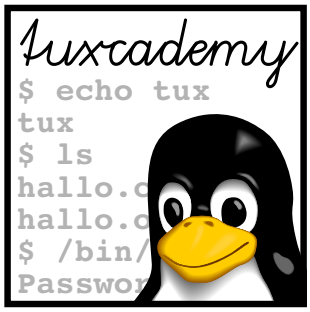
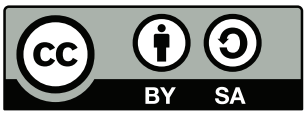
Version 4.0

Introduction to Linux

for Users and Administrators





tuxcademy – Linux and Open Source learning materials for everyone
www.tuxcademy.org · info@tuxcademy.org

This training manual is designed to correspond to the objectives of the LPI-101 (LPIC-1, version 4.0) certification exam promulgated by the Linux Professional Institute. *Further details are available in Appendix C.*

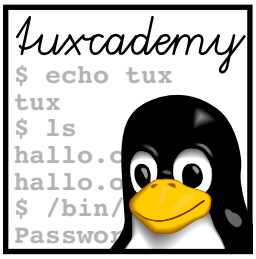
The Linux Professional Institute *does not endorse specific exam preparation materials or techniques. For details, refer to* `info@lpi.org`.

The tuxcademy project aims to supply freely available high-quality training materials on Linux and Open Source topics – for self-study, school, higher and continuing education and professional training.
Please visit `http://www.tuxcademy.org/`! Do contact us with questions or suggestions.

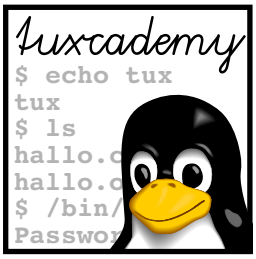
Introduction to Linux for Users and Administrators
Revision: `grd1:62f570f98f89998d:2015-08-04`
`grd1:be27bba8095b329b:2015-08-04` 1–11, B–C
`grd1:EF6EC05fegg6iuNkQRlDOJ`

`http://www.tuxcademy.org` · `info@tuxcademy.org`

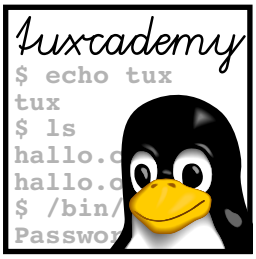
Authors: Tobias Elsner, Anselm Lingnau
Technical Editor: Anselm Lