use later on.

Redirection

The shell can redirect these standard channels for individual commands, without the programs in question noticing anything. These always use the standard channels, even though the output might no longer be written to the screen or terminal window but some arbitrary other file. That file could be a different device, like a printer—but it is also possible to specify a text file which will receive the output. That file does not even have to exist but will be created if required.

The standard input channel can be redirected in the same way. A program no longer receives its input from the keyboard, but takes it from the specified file, which can refer to another device or a file in the proper sense.

The keyboard and screen of the "terminal" you are working on (no matter whether this is a Linux text console, a "genuine" terminal on a serial port, a terminal window in a graphical environment, or a network session using, say, the secure shell) can be accessed by means of the `/dev/tty` file—if you want to read data this means the keyboard, for output the screen (the other way round would be quite silly). The

```
$ grep xyz /dev/tty
```

would be equivalent to our example earlier on in this section. You can find out more about such "special files" from Chapter 10.)

8.1.2 Redirecting Standard Channels

Redirecting standard output

You can redirect the standard output channel using the shell operator ">" (the "greater-than" sign). In the following example, the output of "`ls -laF`" is redirected to a file called `filelist`; the screen output consists merely of

```
$ ls -laF >filelist
$ _
```

If the `filelist` file does not exist it is created. Should a file by that name exist, however, its content will be overwritten. The shell arranges for this even before the program in question is invoked—the output file will thus be created even if the actual command invocation contained typos, or if the program did not indeed write any output at all (in which case the `filelist` file will remain empty).

Protecting existing files

If you want to avoid overwriting existing files using shell output redirection, you can give the `bash` command "`set -o noclobber`". In this case, if output is redirected to an existing file, an error occurs.

You can look at the `filelist` file in the usual way, e. g., using `less`:

```
$ less inhalt
total 7
```

```
drwxr-xr-x  12 joe    users     1024 Aug 26 18:55 ./
drwxr-xr-x   5 root   root      1024 Aug 13 12:52 ../
drwxr-xr-x   3 joe    users     1024 Aug 20 12:30 photos/
-rw-r--r--   1 joe    users        0 Sep  6 13:50 filelist
-rw-r--r--   1 joe    users    15811 Aug 13 12:33 pingu.gif
-rw-r--r--   1 joe    users    14373 Aug 13 12:33 hobby.txt
-rw-r--r--   2 joe    users     3316 Aug 20 15:14 chemistry.txt
```

If you look closely at the content of `filelist`, you can see a directory entry for `filelist` with size 0. This is due to the shell's way of doing things: When parsing the command line, it notices the output redirection first and creates a new `filelist` file (or removes its content). After that, the shell executes the command, in this case `ls`, while connecting `ls`'s standard output to the `filelist` file instead of the terminal.

The file's length in the `ls` output is 0 because the `ls` command looked at the file information for `filelist` before anything was written to that file – even though there are three other entries above that of `filelist`. This is because `ls` first reads all directory entries, then sorts them by file name, and only then starts writing to the file. Thus `ls` sees the newly created (or emptied) file `filelist`, with no content so far.

Appending standard output to a file

If you want to append a command's output to an existing file without replacing its previous content, use the `>>` operator. If that file does not exist, it will be created in this case, too.

```
$ date >> filelist
$ less filelist
total 7
drwxr-xr-x  12 joe    users     1024 Aug 26 18:55 ./
drwxr-xr-x   5 root   root      1024 Aug 13 12:52 ../
drwxr-xr-x   3 joe    users     1024 Aug 20 12:30 photos/
-rw-r--r--   1 joe    users        0 Sep  6 13:50 filelist
-rw-r--r--   1 joe    users    15811 Aug 13 12:33 pingu.gif
-rw-r--r--   1 joe    users    14373 Aug 13 12:33 hobby.txt
-rw-r--r--   2 joe    users     3316 Aug 20 15:14 chemistry.txt
Wed Oct 22 12:31:29 CEST 2003
```

In this example, the current date and time was appended to the `filelist` file.

command substitution

Another way to redirect the standard output of a command is by using "backticks" (`…`). This is also called **command substitution**: The standard output of a command in backticks will be inserted into the command line instead of the command (and backticks); whatever results from the replacement will be executed. For example:

```
$ cat dates                                              Our little diary
22/12 Get presents
23/12 Get Christmas tree
24/12 Christmas Eve
$ date +%d/%m                                            What's the date?
23/12
$ grepdàt̂e +%d/%m.` dates                                      What's up?
23/12 Get Christmas tree
```

A possibly more convenient syntax for "`date`" is "`$(date)`". This makes it easier to nest such calls. However, this syntax is only supported by modern shells such as `bash`.

Redirecting standard input

You can use `<`, the "less-than" sign, to redirect the standard input channel. This will read the content of the specified file instead of keyboard input:

Copyright © 2012 Linup Front GmbH

```
$ wc -w <frog.txt
1397
```

In this example, the `wc` filter command counts the words in file `frog.txt`.

There is no `<<` redirection operator to concatenate multiple input files; to pass the content of several files as a command's input you need to use `cat`:

```
$ cat file1 file2 file3 | wc -w
```

(We shall find out more about the "`|`" operator in the next section.) Most programs, however, do accept one or more file names as command line arguments.

Simultaneous redirection

Of course, standard input and standard output may be redirected at the same time. The output of the word-count example is written to a file called `wordcount` here:

```
$ wc -w <frog.txt >wordcount
$ cat wordcount
1397
```

standard error output

Besides the standard input and standard output channels, there is also the standard error output channel. If errors occur during a program's operation, the corresponding messages will be written to that channel. That way you will see them even if standard output has been redirected to a file. If you want to redirect standard error output to a file as well, you must state the channel number for the redirection operator—this is optional for `stdin` (`0<`) and `stdout` (`1>`) but mandatory for `stderr` (`2>`).

You can use the `>&` operator to redirect a channel to a different one:

```
make >make.log 2>&1
```

redirects standard output *and* standard error output of the `make` command to `make.log`.

Watch out: Order is important here! The two commands

```
make >make.log 2>&1
make 2>&1 >make.log
```

lead to completely different results. In the second case, standard error output will be redirected to wherever standard output goes (`/dev/tty`, where standard error output would go anyway), and then standard output will be sent to `make.log`, which, however, does not change the target for standard error output.

Exercises

8.1 [2] You can use the `-U` option to get `ls` to output a directory's entries without sorting them. Even so, after "`ls -laU >filelist`", the entry for `filelist` in the output file gives length zero. What could be the reason?

8.2 [!2] Compare the output of the commands "`ls /tmp`" and "`ls /tmp >ls-tmp.txt`" (where, in the second case, we consider the content of the `ls-tmp.txt` to be the output). Do you notice something? If so, how could you explain the phenomenon?

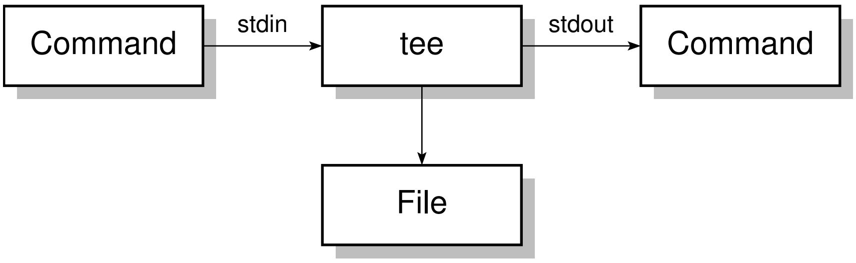


Figure 8.2: The tee command

8.3 [!2] Why isn't it possible to replace a file by a new version in one step, for example using "grep xyz file >file"?

8.4 [!1] And what is wrong with "cat foo >>foo", assuming a non-empty file foo?

8.5 [2] In the shell, how would you output an error message such that it goes to standard error output?

8.1.3 Command Pipelines

Output redirection is frequently used to store the result of a program in order to continue processing it with a different command. However, this type of intermediate storage is not only quite tedious, but you must also remember to get rid of the intermediate files once they are no longer required. Therefore, Linux offers a way of linking commands directly via **pipes**: A program's output automatically becomes another program's input.

pipes

direct connection of several commands

pipeline

This direct connection of several commands into a **pipeline** is done using the | operator. Instead of first redirecting the output of "ls -laF" to a file and then looking at that file using less, you can do the same thing in one step without an intermediate file:

```
$ ls -laF | less
total 7
drwxr-xr-x  12 joe    users     1024 Aug 26 18:55 ./
drwxr-xr-x   5 root   root      1024 Aug 13 12:52 ../
drwxr-xr-x   3 joe    users     1024 Aug 20 12:30 photos/
-rw-r--r--   1 joe    users      449 Sep  6 13:50 filelist
-rw-r--r--   1 joe    users    15811 Aug 13 12:33 pingu.gif
-rw-r--r--   1 joe    users    14373 Aug 13 12:33 hobby.txt
-rw-r--r--   2 joe    users     3316 Aug 20 15:14 chemistry.txt
```

These command pipelines can be almost any length. Besides, the final result can be redirected to a file:

```
$ cut -d: -f1 /etc/passwd | sort | pr -2 >userlst
```

This command pipeline takes all user names from the first comma-separated column of /etc/passwd file, sorts them alphabetically and writes them to the userlst file in two columns. The commands used here will be described in the remainder of this chapter.

intermediate result

Sometimes it is helpful to store the data stream inside a command pipeline at a certain point, for example because the intermediate result at that stage is useful for different tasks. The tee command copies the data stream and sends one copy to standard output and another copy to a file. The command name should be obvious if you know anything about plumbing (see Figure 8.2).

The tee command with no options creates the specified file or overwrites it if it exists; with -a ("append"), the output can be appended to an existing file.

```
$ ls -laF | tee list | less
total 7
drwxr-xr-x  12 joe    users     1024 Aug 26 18:55 ./
drwxr-xr-x   5 root   root      1024 Aug 13 12:52 ../
drwxr-xr-x   3 joe    users     1024 Aug 20 12:30 photos/
-rw-r--r--   1 joe    users      449 Sep  6 13:50 content
-rw-r--r--   1 joe    users    15811 Aug 13 12:33 pingu.gif
-rw-r--r--   1 joe    users    14373 Aug 13 12:33 hobby.txt
-rw-r--r--   2 joe    users     3316 Aug 20 15:14 chemistry.txt
```

In this example the content of the current directory is written both to the `list` file and the screen. (The `list` file does not show up in the `ls` output because it is only created afterwards by `tee`.)

Exercises

8.6 [!2] How would you write the same intermediate result to several files at the same time?

8.2 Filter Commands

toolkit principle

One of the basic ideas of Unix—and, consequently, Linux—is the "toolkit principle". The system comes with a great number of system programs, each of which performs a (conceptually) simple task. These programs can be used as "building blocks" to construct other programs, to save the authors of those programs from having to develop the requisite functions themselves. For example, not every program contains its own sorting routines, but many programs avail themselves of the `sort` command provided by Linux. This modular structure has several advantages:

- It makes life easier for programmers, who do not need to develop (or incorporate) new sorting routines all the time.
- If `sort` receives a bug fix or performance improvement, all programs using `sort` benefit from it, too—and in most cases do not even need to be changed.

Tools that take their input from standard input and write their output to standard output are called "filter commands" or "filters" for short. Without input redirection, a filter will read its input from the keyboard. To finish off keyboard input for such a program, you must enter the key sequence Ctrl+d, which is interpreted as "end of file" by the terminal driver.

Note that the last applies to keyboard input *only*. Files on the disk may of course contain the Ctrl+d character (ASCII 4), without the system believing that the file ended at that point. This as opposed to a certain very popular operating system, which traditionally has a somewhat quaint notion of the meaning of the Control-Z (ASCII 26) character even in text files …

Many "normal" commands, such as the aforementioned `grep`, operate like filters if you do not specify input file names for them to work on.

In the remainder of the chapter you will become familiar with a selection of the most important such commands. Some commands have crept in that are not technically genuine filter commands, but all of them form important building blocks for pipelines.

Table 8.2: Options for `cat` (selection)

Option	Result
`-b`	(engl. *number non-blank lines*) Numbers all non-blank lines in the output, starting at 1.
`-E`	(engl. *end-of-line*) Displays a `$` at the end of each line (useful to detect otherwise invisible space characters).
`-n`	(engl. *number*) Numbers all lines in the output, starting at 1.
`-s`	(engl. *squeeze*) Replaces sequences of empty lines by a single empty line.
`-T`	(engl. *tabs*) Displays tab characters as "`^I`".
`-v`	(engl. *visible*) Makes control characters *c* visible as "`^`*c*", characters *α* with character codes greater than 127 as "`M-`*α*".
`-A`	(engl. *show all*) Same as `-vET`.

8.3 Reading and Writing Files

8.3.1 Outputting and Concatenating Text Files—`cat`

concatenating files

The `cat` ("concatenate") command is really intended to join several files named on the command line into one. If you pass just a single file name, the content of that file will be written to standard output. If you do not pass a file name at all, `cat` reads its standard input—this may seem useless, but `cat` offers options to number lines, make line ends and special characters visible or compress runs of blank lines into one (Table 8.2).

text files

It goes without saying that only text files lead to sensible screen output with `cat`. If you apply the command to other types of files (such as the binary file `/bin/cat`), it is more than probable—on a text terminal at least—that the shell prompt will consist of unreadable characters once the output is done. In this case you can restore the normal character set by (blindly) typing `reset`. If you redirect `cat` output to a file this is of course not a problem.

The "Useless Use of `cat` Award" goes to people using `cat` where it is extraneous. In most cases, commands do accept filenames and don't just read their standard input, so `cat` is not required to pass a single file to them on standard input. A command like "`cat data.txt | grep foo`" is unnecessary if you can just as well write "`grep foo data.txt`". Even if `grep` could only read its standard input, "`grep foo <data.txt`" would be shorter and would not involve an additional `cat` process.

Exercises

8.7 [2] How can you check whether a directory contains files with "weird" names (e. g., ones with spaces at the end or invisible control characters in the middle)?

8.3.2 Beginning and End—`head` and `tail`

Sometimes you are only interested in part of a file: The first few lines to check whether it is the right file, or, in particular with log files, the last few entries. The `head` and `tail` commands deliver exactly that—by default, the first ten and the last ten lines of every file passed as an argument, respectively (or else as usual the first or last ten lines of their standard input). The `-n` option lets you specify a different number of lines: "`head -n 20`" returns the first 20 lines of its standard input, "`tail -n 5 data.txt`" the last 5 lines of file `data.txt`.

Copyright © 2012 Linup Front GmbH

Tradition dictates that you can specify the number *n* of desired lines directly as "-*n*". Officially this is no longer allowed, but the Linux versions of `head` and `tail` still support it.

You can use the `-c` option to specify that the count should be in bytes, not lines: "`head -c 20`" displays the first 20 bytes of standard input, no matter how many lines they occupy. If you append a "`b`", "`k`", or "`m`" (for "blocks", "kibibytes", and "mebibytes", respectively) to the count, the count will be multiplied by 512, 1024, or 1048576, respectively.

`head` also lets you use a minus sign: "`head -c -20`" displays all of its standard input *but* the last 20 bytes.

By way of revenge, `tail` can do something that `head` does not support: If the number of lines starts with "+", it displays everything *starting with* the given line:

```
$ tail -n +3 file                                    Everything from line 3
```

The `tail` command also supports the important `-f` option. This makes `tail` wait after outputting the current end of file, to also output data that is appended later on. This is very useful if you want to keep an eye on some log files. If you pass several file names to `tail -f`, it puts a header line in front of each block of output lines telling what file the new data was written to.

Exercises

8.8 [!2] How would you output just the 13th line of the standard input?

8.9 [3] Check out "`tail -f`": Create a file and invoke "`tail -f`" on it. Then, from another window or virtual console, append something to the file using, e. g., "`echo >>…`", and observe the output of `tail`. What does it look like when `tail` is watching several files simultaneously?

8.10 [3] What happens to "`tail -f`" if the file being observed shrinks?

8.11 [3] Explain the output of the following commands:

```
$ echo Hello >/tmp/hello
$ echo "Hiya World" >/tmp/hello
```

when you have started the command

```
$ tail -f /tmp/hello
```

in a different window after the first `echo` above.

8.4 Data Management

8.4.1 Sorted Files—`sort` and `uniq`

The `sort` command lets you sort the lines of text files according to predetermined criteria. The default setting is ascending (from A to Z) according to the ASCII values[1] of the first few characters of each line. This is why special characters such as German umlauts are frequently sorted incorrectly. For example, the character code of "Ä" is 143, so that character ends up far beyond "Z" with its character code of 91. Even the lowercase latter "a" is considered "greater than" the uppercase letter "Z".

default setting

[1] Of course ASCII only goes up to 127. What is really meant here is ASCII together with whatever extension for the characters with codes from 128 up is currently used, for example ISO-8859-1, also known as ISO-Latin-1.

Of course, sort can adjust itself to different languages and cultures. To sort according to German conventions, set one of the environment variables LANG, LC_ALL, or LC_COLLATE to a value such as "de", "de_DE", or "de_DE@UTF-8" (the actual value depends on your distribution). If you want to set this up for a single sort invocation only, do

```
$ … | LC_COLLATE=de_DE.UTF-8 sort
```

The value of LC_ALL has precedence over the value of LC_COLLATE and that, again, has precedence over the value of LANG. As a side effect, German sort order causes the case of letters to be ignored when sorting.

Sorting by fields

Unless you specify otherwise, the sort proceeds "lexicographically" considering all of the input line. That is, if the initial characters of two lines compare equal, the first differing character within the line governs their relative positioning. Of course sort can sort not just according to the whole line, but more specifically according to the values of certain "columns" or fields of a (conceptual) table. Fields are numbered starting at 1; with the "-k 2" option, the first field would be ignored and the second field of each line considered for sorting. If the values of two lines are equal in the second field, the rest of the line will be looked at, unless you specify the last field to be considered using something like "-k 2,3". Incidentally, it is permissible to specify several -k options with the same sort command.

In addition, sort supports an obsolete form of position specification: Here fields are numbered starting at 0, the initial field is specified as "+*m*" and the final field as "-*n*". To complete the differences to the modern form, the final field is specified "exclusively"—you give the first field that should *not* be taken into account for sorting. The examples above would, respectively, be "+1", "+1 -3", and "+1 -2".

separator

The space character serves as the separator between fields. If several spaces occur in sequence, only the first is considered a separator; the others are considered part of the value of the following field. Here is a little example, namely the list of participants for the annual marathon run of the Lameborough Track & Field Club. To start, we ensure that we use the system's standard language environment ("POSIX") by resetting the corresponding environment variables. (Incidentally, the fourth column gives a runner's bib number.)

```
$ unset LANG LC_ALL LC_COLLATE
$ cat participants.dat
Smith      Herbert  Pantington AC        123 Men
Prowler    Desmond  Lameborough TFC      13  Men
Fleetman   Fred     Rundale Sportsters   217 Men
Jumpabout  Mike     Fairing Track Society 154 Men
de Leaping Gwen     Fairing Track Society 26  Ladies
Runnington Vivian   Lameborough TFC      117 Ladies
Sweat      Susan    Rundale Sportsters   93  Ladies
Runnington Kathleen Lameborough TFC      119 Ladies
Longshanks Loretta  Pantington AC        55  Ladies
O'Finnan   Jack     Fairing Track Society 45  Men
Oblomovsky Katie    Rundale Sportsters   57  Ladies
```

Let's try a list sorted by last name first. This is easy in principle, since the last names are at the front of each line:

```
$ sort participants.dat
Fleetman   Fred     Rundale Sportsters   217 Men
Jumpabout  Mike     Fairing Track Society 154 Men
Longshanks Loretta  Pantington AC        55  Ladies
```

```
O'Finnan   Jack     Fairing Track Society 45  Men
Oblomovsky Katie    Rundale Sportsters    57  Ladies
Prowler    Desmond  Lameborough TFC       13  Men
Runnington Kathleen Lameborough TFC       119 Ladies
Runnington Vivian   Lameborough TFC       117 Ladies
Smith      Herbert  Pantington AC         123 Men
Sweat      Susan    Rundale Sportsters    93  Ladies
de Leaping Gwen     Fairing Track Society 26  Ladies
```

You will surely notice the two small problems with this list: "Oblomovsky" should really be in front of "O'Finnan", and "de Leaping" should end up at the front of the list, not the end. These will disappear if we specify "English" sorting rules:

```
$ LC_COLLATE=en_GB sort participants.dat
de Leaping Gwen     Fairing Track Society 26  Ladies
Fleetman   Fred     Rundale Sportsters    217 Men
Jumpabout  Mike     Fairing Track Society 154 Men
Longshanks Loretta  Pantington AC         55  Ladies
Oblomovsky Katie    Rundale Sportsters    57  Ladies
O'Finnan   Jack     Fairing Track Society 45  Men
Prowler    Desmond  Lameborough TFC       13  Men
Runnington Kathleen Lameborough TFC       119 Ladies
Runnington Vivian   Lameborough TFC       117 Ladies
Smith      Herbert  Pantington AC         123 Men
Sweat      Susan    Rundale Sportsters    93  Ladies
```

(`en_GB` is short for "British English"; `en_US`, for "American English", would also work here.) Let's sort according to the first name next:

```
$ sort -k 2,2 participants.dat
Smith      Herbert  Pantington AC         123 Men
Sweat      Susan    Rundale Sportsters    93  Ladies
Prowler    Desmond  Lameborough TFC       13  Men
Fleetman   Fred     Rundale Sportsters    217 Men
O'Finnan   Jack     Fairing Track Society 45  Men
Jumpabout  Mike     Fairing Track Society 154 Men
Runnington Kathleen Lameborough TFC       119 Ladies
Oblomovsky Katie    Rundale Sportsters    57  Ladies
de Leaping Gwen     Fairing Track Society 26  Ladies
Longshanks Loretta  Pantington AC         55  Ladies
Runnington Vivian   Lameborough TFC       117 Ladies
```

This illustrates the property of `sort` mentioned above: The first of a sequence of spaces is considered the separator, the others are made part of the following field's value. As you can see, the first names are listed alphabetically but only within the same length of last name. This can be fixed using the `-b` option, which treats runs of space characters like a single space:

```
$ sort -b -k 2,2 participants.dat
Prowler    Desmond  Lameborough TFC       13  Men
Fleetman   Fred     Rundale Sportsters    217 Men
Smith      Herbert  Pantington AC         123 Men
O'Finnan   Jack     Fairing Track Society 45  Men
Runnington Kathleen Lameborough TFC       119 Ladies
Oblomovsky Katie    Rundale Sportsters    57  Ladies
de Leaping Gwen     Fairing Track Society 26  Ladies
Longshanks Loretta  Pantington AC         55  Ladies
Jumpabout  Mike     Fairing Track Society 154 Men
```

Table 8.3: Options for `sort` (selection)

Option	Result
`-b` (*blank*)	Ignores leading blanks in field contents
`-d` (*dictionary*)	Sorts in "dictionary order", i. e., only letters, digits and spaces are taken into account
`-f` (*fold*)	Makes uppercase and lowercase letters equivalent
`-i` (*ignore*)	Ignores non-printing characters
`-k` ⟨*field*⟩[`,`⟨*field′*⟩] (*key*)	Sort according to ⟨*field*⟩ (up to and including ⟨*field′*⟩)
`-n` (*numeric*)	Considers field value as a number and sorts according to its numeric value; leading blanks will be ignored
`-o datei` (*output*)	Writes results to a file, whose name may match the original input file
`-r` (*reverse*)	Sorts in descending order, i. e., Z to A
`-t`⟨*char*⟩ (*terminate*)	The ⟨*char*⟩ character is used as the field separator
`-u` (*unique*)	Writes only the first of a sequence of equal output lines

```
Sweat      Susan    Rundale Sportsters    93  Ladies
Runnington Vivian   Lameborough TFC       117 Ladies
```

This sorted list still has a little blemish; see Exercise 8.14.

More detailed field specification

The sort field can be specified in even more detail, as the following example shows:

```
$ sort -br -k 2.2 participants.dat
Sweat      Susan    Rundale Sportsters    93  Ladies
Fleetman   Fred     Rundale Sportsters    217 Men
Longshanks Loretta  Pantington AC         55  Ladies
Runnington Vivian   Lameborough TFC       117 Ladies
Jumpabout  Mike     Fairing Track Society 154 Men
Prowler    Desmond  Lameborough TFC       13  Men
Smith      Herbert  Pantington AC         123 Men
de Leaping Gwen     Fairing Track Society 26  Ladies
Oblomovsky Katie    Rundale Sportsters    57  Ladies
Runnington Kathleen Lameborough TFC       119 Ladies
O'Finnan   Jack     Fairing Track Society 45  Men
```

Here, the `participants.dat` file is sorted in descending order (`-r`) according to the second character of the second table field, i. e., the second character of the first name (very meaningful!). In this case as well it is necessary to ignore leading spaces using the `-b` option. (The blemish from Exercise 8.14 still manifests itself here.)

field separator

With the `-t` ("terminate") option you can select an arbitrary character in place of the field separator. This is a good idea in principle, since the fields then may contain spaces. Here is a more usable (if less readable) version of our example file:

```
Smith:Herbert:Pantington AC:123:Men
Prowler:Desmond:Lameborough TFC:13:Men
Fleetman:Fred:Rundale Sportsters:217:Men
Jumpabout:Mike:Fairing Track Society:154:Men
de Leaping:Gwen:Fairing Track Society:26:Ladies
Runnington:Vivian:Lameborough TFC:117:Ladies
Sweat:Susan:Rundale Sportsters:93:Ladies
Runnington:Kathleen:Lameborough TFC:119:Ladies
Longshanks:Loretta: Pantington AC:55:Ladies
O'Finnan:Jack:Fairing Track Society:45:Men
Oblomovsky:Katie:Rundale Sportsters:57:Ladies
```

Copyright © 2012 Linup Front GmbH

Sorting by first name now leads to correct results using "LC_COLLATE=en_GB sort -t: -k2,2". It is also a lot easier to sort, e. g., by a participant's number (now field 4, no matter how many spaces occur in their club's name:

```
$ sort -t: -k4 participants0.dat
Runnington:Vivian:Lameborough TFC:117:Ladies
Runnington:Kathleen:Lameborough TFC:119:Ladies
Smith:Herbert:Pantington AC:123:Men
Prowler:Desmond:Lameborough TFC:13:Men
Jumpabout:Mike:Fairing Track Society:154:Men
Fleetman:Fred:Rundale Sportsters:217:Men
de Leaping:Gwen:Fairing Track Society:26:Ladies
O'Finnan:Jack:Fairing Track Society:45:Men
Longshanks:Loretta: Pantington AC:55:Ladies
Oblomovsky:Katie:Rundale Sportsters:57:Ladies
Sweat:Susan:Rundale Sportsters:93:Ladies
```

Of course the "number" sort is done lexicographically, unless otherwise specified—"117" and "123" are put before "13", and that in turn before "154". This can be fixed by giving the -n option to force a numeric comparison:

numeric comparison

```
$ sort -t: -k4 -n participants0.dat
Prowler:Desmond:Lameborough TFC:13:Men
de Leaping:Gwen:Fairing Track Society:26:Ladies
O'Finnan:Jack:Fairing Track Society:45:Men
Longshanks:Loretta: Pantington AC:55:Ladies
Oblomovsky:Katie:Rundale Sportsters:57:Ladies
Sweat:Susan:Rundale Sportsters:93:Ladies
Runnington:Vivian:Lameborough TFC:117:Ladies
Runnington:Kathleen:Lameborough TFC:119:Ladies
Smith:Herbert:Pantington AC:123:Men
Jumpabout:Mike:Fairing Track Society:154:Men
Fleetman:Fred:Rundale Sportsters:217:Men
```

These and some more important options for sort are shown in Table 8.3; studying the program's documentation is well worthwhile. sort is a versatile and powerful command which will save you a lot of work.

uniq command

The uniq command does the important job of letting through only the first of a sequence of equal lines in the input (or the last, just as you prefer). What is considered "equal" can, as usual, be specified using options. uniq differs from most of the programs we have seen so far in that it does not accept an arbitrary number of named input files but just one; a second file name, if it is given, is considered the name of the desired output file (if not, standard output is assumed). If no file is named in the uniq call, uniq reads standard input (as it ought).

uniq works best if the input lines are sorted such that *all* equal lines occur one after another. If that is not the case, it is not guaranteed that each line occurs only once in the output:

```
$ cat uniq-test
Hipp
Hopp
Hopp
Hipp
Hipp
Hopp
$ uniq uniq-test
Hipp
Hopp
```

```
Hipp
Hopp
```

Compare this to the output of “`sort -u`”:

```
$ sort -u uniq-test
Hipp
Hopp
```

Exercises

8.12 [!2] Sort the list of participants in `participants0.dat` (the file with colon separators) according to the club’s name and, within clubs, the last and first names of the runners (in that order).

8.13 [3] How can you sort the list of participants by club name in ascending order and, within clubs, by number in descending order? (*Hint:* Read the documentation!)

8.14 [!2] What is the “blemish” alluded to in the examples and why does it occur?

8.15 [2] A directory contains files with the following names:

```
01-2002.txt  01-2003.txt  02-2002.txt  02-2003.txt
03-2002.txt  03-2003.txt  04-2002.txt  04-2003.txt
◁◁◁◁◁
11-2002.txt  11-2003.txt  12-2002.txt  12-2003.txt
```

Give a `sort` command to sort the output of `ls` into “chronologically correct” order:

```
01-2002.txt
02-2002.txt
◁◁◁◁◁
12-2002.txt
01-2003.txt
◁◁◁◁◁
12-2003.txt
```

8.4.2 Columns and Fields—`cut`, `paste` etc.

Cutting columns

While you can locate and “cut out” lines of a text file using `grep`, the `cut` command works through a text file “by column”. This works in one of two ways:

Absolute columns

One possibility is the absolute treatment of columns. These columns correspond to single characters in a line. To cut out such columns, the column number must be given after the `-c` option (“column”). To cut several columns in one step, these can be specified as a comma-separated list. Even column ranges may be specified.

```
$ cut -c 12,1-5 participants.dat
SmithH
ProwlD
FleetF
JumpaM
de LeG
◁◁◁◁◁
```

Copyright © 2012 Linup Front GmbH

In this example, the first letter of the first name and the first five letters of the last name are extracted. It also illustrates the notable fact that the output always contains the columns in the same order as in input. Even if the selected column ranges overlap, every input character is output at most once:

```
$ cut -c 1-5,2-6,3-7 participants.dat
Smith
Prowler
Fleetma
Jumpabo
de Leap
◁◁◁◁◁
```

Relative fields

The second method is to cut relative fields, which are delimited by separator characters. If you want to cut delimited fields, `cut` needs the `-f` (“field”) option and the desired field number. The same rules as for columns apply. The `-c` and `-f` options are mutually exclusive.

separators

The default separator is the tab character; other separators may be specified with the `-d` option (“delimiter”):

```
$ cut -d: -f 1,4 participants0.dat
Smith:123
Prowler:13
Fleetman:217
Jumpabout:154
de Leaping:26
◁◁◁◁◁
```

In this way, the participants’ last names (column 1) and numbers (column 4) are taken from the list. For readability, only the first few lines are displayed.

Incidentally, using the `--output-delimiter` option you can specify a different separator character for the output fields than is used for the input fields:

```
$ cut -d: --output-delimiter=': ' -f 1,4 participants0.dat
Smith: 123
Prowler: 13
Fleetman: 217
Jumpabout: 154
de Leaping: 26
```

If you really want to change the order of columns and fields, you have to bring in the big guns, such as `awk` or `perl`; you could do it using the `paste` command, which will be introduced presently, but that is rather tedious.

Suppressing no-field lines

When files are treated by fields (rather than columns), the `-s` option (“separator”) is helpful. If “`cut -f`” encounters lines that do not contain the separator character, these are normally output in their entirety; `-s` suppresses these lines.

Joining lines of files

The `paste` command joins the lines of the specified files. It is thus frequently used together with `cut`. As you will have noticed immediately, `paste` is not a filter command. You may however give a minus sign in place of one of the input filenames for `paste` to read its standard input at that point. Its output always goes to standard output.

Join files “in parallel”

As we said, `paste` works by lines. If two file names are specified, the first line of the first file and the first of the second are joined (using a tab character as the separator) to form the first line of the output. The same is done with all other lines in the files. To specify a different separator, use the `-d` option.

separator

By way of an example, we can construct a version of the list of marathon runners with the participants’ numbers in front:

```
$ cut -d: -f4 participants0.dat >number.dat
$ cut -d: -f1-3,5 participants0.dat \
>   | paste -d: number.dat - >p-number.dat
$ cat p-number.dat
123:Smith:Herbert:Pantington AC:Men
13:Prowler:Desmond:Lameborough TFC:Men
217:Fleetman:Fred:Rundale Sportsters:Men
154:Jumpabout:Mike:Fairing Track Society:Men
26:de Leaping:Gwen:Fairing Track Society:Ladies
117:Runnington:Vivian:Lameborough TFC:Ladies
93:Sweat:Susan:Rundale Sportsters:Ladies
119:Runnington:Kathleen:Lameborough TFC:Ladies
55:Longshanks:Loretta: Pantington AC:Ladies
45:O'Finnan:Jack:Fairing Track Society:Men
57:Oblomovsky:Katie:Rundale Sportsters:Ladies
```

This file may now conveniently be sorted by number using "sort -n p-number.dat".

Join files serially

With -s ("serial"), the given files are processed in sequence. First, all the lines of the first file are joined into one single line (using the separator character), then all lines from the second file make up the second line of the output etc.

```
$ cat list1
Wood
Bell
Potter
$ cat list2
Keeper
Chaser
Seeker
$ paste -s list*
Wood    Bell    Potter
Keeper  Chaser  Seeker
```

All files matching the list* wildcard pattern—in this case, list1 and list2—are joined using paste. The -s option causes every line of these files to make up one column of the output.

Exercises

8.16 [!2] Generate a new version of the participants.dat file (the one with fixed-width columns) in which the participant numbers and club affiliations do not occur.

8.17 [!2] Generate a new version of the participants0.dat file (the one with fields separated using colons) in which the participant numbers and club affiliations do not occur.

8.18 [3] Generate a version of participants0.dat in which the fields are not separated by colons but by the string ",␣" (a comma followed by a space character).

8.19 [3] How many groups are used as primary groups by users on your system? (The primary group of a user is the fourth field in /etc/passwd.)

Commands in this Chapter

cat	Concatenates files (among other things)	cat(1)	108
cut	Extracts fields or columns from its input	cut(1)	114
head	Displays the beginning of a file	head(1)	108
paste	Joins lines from different input files	paste(1)	115
reset	Resets a terminal's character set to a "reasonable" value	tset(1)	108
sort	Sorts its input by line	sort(1)	109
tail	Displays a file's end	tail(1)	108
uniq	Replaces sequences of identical lines in its input by single specimens	uniq(1)	113

Summary

- Every Linux program supports the standard I/O channels stdin, stdout, and stderr.
- Standard output and standard error output can be redirected using operators > and >>, standard input using operator <.
- Pipelines can be used to connect the standard output and input of programs directly (without intermediate files).
- Using the tee command, intermediate results of a pipeline can be stored to files.
- Filter commands (or "filters") read their standard input, manipulate it, and write the results to standard output.
- sort is a versatile program for sorting.
- The cut command cuts specified ranges of columns or fields from every line of its input.
- With paste, the lines of files can be joined.

9

More About The Shell

Contents

Goals

- Knowing about shell variables and evironment variables

Prerequisites

- Basic shell knowledge (Chapter 4)
- File management and simple filter commands (Chapter 6, Chapter 8)
- Use of a text editor (Chapter 3)

9.1 Simple Commands: `sleep`, `echo`, and `date`

To give you some tools for experiments, we shall now explain some very simple commands:

sleep This command does nothing for the number of seconds specified as the argument. You can use it if you want your shell to take a little break:

```
$ sleep 10
                                   Nothing happens for approximately 10 seconds
$ _
```

Output arguments

echo The command `echo` outputs its arguments (and nothing else), separated by spaces. It is still interesting and useful, since the shell replaces variable references (see Section 9.2) and similar things first:

```
$ p=Planet
$ echo Hello $p
Hello Planet
$ echo Hello ${p}oid
Hello Planetoid
```

(The second `echo` illustrates what to do if you want to append something directly to the value of a variable.)

If `echo` is called with the `-n` option, it does not write a line terminator at the end of its output:

```
$ echo -n Hello
Hello_
```

date and time

date The `date` command displays the current date and time. You have considerable leeway in determining the format of the output—call “`date --help`”, or read the online documentation using “`man date`”.

(When reading through this manual for the second time:) In particular, `date` serves as a world clock, if you first set the `TZ` environment variable to the name of a time zone or important city (usually capital):

```
$ date
Thu Oct  5 14:26:07 CEST 2006
$ export TZ=Asia/Tokyo
$ date
Tue Oct  5 21:26:19 JST 2006
$ unset TZ
```

You can find out about valid time zone and city names by rooting around in `/usr/share/zoneinfo`.

Set the system time

While every user is allowed to read the system time, only the system administrator `root` may change the system time using the `date` command and an argument of the form `MMDDhhmm`, where `MM` is the calendar month, `DD` the calendar day, `hh` the hour, and `mm` the minute. You can optionally add two digits the year (plus possibly another two for the century) and the seconds (separated with a dot), which should, however, prove necessary only in very rare cases.

Copyright © 2012 Linup Front GmbH

```
$ date
Thu Oct  5 14:28:13 CEST 2006
$ date 08181715
date: cannot set date: Operation not permitted
Fri Aug 18 17:15:00 CEST 2006
```

The date command only changes the internal time of the Linux system. This time will not necessarily be transferred to the CMOS clock on the computer's mainboard, so a special command may be required to do so. Many distributions will do this automatically when the system is shut down.

Exercises

9.1 [!3] Assume now is 22 October 2003, 12:34 hours and 56 seconds. Study the date documentation and state formatting instructions to achieve the following output:

1. `22-10-2003`
2. `03-294 (WK43)` (Two-digit year, number of day within year, calendar week)
3. `12h34m56s`

9.2 [!2] What time is it now in Los Angeles?

9.2 Shell Variables and The Environment

Like most common shells, bash has features otherwise found in programming languages. For example, it is possible to store pieces of text or numbers in variables and retrieve them later. Variables also control various aspects of the operation of the shell itself.

Setting variables

Within the shell, a variable is set by means of a command like "foo=bar" (this command sets the foo variable to the textual value bar). Take care *not* to insert spaces in front of or behind the equals sign! You can retrieve the value of the variable by using the variable name with a dollar sign in front:

```
$ foo=bar
$ echo foo
foo
$ echo $foo
bar
```

(note the difference).

environment variables
shell variables

We distinguish **environment variables** from **shell variables**. Shell variables are only visible in the shell in which they have been defined. On the other hand, environment variables are passed to the child process when an external command is started and can be used there. (The child process does not have to be a shell; every Linux process has environment variables). All the environment variables of a shell are also shell variables but not vice versa.

export

Using the export command, you can declare an existing shell variable an environment variable:

```
$ foo=bar                                   foo is now a shell variable
$ export foo                          foo is now an environment variable
```

Or you define a new variable as a shell and environment variable at the same time:

Copyright © 2012 Linup Front GmbH

Table 9.1: Important Shell Variables

Variable	Meaning
PWD	Name of the current directory
EDITOR	Name of the user's favourite editor
PS1	Shell command prompt template
UID	Current user's user name
HOME	Current user's home directory
PATH	List of directories containing executable programs that are eligible as external commands
LOGNAME	Current user's user name (again)

```
$ export foo=bar
```

The same works for several variables simultaneously:

```
$ export foo baz
$ export foo=bar baz=quux
```

You can display all environment variables using the `export` command (with no parameters). The `env` command (also with no parameters) also displays the current environment. All shell variables (including those which are also environment variables) can be displayed using the `set` command. The most common variables and their meanings are shown in Table 9.1.

The `set` command also does many other strange and wonderful things. You will encounter it again in the Linup Front training manual *Advanced Linux*, which covers shell programming.

`env`, too, is actually intended to manipulate the process environment rather than just display it. Consider the following example:

```
$ env foo=bar bash                          Launch child shell with foo
$ echo $foo
bar
$ exit                                         Back to the parent shell
$ echo $foo
                                                            Not defined
$ _
```

At least with `bash` (and relations) you don't really need `env` to execute commands with an extended environment – a simple

```
$ foo=bar bash
```

does the same thing. However, `env` also allows you to remove variables from the environment temporarily (how?).

Delete a variable

If you have had enough of a shell variable, you can delete it using the `unset` command. This also removes it from the environment. If you want to remove a variable from the environment but keep it on as a shell variable, use "`export -n`":

```
$ export foo=bar                         foo is an environment variable
$ export -n foo                           foo is a shell variable (only)
$ unset foo                               foo is gone and lost forever
```

9.3 Command Types – Reloaded

Controlling the shell

One application of shell variables is controlling the shell itself. Here's another example: As we discussed in Chapter 4, the shell distinguishes internal and external commands. External commands correspond to executable programs, which the shell looks for in the directories that make up the value of the PATH environment variable. Here is a typical value for PATH:

```
$ echo $PATH
/home/joe/bin:/usr/local/bin:/usr/bin:/bin:/usr/games
```

Individual directories are separated in the list by colons, therefore the list in the example consists of five directories. If you enter a command like

```
$ ls
```

the shell knows that this isn't an internal command (it knows its internal commands) and thus begins to search the directories in PATH, starting with the leftmost directory. In particular, it checks whether the following files exist:

```
/home/joe/bin/ls                                          Nope …
/usr/local/bin/ls                                  Still no luck …
/usr/bin/ls                                        Again no luck …
/bin/ls                                                   Gotcha!
                              The directory /usr/games is not checked.
```

This implies that the /bin/ls file will be used to execute the ls command.

Of course this search is a fairly involved process, which is why the shell prepares for the future: If it has once identified the /bin/ls file as the implementation of the ls command, it remembers this correspondence for the time being. This process is called "hashing", and you can see that it did take place by applying type to the ls command.

```
$ type ls
ls is hashed (/bin/ls)
```

The hash command tells you which commands your bash has "hashed" and how often they have been invoked in the meantime. With "hash -r" you can delete the shell's complete hashing memory. There are a few other options which you can look up in the bash manual or find out about using "help hash".

Strictly speaking, the PATH variable does not even need to be an environment variable—for the current shell a shell variable would do just fine (see Exercise 9.5). However it is convenient to define it as an environment variable so the shell's child processes (often also shells) use the desired value.

If you want to find out exactly which program the shell uses for a given external command, you can use the which command:

```
$ which grep
/bin/grep
```

which uses the same method as the shell—it starts at the first directory in PATH and checks whether the directory in question contains an executable file with the same name as the desired command.

`which` knows nothing about the shell's internal commands; even though something like "`which test`" returns "`/usr/bin/test`", this does not imply that this program will, in fact, be executed, since internal commands have precedence. If you want to know for sure, you need to use the "`type`" shell command.

The `whereis` command not only returns the names of executable programs, but also documentation (man pages), source code and other interesting files pertaining to the command(s) in question. For example:

```
$ whereis passwd
passwd: /usr/bin/passwd /etc/passwd /etc/passwd.org /usr/share/passwd▷
 ◁ /usr/share/man/man1/passwd.1.gz /usr/share/man/man1/passwd.1ssl.gz▷
 ◁ /usr/share/man/man5/passwd.5.gz
```

This uses a hard-coded method which is explained (sketchily) in `whereis`(1).

Exercises

9.3 [!2] Convince yourself that passing (or not passing) environment and shell variables to child processes works as advertised, by working through the following command sequence:

```
$ foo=bar                                   foo is a shell variable
$ bash                                      New shell (child process)
$ echo $foo
                                            foo is not defined
$ exit                                      Back to the parent shell
$ export foo                                foo is an environment variable
$ bash                                      New shell (child process)
$ echo $foo
bar                                         Environment variable was passed along
$ exit                                      Back to the parent shell
```

9.4 [!2] What happens if you change an environment variable in the child process? Consider the following command sequence:

```
$ foo=bar                                   foo is a shell variable
$ bash                                      New shell (child process)
$ echo $foo
bar                                         Environment variable was passed along
$ foo=baz                                   New value
$ exit                                      Back to the parent shell
$ echo $foo                                 What do we get??
```

9.5 [2] Make sure that the shell's command line search works even if `PATH` is a "only" simple shell variable rather than an environment variable. What happens if you remove `PATH` completely?

9.6 [!1] Which executable programs are used to handle the following commands: `fgrep`, `sort`, `mount`, `xterm`

9.7 [!1] Which files on your system contain the documentation for the "`crontab`" command?

9.4 The Shell As A Convenient Tool

Since the shell is the most often used tool for many Linux users, its developers have spared no trouble to make its use convenient. Here are some more useful trifles:

Command Editor You can edit command lines like in a simple text editor. Hence, you can move the cursor around in the input line and delete or add characters arbitrarily before finishing the input using the return key. The behaviour of this editor can be adapted to that of the most popular editors on Linux (Chapter 3) using the “`set -o vi`” and “`set -o emacs`” commands.

Aborting Commands With so many Linux commands around, it is easy to confuse a name or pass a wrong parameter. Therefore you can abort a command while it is being executed. You simply need to press the Ctrl+c keys at the same time.

The History The shell remembers ever so many of your most recent commands as part of the “history”, and you can move through this list using the ↑ and ↓ cursor keys. If you find a previous command that you like you can either re-execute it unchanged using ↩, or else edit it as described above. You can search the list “incrementally” using Ctrl+r – simply type a sequence of characters, and the shell shows you the most recently executed command containing this sequence. The longer your sequence, the more precise the search.

When you log out of the system, the shell stores the history in the hidden file `~/.bash_history` and makes it available again after your next login. (You may use a different file name by setting the `HISTFILE` variable to the name in question.)

A consequence of the fact that the history is stored in a “plain” file is that you can edit it using a text editor (Chapter 3 tells you how). So in case you accidentally enter your password on the command line, you can (and should!) remove it from the history manually—in particular, if your system is one of the more freewheeling ones where home directories are visible to anybody.

Completing command and file names

Autocompletion A massive convenience is `bash`’s ability to automatically complete command and file names. If you hit the Tab key, the shell completes an incomplete input if the continuation can be identified uniquely. For the first word of a command, `bash` considers all executable programs, within the rest of the command line all the files in the current or specified directory. If several commands or files exist whose names start out equal, the shell completes the name as far as possible and then signals acoustically that the command or file name may still be incomplete. Another Tab press then lists the remaining possibilities.

It is possible to adapt the shell’s completion mechanism to specific programs. For example, on the command line of a FTP client it might offer the names of recently visited FTP servers in place of file names. Check the `bash` documentation for details.

Table 9.2 gives an overview of the most important key strokes within `bash`.

Multiple Commands On One Line You are perfectly free to enter several commands on the same input line. You merely need to separate them using a semicolon:

```
$ echo Today is; date
Today is
Fri  5 Dec 12:12:47 CET 2008
```

In this instance the second command will be executed once the first is done.

Copyright © 2012 Linup Front GmbH

Table 9.2: Key Strokes within bash

Key Stroke	Function
↑ or ↓	Scroll through most recent commands
Ctrl+r	Search command history
← bzw. →	Move cursor within current command line
Home oder Ctrl+a	Jump to the beginning of the command line
End oder Ctrl+e	Jump to the end of the command line
⇐ bzw. Del	Delete character in front of/under the cursor, respectively
Ctrl+t	Swap the two characters in front of and under the cursor
Ctrl+l	Clear the screen
Ctrl+c	Interrupt a command
Ctrl+d	End the input (for login shells: log off)

return value

Conditional Execution Sometimes it is useful to make the execution of the second command depend on whether the first was executed correctly or not. Every Unix process yields a **return value** which states whether it was executed correctly or whether errors of whatever kind have occurred. In the former case, the return value is 0; in the latter, it is different from 0.

You can find the return value of a child process of your shell by looking at the $? variable:

```
$ bash                                          Start a child shell …
$ exit 33                                  … and exit again immediately
exit
$ echo $?
33                                        The value from our exit above
$ _
```

But this really has no bearing on the following.

With && as the "separator" between two commands (where there would otherwise be the semicolon), the second command is only executed when the first has exited successfully. To demonstrate this, we use the shell's -c option, with which you can pass a command to the child shell on the command line (impressive, isn't it?):

```
$ bash -c "exit 0" && echo "Successful"
Successful
$ bash -c "exit 33" && echo "Successful"
                                               Nothing -- 33 isn't success!
```

Conversely, with || as the "separator", the second command is only executed if the first did *not* finish successfully:

```
$ bash -c "exit 0" || echo "Unsuccessful"
$ bash -c "exit 33" || echo "Unsuccessful"
Unsuccessful
```

Exercises

9.8 [3] What is wrong about the command "echo "Hello!""? (*Hint:* Experiment with commands of the form "!-2" or "!ls".)

9.5 Commands From A File

You can store shell commands in a file and execute *en bloc*. (You will learn how to create a file conveniently in Chapter 3.) You just need to invoke the shell and pass the file name as a parameter:

```
$ bash my-commands
```

shell script

Such a file is also called a **shell script**, and the shell has extensive programming features that we can only outline very briefly here. (The Linup Front training manual *Advanced Linux* explains shell programming in great detail.)

You can avoid having to prepend the `bash` command by inserting the magical incantation

```
#!/bin/bash
```

as the first line of your file and making the file "executable":

```
$ chmod +x my-commands
```

(You will find out more about `chmod` and access rights in Chapter 14.) After this, the

```
$ ./my-commands
```

command will suffice.

subshell

If you invoke a shell script as above, whether with a prepended `bash` or as an executable file, it is executed in a subshell, a shell that is a child process of the current shell. This means that changes to, e. g., shell or environment variables do not influence the current shell. For example, assume that the file `assignment` contains the line

```
foo=bar
```

Consider the following command sequence:

```
$ foo=quux
$ bash assignment                                        Contains foo=bar
$ echo $foo
quux                          No change; assignment was only in subshell
```

This is generally considered a feature, but every now and then it would be quite desirable to have commands from a file affect the *current* shell. That works, too: The `source` command reads the lines in a file exactly as if you would type them directly into the current shell—all changes to variables (among other things) hence take effect in your current shell:

```
$ foo=quux
$ source assignment                                      Contains foo=bar
$ echo $foo
bar                                                  Variable was changed!
```

A different name for the `source` command, by the way, is "`.`". (You read correctly – dot!) Hence

```
$ source assignment
```

Copyright © 2012 Linup Front GmbH

is equivalent to

```
$ . assignment
```

Like program files for external commands, the files to be read using source or . are searched in the directories given by the PATH variable.

9.6 The Shell As A Programming Language

Being able to execute shell commands from a file is a good thing, to be sure. However, it is even better to be able to structure these shell commands such that they do not have to do the same thing every time, but—for example—can obtain command-line parameters. The advantages are obvious: In often-used procedures you save a lot of tedious typing, and in seldom-used procedures you can avoid mistakes that might creep in because you accidentally leave out some important step. We do not have space here for a full explanation of the shell als a programming language, but fortunately there is enough room for a few brief examples.

Single parameters

Command-line parameters When you pass command-line parameters to a shell script, the shell makes them available in the variables $1, $2, …. Consider the following example:

```
$ cat hello
#!/bin/bash
echo Hello $1, are you free $2?
$ ./hello Joe today
Hello Joe, are you free today?
$ ./hello Sue tomorrow
Hello Sue, are you free tomorrow?
```

All parameters

The $* contains all parameters at once, and the number of parameters is in $#:

```
$ cat parameter
#!/bin/bash
echo $# parameters: $*
$ ./parameter
0 parameters:
$ ./parameter dog
1 parameters: dog
$ ./parameter dog cat mouse tree
4 parameters: dog cat mouse tree
```

Loops The for command lets you construct loops that iterate over a list of words (separated by white space):

```
$ for i in 1 2 3
> do
>   echo And $i!
> done
And 1!
And 2!
And 3!
```

Here, the i variable assumes each of the listed values in turn as the commands between do and done are executed.

This is even more fun if the words are taken from a variable:

```
$ list='4 5 6'
$ for i in $list
> do
>   echo And $i!
> done
And 4!
And 5!
And 6!
```

Loop over parameters

If you omit the “in …”, the loop iterates over the command line parameters:

```
$ cat sort-wc
#!/bin/bash
# Sort files according to their line count
for f
do
    echo `wc -l <"$f"` lines in $f
done | sort -n
$ ./sort-wc /etc/passwd /etc/fstab /etc/motd
```

(The “wc -l” command counts the lines of its standard input or the file(s) passed on the command line.) Do note that you can redirect the standard output of a *loop* to sort using a pipe line!

Alternatives You can use the aforementioned && and || operators to execute certain commands only under specific circumstances. The

```
#!/bin/bash
# grepcp REGEX
rm -rf backup; mkdir backup
for f in *.txt
do
    grep $1 "$f" && cp "$f" backup
done
```

script, for example, copies a file to the backup directory only if its name ends with .txt (the for loop ensures this) and which contain at least one line matching the regular expression that is passed as a parameter.

test

A useful tool for alternatives is the test command, which can check a large variety of conditions. It returns an exit code of 0 (success), if the condition holds, else a non-zero exit code (failure). For example, consider

```
#!/bin/bash
# filetest NAME1 NAME2 ...
for name
do
   test -d "$name" && echo $name: directory
   test -f "$name" && echo $name: file
   test -L "$name" && echo $name: symbolic link
done
```

This script looks at a number of file names passed as parameters and outputs for each one whether it refers to a directory, a (plain) file, or a symbolic link.

⚠ The test command exists both as a free-standing program in /bin/test and as a built-in command in bash and other shells. These variants can differ subtly especially as far as more outlandish tests are concerned. If in doubt, read the documentation.

Copyright © 2012 Linup Front GmbH

`if` You can use the `if` command to make more than one command depend on a condition (in a convenient and readable fashion). You may write “`[` …`]`” instead of “`test` …”:

```
#!/bin/bash
# filetest2 NAME1 NAME2 ...
for name
do
    if [ -L "$name" ]
    then
        echo $name: symbolic link
    elif [ -d "$name" ]
        echo $name: directory
    elif [ -f "$name" ]
        echo $name: file
    else
        echo $name: no idea
    fi
done
```

If the command after the `if` signals “success” (exit code 0), the commands after `then` will be executed, up to the next `elif`, `else`, or `fi`. If on the other hand it signals “failure”, the command after the next `elif` will be evaluated next and its exit code will be considered. The shell continues the pattern until the matching `fi` is reached. Commands after the `else` are executed if none of the `if` or `elif` commands resulted in “success”. The `elif` and `else` branches may be omitted if they are not required.

More loops With the `for` loop, the number of trips through the loop is fixed at the beginning (the number of words in the list). However, we often need to deal with situations where it is not clear at the beginning how often a loop should be `while` executed. To handle this, the shell offers the `while` loop, which (like `if`) executes a command whose success or failure determines what to do about the loop: On success, the “dependent” commands will be executed, on failure execution will continue after the loop.

The following script reads a file like

```
Aunt Maggie:maggie@example.net:the delightful tea cosy
Uncle Bob:bob@example.com:the great football
```

(whose name is passed on the command line) and constructs a thank-you e-mail message from each line (Linux *is* very useful in daily life):

```
#!/bin/bash
# birthday FILE
IFS=:
while read name email present
do
    (echo $name
     echo ""
     echo "Thank you very much for $present!"
     echo "I enjoyed it very much."
     echo ""
     echo "Best wishes"
     echo "Tim") | mail -s "Many thanks!" $email
done <$1
```

`read` The `read` command reads the input file line by line and splits each line at the colons (variable IFS) into the three fields name, email, and present which are then made available as variables inside the loop. Somewhat counterintuitively, the input redirection for the loop can be found at the very end.

Please test this script with innocuous e-mail addresses only, lest your relations become confused!

Exercises

9.9 [1] What is the difference (as far as loop execution is concerned) between

```
for f; do …; done
```

and

```
for f in $*; do …; done
```

? (Try it, if necessary)

9.10 [2] In the sort-wc script, why do we use the

```
wc -l <$f
```

instead of

```
wc -l $f
```

9.11 [2] Alter the grepcp such that the list of files to be considered is also taken from the command line. (*Hint:* The shift shell command removes the first command line parameter from $ and pulls all others up to close the gap. After a shift, the previous $2 is now $1, $3 is $2 and so on.)

9.12 [2] Why does the filetest script output

```
$ ./filetest foo
foo: file
foo: symbolic link
```

for symbolic links (instead of just »foo: symbolic link«)?

Copyright © 2012 Linup Front GmbH

Commands in this Chapter

`.`	Reads a file containing shell commands as if they had been entered on the command line	`bash`(1)	127
`date`	Displays the date and time	`date`(1)	120
`env`	Outputs the process environment, or starts programs with an adjusted environment	`env`(1)	122
`export`	Defines and manages environment variables	`bash`(1)	121
`hash`	Shows and manages ”‘seen”’ commands in `bash`	`bash`(1)	123
`set`	Manages shell variables and options	`bash`(1)	122
`source`	Reads a file containing shell commands as if they had been entered on the command line	`bash`(1)	127
`test`	Evaluates logical expressions on the command line	`test`(1), `bash`(1)	129
`unset`	Deletes shell or environment variables	`bash`(1)	122
`whereis`	Searches executable programs, manual pages, and source code for given programs	`whereis`(1)	123
`which`	Searches programs along `PATH`	`which`(1)	123

Summary

- The `sleep` command waits for the number of seconds specified as the argument.
- The `echo` command outputs its arguments.
- The date and time may be determined using `date`
- Various `bash` features support interactive use, such as command and file name autocompletion, command line editing, alias names and variables.

10

The File System

Contents

Goals

- Understanding the terms “file” and “file system”
- Recognising the different file types
- Knowing your way around the directory tree of a Linux system
- Knowing how external file systems are integrated into the directory tree

Prerequisites

- Basic Linux knowledge (from the previous chapters)
- Handling files and directories (Chapter 6)

Table 10.1: Linux file types

Type	`ls -l`	`ls -F`	Create using …
plain file	`-`	`name`	diverse programs
directory	`d`	`name/`	`mkdir`
symbolic link	`l`	`name@`	`ln -s`
device file	`b` or `c`	`name`	`mknod`
FIFO (*named pipe*)	`p`	`name|`	`mkfifo`
Unix-domain socket	`s`	`name=`	no command

10.1 Terms

file

Generally speaking, a **file** is a self-contained collection of data. There is no restriction on the type of the data within the file; a file can be a text of a few letters or a multi-megabyte archive containing a user's complete life works. Files do not need to contain plain text. Images, sounds, executable programs and lots of other things can be placed on a storage medium as files. To guess at the type of data contained in a file you can use the `file` command:

`file`

```
$ file /bin/ls /usr/bin/groups /etc/passwd
/bin/ls:         ELF 32-bit LSB executable, Intel 80386,▷
 ◁ version 1 (SYSV), for GNU/Linux 2.4.1,▷
 ◁ dynamically linked (uses shared libs), for GNU/Linux 2.4.1, stripped
/usr/bin/groups: Bourne shell script text executable
/etc/passwd:     ASCII text
```

`file` guesses the type of a file based on rules in the `/usr/share/file` directory. `/usr/share/file/magic` contains a clear-text version of the rules. You can define your own rules by putting them into the `/etc/magic` file. Check `magic`(5) for details.

To function properly, a Linux system normally requires several thousand different files. Added to that are the many files created and owned by the system's various users.

file system

A **file system** determines the method of arranging and managing data on a storage medium. A hard disk basically stores bytes that the system must be able to find again somehow—and as efficiently and flexibly as possible at that, even for very huge files. The details of file system operation may differ (Linux knows lots of different file systems, such as `ext2`, `ext3`, `ext4`, ReiserFS, XFS, JFS, btrfs, …) but what is presented to the user is largely the same: a tree-structured hierarchy of file and directory names with files of different types. (See also Chapter 6.)

In the Linux community, the term "file system" carries several meanings. In addition to the meaning presented here—"method of arranging bytes on a medium"—, a file system is often considered what we have been calling a "directory tree". In addition, a specific medium (hard disk partition, USB key, …) together with the data on it is often called a "file system"—in the sense that we say, for example, that hard links (Section 6.4.2) do not work "across file system boundaries", that is, between two different partitions on hard disk or between the hard disk and a USB key.

10.2 File Types

Linux systems subscribe to the basic premise "Everything is a file". This may seem confusing at first, but is a very useful concept. Six file types may be distinguished in principle:

Copyright © 2012 Linup Front GmbH

Plain files This group includes texts, graphics, sound files, etc., but also executable programs. Plain files can be generated using the usual tools like editors, `cat`, shell output redirection, and so on.

Directories Also called "folders"; their function, as we have mentioned, is to help structure storage. A directory is basically a table giving file names and associated inode numbers. Directories are created using the `mkdir` command.

Symbolic links Contain a path specification redirecting accesses to the link to a different file (similar to "shortcuts" in Windows). See also Section 6.4.2. Symbolic links are created using `ln -s`.

Device files These files serve as interfaces to arbitrary devices such as disk drives. For example, the file `/dev/fd0` represents the first floppy drive. Every write or read access to such a file is redirected to the corresponding device. Device files are created using the `mknod` command; this is usually the system administrator's prerogative and is thus not explained in more detail in this manual.

FIFOs Often called "named pipes". Like the shell's pipes, they allow the direct communication between processes without using intermediate files. A process opens the FIFO for writing and another one for reading. Unlike the pipes that the shell uses for its pipelines, which behave like files from a program's point of view but are "anonymous"—they do not exist within the file system but only between related processes—, FIFOs have file names and can thus be opened like files by arbitrary programs. Besides, FIFOs may have access rights (pipes may not). FIFOs are created using the `mkfifo` command.

Unix-domain sockets Like FIFOs, Unix-domain sockets are a method of interprocess communication. They use essentially the same programming interface as "real" network communications across TCP/IP, but only work for communication peers on the same computer. On the other hand, Unix-domain sockets are considerably more efficient than TCP/IP. Unlike FIFOs, Unix-domain sockets allow bi-directional communications—both participating processes can send as well as receive data. Unix-domain sockets are used, e. g., by the X11 graphic system, if the X server and clients run on the same computer. There is no special program to create Unix-domain sockets.

Exercises

10.1 [3] Check your system for examples of the various file types. (Table 10.1 shows you how to recognise the files in question.)

10.3 The Linux Directory Tree

FHS

A Linux system consists of hundreds of thousands of files. In order to keep track, there are certain conventions for the directory structure and the files comprising a Linux system, the *Filesystem Hierarchy Standard* (**FHS**). Most distributions adhere to this standard (possibly with small deviations). The FHS describes all directories immediately below the file system's root as well as a second level below `/usr`.

root directory

The file system tree starts at the **root directory**, "`/`" (not to be confused with `/root`, the home directory of user `root`). The root directory contains either just subdirectories or else additionally, if no `/boot` directory exists, the operating system kernel.

You can use the "`ls -la /`" command to list the root directory's subdirectories. The result should look similar to Figure 10.1. The individual subdirectories follow FHS and therefore contain approximately the same files on every distribution. We shall now take a closer look at some of the directories:

```
$ cd /
$ ls -l
insgesamt 125
drwxr-xr-x    2 root  root    4096 Dez 20 12:37 bin
drwxr-xr-x    2 root  root    4096 Jan 27 13:19 boot
lrwxrwxrwx    1 root  root      17 Dez 20 12:51 cdrecorder▷
                                                          ◁ -> /media/cdrecorder
lrwxrwxrwx    1 root  root      12 Dez 20 12:51 cdrom -> /media/cdrom
drwxr-xr-x   27 root  root   49152 Mär  4 07:49 dev
drwxr-xr-x   40 root  root    4096 Mär  4 09:16 etc
lrwxrwxrwx    1 root  root      13 Dez 20 12:51 floppy -> /media/floppy
drwxr-xr-x    6 root  root    4096 Dez 20 16:28 home
drwxr-xr-x    6 root  root    4096 Dez 20 12:36 lib
drwxr-xr-x    6 root  root    4096 Feb  2 12:43 media
drwxr-xr-x    2 root  root    4096 Mär 21  2002 mnt
drwxr-xr-x   14 root  root    4096 Mär  3 12:54 opt
dr-xr-xr-x   95 root  root       0 Mär  4 08:49 proc
drwx------   11 root  root    4096 Mär  3 16:09 root
drwxr-xr-x    4 root  root    4096 Dez 20 13:09 sbin
drwxr-xr-x    6 root  root    4096 Dez 20 12:36 srv
drwxrwxrwt   23 root  root    4096 Mär  4 10:45 tmp
drwxr-xr-x   13 root  root    4096 Dez 20 12:55 usr
drwxr-xr-x   17 root  root    4096 Dez 20 13:02 var
```

Figure 10.1: Content of the root directory (SUSE)

There is considerable consensus about the FHS, but it is just as "binding" as anything on Linux, i. e., not that much. On the one hand, there certainly are Linux systems (for example the one on your broadband router or PVR) that are mostly touched only by the manufacturer and where conforming to every nook and cranny of the FHS does not gain anything. On the other hand, you may do whatever you like on your own system, but must be prepared to bear the consequences—your distributor assures you to keep to his side of the FHS bargain, but also expects you not to complain if you are not playing completely by the rules and problems do occur. For example, if you install a program in `/usr/bin` and the file in question gets overwritten during the next system upgrade, this is your own fault since, according to the FHS, you are not supposed to put your own programs into `/usr/bin` (`/usr/local/bin` would have been correct).

The Operating System Kernel—`/boot` The `/boot` directory contains the actual operating system: `vmlinuz` is the Linux kernel. In the `/boot` directory there are also other files required for the boot loader (LILO or GRUB).

General Utilities—`/bin` In `/bin` there are the most important executable programs (mostly system programs) which are necessary for the system to boot. This includes, for example, `mount` and `mkdir`. Many of these programs are so essential that they are needed not just during system startup, but also when the system is running—like `ls` and `grep`. `/bin` also contains programs that are necessary to get a damaged system running again if only the file system containing the root directory is available. Additional programs that are not required on boot or for system repair can be found in `/usr/bin`.

Special System Programs—`/sbin` Like `/bin`, `/sbin` contains programs that are necessary to boot or repair the system. However, for the most part these are system configuration tools that can really be used only by `root`. "Normal" users can use some of these programs to query the system, but can't change anything. As with `/bin`, there is a directory called `/usr/sbin` containing more system programs.

System Libraries—`/lib` This is where the "shared libraries" used by programs in `/bin` and `/sbin` reside, as files and (symbolic) links. Shared libraries are pieces of code that are used by various programs. Such libraries save a lot of resources, since many processes use the same basic parts, and these basic parts must then be loaded into memory only once; in addition, it is easier to fix bugs in such libraries when they are in the system just once and all programs fetch the code in question from one central file. Incidentally, below `/lib/modules` there are **kernel modules**, i. e., kernel code which is not necessarily in use—device drivers, file systems, or network protocols. These modules can be loaded by the kernel when they are needed, and in many cases also be removed after use.

kernel modules

Device Files—`/dev` This directory and its subdirectories contain a plethora of entries for device files. **Device files** form the interface between the shell (or, generally, the part of the system that is accessible to command-line users or programmers) to the device drivers inside the kernel. They have no "content" like other files, but refer to a driver within the kernel via "device numbers".

Device files

In former times it was common for Linux distributors to include an entry in `/dev` for every conceivable device. So even a laptop Linux system included the device files required for ten hard disks with 63 partitions each, eight ISDN adapters, sixteen serial and four parallel interfaces, and so on. Today the trend is away from overfull `/dev` directories with one entry for every imaginable device and towards systems more closely tied to the running kernel, which only contain entries for devices that actually exist. The magic word in this context is `udev` (short for *userspace* `/dev`) and will be discussed in more detail in *Linux Administration I*.

Linux distinguishes between **character devices** and **block devices**. A character device is, for instance, a terminal, a mouse or a modem—a device that provides or processes single characters. A block device treats data in blocks—this includes hard disks or floppy disks, where bytes cannot be read singly but only in groups of 512 (or some such). Depending on their flavour, device files are labelled in "`ls -l`" output with a "`c`" or "`b`":

character devices
block devices

```
crw-rw-rw-  1  root  root  10,  4 Oct 16 11:11 amigamouse
brw-rw----  1  root  disk   8,  1 Oct 16 11:11 sda1
brw-rw----  1  root  disk   8,  2 Oct 16 11:11 sda2
crw-rw-rw-  1  root  root   1,  3 Oct 16 11:11 null
```

Instead of the file length, the list contains two numbers. The first is the "major device number" specifying the device's type and governing which kernel driver is in charge of this device. For example, all SCSI hard disks have major device number 8. The second number is the "minor device number". This is used by the driver to distinguish between different similar or related devices or to denote the various partitions of a disk.

There are several notable **pseudo devices**. The *null device*, `/dev/null`, is like a "dust bin" for program output that is not actually required, but must be directed somewhere. With a command like

pseudo devices

```
$ program >/dev/null
```

the program's standard output, which would otherwise be displayed on the terminal, is discarded. If `/dev/null` is read, it pretends to be an empty file and returns end-of-file at once. `/dev/null` must be accessible to all users for reading and writing.

Copyright © 2012 Linup Front GmbH

The “devices” `/dev/random` and `/dev/urandom` return random bytes of “cryptographic quality” that are created from “noise” in the system—such as the intervals between unpredictable events like key presses. Data from `/dev/random` is suitable for creating keys for common cryptographic algorithms. The `/dev/zero` file returns an unlimited supply of null bytes; you can use these, for example, to create or overwrite files with the `dd` command.

Configuration Files—`/etc` The `/etc` directory is very important; it contains the configuration files for most programs. Files `/etc/inittab` and `/etc/init.d/*`, for example, contain most of the system-specific data required to start system services. Here is a more detailed descriptionof the most important files—except for a few of them, only user `root` has write permission but everyone may read them.

`/etc/fstab` This describes all mountable file systems and their properties (type, access method, “mount point”).

`/etc/hosts` This file is one of the configuration files of the TCP/IP network. It maps the names of network hosts to their IP addresses. In small networks and on freestanding hosts this can replace a name server.

`/etc/inittab` The `/etc/inittab` file is the configuration file for the `init` program and thus for the system start.

`/etc/init.d/*` This directory contains the “init scripts” for various system services. These are used to start up or shut down system services when the system is booted or switched off.

On Red Hat distributions, this directory is called `/etc/rc.d/init.d`.

`/etc/issue` This file contains the greeting that is output before a user is asked to log in. After the installation of a new system this frequently contains the name of the vendor.

`/etc/motd` This file contains the “message of the day” that appears after a user has successfully logged in. The system administrator can use this file to notify users of important facts and events[1].

`/etc/mtab` This is a list of all mounted file systems including their mount points. `/etc/mtab` differs from `/etc/fstab` in that it contains all currently mounted file systems, while `/etc/fstab` contains only settings and options for file systems that *might* be mounted—typically on system boot but also later. Even that list is not exhaustive, since you can mount file systems via the command line where and how you like.

We’re really not supposed to put that kind of information in a file within `/etc`, where files ought to be static. Apparently, tradition has carried the day here.

`/etc/passwd` In `/etc/passwd` there is a list of all users that are known to the system, together with various items of user-specific information. In spite of the name of the file, on modern systems the passwords are not stored in this file but in another one called `/etc/shadow`. Unlike `/etc/passwd`, that file is not readable by normal users.

Accessories—`/opt` This directory is really intended for third-party software—complete packages prepared by vendors that are supposed to be installable without conflicting with distribution files or locally-installed files. Such software packages occupy a subdirectory `/opt/`⟨*package*⟩. By rights, the `/opt` directory should be completely empty after a distribution has been installed on an empty disk.

[1] There is a well-known claim that the only thing all Unix systems in the world have in common is the “message of the day” asking users to remove unwanted files since all the disks are 98% full.

Copyright © 2012 Linup Front GmbH

"Unchanging Files"—`/usr` In `/usr` there are various subdirectories containing programs and data files that are not essential for booting or repairing the system or otherwise indispensable. The most important directories include:

`/usr/bin` System programs that are not essential for booting or otherwise important

`/usr/sbin` More system programs for `root`

`/usr/lib` Further libraries (not used for programs in `/bin` or `/sbin`

`/usr/local` Directory for files installed by the local system administrator. Corresponds to the `/opt` directory—the distribution may not put anything here

`/usr/share` Architecture-independent data. In principle, a Linux network consisting, e. g., of Intel, SPARC and PowerPC hosts could share a single copy of `/usr/share` on a central server. However, today disk space is so cheap that no distribution takes the trouble of actually implementing this.

`/usr/share/doc` Documentation, e. g., HOWTOs

`/usr/share/info` Info pages

`/usr/share/man` Manual pages (in subdirectories)

`/usr/src` Source code for the kernel and other programs (if available)

The name `/usr` is often erroneously considered an acronym of "Unix system resources". Originally this directory derives from the time when computers often had a small, fast hard disk and another one that was bigger but slower. All the frequently-used programs and files went to the small disk, while the big disk (mounted as `/usr`) served as a repository for files and programs that were either less frequently used or too big. Today this separation can be exploited in another way: With care, you can put `/usr` on its own partition and mount that partition "read-only". It is even possible to import `/usr` from a remote server, even though the falling prices for disk storage no longer make this necessary (the common Linux distributions do not support this, anyway).

Read-only `/usr`

A Window into the Kernel—`/proc` This is one of the most interesting and important directories. `/proc` is really a "pseudo file system": It does not occupy space on disk, but its subdirectories and files are created by the kernel if and when someone is interested in their content. You will find lots of data about running processes as well as other information the kernel possesses about the computer's hardware. For instance, in some files you will find a complete hardware analysis. The most important files include:

pseudo file system

`/proc/cpuinfo` This contains information about the CPU's type and clock frequency.

`/proc/devices` This is a complete list of devices supported by the kernel including their major device numbers. This list is consulted when device files are created.

`/proc/dma` A list of DMA channels in use. On today's PCI-based systems this is neither very interesting nor important.

`/proc/interrupts` A list of all hardware interrupts in use. This contains the interrupt number, number of interrupts triggered and the drivers handling that particular interrupt. (An interrupt occurs in this list only if there is a driver in the kernel claiming it.)

`/proc/ioports` Like `/proc/interrupts`, but for I/O ports.

`/proc/kcore` This file is conspicuous for its size. It makes available the computer's complete RAM and is required for debugging the kernel. This file requires `root` privileges for reading. You do well to stay away from it!

`/proc/loadavg` This file contains three numbers measuring the CPU load during the last 1, 5 and 15 minutes. These values are usually output by the `uptime` program

`/proc/meminfo` Displays the memory and swap usage. This file is used by the `free` program

`/proc/mounts` Another list of all currently mounted file systems, mostly identical to `/etc/mtab`

`/proc/scsi` In this directory there is a file called `scsi` listing the available SCSI devices. There is another subdirectory for every type of SCSI host adapter in the system containing a file `0` (`1`, `2`, …, for multiple adapters of the same type) giving information about the SCSI adapter.

`/proc/version` Contains the version number and compilation date of the current kernel.

Back when `/proc` had not been invented, programs like the process status display tool, `ps`, which had to access kernel information, needed to include considerable knowledge about internal kernel data structures as well as the appropriate access rights to read the data in question from the running kernel. Since these data structures used to change fairly rapidly, it was often necessary to install a new version of these programs along with a new version of the kernel. The `/proc` file system serves as an abstraction layer between these internal data structures and the utilities: Today you just need to ensure that after an internal change the data formats in `/proc` remain the same—and `ps` and friends continue working as usual.

Hardware Control—`/sys` The Linux kernel has featured this directory since version 2.6. Like `/proc`, it is made available on demand by the kernel itself and allows, in an extensive hierarchy of subdirectories, a consistent view on the available hardware. It also supports management operations on the hardware via various special files.

Theoretically, all entries in `/proc` that have nothing to do with individual processes should slowly migrate to `/sys`. When this strategic goal is going to be achieved, however, is anybody's guess.

log files

Dynamically Changing Files—`/var` This directory contains dynamically changing files, distributed across different directories. When executing various programs, the user often creates data (frequently without being aware of the fact). For example, the `man` command causes compressed manual page sources to be uncompressed, while formatted man pages may be kept around for a while in case they are required again soon. Similarly, when a document is printed, the print data must be stored before being sent to the printer, e. g., in `/var/spool/cups`. Files in `/var/log` record login and logout times and other system events (the "log files"), `/var/spool/cron` contains information about regular automatic command invocations, and users' unread electronic mail is kept in `/var/mail`.

Just so you heard about it once (it might be on the exam): On Linux, the system log files are generally handled by the "syslog" service. A program called `syslogd` accepts messages from other programs and sorts these according to their origin and priority (from "debugging help" to "error" and "emergency, system is crashing right now") into files below `/var/log`, where you can find them later on. Other than to files, the syslog service can also

Copyright © 2012 Linup Front GmbH

write its messages elsewhere, such as to the console or via the network to another computer serving as a central "management station" that consolidates all log messages from your data center.

Besides the syslogd, some Linux distributions also contain a klogd service. Its job is to accept messages from the operating system kernel and to pass them on to syslogd. Other distributions do not need a separate klogd since their syslogd can do that job itself.

The Linux kernel emits all sorts of messages even before the system is booted far enough to run syslogd (and possibly klogd) to accept them. Since the messages might still be important, the Linux kernel stores them internally, and you can access them using the dmesg command.

Transient Files—/tmp Many utilities require temporary file space, for example some editors or sort. In /tmp, all programs can deposit temporary data. Many distributions can be set up to clean out /tmp when the system is booted; thus you should not put anything of lasting importance there.

According to tradition, /tmp is emptied during system startup but /var/tmp isn't. You should check what your distribution does.

Server Files—/srv Here you will find files offered by various server programs, such as

```
drwxr-xr-x    2 root     root         4096 Sep 13 01:14 ftp
drwxr-xr-x    5 root     root         4096 Sep  9 23:00 www
```

This directory is a relatively new invention, and it is quite possible that it does not yet exist on your system. Unfortunately there is no other obvious place for web pages, an FTP server's documents, etc., that the FHS authors could agree on (the actual reason for the introduction of /srv), so that on a system without /srv, these files could end up somewhere completely different, e. g., in subdirectories of /usr/local or /var.

Access to CD-ROM or Floppies—/media This directory is often generated automatically; it contains additional empty directories, like /media/cdrom and /media/floppy, that can serve as mount points for CD-ROMs and floppies. Depending on your hardware setup you should feel free to add further directories such as /media/dvd, if these make sense as mount points and have not been preinstalled by your distribution vendor.

Access to Other Storage Media—/mnt This directory (also empty) serves as a mount point for short-term mounting of additional storage media. With some distributions, such as those by Red Hat, media mountpoints for CD-ROM, floppy, … might show up here instead of below /media.

User Home Directories—/home This directory contains the home directories of all users except root (whose home directory is located elsewhere).

If you have more than a few hundred users, it is sensible, for privacy protection and efficiency, not to keep all home directories as immediate children of /home. You could, for example, use the users' primary group as a criterion for further subdivision:

```
/home/support/jim
/home/develop/bob
◁◁◁◁◁
```

Table 10.2: Directory division according to the FHS

	static	dynamic
local	`/etc`, `/bin`, `/sbin`, `/lib`	`/dev`, `/var/log`
remote	`/usr`, `/opt`	`/home`, `/var/mail`

Administrator's Home Directory—`/root` The system administrator's home directory is located in `/root`. This is a completely normal home directory similar to that of the other users, with the marked difference that it is not located below `/home` but immediately below the root directory (`/`).

The reason for this is that `/home` is often located on a file system on a separate partition or hard disk. However, `root` must be able to access their own user environment even if the separate `/home` file system is not accessible for some reason.

Lost property—`lost+found` (`ext` file systems only; not mandated by FHS.) This directory is used for files that look reasonable but do not seem to belong to any directory. The file system consistency checker creates liks to such files in the `lost+found` directory on the same file system, so the system administrator can figure out where the file really belongs; `lost+found` is created "on the off-chance" for the file system consistency checker to find in a fixed place (by convention, on the `ext` file systems, it always uses inode number 11).

Another motivation for the directory arrangement is as follows: The FHS divides files and directories roughly according to two criteria—do they need to be available locally or can they reside on another computer and be accessed via the network, and are their contents static (do files only change by explicit administrator action) or do they change while the system is running? (Table 10.2)

The idea behind this division is to simplify system administration: Directories can be moved to file servers and maintained centrally. Directories that do not contain dynamic data can be mounted read-only and are more resilient to crashes.

Exercises

10.2 [1] How many programs does your system contain in the "usual" places?

10.3 [I]f `grep` is called with more than one file name on the command line, it outputs the name of the file in question in front of every matching line. This is possibly a problem if you invoke `grep` with a shell wildcard pattern (such as "`*.txt`"), since the exact format of the `grep` output cannot be foreseen, which may mess up programs further down the pipeline. How can you enforce output of the file name, even if the search pattern expands to a single file name only? (*Hint:* There is a very useful "file" in `/dev`.)

10.4 [T]he "`cp foo.txt /dev/null`" command does basically nothing, but the "`mv foo.txt /dev/null`"—assuming suitable access permissions—replaces `/dev/null` by `foo.txt`. Why?

10.5 [2] On your system, which (if any) software packages are installed below `/opt`? Which ones are supplied by the distribution and which ones are third-party products? Should a distribution install a "teaser version" of a third-party product below `/opt` or elsewhere? What do you think?

10.6 [1] Why is it inadvisable to make backup copies of the directory tree rooted at `/proc`?

10.4 Directory Tree and File Systems

A Linux system's directory tree usually extends over more than one partition on disk, and removable media like CD-ROM disks, USB keys as well as portable MP3 players, digital cameras and so on must be taken into account. If you know your way around Microsoft Windows, you are probably aware that this problem is solved there by means of identifying different "drives" by means of letters—on Linux, all available disk partitions and media are integrated in the directory tree starting at "`/`".

partitioning

In general, nothing prevents you from installing a complete Linux system on a single hard disk partition. However, it is common to put at least the `/home` directory—where users' home directories reside—on its own partition. The advantage of this approach is that you can re-install the actual operating system, your Linux distribution, completely from scratch without having to worry about the safety of your own data (you simply need to pay attention at the correct moment, namely when you pick the target partition(s) for the installation in your distribution's installer.) This also simplifies the creation of backup copies.

server systems

On larger server systems it is also quite usual to assign other directories, typically `/tmp`, `/var/tmp`, or `/var/spool`, their own partitions. The goal is to prevent users from disturbing system operations by filling important partitions completely. For example, if `/var` is full, no protocol messages can be written to disk, so we want to keep users from filling up the file system with large amounts of unread mail, unprinted print jobs, or giant files in `/var/tmp`. On the other hand, all these partitions tend to clutter up the system.

More information and strategies for partitioning are presented in the Linup Front training manual, *Linux Administration I.*

`/etc/fstab`

The `/etc/fstab` file describes how the system is assembled from various disk partitions. During startup, the system arranges for the various file systems to be made available—the Linux insider says "mounted"—in the correct places, which you as a normal user do not need to worry about. What you may in fact be interested in, though, is how to access your CD-ROM disks and USB keys, and these need to be mounted, too. Hence we do well to cover this topic briefly even though it is really administrator country.

To mount a medium, you require both the name of the device file for the medium (usually a block device such as `/dev/sda1`) and a directory somewhere in the directory tree where the content of the medium should appear—the so-called *mount point*. This can be any directory.

The directory doesn't even have to be empty, although you cannot access the original content once you have mounted another medium "over" it. (The content reappears after you unmount the medium.)

In principile, somebody could mount a removable medium over an important system directory such as `/etc` (ideally with a file called `passwd` containing a `root` entry without a password). This is why mounting of file systems in arbitrary places within the directory tree is restricted to the system administrator, who will have no need for shenanigans like these, as they are already `root`.

Earlier on, we called the "device file for the medium" `/dev/sda1`. This is really the first partition on the first SCSI disk drive in the system—the real name may be completely different depending on the type of medium you are using. Still it is an obvious name for USB keys, which for technical reasons are treated by the system as if they were SCSI devices.

With this information—device name and mount point—a system administrator can mount the medium as follows:

Copyright © 2012 Linup Front GmbH

```
# mount /dev/sda1 /media/usb
```

This means that a file called `file` on the medium would appear as `/media/usb/file` in the directory tree. With a command such as

```
# umount /media/usb                                    Note: no "n"
```

the administrator can also unmount the medium again.

Commands in this Chapter

dmesg	Outputs the content of the kernel message buffer	`dmesg`(8)	141
file	Guesses the type of a file's content, according to rules	`file`(1)	134
free	Displays main memory and swap space usage	`free`(1)	140
klogd	Accepts kernel log messages	`klogd`(8)	141
mkfifo	Creates FIFOs (named pipes)	`mkfifo`(1)	135
mknod	Creates device files	`mknod`(1)	135
syslogd	Handles system log messages	`syslogd`(8)	141
uptime	Outputs the time since the last system boot as well as the system load averages	`uptime`(1)	139

Summary

- Files are self-contained collections of data stored under a name. Linux uses the "file" abstraction also for devices and other objects.
- The method of arranging data and administrative information on a disk is called a file system. The same term covers the complete tree-structured hierarchy of directories and files in the system or a specific storage medium together with the data on it.
- Linux file systems contain plain files, directories, symbolic links, device files (two kinds), FIFOs, and Unix-domain sockets.
- The *Filesystem Hierarchy Standard* (FHS) describes the meaning of the most important directories in a Linux system and is adhered to by most Linux distributions.

11

Archiving and Compressing Files

Contents

Goals

- Understanding the terms "archival" and "compression"
- Being able to use `tar`
- Being able to compress and uncompress files with `gzip` and `bzip2`
- Being able to process files with `zip` and `unzip`

Prerequisites

- Use of the shell (Chapter 4)
- Handling files and directories (Chapter 6)
- Use of filters (Chapter 8)

11.1 Archival and Compression

"Archival" is the process of collecting many files into a single on. The typical application is storing a directory tree on magnetic tape—the magnetic tape drive appears within Linux as a device file onto which the output of the archival program can be written. Conversely, you can read the tape drive's device file using a de-archiver and reconstruct the directory tree from the archived data. Since most of the relevant programs can both create and unravel archives, we discuss both operations under the heading of "archival".

"Compression" is the rewriting of data into a representation that saves space compared to the original. Here we are only interested in "lossless" compression, where it is possible to reconstruct the original in identical form from the compressed data.

The alternative is to achieve a higher degree of compression by abandoning the requirement of being able to recreate the original perfectly. This "lossy" approach is taken by compression schemes like JPEG for photographs and "MPEG-1 Audio Layer 3" (better known as "MP3") for audio data. The secret here is to get rid of extraneous data; with MP3, for example, we throw out those parts of the signal that, based on a "psycho-acoustic model" of human hearing, the listener will not be able to make out, anyway, and encode the rest as efficiently as possible. JPEG works along similar lines.

run-length encoding As a simple illustration, you might represent a character string like

```
ABBBBAACCCCCAAAABAAAAAC
```

more compactly as

```
A*4BAA*5C*4AB*5AC
```

Here, "`*4B`" stands for a sequence of four "`B`" characters. This simple approach is called "run-length encoding" and is found even today, for example, in fax machines (with refinements). "Real" compression programs like `gzip` or `bzip2` use more sophisticated methods.

While programs that combine archival and compression are used widely in the Windows world (PKZIP, WinZIP and so on), both steps are commonly handled separately on Linux and Unix. A popular approach is to archive a set of files first using `tar` before compressing the output of `tar` using, say, `gzip`—PKZIP and friends compress each file on its own and then collect the compressed files into a single big one.

The advantage of this approach compared to that of PKZIP and its relatives is that compression can take place across several original files, which yields higher compression rates. However, this also counts as a disadvantage: If the compressed archive is damaged (e. g., due to a faulty medium or flipped bits during transmission), the whole archive can become unusable starting at that point.

Naturally even on Linux nobody keeps you from *first* compressing your files and then archiving them. Unfortunately this is not as convenient as the other approach.

Of course there are Linux implementations of compression and archival programs popular in the Windows world, like `zip` and `rar`.

Exercises

11.1 [1] Why does the run-length encoding example use `AA` instead of `*2A`?

11.2 [2] How would your represent the string "`A*2B****A`" using the run-length encoding method shown above?

Copyright © 2012 Linup Front GmbH

11.2 Archiving Files Using tar

The name tar derives from "tape archive". The program writes individual files to the archival file one after the other and annotates them with additional information (like the date, access permissions, owner, …). Even though tar was originally meant to be used with magnetic tape drives, tar archives can be written directly on various media. Among other uses, tar files are the standard format for disseminating the source code for Linux and other free software packages.

The GNU implementation of tar commonly used on Linux includes various extensions not found in the tar implementations of other Unix variants. For example, GNU tar supports creating multi-volume archives spanning several media. This even allows backup copies to floppy disk, which of course is only worthwhile for small archives.

multi-volume archives

A small remark on the side: The split command lets you cut large files like archives into convenient pieces that can be copied to floppy disks or sent via e-mail, and can be re-joined at their destination using cat.

The advantages of tar include: It is straightforward to use, it is reliable and works well, it can be used universally on all Unix and Linux systems. Its disadvantages are that faults on the medium may lead to problems, and not all versions of tar can store device files (which is only an issue if you want to perform a full backup of your system).

tar archives can contain files and whole directory hierarchies. If Windows media have been mounted into the directory tree across the network, even their content can be archived using tar. Archives created using tar are normally uncompressed, but can be compressed using external compression software (nowadays usually gzip or bzip2). This is not a good idea as far as backup copies are concerned, since bit errors in the compressed data usually lead to the loss of the remainder of the archive.

Typical suffixes for tar archives include .tar, .tar.bz2, or .tar.gz, depending on whether they have been compressed not at all, using bzip2, or using gzip. The .tgz suffix is also common when zipped tar-formatted data need to be stored on a DOS file system. tar's syntax is

```
tar ⟨options⟩ ⟨file⟩||⟨directory⟩ …
```

and the most important include:

tar options

-c ("create") creates a new archive

-f *file* creates the new archive on (or reads an existing archive from) ⟨*file*⟩, where ⟨*file*⟩ can be a plain file or a device file (among others)

-M handles multi-volume archives

-r appends files to the archive (not for magnetic tape)

-t displays the "table of contents" of the archive

-u replaces files which are newer than their version inside the archive. If a file is not archived at all, it is inserted (not for magnetic tape)

-v Verbose mode—displays what tar is doing at the moment

-x extracts files and directories from an archive

-z compresses or decompresses the archive using gzip

-Z compresses or decompresses the archive using compress (not normally available on Linux)

-j compresses or decompresses the archive using bzip2 option syntax

tar's option syntax is somewhat unusual, in that it is possible (as is elsewhere) to "bundle" several options after a single dash, including (extraordinarily) ones such as -f that take a parameter. Option parameters need to be specified after the "bundle" and are matched to the corresponding parameter-taking options within the bundle in turn.

You may leave out the dash in front of the first "option bundle"—you will often see commands like

tar cvf …

However, we don't recommend this.

The following example archives all files within the current directory whose names begin with data to the file data.tar in the user's home directory:

```
# tar -cvf ~/data.tar data*
data1
data10
data2
data3
⊲⊲⊲⊲⊲
```

The -c option arranges for the archive to be newly created, "-f ~/data.tar" gives the name for the archive. The -v option does not change anything about the result; it only causes the names of files to appear on the screen as they are being archived. (If one of the files to be archived is really a directory, the complete content of the directory will also be added to the archive.)

directories

tar also lets you archive complete directories. It is better to do this from the enclosing directory, which will create a subdirectory in the archive which is also recreated when the archive is unpacked. The following example shows this in more detail.

```
# cd /
# tar -cvf /tmp/home.tar /home
```

The system administrator root stores an archive of the /home directory (i. e., all user data) under the name of home.tar. This is stored in the /tmp directory.

If files or directories are given using absolute path names, tar automatically stores them as relative path names (in other words, the "/" at the start of each name is removed). This avoids problems when unpacking the archive on other computers (see Exercise 11.6).

You can display the "table of contents" of an archive using the -t option:

```
$ tar -tf data.tar
data1
data10
data2
⊲⊲⊲⊲⊲
```

The -v option makes tar somewhat more talkative:

```
$ tar -tvf data.tar
-rw-r--r-- joe/joe     7 2009-01-27 12:04 data1
-rw-r--r-- joe/joe     8 2009-01-27 12:04 data10
-rw-r--r-- joe/joe     7 2009-01-27 12:04 data2
⊲⊲⊲⊲⊲
```

Copyright © 2012 Linup Front GmbH

You can unpack the data using the `-x` option:

```
$ tar -xf data.tar
```

In this case `tar` produces no output on the terminal at all—you have to give the `-v` option again:

```
$ tar -xvf data.tar
data1
data10
data2
◁◁◁◁◁
```

If the archive contains a directory hierarchy, this is faithfully reconstructed in the current direcotry. (You will remember that `tar` makes relative path names from all absolute ones.) You can unpack the archive relative to any directory—it always keeps its structure.

You can also give file or directory names on unpacking. In this case only the files or directories in question will be unpacked. However, you need to take care to match the names in the archive exactly:

```
$ tar -cf data.tar ./data
$ tar -tvf data.tar
drwxr-xr-x joe/joe     0 2009-01-27 12:04 ./data/
-rw-r--r-- joe/joe     7 2009-01-27 12:04 ./data/data2
◁◁◁◁◁
$ mkdir data-new
$ cd data-new
$ tar -xvf ../data.tar data/data2                                  ./ missing
tar: data/data2: Not found in archive
tar: Error exit delayed from previous errors
```

Exercises

11.3 [!2] Store a list of the files in your home directory in a file called `content`. Create a `tar` archive from that file. Compare the original file and the archive. What do you notice?

11.4 [2] Create three or four empty files and add them to the archive you just created.

11.5 [2] Remove the original files and then unpack the content of the `tar` archive.

11.6 [2] Why does GNU `tar` prophylactically remove the `/` at the beginning of the path name, if the name of a file or directory to be archived is given as an absolute path name? (*Hint:* Consider the

```
# tar -cvf /tmp/etc-backup.tar /etc
```

command and imagine what will happen if `etc-backup.tar` (a) contains absolute path names, and (b) is transferred to another computer and unpacked there.)

Copyright © 2012 Linup Front GmbH

11.3 Compressing Files with gzip

The most common compression program for Linux is gzip by Jean-loup Gailly and Mark Adler. It is used to compress single files (which, as mentioned earlier, may be archives containing many files).

The gzip program (short for "GNU zip") was published in 1992 to avoid problems with the compress program, which was the standard compression tool on proprietary Unix versions. compress is based on the Lempel-Ziv-Welch algorithm (LZW), which used to be covered by US patent 4,558,302. This patent belonged to the Sperry (later Unisys) corporation and expired on 20 June 2003. On the other hand, gzip uses the DEFLATE method by Phil Katz [RFC1951], which is based on a non-patented precursor of LZW called LZ77 as well as the Huffman encoding scheme and is free of patent claims. Besides, it works better than LZW.

gzip can *decompress* files compressed using compress, because the Unisys patent only covered compression. You can recognise such files by the ".Z" suffix of their names.

gzip is not to be confused with PKZIP and similar Windows programs with "ZIP" in their names. These programs can compress files and then archive them immediately; gzip only takes care of the compression and leaves the archiving to programs like tar or cpio.—gzip can unpack ZIP archives as long as the archive contains exactly one file which has been packed using the DEFLATE method.

gzip processes and replaces single files, appending the File*.gz suffix to their names. This substitution happens independently of whether the resulting file is actually smaller than the original. If several files are to be compressed into a single archive, tar and gzip must be combined.

The most important options of gzip include:

-c writes the compressed file to standard output, instead of replacing the original; the original remains unmodified

-d uncompresses the file (alternatively: gunzip works like gzip -d)

-l ("list") displays important information about the compressed file, such as the file name, original and packed size

-r ("recursive") compresses files in subdirectories

-S ⟨*suffix*⟩ uses the specified suffix in place of .gz

-v outputs the name and compression factor of every file

-1 … -9 specifies a compression factor: -1 (or --fast) works most quickly but does not compress as thoroughly, while -9 (or --best) results in the best compression at a slower speed; the default setting is -6.

The following command compresses the letter.tex file, stores the compressed file as letter.tex.gz and deletes the original:

```
$ gzip letter.tex
```

The file can be unpacked using

```
$ gzip -d letter.tex
```

or

Copyright © 2012 Linup Front GmbH

```
$ gunzip letter.tex
```

Here the compressed file is saved as letter.tex.t instead of letter.tex.gz (-S .t), and the compression rate achieved for the file is output (-v):

```
$ gzip -vS .t letter.tex
```

The -S option must also be specified on decompression, since "gzip -d" expects a file with a .gz suffix:

```
$ gzip -dS .t letter.tex
```

If all .tex files are to be compressed in a file tex-all.tar.gz, the command is

```
$ tar -cvzf tex-all.tar.gz *.tex
```

Remember that tar does not delete the original files! This can be unpacked using

```
$ tar -xvzf tex-all.tar.gz
```

Exercises

11.7 [2] Compress the tar archive from Exercise 11.3 using maximum compression.

11.8 [!3] Inspect the content of the compressed archive. Restore the original tar archive.

11.9 [!2] How would you go about packing all of the contents of your home directory into a gzip-compressed file?

11.4 Compressing Files with bzip2

bzip2 by Julian Seward is a compression program which is largely compatible to gzip. However, it uses a different method which leads to higher compression ratios but requires more time and memory to compress (to decompress, the difference is not as significant).

If you are desperate to know: bzip2 uses a "Burrows-Wheeler transformation" to encode frequently-occurring substrings in the input to sequences of single characters. This intermediate result is sorted according to the "local frequency" of individual characters and the sorted result, after being run-length encoded, is encoded using the Huffman scheme. The Huffman code is then written to a file in a very compact manner.

What about bzip? bzip was a predecessor of bzip2 which used arithmetic encoding rather than Huffman encoding after the block transformation. However, the author decided to give arithmetic coding a wide berth due to the various software patent issues that surround it.

Like gzip, bzip2 accepts one or more file names as parameters for compression. The files are replaced by compressed versions, whose names end in .bz2.

The -c and -d options correspond to the eponymous options to gzip. However, the "quality options" -1 to -9 work differently: They determine the block size used during compression. The default value is -9, while -1 does not offer a significant speed gain.

-9 uses a 900 KiB block size. This corresponds to a memory usage of approximately 3.7 MiB to decompress (7.6 MiB to compress), which on contemporary hardware should not present a problem. A further increase of the block size does not appear to yield an appreciable advantage.—It is worth emphasising that the choice of block size on *compression* determines the amount of memory necessary during *decompression*, which you should keep in mind if you use your multi-Gibibyte PC to prepare .bz2 files for computers with very little memory (toasters, set-top boxes, …). bzip2(1) explains this in more detail.

By analogy to gzip and gunzip, bunzip2 is used to decompress files compressed using bzip2. (This is really just another name for the bzip2 program: You can also use "bzip2 -d" to decompress files.)

11.5 Archiving and Compressing Files Using zip and unzip

To exchange data with Windows computers or on the Internet, it often makes sense to use the widespread ZIP file format (although many file archive programs on Windows can also deal with .tar.gz today). On Linux, there are two separate programs zip (to create archives) and unzip (to unpack archives).

Depending on your distribution you may have to install these programs separately. On Debian GNU/Linux, for example, there are two distinct packages, zip and unzip.

zip The zip program combines archiving and compressing in a way that may be familiar to you from programs like PKZIP. In the simplest case, it collects the files passed on the command line:

```
$ zip test.zip file1 file2
  adding: file1 (deflated 66%)
  adding: file2 (deflated 62%)
$ _
```

(Here test.zip is the name of the resulting archive.)

You can use the -r option to tell zip to descend into subdirectories recursively:

```
$ zip -r test.zip ziptest
  adding: ziptest/ (stored 0%)
  adding: ziptest/testfile (deflated 62%)
  adding: ziptest/file2 (deflated 62%)
  adding: ziptest/file1 (deflated 66%)
```

With the -@ option, zip reads the names of the files to be archived from its standard input:

```
$ find ziptest | zip -@ test
  adding: ziptest/ (stored 0%)
  adding: ziptest/testfile (deflated 62%)
  adding: ziptest/file2 (deflated 62%)
  adding: ziptest/file1 (deflated 66%)
```

(You may omit the .zip suffix from the name of the archive file.)

zip knows about two methods of adding files to an archive. stored means that the file was stored without compression, while deflated denotes compression (and the percentage states *how much* the file was compressed—"deflated

62%”, for example, means that, inside the archive, the file is only 38% of its original size). zip automatically chooses the more sensible approach, unless you disable compression completely using the -0 option.

If you invoke zip with an existing ZIP archive as its first parameter and do not specify anything else, the files to be archived are *added* to the archive on top of its existing content (existing files with the same names are overwritten). In this case zip behaves differently from tar and cpio (just so you know). If you want a “clean” archive, you must remove the file first.

Besides stupidly adding of files, zip supports several other modes of operation: The -u option “updates” the archive by adding files to the archive only if the file mentioned on the command line is newer than a pre-existing file of the same name in the archive (named files that are not in the archive yet are added in any case). The -f option “freshens” the archive—files inside the archive are overwritten with newer versions from the command line, but only if they actually exist in the archive already (no completely new files are added to the archive). The -d option considers the file names on the command line as the names of files within the archive and deletes those.

Newer versions of zip also support the -FS (“filesystem sync”) mode: This mode “synchronises” an archive with the file system by doing essentially what -u does, but also deleting files from the archive that have not been named on the command line (or, in case of -r, are part of a directory being searched). The advantage of this method compared to a full reconstruction of the archive is that any preexisting unchanged files in the archive do not need to be compressed again.

zip supports all sorts of options, and you can use “zip -h” to look at a list (or “ -h2 to look at a more verbose list). The man page, zip(1), is also very informative.

unzip

You can unpack a ZIP archive again using unzip (this can also be a ZIP archive from a Windows machine). It is best to take a peek inside the archive first, using the -v option, to see what is in there—this may save you some hassle with subdirectories (or their absence).

```
$ unzip -v test                                      The .zip suffix may be omitted
Archive:  test.zip
 Length   Method    Size  Cmpr    Date    Time   CRC-32   Name
--------  ------  ------- ---- ---------- ----- --------  ----
       0  Stored        0   0% 2012-02-29 09:29 00000000  ziptest/
   16163  Defl:N     6191  62% 2012-02-29 09:46 0d9df6ad  ziptest/testfile
   18092  Defl:N     6811  62% 2012-02-29 09:01 4e46f4a1  ziptest/file2
   35147  Defl:N    12119  66% 2012-02-29 09:01 6677f57c  ziptest/file1
--------          -------  ---                            -------
 69402            25121  64%                              4 files
```

Calling unzip with the name of the archive as its single parameter suffices to unpack the archive:

```
$ mv ziptest ziptest.orig
$ unzip test
Archive:  test.zip
   creating: ziptest/
  inflating: ziptest/testfile
  inflating: ziptest/file2
  inflating: ziptest/file1
```

Use the -d option to unpack the archive in a different directory than the current one. This directory is created first if necessary:

```
$ unzip -d dir test
Archive:  test.zip
   creating: dir/ziptest/
  inflating: dir/ziptest/testfile
  inflating: dir/ziptest/file2
  inflating: dir/ziptest/file1
```

If you name particular files on the command line, then only these files will be unpacked:

```
$ rm -rf ziptest
$ unzip test ziptest/file1
Archive:  test.zip
  inflating: ziptest/file1
```

(In this case, the `ziptest` directory will also be created.)

Alternatively, you can use the `-x` option to selectively exclude certain files from being unpacked:

```
$ rm -rf ziptest
$ unzip test -x ziptest/file1
Archive:  test.zip
   creating: ziptest/
  inflating: ziptest/testfile
  inflating: ziptest/file2
```

You can also use shell search patterns to unpack certain files (or prevent them from being unpacked):

```
$ rm -rf ziptest
$ unzip test "ziptest/f*"
Archive:  test.zip
  inflating: ziptest/file2
  inflating: ziptest/file1
$ rm -rf ziptest
$ unzip test -x "*/t*"
Archive:  test.zip
   creating: ziptest/
  inflating: ziptest/file2
  inflating: ziptest/file1
```

(Note the quotes, which are used to hide the search patterns from the actual shell so `unzip` gets to see them.) Unlike in the shell, the search patterns refer to the *complete* file names (including any "`/`").

As is to be expected, `unzip` also supports various other options. Look at the program's help information using "`unzip -h`" or "`unzip -hh`", or read `unzip`(1).

Exercises

11.10 [!2] Create some files inside your home directory and store them in a `zip` archive. Look at the content of the archive using "`unzip -v`". Unpack the archive inside the `/tmp` directory.

11.11 [!1] What happens if a file that you are about to unpack using `unzip` already exists in the file system?

Copyright © 2012 Linup Front GmbH

11.12 [2] A ZIP archive `files.zip` contains two subdirectories `a` and `b`, which in turn contain a mixture of files with various suffixes (e. g., `.c`, `.txt`, and `.dat`). Give a `unzip` command to extract the complete content of `a` except for the `.txt` files (in one step).

Commands in this Chapter

`bunzip2`	File decompression program for `.bz2` files	`bzip2`(1)	152
`bzip2`	File compression program	`bzip2`(1)	147
`gzip`	File compression utility	`gzip`(1)	147
`split`	Splits a file into pieces up to a given maximum size	`split`(1)	147
`tar`	File archive manager	`tar`(1)	146
`unzip`	Decompression software for (Windows-style) ZIP archives	`unzip`(1)	153
`zip`	Archival and compression software like PKZIP	`zip`(1)	152

Summary

- "Archival" collects many files into one large file. "Compression" reversibly determines a more compact representation for a file.
- `tar` is the most common archival program on Linux.
- `gzip` is a program for compressing and decompressing arbitrary files. It can be used together with `tar`.
- `bzip2` is another compression program. It can achieve higher compression ratios than `gzip`, but also needs more time and memory.
- The `zip` and `unzip` programs are available to process ZIP archives as used (for example) by the PKZIP program on Windows.

Bibliography

RFC1951 P. Deutsch. "DEFLATE Compressed Data Format Specification version 1.3", May 1996. `http://www.ietf.org/rfc/rfc1951.txt`

12

Introduction to System Administration

Contents

Goals

- Having insight into the role of a system administrator
- Understanding the basics of the operating system kernel and processes
- Knowing about package management concepts

Prerequisites

- Basic use of the shell (Chapter 4)
- Knowledge of Linux file system structure (Chapter 10)

12.1 System Administration Basics

What does a system administrator do? Configure computers, install software (and occasionally remove it), connect peripherals and make them usable, make backups (and occasionally restore them), add and remove user accounts, help users with problems, … it is an impressive list of tasks. In the old days of home computers, the user of a computer was also the administrator, and this idea has been kept going for a long time in systems like Windows (even once Windows had acquired the notion of different users, it was common to have to use the "Administrator" account, simply because various important programs assumed the corresponding privileges were available). Unix—the system that inspired Linux—was set up from its very beginnings to support several users, and hence the separation between an "administrator" with special privileges and "normal" users is rooted far more deeply in the system than with operating systems from the home computer tradition.

The LPI's *Linux Essentials* exam does not emphasise system administration, but you should have at least a general overview—perhaps not to *be* the system administrator, but to achieve a better understanding of what your system administrator does for you, or at some point *become* a system administrator. Knowledge important for system administrators is part of the LPI's LPIC certification track, particularly LPIC-2 and LPIC-3.

In this chapter we shall mention several topics that have less to do with the immediate use of a Linux computer, but include, for example, how to get a picture of what is running on the computer (cue "Why is my computer so slow??"), and how software is managed on the computer. We're really after the big picture here, not details.

`root`

On a Linux system, the system administrator has access to a special user account, `root`. This account is exempt from the access checks (Chapter 14) which are otherwise performed, and can therefore access all files on the system. This is necessary, for example, to install new software—"normal" users may read and execute program files from system directories but, in order to prevent manipulations to the detriment of other users, not write them. The administrator must also be able to read *any* user's files in order to create backup copies (and to write them back if a backup needs to be restored).

It is obvious that the ability to write all files on the system includes the opportunity to damage the system seriously. If you are logged in as `root`, Linux does not prevent you from using a command like

```
# rm -rf /
```

to destroy the whole file system (and there are lots of more subtle ways to do damage). Hence you should avail yourself of `root` privileges only if you actually need them. Surfing the web or reading mail while logged in as `root` is RIGHT OUT.

If you can read all files on thesystem as `root`, you might succumb to the temptation of, say, regularly inspecting your boss's (or spouse's) e-mail. DON'T DO IT. It may be distrustful (at least in the case of your spouse) and/or illegal (at least in the case of your boss) and it can get you into all sorts of trouble that are liable to spoil your fun with Linux and system administration, to say nothing of domestic or workplace peace. There is nothing wrong with taking an isolated brief peek, in consultation with the people involved, into a mailbox file in order to diagnose or repair a problem—but do not let it become a habit.

When in doubt, think of Peter Parker, a. k. a. Spider-Man: "With great power comes great responsibility."

Copyright © 2012 Linup Front GmbH

You should avoid logging in as root directly (especially on a graphical screen). Instead, use the su program, in a terminal session started as a normal user, to obtain a shell running as root:

```
$ /bin/su -
Password: secret                                            Password for root
# _
```

After exiting the root shell (using exit or Ctrl+d) you end up back in the shell where you originally invoked su.

Some distributions try to get by without a separate root account. Ubuntu, for example, allows the first user created during the system's installation to execute single commands with root privileges by putting sudo in front, as in sudo

```
$ sudo less /var/log/syslog                                 Peruse system log
```

(that user can extend this privilege to other users if required). For larger tasks, it is possible to use "sudo -i" to obtain a shell running with administrator privileges.

Most Linux distributions signal that a shell is running with root privileges by outputting a prompt ending in "#". If something else appears—typically "$" or ">"—, the shell is unprivileged.

Exercises

12.1 [2] Try su. Why does the example use an absolute path name to invoke the program?

12.2 [2] You need to know the root password for su. sudo usually asks you for your own password. Which is better?

12.2 System Configuration

While other systems bury their configuration details in databases that can only be modified through special tools and are susceptible to "bit rot" (cue Windows registry), system-wide configuration entries on Linux are usually contained in text files within the /etc directory (a few examples can be seen in Section 10.3), where the system administrator can use a text editor of their choice to modify or extend them. For example, a new user can be added by adding the relevant parameters like the user name, numerical user ID, or home directory name to the /etc/passwd file. A new hard disk may be configured by appending a line to /etc/fstab specifying the name of the device file and the directory where the disk is supposed to appear.

A Linux system is a complex system of software components of widely differing provenance (some of which are older than Linux itself). From this historically grown setup it follows that the different configuration files within /etc are structured in a very non-uniform fashion—some are organised by lines, others contain sections delimited by braces, even others are XML files or even executable shell scripts. This is certainly a nuisance to administrators who need to deal with all these different formats, but it is also not straightforward to change, since all sorts of software packages would need to be modified.

However, there are some very widespread conventions: For example, most configuration files allow comments in lines starting with "#".

While the idea of managing the system configuration in separate text files may seem antediluvial at first, it does have some tangible advantages:

Copyright © 2012 Linup Front GmbH

- It is usually not possible to damage the system as a whole through mistakes in the configuration of a single software package or service. (Of course there are a few configuration files which are so essential for the system's functionality that errors in them could, for example, render the system unbootable. But these are a small minority at best.)

- Most configuration files allow comments. This makes it possible to document the details of individual configuration settings directly where they occur, and thus makes collaboration in a team easier or avoids accidents due to one's own forgetfulness. It is certainly a lot better than having to remember that there is an entry *X* in menu *Y* that lets you open a dialog where on tab *Z* there is a box that absolutely needs to be checked because otherwise nothing is going to work. (Pieces of paper containing this type of wisdom have the unfortunate tendency of disappearing when they are needed most urgently:)

- You can "check in" text files to a revision control system such as Git or Mercurial and thereby not just document big changes spanning various files, but also undo them in an orderly manner if required. This also makes it convenient to store the complete configuration of a computer on a central server, such that it is immediately available if the computer needs to be reinstalled for whatever reason—say, after a catastrophic hardware fault. In data centers this is a very decisive advantage, especially if a detailed "auditing" of all configuration changes is desired.

- Text files allow the convenient configuration of whole computer networks by distributing configuration files from a central server to the computers to be managed. Systems such as "Puppet" or "Salt" make it possible to use "templates" for configuration files that are instantiated using suitable details for the target computer when they are distributed, which can totally obviate the manual configuration of individual computers (the "sneaker net"). This, too, makes the management of big networks easier but is also a definite help with smaller installations.

Exercises

12.3 [3] Snoop around in `/etc`. Many files there have manual pages—try something like "`man fstab`". Are there files in `/etc` that you cannot read as a normal user, and why?

12.3 Processes

PID

A program that is being executed is called a "process". Besides the program code itself (in the machine language of the processor in question), a process includes working storage for the data as well as administrative information like the files currently in use, an environment (for the environment variables), a current directory, and a process number or "PID" identifying the process uniquely within the system. The operating system kernel is in charge of creating processes, assigning them CPU time and storage, and cleaning up after them when they have exited. Processes can call into the operating system kernel to access files, devices, or the network.

New processes come into being when existing processes—not unlike bacteria or other low forms of life—split up into two almost identical copies ("almost identical", because one process is considered the "parent" and the other the "child"). In addition, a process can arrange to execute a different program: For example, if you invoke the `ls` command in the shell, the shell creates a child process which is at first also executing the shell's program code. This code takes care (among other things) of arranging a possible input/output redirection and then replaces itself with the /bin/ls program file. At the end of the ls program, the child process ends, and the shell asks you for the next command.

The first process with the PID of 1 is created by the operating system kernel during boot. According to the convention this is a program called /sbin/init, and it is also called the “init process”. The init process is responsible for booting the system and, e. g., starting additional processes for system services running in the background.

ps You can use the “ps” command to obtain information about the processes running on the system. In the simplest case, ps shows you all processes running on your current terminal (or, nowadays, the current terminal window on your graphical screen):

```
$ ps
  PID TTY   STAT TIME COMMAND
  997 pts/8 S    0:00 -bash
 1005 pts/8 R    0:00 ps
$ _
```

The PID and COMMAND columns speak for themselves. TTY gives the name of the terminal (“pts/something” usually refers to a terminal window), TIME the CPU time used by the processes so far, and STAT the “process state”.

process state

A process on Linux is always in one of a number of states, namely

Runnable (R) The process may be assigned CPU time.

Sleeping (S) The process is waiting for an event, typically input or output—a key press or data from disk.

In deep (uninterruptible) sleep (D) The process is waiting for an event and cannot be disturbed. Processes should not remain in this state for too long because they can only be removed by booting the system. If that happens it is usually due to some error.

Temporarily stopped (T) The process was temporarily stopped by its owner or an administrator, but can continue running later on.

Zombie (Z) The process has really finished, but its exit code has not yet been picked up by its parent. This means the process cannot “die” but remains undead within the system. Unlike in real life, zombies are not really a problem because they do not take up resources other than a slot in the process table. If your system is infested with a horde of zombies, this indicates a problem with the program that created the processes in the first place—terminating that program should make the zombies disappear.

Use parameters to control which information ps provides. For example, you could enter a process number to find out about a particular process:

```
$ ps 1
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 init [2]
```

The l option gives you more detailed information about a process:

```
$ ps l $$
F   UID  PID PPID PRI NI   VSZ  RSS WCHAN  STAT TTY    TIME COMMAND
0  1000 3542 3491  20  0 21152 2288 -      Ss   pts/8  0:00 /bin/bash
```

Copyright © 2012 Linup Front GmbH

("$$" denotes the "current process", the shell).

UID is the numerical ID of the owner of the process (see Chapter 13), PPID the process ID of the process's "parent". PRI is its priority—the higher the number, the lower the priority (!)—, VSZ its size in working memory (in KiB), and RSS its current size in RAM (also in KiB).

VSZ and RSS are not identical, since part of the process may have been moved to disk. After all, you can enlarge the available working memory on a Linux computer by adding swap space on a disk partition or file.

The ps command supports a multitude of options controlling the selection of processes considered and the type and volume of information output for each process. Read ps(1).

ps and similar programs obtain their information from the proc file system, which is usually mounted on /proc and is made available by the operating system kernel. The "files" within this directory contain up-to-date information about processes and other properties of the system. (See also Section 10.3.)

free The free command provides information about system memory:

```
$ free
             total     used     free  shared  buffers  cached
Mem:       3921956  1932696  1989260       0    84964  694640
-/+ buffers/cache:  1153092  2768864
Swap:      8388604        0  8388604
```

The "Mem:" line tells you that this computer has about 4 GiB of RAM (under "total"; the operating system kernel takes up some memory that does not appear here) which is about half full (see "used" and "free"). The operating system uses about 700 MiB to store disk data (the "buffers" and "cached" columns), and the second line tells you how that impacts the free and used memory. The third line ("Swap:") describes swap space utilisation (out of 8 GiB, on this machine).

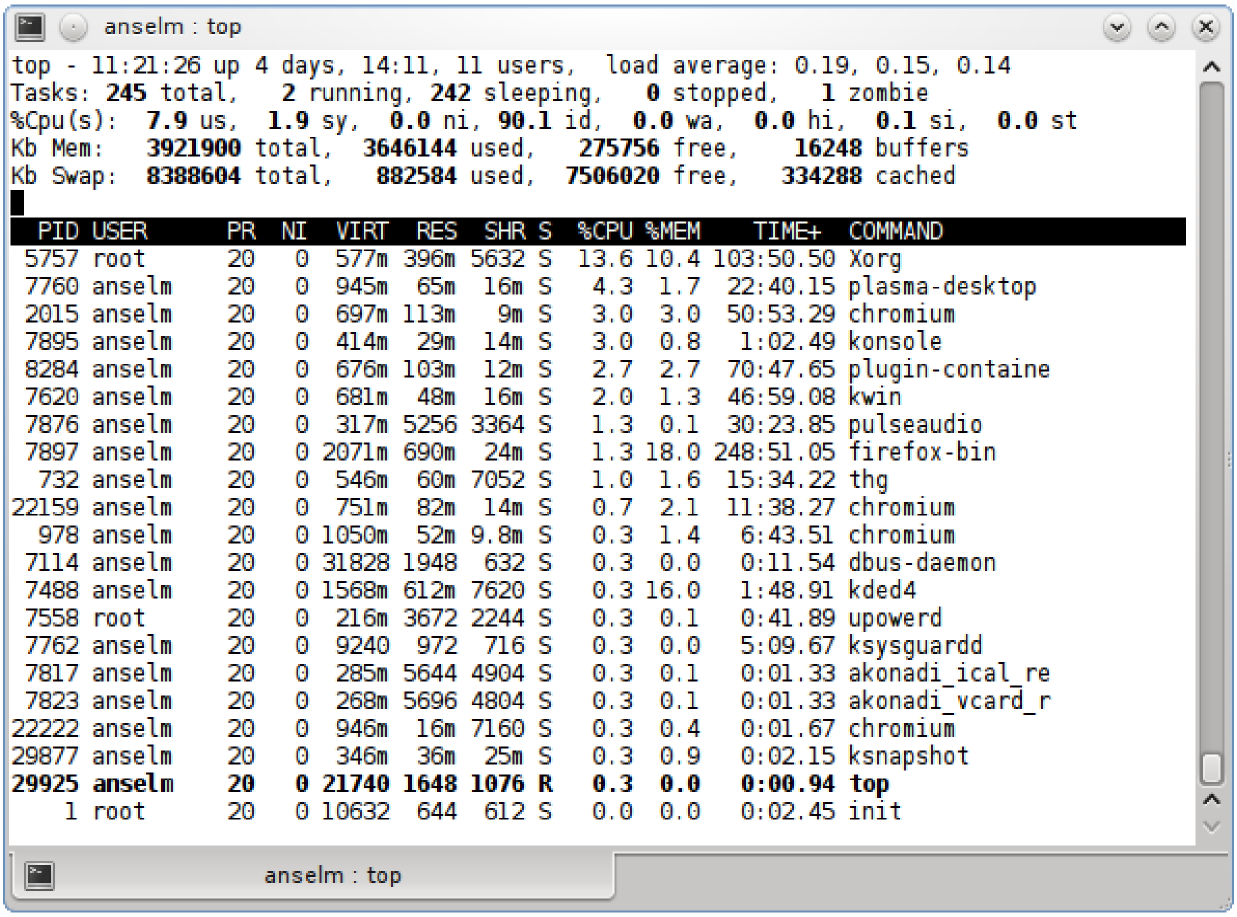
The "shared" column is always zero on modern Linux machines and can therefore be ignored.

free, too, supports a number of options, for example to produce a friendlier output format:

```
$ free --human                                          "-h" would do, too
             total   used  free  shared  buffers  cached
Mem:          3,7G   1,9G  1,8G      0B      84M    678M
-/+ buffers/cache:   1,2G  2,5G
Swap:         8,0G     0B  8,0G
```

Here free uses the "M" and "G" units to refer to the computer-friendly mebibyte and gibibyte. The --si option switches to powers of ten (mega- and gigabyte).

top Finally, the "top" command is like a combination of ps and free with ongoing updates. It displays a full screen of information including system and process information: Figure 12.1 shows an example:



```
top - 11:21:26 up 4 days, 14:11, 11 users,  load average: 0.19, 0.15, 0.14
Tasks: 245 total,   2 running, 242 sleeping,   0 stopped,   1 zombie
%Cpu(s):  7.9 us,  1.9 sy,  0.0 ni, 90.1 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
Kb Mem:   3921900 total,  3646144 used,   275756 free,    16248 buffers
Kb Swap:  8388604 total,   882584 used,  7506020 free,   334288 cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5757 root      20   0  577m 396m 5632 S 13.6 10.4 103:50.50 Xorg
 7760 anselm    20   0  945m  65m  16m S  4.3  1.7  22:40.15 plasma-desktop
 2015 anselm    20   0  697m 113m   9m S  3.0  3.0  50:53.29 chromium
 7895 anselm    20   0  414m  29m  14m S  3.0  0.8   1:02.49 konsole
 8284 anselm    20   0  676m 103m  12m S  2.7  2.7  70:47.65 plugin-containe
 7620 anselm    20   0  681m  48m  16m S  2.0  1.3  46:59.08 kwin
 7876 anselm    20   0  317m 5256 3364 S  1.3  0.1  30:23.85 pulseaudio
 7897 anselm    20   0 2071m 690m  24m S  1.3 18.0 248:51.05 firefox-bin
  732 anselm    20   0  546m  60m 7052 S  1.0  1.6  15:34.22 thg
22159 anselm    20   0  751m  82m  14m S  0.7  2.1  11:38.27 chromium
  978 anselm    20   0 1050m  52m 9.8m S  0.3  1.4   6:43.51 chromium
 7114 anselm    20   0 31828 1948  632 S  0.3  0.0   0:11.54 dbus-daemon
 7488 anselm    20   0 1568m 612m 7620 S  0.3 16.0   1:48.91 kded4
 7558 root      20   0  216m 3672 2244 S  0.3  0.1   0:41.89 upowerd
 7762 anselm    20   0  9240  972  716 S  0.3  0.0   5:09.67 ksysguardd
 7817 anselm    20   0  285m 5644 4904 S  0.3  0.1   0:01.33 akonadi_ical_re
 7823 anselm    20   0  268m 5696 4804 S  0.3  0.1   0:01.33 akonadi_vcard_r
22222 anselm    20   0  946m  16m 7160 S  0.3  0.4   0:01.67 chromium
29877 anselm    20   0  346m  36m  25m S  0.3  0.9   0:02.15 ksnapshot
29925 anselm    20   0 21740 1648 1076 R  0.3  0.0   0:00.94 top
    1 root      20   0 10632  644  612 S  0.0  0.0   0:02.45 init
```

Figure 12.1: The top program

Copyright © 2012 Linup Front GmbH

- In the top part of the output, the first line shows the current wallclock time and the "uptime", i. e., the amount of time elapsed since the system was booted (here, four days, fourteen hours, and change) and the number of logged-in users (the "11" here shouldn't be taken too seriously; every session in a terminal window counts as a user). On the right there are three numbers, the so-called load averages, which describe the system load.

  The load averages specify the number of runnable processes (state `R`), averaged over the last minute, the last five minutes, and the last fifteen minutes respectively. The utility of these values should not be overestimated (!); they don't really tell you all that much. If the value for the last minute is high and the one for the last 15 minutes low, then your system has suddenly got a lot more to do; if the value for the last minute is low but that for the last 15 minutes is high, your system used to have a lot to do but that is over now.

  If the load averages are constantly lower than the number of processor cores in your system, this means that you have unnecessarily overspent on an expensive processor. On an eight-core system, for example, values around 8 (which traditionally would have sent cold shivers down a system administrator's spine) are completely unremarkable; values that are a lot lower than 8 over prolonged periods of time are pathetic.

- The second line gives the number of processes and how they are distributed over the various process states.

- The third line contains percentages according to the type of CPU utilisation: "`us`" is the execution of code outside and "`sy` that of code inside the operating system kernel. "`ni`" is code outside the operating system kernel whose priority was deliberately reduced by a user, and "`id`" is doing nothing. "`wa`" is waiting for I/O, and the other three columns are not that interesting.

- The two following lines basically correspond to the output of `free`.

- The lower part of the screen is a process list similar to that of "`ps l`". Like the upper part, it is updated every few seconds and, by default, is sorted according to the percentage of CPU time the processes in the list are using (the process that the system spends most time on is leading the list).

  If you press the m key, the list is sorted according to memory use—the most obese process is on top. With p, you can go back to the CPU-time list.

You can use the h key to display a help page inside `top`. The `top`(1) man page explains the output and the possible key combinations in detail, and shows you how to adapt the content of the process list to your requirements.

Exercises

12.4 [1] With the `ps` option, `ax`, you can display all processes on the system. Look at the list. Which processes do you recognise?

12.5 [2] Start a long-running process inside a shell session (something like "`sleep 120`" should do). In another session, invoke "`ps ax`" and try to locate the process in the output. (*Hint:* `grep` is your friend.)

12.6 [!2] Use `top` to find out which processes are currently using the most CPU time. Which processes are using the most memory?

12.4 Package Management

Contemporary Linux distributions normally consist of a multitude (typically thousands) of "packages", each containing everything necessary for a certain part of the system's functionality: executable programs, libraries, documentation, … When initially configuring a Linux computer, you as the administrator can specify which packages should be installed on the computer, and of course you can always add arbitrary packages from your distribution later, or remove unused ones.

The details of how functionality is split into packages depends on the distribution. With libraries, it is usual to distinguish between a "run-time package" and a "development package". The run-time package contains the files that must be installed so other programs can use the library (like the actual dynamically loadable library in a `.so` file). You need to install the development package only if you intend to *compile* new or existing programs using the library—this contains the information the C compiler needs to use the library ("include files"), a statically linkable library for debugging, or documentation about the library's content. If the documentation is large, it may be separated into yet another package.

For example, here is the package split for the `rsvg` library (which deals with SVG-format graphics) according to Debian GNU/Linux 6.0 ("Squeeze"):

`librsvg2-2`	*The actual (run-time) library*
`librsvg2-dev`	*Development package*
`librsvg2-bin`	*Command-line programs*
`librsvg2-dbg`	*Debugging information*
`librsvg2-doc`	*Documentation*
`librsvg2-common`	*More command-line programs*
`python-rsvg`	*Python language binding*
`libimage-librsvg-perl`	*Perl language binding*

package database

On every Linux computer[1] there is a "package database" containing information about which packages the computer is aware of and which of those are currently installed. You can periodically synchronise the package database with your distribution's "repositories", or servers containing packages, and thus find out which of the packages on your computer are out of date because the distribution offers newer versions. The package management system then usually offers you the opportunity to selectively update the packages in question.

How well that works in practice depends (once again) on your distribution. In particular, the issue may be more complicated than it seems: The newer version of a package could, for example, require that a library (available as its own package) must be installed in a newer version, and that can lead to problems if another installed program absolutely requires the *old* version of the library. Sometimes it is possible that a package cannot be updated without making drastic changes elsewhere in the system. Good package management systems detect such situations and warn you as the administrator and/or give you the opportunity to intervene.

As pointed out in Section 2.4.7, the major Linux distributions use either of two different package management systems, both of which come with their own tools and their own format for package files—the package management system of Debian GNU/Linux and its derivatives, and the RPM package manager as used by Red Hat, SUSE etc. In principle, both solve the same problem, but differ in details such as the commands used for package management. For example, on an

[1] Those using one of the major distributions, at any rate—there are a few distributions that seem to get by without a package management system, but these are for nerds.

RPM-based system such as RHEL, Fedora, or openSUSE you can display a list of all installed packages using the

```
$ rpm --query --all                                        “-qa” would do
```

command, while a Debian-based system would require the

```
$ dpkg --list                                               “-l” would do
```

command instead.

The package databases themselves are usually found below `/var/lib`; on Debian-like systems within `/var/lib/dpkg` (`/var/cache/apt` contains the tables of contents of repository servers as well as any downloaded package files), and within `/var/lib/rpm` on RPM-based systems.

Today, programs such as `dpkg` and `rpm` form the “foundation” of a package management system. Administrators prefer more convenient tools that build on these base programs and include, for example, easy access to package repositories and the automatic resolution of dependencies between packages. In the Debian world, these include “Aptitude” and “Synaptic”, while, on the RPM side of things, Red Hat relies on a program called YUM and SUSE on one called “Zypper” (even though package management has also been integrated into the general administration tool, YaST).

Some of these tools are even independent of the underlying package management system. “PackageKit”, for example, can not only use either Debian or RPM package management, but also, under controlled circumstances, allow normal users without administrator privileges to install or update packages.

Exercises

12.7 [2] How many packages are installed on your system? Use the `rpm` or `dpkg` invocation shown above and count the lines in the output. (*Caution:* “`dpkg --list`” also displays packages that used to be installed but have been removed or superseded by newer versions. Count only those lines in the output that begin with “`ii`”.)

Commands in this Chapter

dpkg	Debian GNU/Linux package management tool	dpkg(8)	166
free	Displays main memory and swap space usage	free(1)	162
ps	Outputs process status information	ps(1)	161
rpm	Package management tool used by various Linux distributions (Red Hat, SUSE, …)	rpm(8)	166
su	Starts a shell using a different user’s identity	su(1)	158
sudo	Allows normal users to execute certain commands with administrator privileges	sudo(8)	159
top	Screen-oriented tool for process monitoring and control	top(1)	162

Copyright © 2012 Linup Front GmbH

Summary

- Linux distinguishes between “normal” users and the system administrator, `root`. `root` is exempt from the usual privilege checks.
- As a normal user, you can obtain temporary administrator privileges by means of `su` or `sudo`.
- A Linux computer’s configuration is contained in text files within the `/etc` directory.
- Processes are programs that are being executed.
- Commands such as `ps` and `top` allow you insight into the current system state.
- The important Linux distributions use either the package management system of GNU/Linux or the RPM system originally developed by Red Hat.
- Based on foundation tools, most distributions offer convenient software for managing, installing, and removing software packages including dependencies.

Copyright © 2012 Linup Front GmbH

13

User Administration

Contents

Goals

- Understanding the user and group concepts of Linux
- Knowing how user and group information is stored on Linux
- Being able to use the user and group administration commands

Prerequisites

- Knowledge about handling configuration files

13.1 Basics

13.1.1 Why Users?

Computers used to be large and expensive, but today an office workplace without its own PC ("personal computer") is nearly inconceivable, and a computer is likely to be encountered in most domestic "dens" as well. And while it may be sufficient for a family to agree that Dad, Mom and the kids will put their files into different directories, this will no longer do in companies or universities—once shared disk space or other facilities are provided by central servers accessible to many users, the computer system must be able to distinguish between different users and to assign different access rights to them. After all, Ms Jones from the Development Division has as little business looking at the company's payroll data as Mr Smith from Human Resources has accessing the detailed plans for next year's products. And a measure of privacy may be desired even at home—the Christmas present list or teenage daughter's diary (erstwhile fitted with a lock) should not be open to prying eyes as a matter of course.

We shall be discounting the fact that teenage daughter's diary may be visible to the entire world on Facebook (or some such); and even if that is the case, the entire world should surely not be allowed to *write* to teenage daughter's dairy. (Which is why even Facebook supports the notion of different users.)

The second reason for distinguishing between different users follows from the fact that various aspects of the system should not be visible, much less changeable, without special privileges. Therefore Linux manages a separate user identity (`root`) for the system administrator, which makes it possible to keep information such as users' passwords hidden from "common" users. The bane of older Windows systems—programs obtained by e-mail or indiscriminate web surfing that then wreak havoc on the entire system—will not plague you on Linux, since anything you can execute as a common user will not be in a position to wreak system-wide havoc.

Unfortunately this is not entirely correct: Every now and then a bug comes to light that enables a "normal user" to do things otherwise restricted to administrators. This sort of error is extremely nasty and usually corrected very quickly after having been found, but there is a considerable chance that such a bug has remained undetected in the system for an extended period of time. Therefore, on Linux (as on all other operating systems) you should strive to run the most current version of critical system parts like the kernel that your distributor supports.

Even the fact that Linux safeguards the system configuration from unauthorised access by normal users should not entice you to shut down your brain. We do give you some advice (such as not to log in to the graphical user interface as `root`), but you should keep thinking along. E-mail messages asking you to view web site *X* and enter your credit card number and PIN there can reach you even on Linux, and you should disregard them in the same way as everywhere else.

user accounts

Linux distinguishes between different users by means of different **user accounts**. The common distributions typically create two user accounts during installation, namely `root` for administrative tasks and another account for a "normal" user. You (as the administrator) may add more accounts later, or, on a client PC in a larger network, they may show up automatically from a user account database stored elsewhere.

Linux distinguishes between *user accounts*, not users. For example, no one keeps you from using a separate user account for reading e-mail and surfing the web, if you want to be 100% sure that things you download from the

Net have no access to your important data (which might otherwise happen in spite of the user/administrator divide). With a little cunning you can even display a browser and e-mail program running under your "surfing account" among your "normal" programs[1].

Under Linux, every user account is assigned a unique number, the so-called *user ID* (or **UID**, for short). Every user account also features a textual **user name** (such as `root` or `joe`) which is easier to remember for humans. In most places where it counts—e. g., when logging in, or in a list of files and their owners—Linux will use the textual name whenever possible.

UID
user name

💡 The Linux kernel does not know anything about textual user names; process data and the ownership data in the filesystem use the UID exclusively. This may lead to difficulties if a user is deleted while he still owns files on the system, and the UID is reassigned to a different user. That user "inherits" the previous UID owner's files.

💡 There is no technical problem with assigning the same (numerical) UID to different user names. These users have equal access to all files owned by that UID, but every user can have his own password. You should not actually use this (or if you do, use it only with great circumspection).

13.1.2 Users and Groups

To work with a Linux computer you need to log in first. This allows the system to recognise you and to assign you the correct access rights (of which more later). Everything you do during your session (from logging in to logging out) happens under your user account. In addition, every user has a **home directory**, where only they can store and manage their own files, and where other users often have no read permission and very emphatically no write permission. (Only the system administrator—`root`—may read and write all files.)

home directory

⚠ Depending on which Linux distribution you use (cue: Ubuntu) it may be possible that you do not have to log into the system explicitly. This is because the computer "knows" that it will usually be you and simply assumes that this is going to be the case. You are trading security for convenience; this particular deal probably makes sense only where you can stipulate with reasonable certainty that nobody except you will switch on your computer—and hence should be restricted *by rights* to the computer in your single-person household without a cleaner. We told you so.

Several users who want to share access to certain system resources or files can form a **group**. Linux identifies group members either fixedly by name or transiently by a login procedure similar to that for users. Groups have no "home directories" like users do, but as the administrator you can of course create arbitrary directories meant for certain groups and having appropriate access rights.

group

Groups, too, are identified internally using numerical identifiers ("group IDs" or GIDs).

💡 Group names relate to GIDs as user names to UIDs: The Linux kernel only knows about the former and stores only the former in process data or the file system.

Every user belongs to a *primary group* and possibly several *secondary* or *additional groups*. In a corporate setting it would, for example, be possible to introduce project-specific groups and to assign the people collaborating on those projects to the appropriate group in order to allow them to manage common data in a directory only accessible to group members.

[1] Which of course is slightly more dangerous again, since programs runninig on the same screen can communicate with one another

Copyright © 2012 Linup Front GmbH

For the purposes of access control, all groups carry equivalent weight—every user always enjoys all rights deriving from all the groups that he is a member of. The only difference between the primary and secondary groups is that files newly created by a user are usually[2] assigned to his primary group.

Up to (and including) version 2.4 of the Linux kernel, a user could be a member of at most 32 additional groups; since Linux 2.6 the number of secondary groups is unlimited.

You can find out a user account's UID, the primary and secondary groups and the corresponding GIDs by means of the `id` program:

```
$ id
uid=1000(joe) gid=1000(joe) groups=24(cdrom),29(audio),44(video),▷
 ◁ 1000(joe)
$ id root
uid=0(root) gid=0(root) groups=0(root)
```

With the options `-u`, `-g`, and `-G`, `id` lets itself be persuaded to output just the account's UID, the GID of the primary group, or the GIDs of the secondary groups. (These options cannot be combined.) With the additional option `-n` you get names instead of numbers:

```
$ id -G
1000 24 29 44
$ id -Gn
joe cdrom audio video
```

The `groups` command yields the same result as the ”‘`id -Gn`”’ command.

last You can use the `last` command to find who logged into your computer and when (and, in the case of logins via the network, from where):

```
$ last
joe      pts/1       pcjoe.example.c  Wed Feb 29 10:51   still logged in
bigboss  pts/0       pc01.example.c   Wed Feb 29 08:44   still logged in
joe      pts/2       pcjoe.example.c  Wed Feb 29 01:17 - 08:44  (07:27)
sue      pts/0       :0               Tue Feb 28 17:28 - 18:11  (00:43)
◁◁◁◁◁
reboot   system boot 3.2.0-1-amd64    Fri Feb  3 17:43 - 13:25 (4+19:42)
◁◁◁◁◁
```

For network-based sessions, the third column specifies the name of the `ssh` client computer. "`:0`" denotes the graphical screen (the first X server, to be exact—there might be more than one).

Do also note the `reboot` entry, which tells you that the computer was started. The third column contains the version number of the Linux operating system kernel as provided by "`uname -r`".

With a user name, `last` provides information about a particular user:

```
$ last
joe      pts/1       pcjoe.example.c  Wed Feb 29 10:51   still logged in
joe      pts/2       pcjoe.example.c  Wed Feb 29 01:17 - 08:44  (07:27)
◁◁◁◁◁
```

[2] The exception occurs where the owner of a directory has decreed that new files and subdirectories within this directory are to be assigned to the same group as the directory itself. We mention this strictly for completeness.

You might be bothered (and rightfully so!) by the fact that this somewhat sensitive information is apparently made available on a casual basis to arbitrary system users. If you (as the administrator) want to protect your users' privacy somewhat better than you Linux distribution does by default, you can use the

```
# chmod o-r /var/log/wtmp
```

command to remove general read permissions from the file that `last` consults for the telltale data. Users without administrator privileges then get to see something like

```
$ last
last: /var/log/wtmp: Permission denied
```

13.1.3 People and Pseudo-Users

Besides "natural" persons—the system's human users—the user and group concept is also used to allocate access rights to certain parts of the system. This means that, in addition to the personal accounts of the "real" users like you, there are further accounts that do not correspond to actual human users but are assigned to administrative functions internally. They define functional "roles" with their own accounts and groups.

pseudo-users

After installing Linux, you will find several such pseudo-users and groups in the `/etc/passwd` and `/etc/group` files. The most important role is that of the `root` user (which you know) and its eponymous group. The UID and GID of `root` are 0 (zero).

`root`'s privileges are tied to UID 0; GID 0 does not confer any additional access privileges.

Further pseudo-users belong to certain software systems (e. g., `news` for Usenet news using INN, or `postfix` for the Postfix mail server) or certain components or devices (such as printers, tape or floppy drives). You can access these accounts, if necessary, like other user accounts via the `su` command. These pseudo-users are helpful as file or directory owners, in order to fit the access rights tied to file ownership to special requirements without having to use the `root` account. The same appkies to groups; the members of the `disk` group, for example, have block-level access to the system's disks.

pseudo-users for privileges

Exercises

13.1 [1] How does the operating system kernel differentiate between various users and groups?

13.2 [2] What happens if a UID is assigned to two different user names? Is that allowed?

13.3 [1] What is a pseudo-user? Give examples!

13.4 [2] (On the second reading.) Is it acceptable to assign a user to group `disk` who you would not want to trust with the `root` password? Why (not)?

13.2 User and Group Information

13.2.1 The `/etc/passwd` File

The `/etc/passwd` file is the system user database. There is an entry in this file for every user on the system—a line consisting of attributes like the Linux user name,

Copyright © 2012 Linup Front GmbH

"real" name, etc. After the system is first installed, the file contains entries for most pseudo-users.

The entries in `/etc/passwd` have the following format:

⟨*user name*⟩:⟨*password*⟩:⟨*UID*⟩:⟨*GID*⟩:⟨*GECOS*⟩:⟨*home directory*⟩:⟨*shell*⟩

⟨*user name*⟩ This name should consist of lowercase letters and digits; the first character should be a letter. Unix systems often consider only the first eight characters—Linux does not have this limitation but in heterogeneous networks you should take it into account.

Resist the temptation to use umlauts, punctuation and similar special characters in user names, even if the system lets you do so—not all tools that create new user accounts are picky, and you could of course edit `/etc/passwd` by hand. What seems to work splendidly at first glance may lead to problems elsewhere later.

You should also stay away from user names consisting of only uppercase letters or only digits. The former may give their owners trouble logging in (see Exercise 13.6), the latter can lead to confusion, especially if the numerical user name does not equal the account's numerical UID. Commands such as ”‘`ls -l`”’ will display the UID if there is no corresponding entry for it in `/etc/passwd`, and it is not exactly straightforward to tell UIDs from purely numerical user names in `ls` output.

⟨*password*⟩ Traditionally, this field contains the user's encrypted password. Today, most Linux distributions use "shadow passwords"; instead of storing the password in the publically readable `/etc/passwd` file, it is stored in `/etc/shadow` which can only be accessed by the administrator and some privileged programs. In `/etc/passwd`, a "`x`" calls attention to this circumstance. Every user can avail himself of the `passwd` program to change his password.

⟨*UID*⟩ The numerical user identifier—a number between 0 and $2^{32} - 1$. By convention, UIDs from 0 to 99 (inclusive) are reserved for the system, UIDs from 100 to 499 are for use by software packages if they need pseudo-user accounts. With most popular distributions, "real" users' UIDs start from 500 (or 1000).

Precisely because the system differentiates between users not by name but by UID, the kernel treats two accounts as completely identical if they contain different user names but the same UID—at least as far as the access privileges are concerned. Commands that display a user name (e. g., ”‘`ls -l`”’ or `id`) show the one used when the user logged in.

primary group

⟨*GID*⟩ The GID of the user's **primary group** after logging in.

The Novell/SUSE distributions (among others) assign a single group such as `users` as the shared primary group of all users. This method is quite established as well as easy to understand.

Many distributions, such as those by Red Hat or Debian GNU/Linux, create a new group whenever a new account is created, with the GID equalling the account's UID. The idea behind this is to allow more sophisticated assignments of rights than with the approach that puts all users into the same group `users`. Consider the following situation: Jim (user name `jim`) is the personal assistant of CEO Sue (user name `sue`). In this capacity he sometimes needs to access files stored inside Sue's home directory that other users should not be able to get at. The method used by Red Hat, Debian & co., "one group per user", makes it straightforward to put user `jim` into group `sue` and to arrange for Sue's

Copyright © 2012 Linup Front GmbH

files to be readable for all group members (the default case) but not others. With the "one group for everyone" approach it would have been necessary to introduce a new group completely from scratch, and to reconfigure the `jim` and `sue` accounts accordingly.

By virtue of the assignment in `/etc/passwd`, every user must be a member of at least one group.

The user's secondary groups (if applicable) are determined from entries in the `/etc/group` file.

⟨*GECOS*⟩ This is the comment field, also known as the "GECOS field".

GECOS stands for "General Electric Comprehensive Operating System" and has nothing whatever to do with Linux, except that in the early days of Unix this field was added to `/etc/passwd` in order to keep compatibility data for a GECOS remote job entry service.

This field contains various bits of information about the user, in particular his "real" name and optional data such as the office number or telephone number. This information is used by programs such as `mail` or `finger`. The full name is often included in the sender's address by news and mail software.

Theoretically there is a program called `chfn` that lets you (as a user) change the content of your GECOS field. Whether that works in any particular case is a different question, since at least in a corporate setting one does not necessarily want to allow people to change their names at a whim.

⟨*home directory*⟩ This directory is that user's personal area for storing his own files. A newly created home directory is by no means empty, since a new user normally receives a number of "profile" files as his basic equipment. When a user logs in, his shell uses his home directory as its current directory, i. e., immediately after logging in the user is deposited there.

⟨*shell*⟩ The name of the program to be started by `login` after successful authentication—this is usually a shell. The seventh field extends through the end of the line.

The user can change this entry by means of the `chsh` program. The eligible programs (shells) are listed in the `/etc/shells` file. If a user is not supposed to have an interactive shell, an arbitrary program, with arguments, can be entered here (a common candidate is `/bin/true`). This field may also remain empty, in which case the standard shell `/bin/sh` will be started.

If you log in to a graphical environment, various programs will be started on your behalf, but not necessarily an interactive shell. The shell entry in `/etc/passwd` comes into its own, however, when you invoke a terminal emulator such as `xterm` or `konsole`, since these programs usually check it to identify your preferred shell.

Some of the fields shown here may be empty. Absolutely necessary are only the user name, UID, GID and home directory. For most user accounts, all the fields will be filled in, but pseudo-users might use only part of the fields.

home directories

The home directories are usually located below `/home` and take their name from their owner's user name. In general this is a fairly sensible convention which makes a given user's home directory easy to find. In theory, a home directory might be placed anywhere in the file system under a completely arbitrary name.

On large systems it is common to introduce one or more additional levels of directories between `/home` and the "user name" directory, such as

Copyright © 2012 Linup Front GmbH

`/home/hr/joe`	*Joe from Human Resources*
`/home/devel/sue`	*Sue from Development*
`/home/exec/bob`	*Bob the CEO*

There are several reasons for this. On the one hand this makes it easier to keep one department's home directory on a server within that department, while still making it available to other client computers. On the other hand, Unix (and some Linux) file systems used to be slow dealing with directories containing very many files, which would have had an unfortunate impact on a `/home` with several thousand entries. However, with current Linux file systems (`ext3` with `dir_index` and similar) this is no longer an issue.

tools

Note that as an administrator you should not really be editing `/etc/passwd` by hand. There is a number of programs that will help you create and maintain user accounts.

In principle it is also possible to store the user database elsewhere than in `/etc/passwd`. On systems with very many users (thousands), storing user data in a relational database is preferable, while in heterogeneous networks a shared multi-platform user database, e. g., based on an LDAP directory, might recommend itself. The details of this, however, are beyond the scope of this course.

13.2.2 The `/etc/shadow` File

For security, nearly all current Linux distributions store encrypted user passwords in the `/etc/shadow` file ("shadow passwords"). This file is unreadable for normal users; only `root` may write to it, while members of the `shadow` group may read it in addition to `root`. If you try to display the file as a normal user an error occurs.

Use of `/etc/shadow` is not mandatory but highly recommended. However there may be system configurations where the additional security afforded by shadow passwords is nullified, for example if NIS is used to export user data to other hosts (especially in heterogeneous Unix environments).

format

Again, this file contains one line for each user, with the following format:

```
⟨user name⟩:⟨password⟩:⟨change⟩:⟨min⟩:⟨max⟩▷
 ◁:⟨warn⟩:⟨grace⟩:⟨lock⟩:⟨reserved⟩
```

For example:

```
root:gaY2L19jxzHj5:10816:0:10000::::
daemon:*:8902:0:10000::::
joe:GodY6c5pZk1xs:10816:0:10000::::
```

Here is the meaning of the individual fields:

user name This must correspond to an entry in the `/etc/passwd` file. This field "joins" the two files.

password The user's encrypted password. An empty field generally means that the user can log in without a password. An asterisk or an exclamation point prevent the user in question from logging in. It is common to lock user's accounts without deleting them entirely by placing an asterisk or exclamation point at the beginning of the corresponding password.

change The date of the last password change, in days since 1 January 1970.

min The minimal number of days that must have passed since the last password change before the password may be changed again.

max The maximal number of days that a password remains valid without having to be changed. After this time has elapsed the user must change his password.

warn The number of days before the expiry of the ⟨*max*⟩ period that the user will be warned about having to change his password. Generally, the warning appears when logging in.

grace The number of days, counting from the expiry of the ⟨*max*⟩ period, after which the account will be locked if the user does not change his password. (During the time from the expiry of the ⟨*max*⟩ period and the expiry of this grace period the user may log in but must immediately change his password.)

lock The date on which the account will be definitively locked, again in days since 1 January 1970.

Some brief remarks concerning password encryption are in order. You might think that if passwords are encrypted they can also be *decrypted* again. This would open all of the system's accounts to a clever cracker who manages to obtain a copy of `/etc/shadow`. However, in reality this is not the case, since password "encryption" is a one-way street. It is impossible to recover the decrypted representation of a Linux password from the "encrypted" form because the method used for encryption prevents this. The only way to "crack" the encryption is by encrypting likely passwords and checking whether they match what is in `/etc/shadow`.

Let's assume you select the characters of your password from the 95 visible ASCII characters (uppercase and lowercase letters are distinguished). This means that there are 95 different one-character passwords, $95^2 = 9025$ two-character passwords, and so on. With eight characters you are already up to 6.6 quadrillion ($6.6 \cdot 10^{15}$) possibilities. Stipulating that you can trial-encrypt 250,000 passwords per second (not entirely unrealistic on current hardware), this means you would require approximately 841 years to work through all possible passwords.

But do not feel too safe yet. The traditional method (usually called "crypt" or "DES"—the latter because it is based on, but not identical to, the eponymous encryption method[3]) should no longer be used if you can avoid it. It has the unpleasant property of only looking at the first eight characters of the entered password, and clever crackers can nowadays buy enough disk space to build a pre-encrypted cache of the 50 million (or so) most common passwords. To "crack" a password they only need to search their cache for the encrypted password, which can be done extremely quickly, and read off the corresponding clear-text password.

To make things even more laborious, when a newly entered password is encrypted the system traditionally adds a random element (the so-called "salt") which selects one of 4096 different possibilities for the encrypted password. The main purpose of the salt is to avoid random hits resulting from user X, for some reason or other, getting a peek at the content of `/etc/shadow` and noting that his encrypted password looks just like that of user Y (hence letting him log into user Y's account using his own *clear-text* password). For a pleasant side effect, the disk space required for the

[3] If you must know exactly: The clear-text password is used as the key (!) to encrypt a constant string (typically a sequence of zero bytes). A DES key is 56 bits, which just happens to be 8 characters of 7 bits each (as the leftmost bit in each character is ignored). This process is repeated for a total of 25 rounds, with the previous round's output serving as the new input. Strictly speaking the encryption scheme used isn't quite DES but changed in a few places, to make it less feasible to construct a special password-cracking computer from commercially available DES encryption chips.

Copyright © 2012 Linup Front GmbH

cracker's pre-encrypted dictionary from the previous paragraph is blown up by a factor of 4096.

Nowadays, password encryption is commonly based on the MD5 algorithm, allows for passwords of arbitrary length and uses a 48-bit salt instead of the traditional 12 bits. Kindly enough, the encryption works much more slowly than "crypt", which is irrelevant for the usual purpose (checking a password upon login—you can still encrypt several hundred passwords per second) but does encumber clever crackers to a certain extent. (You should not let yourself be bothered by the fact that cryptographers poo-poo the MD5 scheme as such due to its insecurity. As far as password encryption is concerned, this is quite meaningless.)

You should not expect too much of the various password administration parameters. They are being used by the text console login process, but whether other parts of the system (such as the graphical login screen) pay them any notice depends on your setup. Nor is there usually an advantage in forcing new passwords on users at short intervals—this usually results in a sequence like `bob1`, `bob2`, `bob3`, …, or users alternate between two passwords. A *minimal interval* that must pass before a user is allowed to change their password again is outright dangerous, since it may give a cracker a "window" for illicit access even though the user knows their password has been compromised.

The problem you need to cope with as a system administrator is usually not people trying to crack your system's passwords by "brute force". It is much more promising, as a rule, to use "social engineering". To guess your password, the clever cracker does not start at `a`, `b`, and so on, but with your spouse's first name, your kids' first names, your car's plate number, your dog's birthday et cetera. (We do not in any way mean to imply that *you* would use such a stupid password. No, no, not *you* by any means. However, we are not quite so positive about your boss …) And then there is of course the time-honoured phone call approach: "Hi, this is the IT department. We're doing a security systems test and urgently require your user name and password."

There are diverse ways of making Linux passwords more secure. Apart from the improved encryption scheme mentioned above, which by now is used by default by most Linux distributions, these include complaining about (too) weak passwords when they are first set up, or proactively running software that will try to identify weak encrypted passwords, just like clever crackers would (*Caution:* Do this in your workplace only with written (!) pre-approval from your boss!). Other methods avoid passwords completely in favour of constantly changing magic numbers (as in SecurID) or smart cards. All of this is beyond the scope of this manual, and therefore we refer you to the Linup Front manual *Linux Security*.

13.2.3 The `/etc/group` File

group database

By default, Linux keeps group information in the `/etc/group` file. This file contains one-line entry for each group in the system, which like the entries in `/etc/passwd` consists of fields separated by colons (`:`). More precisely, `/etc/group` contains four fields per line.

⟨*group name*⟩`:`⟨*password*⟩`:`⟨*GID*⟩`:`⟨*members*⟩

Their meaning is as follows:

⟨*group name*⟩ The name of the group, for use in directory listings, etc.

⟨*password*⟩ An optional password for this group. This lets users who are not members of the group via `/etc/shadow` or `/etc/group` assume membership of the group using `newgrp`. A "*" as an invalid character prevents normal users from changing to the group in question. A "x" refers to the separate password file /etc/gshadow.

⟨*GID*⟩ The group's numerical group identifier.

⟨*Members*⟩ A comma-separated list of user names. This list contains all users who have this group as a secondary group, i. e., who are members of this group but have a different value in the GID field of their /etc/passwd entry. (Users with this group as their primary group may also be listed here but that is unnecessary.)

A /etc/group file could, for example, look like this:

```
root:x:0:root
bin:x:1:root,daemon
users:x:100:
project1:x:101:joe,sue
project2:x:102:bob
```

administrative groups

The entries for the root and bin groups are entries for administrative groups, similar to the system's pseudo-user accounts. Many files are assigned to groups like this. The other groups contain user accounts.

GID values

Like UIDs, GIDs are counted from a specific value, typically 100. For a valid entry, at least the first and third field (group name and GID) must be filled in. Such an entry assigns a GID (which might occur in a user's primary GID field in /etc/passwd) a textual name.

membership list

group password

The password and/or membership fields must only be filled in for groups that are assigned to users as secondary groups. The users listed in the membership list are not asked for a password when they want to change GIDs using the newgrp command. If an encrypted password is given, users without an entry in the membership list can authenticate using the password to assume membership of the group.

In practice, group passwords are hardly if ever used, as the administrative overhead barely justifies the benefits to be derived from them. On the one hand it is more convenient to assign the group directly to the users in question (since, from version 2.6 of the Linux kernel on, there is no limit to the number of secondary groups a user can join), and on the other hand a *single* password that must be known by *all* group members does not exactly make for bullet-proof security.

If you want to be safe, ensure that there is an asterisk ("*") in every group password slot.

13.2.4 The /etc/gshadow File

As for the user database, there is a shadow password extension for the group database. The group passwords, which would otherwise be encrypted but readable for anyone in /etc/group (similar to /etc/passwd), are stored in the separate file /etc/gshadow. This also contains additional information about the group, for example the names of the group administrators who are entitled to add or remove members from the group.

Exercises

13.5 [1] Which value will you find in the second column of the /etc/passwd file? Why do you find that value there?

13.6 [2] Switch to a text console (using, e. g., Alt+F1) and try logging in but enter your user name in uppercase letters. What happens?

Copyright © 2012 Linup Front GmbH

13.3 Managing User Accounts and Group Information

After a new Linux distribution has been installed, there is often just the `root` account for the system administrator and the pseudo-users' accounts. Any other user accounts must be created first (and most distributions today will gently but firmly nudge the installing person to create at least *one* "normal" user account).

tools for user management

As the administrator, it is your job to create and manage the accounts for all required users (real and pseudo). To facilitate this, Linux comes with several tools for user management. With them, this is mostly a straightforward task, but it is important that you understand the background.

13.3.1 Creating User Accounts

The procedure for creating a new user account is always the same (in principle) and consists of the following steps:

1. You must create entries in the `/etc/passwd` (and possibly `/etc/shadow`) files.
2. If necessary, an entry (or several) in the `/etc/group` file is necessary.
3. You must create the home directory, copy a basic set of files into it, and transfer ownership of the lot to the new user.
4. If necessary, you must enter the user in further databases, e. g., for disk quotas, database access privilege tables and special applications.

All files involved in adding a new account are plain text files. You can perform each step manually using a text editor. However, as this is a job that is as tedious as it is elaborate, it behooves you to let the system help you, by means of the `useradd` program.

`useradd`

In the simplest case, you pass `useradd` merely the new user's user name. Optionally, you can enter various other user parameters; for unspecified parameters (typically the UID), "reasonable" default values will be chosen automatically. On request, the user's home directory will be created and endowed with a basic set of files that the program takes from the `/etc/skel` directory. The `useradd` command's syntax is:

```
useradd [⟨options⟩] ⟨user name⟩
```

The following options (among others) are available:

`-c` *comment* GECOS field entry

`-d` *home directory* If this option is missing, `/home/`⟨*user name*⟩ is assumed

`-e` *date* On this date the account will be deactivated automatically (format "YYYY-MM-DD")

`-g` *group* The new user's primary group (name or GID). This group must exist.

`-G` *group*[,*group*]... Supplementary groups (names or GIDs). These groups must also exist.

`-s` *shell* The new user's login shell

`-u` *UID* The new user's numerical UID. This UID must not be already in use, unless the "`-o`" option is given

`-m` Creates the home directory and copies the basic set of files to it. These files come from `/etc/skel`, unless a different directory was named using "`-k` ⟨*directory*⟩".

For instance, the

Copyright © 2012 Linup Front GmbH

```
# useradd -c "Joe Smith" -m -d /home/joe -g devel -k /etc/skel.devel
```

command creates an account by the name of `joe` for a user called Joe Smith, and assigns it to the `devel` group. `joe`'s home directory is created as `/home/joe`, and the files from `/etc/skel.devel` are being copied into it.

With the `-D` option (on SUSE distributions, `--show-defaults`) you may set default values for some of the properties of new user accounts. Without additional options, the default values are displayed:

```
# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/sh
SKEL=/etc/skel
CREATE_MAIL_SPOOL=no
```

You can change these values using the `-g`, `-b`, `-f`, `-e`, and `-s` options, respectively:

```
# useradd -D -s /usr/bin/zsh                           zsh as the default shell
```

The final two values in the list cannot be changed.

`useradd` is a fairly low-level tool. In real life, you as an experienced administrator will likely not be adding new user accounts by means of `useradd`, but through a shell script that incorporates your local policies (just so you don't have to remember them all the time). Unfortunately you will have to come up with this shell script by yourself—at least unless you are using Debian GNU/Linux or one of its derivatives (see below).

Watch out: Even though every serious Linux distribution comes with a program called `useradd`, the implementations differ in their details.

The Red Hat distributions include a fairly run-of-the-mill version of `useradd`, without bells and whistles, which provides the features discussed above.

The SUSE distributions' `useradd` is geared towards optionally adding users to a LDAP directory rather than the `/etc/passwd` file. (This is why the `-D` option cannot be used to query or set default values like it can elsewhere—it is already spoken for to do LDAPy things.) The details are beyond the scope of this manual.

On Debian GNU/Linux and Ubuntu, `useradd` does exist but the recommended method to create new user accounts is a program called `adduser` (thankfully this is not confusing). The advantage of `adduser` is that it plays according to Debian GNU Linux's rules, and furthermore makes it possible to execute arbitrary other actions for a new account besides creating the actual account. For example, one might create a directory in a web server's document tree so that the new user (and nobody else) can publish files there, or the user could automatically be authorised to access a database server. You can find the details in `adduser`(8) and `adduser.conf`(5).

After it has been created using `useradd`, the new account is not yet accessible; the system administrator must first set up a password. We shall be explaining this presently. password

13.3.2 The passwd Command

The passwd command is used to set up passwords for users. If you are logged in as root, then

```
# passwd joe
```

asks for a new password for user john (You must enter it twice as it will not be echoed to the screen).

The passwd command is also available to normal users, to let them change their own passwords (changing other users' passwords is root's prerogative):

```
$ passwd
Changing password for joe.
(current) UNIX password: secret123
Enter new UNIX password: 321terces
Retype new UNIX password: 321terces
passwd: password updated successfully
```

Normal users must enter their own password correctly once before being allowed to set a new one. This is supposed to make life difficult for practical jokers that play around on your computer if you had to step out very urgently and didn't have time to engage the screen lock.

On the side, passwd serves to manage various settings in /etc/shadow. For example, you can look at a user's "password state" by calling the passwd command with the -S option:

```
# passwd -S bob
bob LK 10/15/99 0 99999 7 0
```

The first field in the output is (once more) the user name, followed by the password state: "PS" or "P" if a password is set, "LK" or "L" for a locked account, and "NP" for an account with no password at all. The other fields are, respectively, the date of the last password change, the minimum and maximum interval for changing the password, the expiry warning interval and the "grace period" before the account is locked completely after the password has expired. (See also Section 13.2.2.)

You can change some of these settings by means of passwd options. Here are a few examples:

```
# passwd -l joe                                   Lock the account
# passwd -u joe                                 Unlock the account
# passwd -m 7 joe                Password change at most every 7 days
# passwd -x 30 joe              Password change at lesat every 30 days
# passwd -w 3 joe           3 days grace period before password expires
```

Locking and unlocking accounts by means of -l and -u works by putting a "!" in front of the encrypted password in /etc/shadow. Since "!" cannot result from password encryption, it is impossible to enter something upon login that matches the "encrypted password" in the user database—hence access via the usual login procedure is prevented. Once the "!" is removed, the original password is back in force. (Astute, innit?) However, you should keep in mind that users may be able to gain access to the system by other means that do not refer to the encrypted password in the user database, such as the secure shell with public-key authentication.

Changing the remaining settings in /etc/shadow requires the chage command:

```
# chage -E 2009-12-01 joe                     Lock account from 1 Dec 2009
# chage -E -1 joe                                       Cancel expiry date
# chage -I 7 joe                 Grace period 1 week from password expiry
# chage -m 7 joe                                           Like passwd -m
# chage -M 7 joe                                     Like passwd -x (Grr.)
# chage -W 3 joe                                Like passwd -w (Grr, grr.)
```

(chage can change all settings that passwd can change, and then some.)

If you cannot remember the option names, invoke chage with the name of a user account only. The program will present you with a sequence of the current values to change or confirm.

You cannot retrieve a clear-text password even if you are the administrator. Even checking /etc/shadow doesn't help, since this file stores all passwords already encrypted. If a user forgets their password, it is usually sufficient to reset their password using the passwd command.

Should you have forgotten the root password and not be logged in as root by any chance, your last option is to boot Linux to a shell, or boot from a rescue disk or CD. After that, you can use an editor to clear the ⟨*password*⟩ field of the root entry in /etc/passwd.

Exercises

13.7 [3] Change user joe's password. How does the /etc/shadow file change? Query that account's password state.

13.8 [!2] The user dumbo has forgotten his paassword. How can you help him?

13.9 [!3] Adjust the settings for user joe's password such that he can change his password after at least a week, and must change it after at most two weeks. There should be a warning two days before the two weeks are up. Check the settings afterwards.

13.3.3 Deleting User Accounts

To delete a user account, you need to remove the user's entries from /etc/passwd and /etc/shadow, delete all references to that user in /etc/group, and remove the user's home directory as well as all other files created or owned by that user. If the user has, e. g., a mail box for incoming messages in /var/mail, that should also be removed.

userdel

Again there is a suitable command to automate these steps. The userdel command removes a user account completely. Its syntax:

```
userdel [-r] ⟨user name⟩
```

The -r option ensures that the user's home directory (including its content) and his mail box in /var/mail will be removed; other files belonging to the user—e. g., crontab files—must be delete manually. A quick way to locate and remove files belonging to a certain user is the

```
find / -uid ⟨UID⟩ -delete
```

command. Without the -roption, only the user information is removed from the user database; the home directory remains in place.

Copyright © 2012 Linup Front GmbH

13.3.4 Changing User Accounts and Group Assignment

User accounts and group assignments are traditionally changed by editing the /etc/passwd and /etc/group files. However, many systems contain commands like usermod and groupmod for the same purpose, and you should prefer these since they are safer and—mostly—more convenient to use.

usermod

The usermod program accepts mostly the same options as useradd, but changes existing user accounts instead of creating new ones. For example, with

```
usermod -g ⟨group⟩ ⟨user name⟩
```

you could change a user's primary group.

Changing UIDs

Caution! If you want to change an existing user account's UID, you could edit the ⟨*UID*⟩ field in /etc/passwd directly. However, you should at the same time transfer that user's files to the new UID using chown: "chown -R tux /home/tux" re-confers ownership of all files below user tux's home directory to user tux, after you have changed the UID for that account. If "ls -l" displays a numerical UID instead of a textual name, this implies that there is no user name for the UID of these files. You can fix this using chown.

13.3.5 Changing User Information Directly—vipw

The vipw command invokes an editor (vi or a different one) to edit /etc/passwd directly. At the same time, the file in question is locked in order to keep other users from simultaneously changing the file using, e. g., passwd (which changes would be lost). With the -s option, /etc/shadow can be edited.

The actual editor that is invoked is determined by the value of the VISUAL environment variable, alternatively that of the EDITOR environment variable; if neither exists, vi will be launched.

Exercises

13.10 [!2] Create a user called test. Change to the test account and create a few files using touch, including a few in a different directory than the home directory (say, /tmp). Change back to root and change test's UID. What do you see when listing user test's files?

13.11 [!2] Create a user called test1 using your distribution's graphical tool (if available), test2 by means of the useradd command, and another, test3, manually. Look at the configuration files. Can you work without problems using any of these three accounts? Create a file using each of the new accounts.

13.12 [!2] Delete user test2's account and ensure that there are no files left on the system that belong to that user.

13.13 [2] Change user test1's UID. What else do you need to do?

13.14 [2] Change user test1's home directory from /home/test1 to /home/user/test1.

13.3.6 Creating, Changing and Deleting Groups

Like user accounts, you can create groups using any of several methods. The "manual" method is much less tedious here than when creating new user accounts: Since groups do not have home directories, it is usually sufficient to edit the /etc/group file using any text editor, and to add a suitable new line (see below for vigr). When group passwords are used, another entry must be added to /etc/gshadow.

Incidentally, there is nothing wrong with creating directories for groups. Group members can place the fruits of their collective labour there. The approach is similar to creating user home directories, although no basic set of configuration files needs to be copied.

For group management, there are, by analogy to `useradd`, `usermod`, and `userdel`, the `groupadd`, `groupmod`, and `groupdel` programs that you should use in favour of editing `/etc/group` and `/etc/gshadow` directly. With `groupadd` you can create new groups simply by giving the correct command parameters:

`groupadd`

```
groupadd [-g ⟨GID⟩] ⟨group name⟩
```

The `-g` option allows you to specify a given group number. As mentioned before, this is a positive integer. The values up to 99 are usually reserved for system groups. If `-g` is not specified, the next free GID is used.

You can edit existing groups with `groupmod` without having to write to `/etc/group` directly:

`groupmod`

```
groupmod [-g ⟨GID⟩] [-n ⟨name⟩] ⟨group name⟩
```

The "`-g` ⟨*GID*⟩" option changes the group's GID. Unresolved file group assignments must be adjusted manually. The "`-n` ⟨*name*⟩" option sets a new name for the group without changing the GID; manual adjustments are not necessary.

There is also a tool to remove group entries. This is unsurprisingly called `groupdel`:

`groupdel`

```
groupdel ⟨group name⟩
```

Here, too, it makes sense to check the file system and adjust "orphaned" group assignments for files with the `chgrp` command. Users' primary groups may not be removed—the users in question must either be removed beforehand, or they must be reassigned to a different primary group.

The `gpasswd` command is mainly used to manipulate group passwords in a way similar to the `passwd` command. The system administrator can, however, delegate the administration of a group's membership list to one or more group administrators. Group administrators also use the `gpasswd` command:

`gpasswd`

group administrator

```
gpasswd -a ⟨user⟩ ⟨group⟩
```

adds the ⟨*user*⟩ to the ⟨*group*⟩, and

```
gpasswd -d ⟨user⟩ ⟨group⟩
```

removes him again. With

```
gpasswd -A ⟨user⟩,… ⟨group⟩
```

the system administrator can nominate users who are to serve as group administrators.

SUSE The SUSE distributions haven't included `gpasswd` for some time. Instead there are modified versions of the user and group administration tools that can handle an LDAP directory.

As the system administrator, you can change the group database directly using the `vigr` command. It works like `vipw`, by invoking an editor for "exclusive" access to `/etc/group`. Similarly, "`vigr -s`" gives you access to `/etc/gshadow`.

`vigr`

Copyright © 2012 Linup Front GmbH

Exercises

13.15 [2] What are groups needed for? Give possible examples.

13.16 [1] Can you create a directory that all members of a group can access?

13.17 [!2] Create a supplementary group `test`. Only user `test1` should be a member of that group. Set a group password. Log in as user `test1` or `test2` and try to change over to the new group.

Commands in this Chapter

`adduser`	Convenient command to create new user accounts (Debian)	`adduser`(8)	181
`chfn`	Allows users to change the GECOS field in the user database	`chfn`(1)	175
`gpasswd`	Allows a group administrator to change a group's membership and update the group password	`gpasswd`(1)	185
`groupadd`	Adds user groups to the system group database	`groupadd`(8)	185
`groupdel`	Deletes groups from the system group database	`groupdel`(8)	185
`groupmod`	Changes group entries in the system group database	`groupmod`(8)	185
`groups`	Displays the groups that a user is a member of	`groups`(1)	172
`id`	Displays a user's UID and GIDs	`id`(1)	172
`last`	List recently-logged-in users	`last`(1)	172
`useradd`	Adds new user accounts	`useradd`(8)	180
`userdel`	Removes user accounts	`userdel`(8)	183
`usermod`	Modifies the user database	`usermod`(8)	183
`vigr`	Allows editing `/etc/group` or `/etc/gshadow` with "file locking", to avoid conflicts	`vipw`(8)	185

Summary

- Access to the system is governed by user accounts.
- A user account has a numerical UID and (at least) one textual user name.
- Users can form groups. Groups have names and numerical GIDs.
- "Pseudo-users" and "pseudo-groups" serve to further refine access rights.
- The central user database is (normally) stored in the `/etc/passwd` file.
- The users' encrypted passwords are stored—together with other password parameters—in the `/etc/shadow` file, which is unreadable for normal users.
- Group information is stored in the `/etc/group` and `/etc/gshadow` files.
- Passwords are managed using the `passwd` program.
- The `chage` program is used to manage password parameters in `/etc/shadow`.
- User information is changed using `vipw` or—better—using the specialised tools `useradd`, `usermod`, and `userdel`.
- Group information can be manipulated using the `groupadd`, `groupmod`, `groupdel` and `gpasswd` programs.

Copyright © 2012 Linup Front GmbH

14

Access Control

Contents

Goals

- Understanding the Linux access control/privilege mechanisms
- Being able to assign access permissions to files and directories
- Knowing about SUID, SGID and the "sticky bit"

Prerequisites

- Knowledge of Linux user and group concepts (see Chapter 13)
- Knowledge of Linux files and directories

14.1 The Linux Access Control System

access control system

Whenever several users have access to the same computer system there must be an access control system for processes, files and directories in order to ensure that user A cannot access user B's private files just like that. To this end, Linux implements the standard system of Unix privileges.

separate privileges

access mode

In the Unix tradition, every file and directory is assigned to exactly one user (its owner) and one group. Every file supports separate privileges for its owner, the members of the group it is assigned to ("the group", for short), and all other users ("others"). Read, write and execute privileges can be enabled individually for these three sets of users. The owner may determine a file's access privileges. The group and others may only access a file if the owner confers suitable privileges to them. The sum total of a file's access permissions is also called its **access mode**.

access control

In a multi-user system which stores private or group-internal data on a generally accessible medium, the owner of a file can keep others from reading or modifying his files by instituting suitable access control. The rights to a file can be determined separately and independently for its owner, its group and the others. Access permissions allow users to map the responsibilities of a group collaborative process to the files that the group is working with.

14.2 Access Control For Files And Directories

14.2.1 The Basics

For each file and each directory in the system, Linux allows separate access rights for each of the three classes of users—owner, members of the file's group, others. These rights include read permission, write permission, and execute permission.

file permissions

As far as files are concerned, these permissions control approximately what their names suggest: Whoever has read permission may look at the file's content, whoever has write permission is allowed to change its content. Execute permission is necessary to launch the file as a process.

💡 Executing a binary "machine-language program" requires only execute permission. For files containing shell scripts or other types of "interpreted" programs, you also need read permission.

directory permissions

For directories, things look somewhat different: Read permission is required to look at a directory's content—for example, by executing the `ls` command. You need write permission to create, delete, or rename files in the directory. "Execute" permission stands for the possibility to "use" the directory in the sense that you can change into it using `cd`, or use its name in path names referring to files farther down in the directory tree.

💡 In directories where you have only read permission, you may read the file names but cannot find out anything else about the files. If you have only "execute permission" for a directory, you can access files as long as you know their names.

Usually it makes little sense to assign write and execute permission to a directory separately; however, it may be useful in certain special cases.

⚠ It is important to emphasise that write permission on a *file* is completely immaterial if the file is to be *deleted*—you need write permission to the *directory* that the file is in and nothing else! Since "deleting" a file only removes a reference to the actual file information (the inode) from the directory, this is purely a directory operation. The `rm` command does warn you if you're trying to delete a file that you do not have write permission for, but if you confirm the operation and have write permission to the directory involved, nothing will stand in the way of the operation's success. (Like any other

Copyright © 2012 Linup Front GmbH

Unix-like system, Linux has no way of "deleting" a file outright; you can only remove all references to a file, in which case the Linux kernel decides on its own that no one will be able to access the file any longer, and gets rid of its content.)

If you do have write permission to the file but not its directory, you cannot remove the file completely. You can, however, truncate it down to 0 bytes and thereby remove its *content*, even though the file itself still exists in principle.

For each user, Linux determines the "most appropriate" access rights. For example, if the members of a file's group do not have read permission for the file but "others" do, then the group members may not read the file. The (admittedly enticing) rationale that, if all others may look at the file, then the group members, who are in some sense also part of "all others", should be allowed to read it as well, does not apply.

14.2.2 Inspecting and Changing Access Permissions

information

You can obtain information about the rights, user and group assignment that apply to a file using "`ls -l`":

```
$ ls -l
-rw-r--r--  1  joe  users   4711  Oct 4 11:11 datei.txt
drwxr-x---  2  joe  group2  4096  Oct 4 11:12 testdir
```

The string of characters in the first column of the table details the access permissions for the owner, the file's group, and others (the very first character is just the file type and has nothing to do with permissions). The third column gives the owner's user name, and the fourth that of the file's group.

In the permissions string, "`r`", "`w`", and "`x`" signify existing read, write, and execute permission, respectively. If there is just a "`-`" in the list, then the corresponding category does not enjoy the corresponding privilege. Thus, "`rw-r--r--`" stands for "read and write permission for the owner, but read permission only for group members and others".

chmod command

As the file owner, you may set access permissions for a file using the `chmod` command (from "change mode"). You can specify the three categories by means of the abbreviations "`u`" (user) for the owner (yourself), "`g`" (group) for the file's group's members, and "`o`" (others) for everyone else. The permissions themselves are given by the already-mentioned abbreviations "`r`", "`w`", and "`x`". Using "`+`", "`-`", and "`=`", you can specify whether the permissions in question should be added to any existing permissions, "subtracted" from the existing permissions, or used to replace whatever was set before. For example:

```
$ chmod u+x file                           Execute permission for owner
$ chmod go+w file          sets write permission for group and others
$ chmod g+rw file            sets read and write permission for group
$ chmod g=rw,o=r file                 sets read and write permission,
                                   removes group execute permission;
                                    sets just read permission for others
$ chmod a+w file                                  equivalent to ugo+w
```

In fact, permission specifications can be considerably more complex. Consult the `info` documentation for `chmod` to find out all the details.

A file's owner is the single user (apart from `root`) who is allowed to change a file's or directory's access permissions. This privilege is independent of the actual permissions; the owner may take away all their own permissions, but that does not keep them from giving them back later.

The general syntax of the `chmod` command is

Copyright © 2012 Linup Front GmbH

```
chmod [⟨options⟩] ⟨permissions⟩ ⟨name⟩ …
```

You can give as many file or directory names as desired. The most important options include:

-R If a directory name is given, the permissions of files and directories inside this directory will also be changed (and so on all the way down the tree).

--reference=⟨*name*⟩ Uses the access permissions of file ⟨*name*⟩. In this case no ⟨*permissions*⟩ must be given with the command.

You may also specify a file's access mode "numerically" instead of "symbolically" (what we just discussed). In practice this is very common for setting all permissions of a file or directory at once, and works like this: The three permission triples are represented as a three-digit octal number—the first digit describes the owner's rights, the second those of the file's group, and the third those that apply to "others". Each of these digits derives from the sum of the individual permissions, where read permission has value 4, write permission 2, and execute permission 1. Here are a few examples for common access modes in "`ls -l`" and octal form:

```
rw-r--r--  644
r--------  400
rwxr-xr-x  755
```

Using numerical access modes, you can only set all permissions at once—there is no way of setting or removing individual rights while leaving the others alone, like you can do with the "+" and "-" operators of the symbolic representation. Hence, the command

```
$ chmod 644 file
```

is equivalent to the symbolic

```
$ chmod u=rw,go=r file
```

14.2.3 Specifying File Owners and Groups—`chown` and `chgrp`

The `chown` command lets you set the owner and group of a file or directory. This command takes the desired owner's user name and/or group name and the file or directory name the change should apply to. It is called like

```
chown ⟨user name⟩[:][⟨group name⟩] ⟨name⟩ …
chown :⟨group name⟩ ⟨name⟩ …
```

If both a user and group name are given, both are changed; if just a user name is given, the group remains as it was; if a user name followed by a colon is given, then the file is assigned to the user's primary group. If just a group name is given (with the colon in front), the owner remains unchanged. For example:

```
# chown joe:devel letter.txt
# chown www-data foo.html                              new user www-data
# chown :devel /home/devel                             new group devel
```

`chown` also supports an obsolete syntax where a dot is used in place of the colon.

Copyright © 2012 Linup Front GmbH

To "give away" files to other users or arbitrary groups you need to be `root`. The main reason for this is that normal users could otherwise annoy one another if the system uses quotas (i.e., every user can only use a certain amount of storage space).

Using the `chgrp` command, you can change a file's group even as a normal user—as long as you own the file and are a member of the *new* group:

```
chgrp ⟨group name⟩ ⟨name⟩ …
```

Changing a file's owner or group does not change the access permissions for the various categories.

`chown` and `chgrp` also support the `-R` option to apply changes recursively to part of the directory hierarchy.

Of course you can also change a file's permissions, group, and owner using most of the popular file browsers (such as Konqueror or Nautilus).

Exercises

14.1 [!2] Create a new file. What is that file's group? Use `chgrp` to assign the file to one of your secondary groups. What happens if you try to assign the file to a group that you are not a member of?

14.2 [4] Compare the mechanisms that various file browsers (like Konqueror, Nautilus, …) offer for setting a file's permissions, owner, group, … Are there notable differences?

14.3 Process Ownership

Linux considers not only the data on a storage medium as objects that can be owned. The processes on the system have owners, too.

Many commands create a process in the system's memory. During normal use, there are always several processes that the system protects from each other. Every process together with all data within its virtual address space is assigned to a user, its owner. This is most often the user who started the process—but processes created using administrator privileges may change their ownership, and the SUID mechanism (Section 14.4) can also have a hand in this.

Processes have owners

The owners of processes are displayed by the `ps` program if it is invoked using the `-u` option.

```
# ps -u
USER   PID %CPU  %MEM SIZE      RSS TTY STAT  START  TIME COMMAND
bin     89  0.0   1.0  788      328  ?    S  13:27  0:00 rpc.portmap
test1  190  0.0   2.0 1100       28  3    S  13:27  0:00 bash
test1  613  0.0   1.3  968       24  3    S  15:05  0:00 vi XF86.tex
nobody 167  0.0   1.4  932       44  ?    S  13:27  0:00 httpd
root     1  0.0   1.0  776       16  ?    S  13:27  0:03 init [3]
root     2  0.0   0.0    0        0  ?   SW  13:27  0:00 (kflushd)
```

14.4 Special Permissions for Executable Files

When listing files using the "`ls -l`" command, you may sometimes encounter permission sets that differ from the usual `rwx`, such as

```
-rwsr-xr-x   1 root shadow  32916 Dec 11 20:47 /usr/bin/passwd
```

Copyright © 2012 Linup Front GmbH

What does that mean? We have to digress here for a bit:

Assume that the passwd program carries the usual access mode:

```
-rwxr-xr-x   1 root shadow  32916 Dec 11 20:47 /usr/bin/passwd
```

A normal (unprivileged) user, say joe, wants to change his password and invokes the passwd program. Next, he receives the message "permission denied". What is the reason? The passwd process (which uses joe's privileges) tries to open the /etc/shadow file for writing and fails, since only root may write to that file—this cannot be different since otherwise, everybody would be able to manipulate passwords arbitrarily and, for example, change the root password.

SUID bit

By means of the **set-UID bit** (frequently called "SUID bit", for short) a program can be caused to run not with the invoker's privileges but those of the file owner—here, root. In the case of passwd, the *process* executing passwd has write permission to /etc/shadow, even though the invoking user, not being a system administrator, generally doesn't. It is the responsibility of the author of the passwd program to ensure that no monkey business goes on, e. g., by exploiting programming errors to change arbitrary files except /etc/shadow, or entries in /etc/shadow except the password field of the invoking user. On Linux, by the way, the set-UID mechanism works only for binary programs, not shell or other interpreter scripts.

Bell Labs used to hold a patent on the SUID mechanism, which was invented by Dennis Ritchie [SUID]. Originally, AT&T distributed Unix with the caveat that license fees would be levied after the patent had been granted; however, due to the logistical difficulties of charging hundreds of Unix installations small amounts of money retroactively, the patent was released into the public domain.

SGID bit

By analogy to the set-UID bit there is a SGID bit, which causes a process to be executed with the program file's group and the corresponding privileges (usually to access other files assigned to that group) rather than the invoker's group setting.

chmod syntax

The SUID and SGID modes, like all other access modes, can be changed using the chmod program, by giving symbolic permissions such as u+s (sets the SUID bit) or g-s (deletes the SGID bit). You can also set these bits in octal access modes by adding a fourth digit at the very left: The SUID bit has the value 4, the SGID bit the value 2—thus you can assign the access mode 4755 to a file to make it readable and executable to all users (the owner may also write to it) and to set the SUID bit.

ls output

You can recognise set-UID and set-GID programs in the output of "ls -l" by the symbolic abbreviations "s" in place of "x" for executable files.

14.5 Special Permissions for Directories

SGID for directories

There is another exception from the principle of assigning file ownership according to the identity of its creator: a directory's owner can decree that files created in that directory should belong to the same group as the directory itself. This can be specified by setting the SGID bit on the directory. (As directories cannot be executed, the SGID bit is available to be used for such things.)

A directory's access permissions are not changed via the SGID bit. To create a file in such a directory, a user must have write permission in the category (owner, group, others) that applies to him. If, for example, a user is neither the owner of a directory nor a member of the directory's group, the directory must be writable for "others" for him to be able to create files there. A file created in a SGID directory then belongs to that directory's group, even if the user is not a member of that group at all.

The typical application for the SGID bit on a directory is a directory that is used as file storage for a "project group". (Only) the members of the project group are supposed to be able to read and write all files in the directory, and

Copyright © 2012 Linup Front GmbH

to create new files. This means that you need to put all users collaborating on the project into a project group (a secondary group will suffice):

```
# groupadd project                                    Create new group
# usermod -a -G project joe                          joe into the group
# usermod -a -G project sue                                     sue too
⊲⊲⊲⊲⊲
```

Now you can create the directory and assign it to the new group. The owner and group are given all permissions, the others none; you also set the SGID bit:

```
# cd /home/project
# chgrp project /home/project
# chmod u=rwx,g=srwx /home/project
```

Now, if user `hugo` creates a file in `/home/project`, that file should be assigned to group `project`:

```
$ id
uid=1000(joe) gid=1000(joe) groups=101(project),1000(joe)
$ touch /tmp/joe.txt                            Test: standard directory
$ ls -l /tmp/joe.txt
-rw-r--r-- 1 joe joe 0 Jan  6 17:23 /tmp/joe.txt
$ touch /home/project/joe.txt                          project directory
$ ls -l /home/project/joe.txt
-rw-r--r-- 1 joe project 0 Jan  6 17:24 /home/project/joe.txt
```

There is just a little fly in the ointment, which you will be able to discern by looking closely at the final line in the example: The file does belong to the correct group, but other members of group `project` may nevertheless only read it. If you want all members of group `project` to be able to write to it as well, you must either apply `chmod` after the fact (a nuisance) or else set the umask such that group write permission is retained (see Exercise 14.4).

The SGID mode only changes the system's behaviour when new files are created. Existing files work just the same as everywhere else. This means, for instance, that a file created outside the SGID directory keeps its existing group assignment when moved into it (whereas on copying, the new copy would be put into the directory's group).

The `chgrp` program works as always in SGID directories, too: the owner of a file can assign it to any group he is a member of. Is the owner not a member of the directory's group, he cannot put the file into that group using `chgrp`—he must create it afresh within the directory.

It is possible to set the SUID bit on a directory—this permission does not signify anything, though.

Linux supports another special mode for directories, where only a file's owner may delete or remove files within that directory:

```
drwxrwxrwt   7 root   root   1024 Apr  7 10:07 /tmp
```

This `t` mode, the "sticky bit", can be used to counter a problem which arises when public directories are in shared use: Write permission to a directory lets a user delete other users' files, regardless of their access mode and owner! For example, the `/tmp` directories are common ground, and many programs create their temporary files there. To do so, all users have write permission to that directory. This implies that any user has permission to delete files there.

Copyright © 2012 Linup Front GmbH

Usually, when deleting or renaming a file, the system does not consider that file's access permissions. If the "sticky bit" is set on a directory, a file in that directory can subsequently be deleted only by its owner, the directory's owner, or `root`. The "sticky bit" can be set or removed by specifying the symbolic `+t` and `-t` modes; in the octal representation it has value 1 in the same digit as SUID and SGID.

The "sticky bit" derives its name from an additional meaning it used to have in earlier Unix systems: At that time, programs were copied to swap space in their entirety when started, and removed completely after having terminated. Program files with the sticky bit set would be left in swap space instead of being removed. This would accelerate subsequent invocations of those programs since no copy would have to be done. Like most current Unix systems, Linux uses demand paging, i. e., it fetches only those parts of the code from the program's executable file that are really required, and does not copy anything to swap space at all; on Linux, the sticky bit never had its original meaning.

Exercises

14.3 [2] What does the special "`s`" privilege mean? Where do you find it? Can you set this privilege on a file that you created yourself?

14.4 [!1] Which `umask` invocation can be used to set up a umask that would, in the project directory example above, allow all members of the `project` group to read *and* write files in the project directory?

14.5 [2] What does the special "`t`" privilege mean? Where do you find it?

14.6 [4] (For programmers.) Write a C program that invokes a suitable command (such as `id`). Set this program SUID `root` (or SGID `root`) and observe what happens when you execute it.

14.7 [I]f you leave them alone for a few minutes with a `root` shell, clever users might try to stash a SUID `root` shell somewhere in the system, in order to assume administrator privileges when desired. Does that work with `bash`? With other shells?

Commands in this Chapter

`chgrp`	Sets the assigned group of a file or directory	`chgrp`(1)	190
`chmod`	Sets access modes for files and directories	`chmod`(1)	189
`chown`	Sets the owner and/or assigned group of a file or directory	`chown`(1)	190

Summary

- Linux supports file read, write and execute permissions, where these permissions can be set separately for a file's owner, the members of the file's group and "all others".
- The sum total of a file's permissions is also called its access mode.
- Every file (and directory) has an owner and a group. Access rights—read, write, and execute permission—are assigned to these two categories and "others" separately. Only the owner is allowed to set access rights.
- Access rights do not apply to the system administrator (`root`). He may read or write all files.
- File permissions can be manipulated using the `chmod` command.
- Using `chown`, the system administrator can change the user and group assignment of arbitrary files.
- Normal users can use `chgrp` to assign their files to different groups.
- The SUID and SGID bits allow the execution of programs with the privileges of the file owner or file group instead of those of the invoker.
- The SGID bit on a directory causes new files in that directory to be assigned the directory's group (instead of the primary group of the creating user).
- The "sticky bit" on a directory lets only the owner (and the system administrator) delete files.

Bibliography

SUID Dennis M. Ritchie. "Protection of data file contents". US patent 4,135,240.

Copyright © 2012 Linup Front GmbH

15

Linux Networking

Contents

Goals

- Knowing basic networking concepts
- Understanding the requirements for integrating a Linux computer into an (existing) LAN
- Knowing important troubleshooting commands
- Knowing about important network services

Prerequisites

- File handling (Chapter 6) and use of a text editor
- Knowledge of Linux file system structure (Chapter 10)
- Knowledge about TCP/IP and using network services is helpful

15.1 Networking Basics

15.1.1 Introduction and Protocols

In the 21st century, computers are only really fun when they are connected to the Internet—via a company-wide "local area network", a home DSL router or, on the go, a wireless connection (WiFi or cellular). There are various methods of getting on the net—but whichever way you're using, the Internet itself works pretty much the same.

TCP/IP

The Internet is based on a "protocol family" called "TCP/IP". A protocol is an agreement about how computers should talk to one another, and can cover anything from particular electrical, optical or radio signals up to requests and responses from and to, e. g., a web server (but not all at the same time). Without cutting things too finely, it is possible to distinguish three different types of protocol:

Medium access protocols govern data transmission (casually speaking) at the level of network cards and cables. They include, for example, Ethernet (for LANs) or WLAN protocols such as IEEE 802.11.

Communication protocols govern the communication between stations on different networks. If you want to access a web site in Australia from Britain, the communication protocols IP and TCP arrange for the data bytes you're sending to actually arrive "down under" and vice-versa.

Application protocols make sure that the recipient of your data bytes in Australia can actually do something with them. The web application protocol, HTTP, for example, allows you to retrieve a picture of a koala; the server in Australia interprets your request and sends you the koala picture (instead of the kangaroo picture), while your web browser in Britain can figure out that you have actually been sent a picture and not a (textual) error message.

This multi-tier setup has the considerable advantage that each layer must only communicate with the layer immediately above and the layer immediately below. The application protocol, HTTP, does not need to know how bytes are moved from Britain to Australia (fortunately!), because it delegates that job to the communication protocol. The communication protocol in turn leaves the actual transmission of the bytes to the medium access protocol.

At this point it is customary to refer to the "ISO/OSI reference model", which stipulates no less than seven (!) layers for the organisation of a computer network. We will spare you the details in the interest of brevity.

The TCP/IP application protocols—besides HTTP, these include SMTP (for e-mail), SSH (for interactive sessions on remote machines), DNS (for resolving host names) or SIP (for Internet-based telephony), among many others—are usually based on either of two communication protocols, UDP or TCP. While UDP provides a connectionless, unreliable service (in plain language: data can get lost during transmission or arrive at the destination in a different order from that in which they were sent), TCP offers a connection-oriented, reliable service (which means all data will arrive at the other end in the proper sequence).

The application determines which communication protocol is more appropriate. On the Web, you normally do not want data to go missing during transmission (a piece of text, image, or downloaded software might get lost, with annoying to catastrophic results), hence TCP is the correct choice. For television or voice chat, it is usually preferrable to live with small breaks in the service (a pixellated picture or a brief burst of static) than for everything to grind to a halt while the system arranges for a missing datagram to be

Copyright © 2012 Linup Front GmbH

retransmitted. At this point, UDP makes more sense. UDP is also better for services like DNS, where small requests should result in very quick and brief answers.

The third TCP/IP communication protocol, ICMP, is used for control and troubleshooting purposes and is normally not used directly by user-level applications.

15.1.2 Addressing and Routing

In order to allow direct access via the Internet—your computer sends a request to a specific server, and the response to that request must somehow find its way back to your computer—, every computer uses a unique address, its "IP address". In the (still) most popular scheme, "IPv4", this is a sequence of four "octets", i. e., numbers from 0 to 255, such as `192.168.178.10`.

You cannot simply pick any arbitrary IP address for your computer. Instead, it is assigned one. In a company, this will be done by the systems or network administrator, while your ISP will take care of it for your home connection.

In the company you are likely to use the same IP address all of the time. Your ISP, on the other hand, will only "lend" you an address for a certain limited time; the next time you will get a different one. On the one hand this enables your ISP to use fewer addresses than the number of its customers (since not every customer is online all the time), and on the other hand this is supposed to discourage you from offering services for which a constantly changing IP address would be a nuisance.

The IP addresses themselves do not fall from the sky, but are—as far as possible—assigned with forethought, in order to keep the exchange of data between different networks, or "routing", straightforward. We will look at this in more detail later on.

loopback address

Computers usually have more than one IP address. The "loopback address" `127.0.0.1` is available on every computer and refers to that computer itself; it is not accessible from the outside. A service that is bound to the `127.0.0.1` address can only be accessed by clients that are running on the same machine.

That sounds absurd and useless, but does in fact make eminent sense: For example, you might be developing a web site that must be able to send mail to users (say, containing activation links for user accounts). To test this on your development machine, you could install a local mail server that only accepts mail on the loopback address—which will be perfectly adequate for your project while keeping you safe from being inundated with spam from elsewhere on the Internet.

On computers featuring Ethernet or WiFi, the corresponding interfaces also have IP addresses—at least if the computer is currently connected to the network. In addition, nothing prevents you from assigning additional addresses to the network interfaces for testing or specialised configurations.

The challenge consists of arranging for an arbitrary computer *X* on the Internet to be able to communicate with another arbitrary computer *Y*, as long as it knows *Y*'s IP address. (If *X* is located in Britain and *Y* in Australia, it is not 100% obvious how *X*'s bytes should reach *Y*.) The magic word which makes this possible is "routing". And it works approximately like this:

routing

- Your computer *X* can find out *Y*'s IP address (this is what the DNS is for). Let's assume it is `10.11.12.13`.
- Computer *X* can figure out that the IP address `10.11.12.13` is not on the same local area network as its own (which is no wonder considering that *Y* is in Australia). This means that computer *X* cannot send data directly to computer *Y* (big surprise).

Copyright © 2012 Linup Front GmbH

default route

- Computer X's network configuration contains a "default route", otherwise known as a recipe for what to do with data that cannot be sent to their destination directly. This might look like "Send anything you can't immediately deliver on to computer Z". The computer Z is also called a "default gateway".

- Computer Z—possibly your DSL router—cannot deliver the data to computer Y directly, either. However, it knows what to do with data that is not addressed to computers like X that are directly connected to itself, namely to send it on to your ISP.

- This game continues across a few more levels until your data arrives at a computer which knows that data addressed to `10.11.`x.y must be sent to the Australian ISP (let's call it "Billabong-Net"), so it is sent there.

- Billabong-Net knows (hopefully), how incoming data to `10.11.12.13` must be forwarded to their ultimate destination. The particular computer Y may be located at one of Billabong-Net's customer's, or one of that customer's customer's, but if everything is configured correctly it will work out right.

- Any responses from Y to X take a similar route back.

The important observation here is that the actual route that data take from computer X to computer Y is determined while the data is being transmitted. Computer X does not need to specify a complete list of intermediate destinations, but relies on every intermediate destination doing the Right Thing. Conversely, every computer only needs "local" knowledge and does not need to know what is where on the whole Internet—which would be quite impossible.

One of the properties of IP as a communication protocol is that the routing can, in principle, change from one packet to the next (even if this usually does not happen). This enables the Internet to react to congestion or connection outages in a flexible manner.

This is essentially similar to "snail mail". You don't need to know the exact route your birthday card takes to reach Auntie Fran in Sydney; you simply put the correctly stamped envelope into your friendly neighbourhood postbox.

At the end of the day this means that your Linux computer needs to know three things: its own IP address, the set of addresses it can reach directly, and a default gateway for the rest. The set of addresses that your computer can reach directly is described by the IP address of your local network together with a "subnet mask", which today is most commonly specified as a number.

subnet mask

Imagine your computer has the IP address `192.168.178.111` within the local network `192.168.178.0/24`. Here, 24 is the subnet mask—it specifies that the first 24 bits of your address (the first three octets, namely `192.168.178`) address the network itself. The final octet (or the last 8 bits to make up 32) is available for local addresses. This means that all computers with addresses from `192.168.178.0` to `192.168.178.255`—if they exist at all—can be reached directly; the default gateway (which must have an address within the local network) must be used to reach any other IP address.

Actually, in our example the addresses `192.168.178.0` and `192.168.178.255` would not be available for computers since they have a special meaning, but we only mention this for completeness.

You can in principle set these essential networking parameters—IP address, subnet (mask), and default gateway—manually as part of your computer's network configuration. (The details depend on your distribution.) However, it is very

Copyright © 2012 Linup Front GmbH

probable that your network provides a "DHCP server" which will make these parameters available to your computer without you having to worry about them.

Hence, if you want to add a Linux computer as a LAN client, it should normally suffice to switch it on and to plug an Ethernet cable into the appropriate socket or to select a WiFi network from the appropriate menu and possibly enter the corresponding password. If there are any problems whatsoever, then call loudly and persistently for your system or network administrator.

15.1.3 Names and the DNS

IP addresses are nice and important, but somewhat inconvenient to use. It would be very aggravating if you had to enter (much less remember) the address `213.157.7.75` to access the server offering the latest version of this manual. `shop.linupfront.com` is that much handier.

The way to get to inconvenient IP addresses given convenient names (and vice versa) is via the *Domain Name System*, or DNS. DNS is a globally distributed database for host names, IP addresses, and various other items, and it can be accessed via DNS servers (a. k. a. "name servers"). Your company or ISP should be running one (or, even better, two for redundancy) DNS server on your behalf.

DNS servers

You can do it yourself if you want—based on Linux, of course—, and by the time you're running a reasonably-sized LAN for your company, with a web and mail server, this is usually a good idea. However, that places us dangerously close to LPIC-2 territory.

The address of the DNS server(s) goes together with the other "essential network parameters" from the previous section—IP address, subnet (mask), default gateway—that every Linux machine ought to have.

As with IP addresses and routes, no single DNS server must have complete knowledge of all existing names (there are much too many of those by now, anyway). In fact, there is a hierarchy:

- The "root-level name servers" know about the part of a name on the very right—like `.de`, `.com`, `.tv`, whatever—and know which name servers are in charge of the *content* of these zones.
- The name servers for `.de` (by way of an example) know all the names of the form *something*`.de` and can tell which name servers know about names below *those* names.
- The name servers for a name like *something*`.de` (which are usually situated at the company in question or their ISP) know the IP address for a name like `www.`*something*`.de` and can supply it if required.

This means that to "resolve" a name like `shop.linupfront.de`, your computer first asks a root-level name server for the name servers in charge of `de`. Then it asks one of those name servers for the name servers in charge of `linupfront.de`. Finally it asks a `linupfront.de` nameserver for the address of `shop.linupfront.de`.

Actually it's not "your computer" doing the work, it's the DNS server your computer is using. But that doesn't detract from the principle.

Of course this is a fairly involved scheme, and this is why your system keeps any answers around for a while. If you have found out that `shop.linupfront.de` corresponds to the IP address `213.157.7.75`, the assumption is that this will stay the same for a while, so the resolution process is only repeated after that time has expired.

The advantage of this scheme is that we at Linup Front are free to dispose of names "below" `linupfront.de` and can add them to our DNS server as we wish. Other people get them directly from there. It would be much more of a hassle to have to petition the "Internet office" for a new name, and to have to wait for it to be added to the official list. (Think of changes to the land register and how long these usually take.)

15.1.4 IPv6

IP as a communications protocol has been around for something like 30 years, and we have found out in the meantime that some assumptions that were made back then must have been somewhat naive. For example, IPv4 (the current version of the protocol) allows, in principle, 2^{32}, or approximately 4 billion, addresses. Due to limitations in the protocol as well as various awkwardnesses with their distribution, there are very few if any unused IPv4 addresses left—and in an age where nearly everybody carries an Internet-enabled smartphone and even more people would like to have one, this is a definite problem.

There are ways of alleviating this problem—for example, not every single Internet-enabled cellphone gets an IP address that is visible from everywhere on the Internet. Instead, the operators wall off their networks from the actual Internet in order to be able to distribute more addresses (cue "network address translation", NAT). These methods are fairly disgusting and do imply other problems.

IPv6, the designated successor to IPv4[1], has been available since the late 1990s. IPv6 does away with various restrictions of IPv4, but ISPs are still somewhat reluctant to roll IPv6 out comprehensively. Linux does deal very well with IPv6, and since you can quite happily run IPv4 and IPv6 in parallel, there is nothing preventing you from setting up an IPv6-based infrastructure in your company (or even your domestic LAN—many DSL routers support IPv6 by now). Here are some of the more important properties of IPv6:

Extended address space Instead of 32-bit addresses, IPv6 uses 128-bit addresses, in the expectation that this will suffice for the foreseeable future (chances are fairly good). IPv6 addresses are notated by writing down chunks of two bytes in hexadecimal (base 16), using a colon as the separator:

```
fe80:0000:0000:0000:025a:b6ff:fe9c:406a
```

Leading zeroes may be removed from every block of four digits:

```
fe80:0:0:0:25a:b6ff:fe9c:406a
```

Furthermore, at most one sequence of blocks of zeroes may be replaced by "`::`":

```
fe80::25a:b6ff:fe9c:406a
```

The loopback address, i. e., the moral equivalent to IPv4's `127.0.0.1`, is `::1`.

Address assignment With IPv4, your ISP assigns you one IPv4 address or at most a few (unless you are a really big company or another ISP—and even for those, addresses are by now fairly scarce). If you need more addresses for your computers you need to become devious. With IPv6, you are instead assigned a complete *network*, namely a "subnet prefix" that fixes only 48 or 56 of the possible 128 address bits. You are then free to assign 2^{64} addresses in each of a number of subnets, and this is probably more than you will be able to use (according to IPv4, the *whole Internet* only uses 2^{32} addresses—a small fraction of this).

Simple configuration While with IPv4 a computer must be assigned a local IP address—possibly with the aid of a DHCP server—, using IPv6 a computer can assign itself an address that is suitable to communicate with other computers in the immediate vicinity. With IPv6, a computer can also, without DHCP, locate routers in the neighbourhood which are prepared to forward

[1] IPv5 never really existed.

Copyright © 2012 Linup Front GmbH

data to the Internet. This avoids various problems with DHCP on IPv4. Incidentally, IPv6 does not use a "default route".

Other improvements The format of IP datagrams was changed to enable more efficient routing. In addition, IPv6 defines methods to change a network's subnet prefix much more easily than a network's address could be changed in IPv6—this is also an attempt to simplify routing. Furthermore, IPv6 supports encrypted networks (IPsec) and mobility where computers—think of cell phones—can migrate from one network to another without changing addresses or interrupting existing connections ("mobile IPv6").

Compatibility The introduction of IPv6 only impacts IP—protocols like TCP and UDP or the application protocols on top of those remain unchanged. As we have mentioned, it is also possible to run IPv4 and IPv6 in parallel.

Exercises

15.1 [!1] Which medium access protocols (except Ethernet or IEEE-802.11) do you know? Which communication protocols (except IP, TCP, and UDP)? Which application protocols except those mentioned above?

15.2 [!1] How many useable IP addresses does the network `10.11.12.0/22` contain? (Subtract the first and last addresses because of their special meaning.)

15.2 Linux As A Networking Client

15.2.1 Requirements

We have already outlined the essentials necessary to add a Linux-based computer to an existing network as an (IPv4) client (the only use case pertinent to *Linux Essentials*):

- An IP address for the computer itself
- A network address and subnet mask for the computer (so it can tell which IP addresses are "local")
- The address of a default gateway on the local network
- The address(es) of one (better two) DNS server(s).

In the ideal case your computer will obtain these settings automatically via DHCP when it is booted, when the LAN cable is plugged in, or when you log into a wireless network from a mobile system. If this is not the case, you must configure these parameters yourself. The details depend on your distribution:

On Debian GNU/Linux and its derivative distributions, the network configuration is stored in a file called `/etc/network/interfaces`. The format is largely self-explanatory, and there is a commented example in `/usr/share/doc/ifupdown/examples/network-interfaces.gz`. In a pinch, there is documentation in `interfaces`(5).

On the SUSE distributions, you configure networking parameters most straightforwardly via YaST (see "Network Devices/Network Cards"). Otherwise there are configuration files below `/etc/sysconfig/network`.

On Red Hat distributions and their derivatives, there are configuration files in the `/etc/sysconfig/network-scripts` directory.

Copyright © 2012 Linup Front GmbH

`ifconfig` For short-term experiments you can also use the `ifconfig` command. Something like

```
# ifconfig eth0 192.168.178.111 netmask 255.255.255.0
```

assigns the IP address `192.168.178.111` to the network interface `eth0` (Ethernet). The local network is `192.168.178.0/24`—the `255.255.255.0` is a roundabout method of writing down the subnet mask (24). The default gateway (e. g, `192.168.178.1`) is configured using the `route` command:

default gateway

```
# route add -net default gw 192.168.178.1
```

These settings only persist until the next reboot!

`/etc/resolv.conf` The addresses of DNS servers are usually stored in the `/etc/resolv.conf` file, which might look approximately like

```
# /etc/resolv.conf
search example.com
nameserver 192.168.178.1
nameserver 192.168.178.2
```

(the "`search example.com`" will append "`example.com`" to any names specified without a period—so if you use `www`, the name being actually looked up will instead be "`www.example.com`").

If your system configures networking automatically—for instance when using WiFi—the content of `/etc/resolv.conf` is often overwritten without mercy. Do check your distribution's documentation to find out how and where to configure name servers persistently.

15.2.2 Troubleshooting

If the simple approaches of joining the Internet (plugging in or connecting to a WiFi network) do not work or if other problems occur—like interminable delays when accessing web sites, or inexplicable connection breakdown—you should consult a systems or network administrator or, generally speaking, somebody who is more familiar with the subject than you are. (At least until you have passed your LPIC-2 exams; at that point people will fetch *you* if they are in trouble.)

On the other hand, it always goes down well if you have excluded the most obvious problems yourself or narrowed the error down somewhat. This may save your administrator some work, or, if nothing else, lets you appear to your administrator like someone to be reckoned with rather than a complete rookie.

The rest of this section explains the most important troubleshooting tools and how to use them.

`ifconfig` We just introduced the `ifconfig` command for network configuration. `ifconfig` can also be used to query the setup of a network interface:

```
$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 70:5a:b6:9c:40:6a
          inet addr:192.168.178.130  Bcast:192.168.178.255▷
                                                   ◁ Mask:255.255.255.0
          inet6 addr: 2002:4fee:5912:0:725a:b6ff:fe9c:406a/64▷
                                                        ◁ Scope:Global
          inet6 addr: fe80::725a:b6ff:fe9c:406a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:112603 errors:0 dropped:0 overruns:0 frame:0
          TX packets:98512 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:102695538 (97.9 MiB)  TX bytes:13169349 (12.5 MiB)
          Interrupt:20 Memory:d7400000-d7420000
```

The most interesting bits include the various addresses:

- The first line of the output contains the "hardware" or "MAC address" of the interface. It is assigned by the manufacturer of the interface (here, an Ethernet interface).
- The second line contains the IP(v4) address assigned to the interface. On the very right there is the subnet mask in the oldfashioned/tedious notation.
- The third and fourth lines specify various IPv6 addresses. The fourth line is the local address that the computer assigned itself, while the third contains a subnet prefix which would theoretically be reachable from the Internet.

  If you look closely you will recognise the MAC address of the interface in the second half of each of the IPv6 addresses.

The "`UP`" at the start of the fifth line denotes that the interface is actually switched on.

If you execute `ifconfig` without any parameters, it outputs information about all active network interfaces on the computer. When the `-a` option is given, it also shows the ones that are currently not active.

`ping` You can use the `ping` command for low-level (IP) connectivity checks between your computer and others. `ping` uses the control protocol, ICMP, to ask another computer for "signs of life". If these indications arrive back at your computer, you know that (a) your computer can send data to the other computer, and (b) the other computer can send data to *your* computer (the one does not necessarily imply the other).

In the simplest case, you invoke `ping` with the name of the computer you'd like to communicate with:

```
$ ping fritz.box
PING fritz.box (192.168.178.1) 56(84) bytes of data.
64 bytes from fritz.box (192.168.178.1): icmp_req=1 ttl=64 time=3.84 ms
64 bytes from fritz.box (192.168.178.1): icmp_req=2 ttl=64 time=5.09 ms
64 bytes from fritz.box (192.168.178.1): icmp_req=3 ttl=64 time=3.66 ms
64 bytes from fritz.box (192.168.178.1): icmp_req=4 ttl=64 time=3.69 ms
64 bytes from fritz.box (192.168.178.1): icmp_req=5 ttl=64 time=3.54 ms
                                              Stop the program using Ctrl+c …
--- fritz.box ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 3.543/3.967/5.095/0.575 ms
```

Everything is in order here: All five transmitted packets have arrived back, the sequence is correct, and the transmission times make sense for a local network. If instead you see nothing for a while before

```
From 192.168.178.130 icmp_seq=1 Destination Host Unreachable
From 192.168.178.130 icmp_seq=2 Destination Host Unreachable
From 192.168.178.130 icmp_seq=3 Destination Host Unreachable
```

appears, something is fishy—the target computer cannot be contacted.

If you cannot connect to a remote computer on the Internet, the problem can in principle be anywhere between you and the remote computer. For a systematic approach you could use the following tactics:

- Use `ping` to check whether you can reach the loopback interface, `127.0.0.1`. If that doesn't work, something is very wrong with *your* computer.

- Use ping to check whether you can reach the address of the network interface (Ethernet, WLAN, …) you're currently using (or believe you are using) to access the Internet. You can find that address, if required, by means of something like

  ```
  $ /sbin/ifconfig eth0
  ```

  That ought to work, too—if not, then there is a local problem.

- Use ping to check whether you can reach your local default gateway. (If you don't know that address by heart, you can find out about it using route. In

  ```
  $ /sbin/route
  Kernel IP routing table
  Destination    Gateway    Genmask        Flags Metric Ref Use Iface
  default        fritz.box  0.0.0.0        UG    0      0     0 eth0
  link-local     *          255.255.0.0    U     1000   0     0 lo
  192.168.178.0  *          255.255.255.0  U     0      0     0 eth0
  ```

  the default entry below Destination tells you what to use—"ping fritz.box" here.) If that doesn't work and you're getting messages like

  ```
  Destination Host Unreachable
  ```

  then there is a problem with your local network. If you can, ask a colleague who is just accessing the Internet, or try another computer: If everything seems to be OK there, then again your computer is likely to be the culprit. Otherwise—and quite likely in that case, too—it is time for the system administrator.

  For example, your default route might be incorrect and point to the wrong computer. (This would be more likely when networking has been configured manually.) That would be unlikely to impact the users of other computers, where the configuration is probably correct.

- If you can in fact reach the default gateway correctly, then the problem is either outside your LAN, somewhere on the Internet (and may be not only out of your reach but even out of that of your administrator), or somewhere farther up the "protocol stack". For example, it might be possible for you to reach a remote web server using ping, but your (company?) Internet access might not allow direct access to the Web because you are supposed to use a "proxy server" (and forgot about configuring it). Your system administrator will be able to help.

A network connection that sometimes works and sometimes doesn't (kink in the cable? rodent damage?) can be tested using "ping -f". Instead of sending one packet per second as usual, ping sends data as fast as it can. It outputs a dot for every packet sent and one backspace character for every packet received. If you're losing packets, there will be a lengthening line of dots.

If you're not root but an ordinary user, you must make do with a minimal interval of 0.2 seconds between two packets sent. You may only flood the network if you are an administrator.

To check IPv6 connections, you must use the ping6 command instead of ping:

```
$ ping6 ::1
```

dig The dig command is used to test DNS name resolution. Unless you specify otherwise, it tries to find an IP address corresponding to a name given on the command line:

```
$ dig www.linupfront.de

; <<>> DiG 9.8.1-P1 <<>> www.linupfront.de
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34301
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;www.linupfront.de.             IN      A

;; ANSWER SECTION:
www.linupfront.de.      3600    IN      CNAME   s0a.linupfront.de.
s0a.linupfront.de.      3600    IN      A       31.24.175.68

;; Query time: 112 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Thu Mar  1 16:06:06 2012
;; MSG SIZE  rcvd: 69
```

This (very verbose) output tells us in the "QUESTION SECTION" that we were looking for www.linupfront.de (not that we didn't know that already). The "ANSWER SECTION" lets on that actually no IP address corresponds to www.linupfront.de, but that www.linupfront.de is in fact a "nickname" for the computer called s0a.linupfront.de (a popular approach). s0a.linupfront.de, however, has the address 31.24.175.68. Finally, the last block tells us that this response came from the DNS server running on 127.0.0.1.

If there is no answer for some time and then something like

```
; <<>> DiG 9.8.1-P1 <<>> www.linupfront.de
;; global options: +cmd
;; connection timed out; no servers could be reached
```

appears, then there is something rotten in the state of Denmark. Either your settings in /etc/resolv.conf are incorrect, or the name server doesn't do what it should.

You can ask a specific name server by mentioning it on the command line:

```
$ dig www.linupfront.de @fritz.box                    Frage fritz.box
```

Of course resolving this name should not entail an expensive DNS query (or else you might have a chicken-egg problem); when in doubt, you can always specify an IP address directly:

```
$ dig www.linupfront.de @192.168.178.1
```

If you know your way around DNS, you can use dig to look for RR types other than A records. Just tell dig on the command line what you want:

```
$ dig linupfront.de mx
```

To find the name belonging to a given IP address (if any), you must specify the -x option:

```
$ dig +short -x 31.24.175.68
s0a.linupfront.de.
```

(With the `+short` option, `dig` produces very brief output.)

Of course `dig` can do a lot more, but most of that is only of interest to people who need to configure name servers or keep them running. If you can't resist, there is ample detail in `dig`(1).

netstat The `netstat` command is a kind of Swiss army knife which provides all sorts of information about your computer and its network connection. If you just execute

```
$ netstat
```

you get a list of all active connections. This includes not only TCP connection, but also local connections via Unix domain sockets, which are as stiflingly voluminous as they are utterly boring. It is more interesting to use something like

```
$ netstat -tl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address       Foreign Address  State
tcp        0      0 ceol:domain         *:*              LISTEN
tcp        0      0 ceol-eth.fri:domain *:*              LISTEN
tcp        0      0 *:ssh               *:*              LISTEN
tcp        0      0 ceol:ipp            *:*              LISTEN
tcp        0      0 ceol:postgresql     *:*              LISTEN
tcp        0      0 ceol:smtp           *:*              LISTEN
◁◁◁◁◁
```

to obtain a list of TCP ports where services are listening for incoming connections on this computer.

TCP and UDP use "ports" to allow the same computer to offer or access several services at the same time. Many protocols use fixed port numbers—a list is in the `/etc/services` file.

This example output tells you that, on the address `ceol`, the computer provides a DNS server (service `domain`), a CUPS server for printing (service `ipp`) and a mail server (service `smtp`). It even offers the `ssh` service (secure shell) on all configured addresses.

"`netstat -tl`" is an important troubleshooting tool in connection with networ services. If a service does not appear here but you think it actually ought to, that indicates that something is wrong with its configuration—possibly it does not use the correct address/port, or something went catastrophically wrong when the service was started so it is not running at all.

The "`-u`" option in place of "`-t`" displays the UDP-based services, and "`-p`" will also display the name and PID of the process providing the service. The latter feature is only available if you invoke the command as the `root` user.

The "`-n`" option will display everything with IP addresses and port numbers instead of names. This is sometimes more revealing, at least as long as you have a working knowledge of the port numbers.

```
$ netstat -tln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address       Foreign Address State
```

```
tcp        0      0 127.0.0.1:53          0.0.0.0:*         LISTEN
tcp        0      0 192.168.178.130:53    0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:22            0.0.0.0:*         LISTEN
◁◁◁◁◁
```

"`netstat -s`" shows you statistics like

```
Ip:
    145845 total packets received
    8 with invalid addresses
    0 forwarded
    0 incoming packets discarded
    145837 incoming packets delivered
    138894 requests sent out
    16 outgoing packets dropped
    172 dropped because of missing route
Icmp:
    30 ICMP messages received
    0 input ICMP message failed.
◁◁◁◁◁
```

And "`netstat -r`" is essentially the same as "`route`" (without parameters).

Exercises

15.3 [!2] Use `ping` to make sure you can reach a well-known server on the Internet (perhaps `www.google.com`).

15.4 [1] Use `dig` to check which IP address corresponds to `www.heise.de`.

15.5 [2] `dig`'s `+trace` option causes the program to document the complete lookup chain for a name, starting from the root-level name servers. Try that for some interesting names and check the intermediate steps.

15.6 [2] Which network services does your computer offer? Which of those are reachable from other computers?

Commands in this Chapter

`dig`	Searches DNS for information (very convenient)	`dig`(1)	206
`ifconfig`	Configures network interfaces	`ifconfig`(8)	203, 204
`netstat`	Displays information about network connections, servers, routing	`netstat`(8)	208
`ping`	Checks basic network connectivity using ICMP	`ping`(8)	205
`ping6`	Checks basic network connectivity (for IPv6)	`ping`(8)	206
`route`	Manages the Linux kernel's static routing table	`route`(8)	203

Copyright © 2012 Linup Front GmbH

Summary

- There are three kinds of network protocols: medium access protocols, communication protocols, and application protocols.
- In a "protocol stack", each protocol only interacts with the protocols immediately above and below.
- TCP/IP contains the communication protocols IP, TCP (reliable and connection-oriented), and UDP (unreliable and connectionless), the control protocol ICMP, and a multitude of application protocols based on TCP or UDP.
- Computers on the Internet have unique IP addresses.
- Routing allows communication between computers that are not directly connected to each other.
- To connect to a network, a Linux computer requires an IP address, a subnet address with a mask, a default gateway, and the address of at least one DNS server.
- The DNS is a distributed database that maps host names to IP addresses and vice versa (among other things).
- IPv6 is the successor standard to IPv4, which removes various limitations and incorporates improvements.
- `ifconfig` and `route` are used for the manual configuration of networking. Distributions use various schemes for persistent networking configuration.
- The `ifconfig`, `ping`, `dig`, and `netstat` programmes are used for network troubleshooting.

Copyright © 2012 Linup Front GmbH

A

Sample Solutions

This appendix contains sample solutions for selected exercises.

1.1 Your author first cut his teeth on a “real” computer in 1982, using an Apple II plus. That machine had a 6502 processor with the awesome clock frequency of 1 MHz and the inconceivable (at the time) RAM capacity of 48 KiB. There was no hard disk, but there were 5¼-inch floppy disks storing 143 KiB of data each – on each side. You were really supposed to use only one side of a “diskette”, but with care or a special punch it was possible to cut a “write-protection notch” into the opposite side of the plastic sleeve to be able to flip the disk over. (At $20 per box of 10 disks this was badly required.)

2.1 A good source is `http://oreilly.com/catalog/opensources/book/appa.html`. Do also read `http://www.cs.vu.nl/~ast/reliable-os/`.

2.2 `ftp.kernel.org` has a file called `linux-0.01.tar.gz`.

2.3

1. False. GPL software may be sold for arbitrary amounts of money, as long as the buyer receives the source code (etc.) and the GPL rights.

2. False. Companies are encouraged to develop products based on GPL code, but these products must also be distributed under the GPL. Of course a company is not required to give away their product to the world at large—it only needs to make the source code available to its direct customers who bought the executables, but these may make full use of their rights to the software under the GPL.

3. True.

4. False. You may *use* a program freely without having accepted the GPL (it is not a contract). The GPL governs just the *redistribution* of the software, and you can peruse the GPL before doing that. (Interactive programs are supposed to call your attention to the GPL.) The observation is true that only those conditions can be valid that the software recipient could know *before* the purchase of the product; since the GPL gives to the recipient rights that he would otherwise not have had at all—such as the right to distribute original or modified code—this is not a problem: One may ignore the GPL completely and still do all with the software that copyright allows for. This is a marked difference to the EULAs of proprietary programs; these try to establish a contract relationship in which the buyer explicitly *gives away* rights

that he would otherwise have been entitled to by copyright law (such as the right to inspect the program to find out its structure). This of course only works *before* the purchase (if at all).

3.2 In both cases, the message "`Login incorrect`" appears, but only after the password has been prompted for and entered. This is supposed to make it difficult to guess valid user names (those that do not elicit an error message right away). The way the system is set up, a "cracker" cannot tell whether the user name was invalid already, or whether the password was wrong, which makes breaking into the system a lot more difficult.

4.1 In the login shell, the output is "`-bash`", whereas in the "subshell" it is "`bash`". The minus sign at the beginning tells the shell to behave as a login shell rather than a "normal" shell, which pertains to the initialisation.

4.2 `alias` is an internal command (does not work otherwise). `rm` is an external command. Within bash, `echo` and `test` are internal commands but are also available as external commands (executable program files), since other shells do not implement them internally. In bash's case, they are internal mostly for reasons of efficiency.

5.2 Try "`apropos process`" or "`man -k process`".

5.5 The format and tools for info files were written in the mid-1980s. HTML wasn't even invented then.

6.1 In Linux, the current directory is a process attribute, i. e., every process has its own current directory (with DOS, the current directory is a feature of the drive, which of course is inappropriate in a multi-user system). Therefore `cd` must be an internal command. If it was an external command, it would be executed in a new process, change that process's current directory and quit, while the invoking shell's current directory would remain unchanged throughout the process.

6.4 If a file name is passed to `ls`, it outputs information about that file only. With a directory name, it outputs information about all the files in that directory.

6.5 The `-d` option to `ls` does exactly that.

6.6 This could look approximately like so:

```
$ mkdir -p grd1-test/dir1 grd1-test/dir2 grd1-test/dir3
$ cd grd1-test/dir1
$ vi hello
$ cd
$ vi grd1-test/dir2/howdy
$ ls grd1-test/dir1/hallo grd1-test/dir2/howdy
grd1-test/dir1/hello
grd1-test/dir2/howdy
$ rmdir grd1-test/dir3
$ rmdir grd1-test/dir2
rmdir: grd1-test/dir2: Directory not empty
```

To remove a directory using `rmdir`, it must be empty (except for the entries "`.`" and "`..`", which cannot be removed).

6.7 The matching names are, respectively

(a) `prog.c`, `prog1.c`, `prog2.c`, `progabc.c`

(b) `prog1.c`, `prog2.c`

(c) `p1.txt`, `p2.txt`, `p21.txt`, `p22.txt`

(d) `p1.txt`, `p21.txt`, `p22.txt`, `p22.dat`

(e) all names

(f) all names except `prog` (does not contain a period)

6.8 "`ls`" without arguments lists the content of the current directory. Directories in the current directory are only mentioned by name. "`ls`" with arguments, on the other hand (and in particular "`ls *`"—`ls` does not get to see the search pattern, after all) lists information about the given arguments. For directories this means that the *content* of the directories is listed as well.

6.9 The "`-l`" file (visible in the output of the first command) is interpreted as an option by the `ls` command. Thus it does not show up in the output of the second command, since `ls` with path name arguments only outputs information about the files specified as arguments.

6.10 If the asterisk matched file names starting with a dot, the recursive deletion command "`rm -r *`" would also apply to the "`..`" entry of a directory. This would delete not just subdirectories of the current directory, but also the enclosing directory and so on.

6.11 Here are the commands:

```
$ cd
$ cp /etc/services myservices
$ mv myservices src.dat
$ cp src.dat /tmp
$ rm src.dat /tmp/src.dat
```

6.12 When you rename a directory, all its files and subdirectories will automatically be "moved" so as to be within the directory with its new name. An `-R` to `mv` is therefore completely unnecessary.

6.13 The simple-minded approach—something like "`rm -file`"—fails because `rm` misinterprets the file name as a sequence of options. The same goes for commands like "`rm "-file"`" or "`rm '-file'`". The following methods work better:

1. With "`rm ./-file`", the dash is no longer at the start of the parameter and thus no longer introduces an option.
2. With "`rm -- -file`", you tell `rm` that there are definitely no options after the "`--`" but only path names. This also works with many other programs.

6.14 During the replacement of the "`*`", the "`-i`" file is picked up as well. Since the file names are inserted into the command line in ASCII order, `rm` sees a parameter list like

```
-i a.txt b.jpg c.dat                                        or whatever
```

and considers the "`-i`" the *option* `-i`, which makes it remove files only with confirmation. We hope that this is sufficient to get you to think things over.

Copyright © 2012 Linup Front GmbH

6.15 If you edit the file via one link, the new content should also be visible via the other link. However, there are "clever" editors which do not overwrite your file when saving, but save a new file and rename it afterwards. In this case you will have two different files again.

6.16 If the target of a symbolic link does not exist (any longer), accessing that "dangling" link will lead to an error message.

6.17 To itself. You can recognise the file system root directory by this.

6.18 On this system, the `/home` directory is on a separate partition and has inode number 2 on that partition, while the `/` directory is inode number 2 on its own file system. Since inode numbers are only unique within the same physical file system, the same number can show up for different files in "`ls -i`" output; this is no cause for concern.

6.19 Hard links are indistinguishable, equivalent names for the same file (or, hypothetically, directory). But every directory has a "link" called "`..`" referring to the directory "above". There can be just one such link per directory, which does not agree with the idea of several equivalent names for that directory. Another argument against hard links on directories is that for every name in the file system tree there must be a unique path leading to the root directory (`/`) in a finite number of steps. If hard links to directories were allowed, a command sequence such as

```
$ mkdir -p a/b
$ cd a/b
$ ln .. c
```

could lead to a loop.

6.20 The reference counter for the subdirectory has the value 2 (one link results from the name of the subdirectory in `~`, one from the "`.`" link in the subdirectory itself). If there were additional subdirectories within the directory, their "`..`" links would increment the reference counter beyond its minimum value of 2.

6.21 Hard links need hardly any space, since they are only additional directory entries. Symbolic links are separate files and need one inode at least (every file has its own inode). Also, some space is required to store the name of the target file. In theory, disk space is assigned to files in units of the file system's block size (1 KiB or more), but there is a special exception in the `ext2` and `ext3` file systems for "short" symbolic links (smaller than approximately 60 bytes), which can be stored within the inode itself and do not require a full data block. More advanced file systems such as the Reiser file system can handle short files of any type very efficiently, thus the space required for symbolic links ought to be negligible.

6.22 One possible command could be "`find / -size +1024k -print`".

6.23 The basic approach is something like

```
find . -maxdepth 1 ⟨tests⟩ -ok rm '{}' \;
```

The ⟨*tests*⟩ should match the file as closely as possible. The "`-maxdepth 1`" option restricts the search to the current directory (no subdirectories). In the simplest case, use "`ls -i`" to determine the file's inode number (e.g., 4711) and then use

```
find . -maxdepth 1 -inum 4711 -exec rm -f '{}' \;
```

to delete the file.

Copyright © 2012 Linup Front GmbH

6.24 Add a line like

```
find /tmp -user $LOGNAME -type f -exec rm '{}' \;
```

or—more efficiently—

```
find /tmp -user $LOGNAME -type f -print0 \
    | xargs -0 -r rm -f
```

to the file `.bash_logout` in your home directory. (The `LOGNAME` environment variable contains the current user name.)

6.25 Use a command like ”‘`locate '*/README'`”’. Of course, something like ”‘`find / -name README`”’ would also do the trick, but it will take *a lot* longer.

6.26 Immediately after its creation the new file does not occur in the database and thus cannot be found (you need to run `updatedb` first). The database also doesn't notice that you have deleted the file until you invoke `updatedb` again.—It is best not to invoke `updatedb` directly but by means of the shell script that your distribution uses (e. g., `/etc/cron.daily/find` on Debian GNU/Linux). This ensures that `updatedb` uses the same parameters as always.

6.27 `slocate` should only return file names that the invoking user may access. The `/etc/shadow` file, which contains the users' encrypted passwords, is restricted to the system administrator (see *Linux Administration I*).

7.1 The regular expression *r*`+` is a mere abbreviation of *rr*`*`, so we could do without `+`. Things are different with `?`, for which there is no convenient substitute, at least if we must assume (as in `grep` vs. `egrep`) that we cannot substitute *r*`?` by `\(\|`*r*`\)` (GNU `grep` supports the synonymous *r*`{,1}`—see Table 7.1—but this is not supported by the `grep` implementations of the traditional Unix vendors.

7.2 You want a sample solution for this? Don't be ridiculous.—Well, if you insist …

```
egrep "\<king('s daughter)?\>" frog.txt
```

7.3 One possibility might be

```
grep :/bin/bash$ /etc/passwd
```

7.4 We're looking for words starting with a (possibly empty) sequence of consonants, then there is an "a", then possibly consonants again, then an "e", and so on. We must take care, in particular, not to let extra vowels "slip through". The resulting regular expression is fairly unsavoury, so we allow ourselves some notational simplification:

```
$ k='[âeiou]*'
$ grep -i{$k}^a${k}e${k}i${k}o${k}u${k}$ /usr/share/dict/words
abstemious
abstemiously
abstentious
acheilous
acheirous
acleistous
affectious
annelidous
```

```
arsenious
arterious
bacterious
caesious
facetious
facetiously
fracedinous
majestious
```

(You may look up the words on your own time.)

7.5 Try

```
egrep '(\<[A-Za-z]{4,}\>).*\<\1\>' frog.txt
```

We need `egrep` for the back reference. The word brackets are also required (try it without them!).

8.1 A (probable) explanation is that the `ls` program works roughly like this:

```
Read directory information to list $l$;
if (option -U not specified) {
    Sort the entries of $l$;
}
Write $l$ to standard output;
```

That is, everything is being read, then sorted (or not), and then output.

The other explanation is that, at the time the `filelist` entry is being read, there has not in fact been anything written to the file to begin with. For efficiency, most file-writing programs buffer their output internally and only call upon the operating system to write to the file if a substantial amount of data has been collected (e. g. 8192 bytes). This can be observed with commands that produce very much output relatively slowly; the output file will grow by 8192 bytes at a time.

8.2 When `ls` writes to the screen (or, generally, a "screen-like" device), it formats the output differently from when it writes to a "real" file: It tries to display several file names on the same line if the file names' length permits, and can also colour the file names according to their type. When output is redirected to a "real" file, just the names will be output one per line, with no formatting.

At first glance this seems to contradict the claim that programs do not know whether their output goes to the screen or elsewhere. This claim is correct in the normal case, but if a program is seriously interested in whether its output goes to a screen-like device (a "terminal") it can ask the system. In the case of `ls`, the reasoning behind this is that terminal output is usually looked at by people who deserve as much information as possible. Redirected output, on the other hand, is processed by other programs and should therefore be simple; hence the limitation to one file name per line and the omission of colors, which must be set up using terminal control characters which would "pollute" the output.

8.3 The shell arranges for the output redirection before the command is invoked. Therefore the command sees only an empty input file, which usually does not lead to the desired result.

8.4 The file is read from the beginning, and all that is read is appended to the file at the same time, so that it grows until it takes up all the free space on the disk.

8.5 You need to redirect standard output to standard error output:

```
echo Error >&2
```

8.6 There is nothing wrong in principle with

```
… | tee foo | tee bar | …
```

However, it is easier to write

```
… | tee foo bar | …
```

See also `tee`'s documentation (man page or info page).

8.7 Pipe the list of file names through "`cat -b`".

8.8 One method would be "`head -n 13 | tail -n 1`".

8.10 `tail` notices it, emits a warning, and continues from the new end of file.

8.11 The `tail` window displays

```
Hello
orld
```

The first line results from the first `echo`; the second `echo` overwrites the complete file, but "`tail -f`" knows that it has already written the first six characters of the file ("`Hello`" and a newline character)—it just waits for the file to become longer, and then outputs whatever is new, in particular, "`orld`" (and a newline character).

8.14 The line containing the name "de Leaping" is sorted wrongly, since on that line the second field isn't really the first name but the word "Leaping". If you look closely at the examples you will note that the sorted output is always correct—regarding "Leaping", not "Gwen". This is a strong argument for the second type of input file, the one with the colon as the separator character.

8.15 You can sort the lines by year using "`sort -k 1.4,1.8`". If two lines are equal according to the sort key, `sort` makes an "emergency comparison" considering the whole line, which in this case leads to the months getting sorted correctly within every year. If you want to be sure and very explicit, you could also write "`sorkt -k 1.4,1.8 -k 1.1,1.2`".

8.19 Use something like

```
cut -d: -f 4 /etc/passwd | sort -u | wc -l
```

The `cut` command isolates the group number in each line of the user database. "`sort -u`" constructs a sorted list of all group numbers containing each group number exactly once. Finally, "`wc -l`" counts the number of lines in that list. The result is the number of different primary groups in use on the system.

9.1 For example:

1. `%d-%m-%Y`
2. `%y-%j (WK%V)`
3. `%Hh%Mm%Ss`

9.2 We don't know either, but try something like "`TZ=America/Los_Angeles date`".

9.4 If you change an environment variable in the child process, its value in the parent process remains unmodified. There are ways and means to pass information back to the parent process but the environment is not one.

9.5 Start a new shell and remove the `PATH` variable from the environment (without deleting the variable itself). Try starting external programs.—If `PATH` does not exist at all, the shell will not start external programs.

9.6 Unfortunately we cannot offer a system-independent sample solution; you need to see for yourself (using `which`).

9.7 Using `whereis`, you should be able to locate two files called `/usr/share/man/man1/crontab.1.gz` and `/usr/share/man/man5/crontab.5.gz`. The former contains the documentation for the actual `crontab` command, the latter the documentation for the format of the files that `crontab` creates. (The details are irrelevant for this exercise; see *Advanced Linux*.)

9.8 `bash` uses character sequences of the form "`!`⟨*character*⟩" to access previous commands (an alternative to keyboard functions such as Ctrl+r which have migrated from the C shell to `bash`). The "`!"`" character sequence, however, counts as a syntax error.

9.9 None.

9.10 If the file name is passed as a parameter, `wc` insists on outputting it together with the number of lines. If `wc` reads its standard input, it only outputs the line count.

9.11 Try something like

```
#!/bin/bash
pattern=$1
shift
◁◁◁◁◁
for f
do
    grep $pattern "$f" && cp "$f" backup
done
```

After the `shift`, the regular expression is no longer the first parameter, and that must be taken into account for "`for f`".

9.12 If the `-f` file test is applied to a symbolic link, it always applies to the file (or directory, or whatever) that the link refers to. Hence it also succeeds if the name in question is really just a symbolic link. (Why does this problem *not* apply to `filetest2`?)

10.2 You can find out about this using something like

```
ls /bin /sbin /usr/bin /usr/sbin | wc -l
```

Alternatively, you can hit Tab twice at a shell prompt—the shell will answer something like

```
Display all 2371 possibilities? (y or n)
```

and that is—depending on `PATH`—your answer. (If you are logged in as a normal—non-privileged—user, the files in `/sbin` and `/usr/sbin` will not normally be included in the total.)

10.3 Use "`grep` ⟨*pattern*⟩ `*.txt /dev/null`" instead of "`grep` ⟨*pattern*⟩ `*.txt`". Thus `grep` always has at least two file name parameters, but `/dev/null` does not otherwise change the output.—The GNU implementation of `grep`, which is commonly found on Linux, supports an `-H` option which does the same thing but in a non-portable manner.

10.4 With `cp` to an existing file, the file is opened for writing and truncated to length 0, before the source data is written to it. For `/dev/null`, this makes the data disappear. With `mv` to an existing file, the target file is first removed—and that is a directory operation which, disregarding the special nature of `/dev/null`, simply removes the name `null` from the directory `/dev` and creates a new file called `null` with the content of `foo.txt` in its place.

10.6 It is inadvisable because firstly it doesn't work right, secondly the data in question isn't worth backing up anyway since it is changing constantly (you would be wasting lots of space on backup media and time for copying), and thirdly because such a backup could never be restored. Uncontrolled write operations to, say, `/proc/kcore` will with great certainty lead to a system crash.

11.1 Because `AA` is shorter than `*2A`.

11.2 The main problem is representing the asterisk. In the simplest case you could write something like "`A*12B*4*A`". Of course compression suffers by representing the single asterisk by three characters; you could institute an exception to let, for example, `**` stand for a single asterisk, but this makes the decompression step more complicated.

11.3 Use "`ls -l >content`" and "`tar -cvf content.tar content`". You will notice that the archive is considerably bigger than the original. This is due to the metadata in the archive. `tar` does not compress; it archives. To create an archive (a file) from a single file is not really a workable idea.

11.4 For example, enter "`touch file{1,2,3}`" and "`tar -rvf content.tar file*`".

11.5 Unpack the archive using "`tar -xvf content.tar`".

11.6 If you want to unpack `etc-backup.tar` on the other computer (e. g., because you want to see what is in there) and the archive contains absolute path names, the data will not be written to a subdirectory `etc` of the current directory, but they end up in the `/etc` directory of the remote computer. This is very likely not what you had in mind. (Of course you should take care not to unpack an archive containing relative path names while `/` is your current directory.)

11.7 If you want to use `gzip`, enter "`gzip -9 contents.tar`".

11.8 Take care: To handle `gzip`-compressed `tar` archives, `tar` requires the `-z` option: "`tar -tzf contents.tar.gz`". To restore the original *archive*, you need the "`gunzip contents.tar.gz`" command.

Copyright © 2012 Linup Front GmbH

11.9 Try “`tar -cvzf /tmp/homearchive.tar ~`”.

11.11 `unzip` offers to ignore the file in the archive, to rename it, or to overwrite the existing file.

11.12 Try something like

```
$ unzip files.zip "a/*" -x "*/*.c"
```

12.1 A simple `su` (without `/bin/` in front) would in principle work just as well. However, it makes sense to get used to `/bin/su` because it protects you better against “Trojan horses”: A devious user could place a program called `su` in their `$PATH` such that it is found before `/bin/su`. Then they call you to their machine because of some feigned problem. You say to yourself “I’ll have this fixed in no time, I’ll just borrow this terminal session for a moment to become `root`”, innocuously call “`su`”, and your devious user’s program asks you for the `root` password just like the genuine `su` would. Except that it saves the password to some file, outputs an error message, and removes itself. You are likely to assume that you made a typo, call “`su`” again, get the genuine program, and everything is fine from here—except that your devious user now knows the `root` password. If you use “`/bin/su`” in the first place, this attack isn’t as straightforward.

12.2 Using `su`, any arbitrary user can obtain administrator privileges simply by virtue of knowing the `root` password. With `sudo`, however, you will only be successful if you are on `sudo`’s list of authorised users. Hence, `sudo` does not ask you for the password to figure out whether you are allowed to *be* `root`, but to establish that you are *yourself*. (Someone else might have taken over your computer while you have stepped out for a moment.) Your own password works for this just as well (or even better) than `root`’s. `sudo` is particularly useful if you share the system administrator job with several colleagues, because then *nobody* needs to know the actual `root` password—you can assign something very long and complicated and place it in a sealed envelope in the safe after the computer has been installed (for emergencies). With `su`, *all* administrators need to know the password, which makes changing it difficult. (The SUSE distributions use the `root` password with `sudo`, by default, and they even seem to be proud of that type of brain damage.)

12.7 Here is the RPM variant:

```
$ rpm --query --all | wc -l
```

And on a Debian-like system you should use something like

```
$ dpkg --list | grep ^ii | wc -l
```

13.1 By their respective numerical UIDs and GIDs.

13.2 This works but is not necessarily a good idea. As far as the system is concerned, the two are a single user, i. e., all files and processes with that UID belong to both user names.

13.3 A pseudo-user’s UID is used by programs in order to obtain particular well-defined access rights.

Copyright © 2012 Linup Front GmbH

13.4 Whoever is in group disk has block-level read and write permission to the system's disks. With knowledge of the file system structure it is easy to make a copy of /bin/sh into a SUID root shell (Section 14.4) by changing the file system metadata directly on disk. Thus, group disk membership is tantamount to root privileges; you should put nobody into the disk group whom you would not want to tell the root password outright.

13.5 You will usually find an "x". This is a hint that the password that would usually be stored there is indeed stored in another file, namely /etc/shadow, which unlike the former file is readable only for root.

13.6 There are basically two possibilities:

1. Nothing. In this case the system should turn you away after you entered your password, since no user account corresponds to the all-uppercase user name.
2. From now on, the system talks to you in uppercase letters only. In this case your Linux system assumes that you are sitting in front of an absolutely antediluvial terminal (1970s vintage or so) that does not support lowercase letters, and kindly switches its processing of input and output data such that uppercase letters in the input are interpreted as lowercase, and lowercase letters in the output are displayed as uppercase. Today this is of limited benefit (except if you work in a computer museum), and you should log out as quickly again as possible before your head explodes. Since this behaviour is so atavistic, not every Linux distribution goes along with it, though.

13.7 Use the passwd command if you're logged in as user joe, or "passwd joe" as root. In joe's entry in the /etc/shadow file there should be a different value in the second field, and the date of the last password change (field 3) should show the current date (in what unit?)

13.8 As root, you set a new password for him using "passwd dumbo", as you cannot retrieve his old one even though you are the administrator.

13.9 Use the command "passwd -n 7 -m 14 -w 2 joe". You can verify the settings using "passwd -S joe".

13.10 Use the useradd command to create the user, "usermod -u" to modify the UID. Instead of a user name, the files should display a UID as their owner, since no user name is known for that UID …

13.11 For each of the three user accounts there should be one line in /etc/passwd and one in /etc/shadow. To work with the accounts, you do not necessarily need a password (you can use su as root), but if you want to login you do. You can create a file without a home directory by placing it in /tmp (in case you forgot—a home directory for a new user would however be a good thing).

13.12 Use the userdel command to delete the account. To remove the files, use the "find / -uid ⟨*UID*⟩ -delete" command.

13.13 If you use "usermod -u", you must reassign the user's file to the new UID, for example by means of "find / -uid ⟨*UID*⟩ -exec chown test1 {} \;" or (more efficiently) "chown -R --from=⟨*UID*⟩ test1 /". In each case, ⟨*UID*⟩ is the (numerical) former UID.

13.14 You can either edit /etc/passwd using vipw or else call usermod.

Copyright © 2012 Linup Front GmbH

13.15 Groups make it possible to give specific privileges to groups [sic!] of users. You could, for example, add all HR employees to a single group and assign that group a working directory on a file server. Besides, groups can help organise access rights to certain peripherals (e. g., by means of the groups `disk`, `audio`, or `video`).

13.16 Use the "`mkdir` ⟨*directory*⟩" command to create the directory and "`chgrp` ⟨*groupname*⟩ ⟨*directory*⟩" to assign that directory to the group. You should also set the SGID bit to ensure that newly created files belong to the group as well.

13.17 Use the following commands:

```
# groupadd test
# gpasswd -a test1 test
Adding user test1 to group test
# gpasswd -a test2 test
Adding user test2 to group test
# gpasswd test
Changing the password for group test
New Password:x9q.Rt/y
Re-enter new password:x9q.Rt/y
```

To change groups, use the "`newgrp test`" command. You will be asked for the password only if you are not a member of the group in question.

14.1 A new file is assigned to your current primary group. You can't assign a file to a group that you are not a member of—unless you are `root`.

14.3 This is the SUID or SGID bit. The bits cause a process to assume the UID/GID of the executable file rather than that of the executing user. You can see the bits using "`ls -l`". Of course you may change all the permissions on your own files. However, at least the SUID bit only makes sense on binary executable files, not shell scripts and the like.

14.4 One of the two following (equivalent) commands will serve:

```
$ umask 007
$ umask -S u=rwx,g=rwx
```

You may perhaps ask yourself why this umask contains `x` bits. They are indeed irrelevant for files, as files are not created executable by default. However it might be the case that subdirectories are desired in the project directory, and it makes sense to endow these with permissions that allow them to be used reasonably.

14.5 The so-called "sticky bit" on a directory implies that only the owner of a file (or the owner of the directory) may delete or rename it. You will find it, e. g., on the `/tmp` directory.

14.7 This doesn't work with the `bash` shell (at least not without further trickery). We can't speak for other shells here.

15.2 1022 (= $2^{32-22} - 2$). A useful tool for this sort of calculation—if you prefer not to do them in your head—is called `ipcalc`.

B

Example Files

In various places, the fairy tale *The Frog King*, more exactly *The Frog King, or Iron Henry*, from *German Children's and Domestic Fairy Tales* by the brothers Grimm, is used as an example. The fairy tale is presented here in its entirety to allow for comparisons with the examples.

```
The Frog King, or Iron Henry

In olden times when wishing still helped one, there lived a king whose
daughters were all beautiful, but the youngest was so beautiful that
the sun itself, which has seen so much, was astonished whenever it
shone in her face.

Close by the king's castle lay a great dark forest, and under an old
lime-tree in the forest was a well, and when the day was very warm,
the king's child went out into the forest and sat down by the side of
the cool fountain, and when she was bored she took a golden ball, and
threw it up on high and caught it, and this ball was her favorite
plaything.

Now it so happened that on one occasion the princess's golden ball did
not fall into the little hand which she was holding up for it, but on
to the ground beyond, and rolled straight into the water.  The king's
daughter followed it with her eyes, but it vanished, and the well was
deep, so deep that the bottom could not be seen. At this she began to
cry, and cried louder and louder, and could not be comforted.

And as she thus lamented someone said to her, »What ails you, king's
daughter? You weep so that even a stone would show pity.«

She looked round to the side from whence the voice came, and saw a
frog stretching forth its big, ugly head from the water. »Ah, old
water-splasher, is it you,« she said, »I am weeping for my golden
ball, which has fallen into the well.«

»Be quiet, and do not weep,« answered the frog, »I can help you, but
what will you give me if I bring your plaything up again?«

»Whatever you will have, dear frog,« said she, »My clothes, my pearls
and jewels, and even the golden crown which I am wearing.«
```

```
The frog answered, »I do not care for your clothes, your pearls and
jewels, nor for your golden crown, but if you will love me and let me
be your companion and play-fellow, and sit by you at your little
table, and eat off your little golden plate, and drink out of your
little cup, and sleep in your little bed - if you will promise me this
I will go down below, and bring you your golden ball up again.«

»Oh yes,« said she, »I promise you all you wish, if you will but bring
me my ball back again.« But she thought, »How the silly frog does
talk. All he does is to sit in the water with the other frogs, and
croak. He can be no companion to any human being.«

But the frog when he had received this promise, put his head into the
water and sank down; and in a short while came swimming up again with
the ball in his mouth, and threw it on the grass. The king's daughter
was delighted to see her pretty plaything once more, and picked it up,
and ran away with it.

»Wait, wait,« said the frog. »Take me with you. I can't run as you
can.« But what did it avail him to scream his croak, croak, after her,
as loudly as he could. She did not listen to it, but ran home and soon
forgot the poor frog, who was forced to go back into his well again.

The next day when she had seated herself at table with the king and
all the courtiers, and was eating from her little golden plate,
something came creeping splish splash, splish splash, up the marble
staircase, and when it had got to the top, it knocked at the door and
cried, »Princess, youngest princess, open the door for me.«

She ran to see who was outside, but when she opened the door, there
sat the frog in front of it. Then she slammed the door to, in great
haste, sat down to dinner again, and was quite frightened.

The king saw plainly that her heart was beating violently, and said,
»My child, what are you so afraid of? Is there perchance a giant
outside who wants to carry you away?«

»Ah, no,« replied she. »It is no giant but a disgusting frog.«

»What does that frog want from you?«

»Yesterday as I was in the forest sitting by the well, playing, my
golden ball fell into the water. And because I cried so, the frog
brought it out again for me, and because he so insisted, I promised
him he should be my companion, but I never thought he would be able to
come out of his water. And now he is outside there, and wants to come
in to me.«

In the meantime it knocked a second time, and cried, »Princess,
youngest princess, open the door for me, do you not know what you said
to me yesterday by the cool waters of the well. Princess, youngest
princess, open the door for me.«

Then said the king, »That which you have promised must you perform.
Go and let him in.«

She went and opened the door, and the frog hopped in and followed her,
step by step, to her chair. There he sat and cried, »Lift me up beside
```

Copyright © 2012 Linup Front GmbH

```
you.« She delayed, until at last the king commanded her to do it. Once
the frog was on the chair he wanted to be on the table, and when he
was on the table he said, »Now, push your little golden plate nearer
to me that we may eat together.« The frog enjoyed what he ate, but
almost every mouthful she took choked her.

At length he said, »I have eaten and am satisfied, now I am tired,
carry me into your little room and make your little silken bed ready,
and we will both lie down and go to sleep.«  The king's daughter began
to cry, for she was afraid of the cold frog which she did not like to
touch, and which was now to sleep in her pretty, clean little bed.

But the king grew angry and said, »He who helped you when you were in
trouble ought not afterwards to be despised by you.«

So she took hold of the frog with two fingers, carried him upstairs,
and put him in a corner, but when she was in bed he crept to her and
said, »I am tired, I want to sleep as well as you, lift me up or I
will tell your father.«

At this she was terribly angry, and took him up and threw him with all
her might against the wall. »Now, will you be quiet, odious frog,«
said she. But when he fell down he was no frog but a king's son with
kind and beautiful eyes. He by her father's will was now her dear
companion and husband. Then he told her how he had been bewitched by a
wicked witch, and how no one could have delivered him from the well
but herself, and that to-morrow they would go together into his
kingdom.

And indeed, the next morning a carriage came driving up with eight
white horses, which had white ostrich feathers on their heads, and
were harnessed with golden chains, and behind stood the young king's
servant Faithful Henry.

Faithful Henry had been so unhappy when his master was changed into a
frog, that he had caused three iron bands to be laid round his heart,
lest it should burst with grief and sadness. The carriage was to
conduct the young king into his kingdom. Faithful Henry helped them
both in, and placed himself behind again, and was full of joy because
of this deliverance.

And when they had driven a part of the way the king's son heard a
cracking behind him as if something had broken. So he turned round and
cried, »Henry, the carriage is breaking.« »No, master, it is not the
carriage. It is a band from my heart, which was put there in my great
pain when you were a frog and imprisoned in the well.«

Again and once again while they were on their way something cracked,
and each time the king's son thought the carriage was breaking, but it
was only the bands which were springing from the heart of Faithful
Henry because his master was set free and was happy.
```

(Linup Front GmbH would like to point out that the authors strongly disapprove of any cruelty to animals.)

Copyright © 2012 Linup Front GmbH

C

Linux Essentials Certification

The *Linux Professional Institute* (LPI) is a vendor-independent non-profit organization dedicated to furthering the professional use of Linux. One aspect of the LPI's work concerns the creation and delivery of distribution-independent certification exams, for example for Linux professionals. These exams are available world-wide and enjoy considerable respect among Linux professionals and employers.

This training manual serves as an aid in preparing for the *Linux Essentials* exam. On its web site, the LPI explains:

> The purpose of the Linux Essentials Certificate is to define the basic knowledge required to competently use a desktop or mobile device using a Linux Operating System. The associated Linux Essentials Program will guide and encourage youth (and those new to Linux and Open Source) to understand the place of Linux and Open Source in the context of the broader IT industry.

The LPI has compiled the knowledge necessary to pass the exam in the form of a list of **exam objectives** which are published on its web site at `http://www.lpi.org/`.

exam objectives

The following table shows which chapters in the manual cover the content of which exam objectives. The exam objectives themselves are listed later in this appendix. Do note that the *Linux Essentials* objectives are not suitable or intended as a guideline for the didactic delivery of an introductory class to Linux. For this reason, this manual does not follow the order of objectives in the LPI list, but deviates from them in the interest of a logical sequence of topics aimed at making the material more understandable.

⚠ Be aware that the exam objectives on the LPI web site may be modified occasionally, and that, e. g., supplementary information may appear there that has not yet been incorporated into this training manual. For safety, refer to the LPI's version of the exam objectives.

C.1 Exam Objective Overview

The following table shows the objectives for the *Linux Essentials* exam and the chapters covering these objectives. The numbers in the right-hand column refer to the chapters containing the material in question.

No	Wt	Title	LXES
1.1	2	Linux Evolution and Popular Operating Systems	2
1.2	2	Major Open Source Applications	2
1.3	1	Understanding Open Source Software and Licensing	2
1.4	2	ICT Skills and Working in Linux	2–3, 13
2.1	2	Command Line Basics	4, 6, 9
2.2	2	Using the Command Line to Get Help	5
2.3	2	Using Directories and Listing Files	6
2.4	2	Creating, Moving and Deleting Files	6
3.1	2	Archiving Files on the Command Line	11
3.2	4	Searching and Extracting Data from Files	7–8
3.3	4	Turning Commands into a Script	3, 9
4.1	1	Choosing an Operating System	1–2
4.2	2	Understanding Computer Hardware	1
4.3	3	Where Data is Stored	10
4.4	2	Your Computer on the Network	15
5.1	2	Basic Security and Identifying User Types	10, 13
5.2	2	Creating Users and Groups	13
5.3	2	Managing File Permissions and Ownership	14
5.4	1	Special Directories and Files	6, 10, 14

C.2 Exam Objectives For *Linux Essentials*

1.1 Linux Evolution and Popular Operating Systems

Weight 2

Description Knowledge of Linux development and major distributions.

Key Knowledge Areas

- Open Source Philosophy
- Distributions
- Embedded Systems

The following is a partial list of the used files, terms and utilities:

- Android
- Debian
- CentOS

1.2 Major Open Source Applications

Weight 2

Description Awareness of major applications and their uses.

Key Knowledge Areas

- Desktop Applications
- Server Applications
- Mobile Applications
- Development Languages
- Package Management Tools and repositories

The following is a partial list of the used files, terms and utilities:

- OpenOffice.org, LibreOffice, Thunderbird, Firefox
- Blender, Gimp, Audacity, ImageMagick
- Apache, MySQL, PostgreSQL
- NFS, Samba, OpenLDAP, Postfix, DNS, DHCP
- C, Java, Perl, shell, Python, PHP

1.3 Understanding Open Source Software and Licensing

Weight 1

Description Open communities and licensing Open Source Software for business.

Key Knowledge Areas

- Licensing
- Free Software Foundation (FSF), Open Source Initiative (OSI)

The following is a partial list of the used files, terms and utilities:

- GPL, BSD, Creative Commons
- Free Software, Open Source Software, FOSS, FLOSS
- Open Source business models

Nice to know:

- Intellectual Property (IP): copyright, trademarks and patents
- Apache License, Mozilla License

1.4 ICT Skills and Working in Linux

Weight 2 Description Basic Information and Communication Technology (ICT) skills and working in Linux.

Key Knowledge Areas

- Desktop Skills
- Getting to the Command Line
- Industry uses of Linux, Cloud Computing and Virtualization

The following is a partial list of the used files, terms and utilities:

- Using a browser, privacy concerns, configuration options, searching the web and saving content
- Terminal and Console
- Password issues
- Privacy issues and tools
- Use of common open source applications in presentations and projects

2.1 Command Line Basics

Weight 2

Description Basics of using the Linux command line.

Key Knowledge Areas

- Basic shell
- Formatting commands
- Working With Options
- Variables
- Globbing
- Quoting

The following is a partial list of the used files, terms and utilities:

- `echo`
- `history`
- `PATH` env variable

Copyright © 2012 Linup Front GmbH

- `export`
- `which`

Nice to know:

- Substitutions
- `||`, `&&` and `;` control operators

2.2 Using the Command Line to Get Help

Weight 2

Description Running help commands and navigation of the various help systems.

Key Knowledge Areas

- Man
- Info

The following is a partial list of the used files, terms and utilities:

- `man`
- `info`
- Man pages
- `/usr/share/doc`
- `locate`

Nice to know:

- `apropos`, `whatis`, `whereis`

2.3 Using Directories and Listing Files

Weight 2

Description Navigation of home and system directories and listing files in various locations.

Key Knowledge Areas

- Files, directories
- Hidden files and directories
- Home
- Absolute and relative paths

The following is a partial list of the used files, terms and utilities:

- Common options for `ls`
- Recursive listings
- `cd`
- `.` and `..`
- home and ~

2.4 Creating, Moving and Deleting Files

Weight 2

Description Create, move and delete files and directories under the home directory.

Key Knowledge Areas

- Files and directories

Copyright © 2012 Linup Front GmbH

- Case sensitivity
- Simple globbing and quoting

The following is a partial list of the used files, terms and utilities:

- mv, cp, rm, touch
- mkdir, rmdir

3.1 Archiving Files on the Command Line

Weight 2
Description Archiving files in the user home directory.
Key Knowledge Areas

- Files, directories
- Archives, compression

The following is a partial list of the used files, terms and utilities:

- tar
- Common tar options
- gzip, bzip2
- zip, unzip

Nice to know:

- Extracting individual files from archives

3.2 Searching and Extracting Data from Files

Weight 4
Description Search and extract data from files in the home directory.
Key Knowledge Areas

- Command line pipes
- I/O re-direction
- Partial POSIX Regular Expressions (., [ ], *, ?)

The following is a partial list of the used files, terms and utilities:

- find
- grep
- less
- cat, head, tail
- sort
- cut
- wc

Nice to know:

- Partial POSIX Basic Regular Expressions ([^], ^, $)
- Partial POSIX Extended Regular Expressions (+, ( ), |)
- xargs

3.3 Turning Commands into a Script

Weight 4
Description Turning repetitive commands into simple scripts.
Key Knowledge Areas

Copyright © 2012 Linup Front GmbH

- Basic text editing
- Basic shell scripting

The following is a partial list of the used files, terms and utilities:

- `/bin/sh`
- Variables
- Arguments
- `for` loops
- `echo`
- Exit status
- names of common text editors

Nice to know:

- use of `pico`, `nano`, `vi` (only basics for creating scripts)
- Bash
- `if`, `while`, `case` statements
- `read` and `test`, and `[` commands

4.1 Choosing an Operating System

Weight 1
Description Knowledge of major operating systems and Linux distributions.
Key Knowledge Areas

- Windows, Mac, Linux differences
- Distribution life cycle management

The following is a partial list of the used files, terms and utilities:

- GUI versus command line, desktop configuration
- Maintenance cycles, Beta and Stable

4.2 Understanding Computer Hardware

Weight 2
Description Familiarity with the components that go into building desktop and server computers.
Key Knowledge Areas

- Hardware

The following is a partial list of the used files, terms and utilities:

- Hard drives and partitions, motherboards, processors, power supplies, optical drives, peripherals
- Display types
- Drivers

4.3 Where Data is Stored

Weight 3
Description Where various types of information are stored on a Linux system.
Key Knowledge Areas

- Kernel
- Processes

Copyright © 2012 Linup Front GmbH

- `syslog`, `klog`, `dmesg`
- `/lib`, `/usr/lib`, `/etc`, `/var/log`

The following is a partial list of the used files, terms and utilities:

- Programs, libraries, packages and package databases, system configuration
- Processes and process tables, memory addresses, system messaging and logging
- `ps`, `top`, `free`

4.4 Your Computer on the Network

Weight 2
Description Querying vital networking settings and determining the basic requirements for a computer on a Local Area Network (LAN).
Key Knowledge Areas

- Internet, network, routers
- Domain Name Service
- Network configuration

The following is a partial list of the used files, terms and utilities:

- `route`
- `resolv.conf`
- IPv4, IPv6
- `ifconfig`
- `netstat`
- `ping`

Nice to know:

- `ssh`
- `dig`

5.1 Basic Security and Identifying User Types

Weight 2
Description Various types of users on a Linux system.
Key Knowledge Areas

- Root and Standard Users
- System users

The following is a partial list of the used files, terms and utilities:

- `/etc/passwd`, `/etc/group`
- `id`, `who`, `w`
- `sudo`

Nice to know:

- `su`

5.2 Creating Users and Groups

Weight 2
Description Creating users and groups on a Linux system.
Key Knowledge Areas

- User and group commands
- User IDs

The following is a partial list of the used files, terms and utilities:

- `/etc/passwd`, `/etc/shadow`, `/etc/group`
- `id`, `last`
- `useradd`, `groupadd`
- `passwd`

Nice to know:

- `usermod`, `userdel`
- `groupmod`, `groupdel`

5.3 Managing File Permissions and Ownership

Weight 2

Description Understanding and manipulating file permissions and ownership settings.

Key Knowledge Areas

- File/directory permissions and owners

The following is a partial list of the used files, terms and utilities:

- `ls -l`
- `chmod`, `chown`

Nice to know:

- `chgrp`

5.4 Special Directories and Files

Weight 1

Description Special directories and files on a Linux system including special permissions.

Key Knowledge Areas

- System files, libraries
- Symbolic links

The following is a partial list of the used files, terms and utilities:

- `/etc`, `/var`
- `/tmp`, `/var/tmp` and Sticky Bit
- `ls -d`
- `ln -s`

Nice to know:

- Hard links
- Setuid/Setgid

Copyright © 2012 Linup Front GmbH

D

Command Index

This appendix summarises all commands explained in the manual and points to their documentation as well as the places in the text where the commands have been introduced.

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Index

This index points to the most important key words in this document. Particularly important places for the individual key words are emphasised by **bold** type. Sorting takes place according to letters only; "`~/.bashrc`" is therefore placed under "B".

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH

Copyright © 2012 Linup Front GmbH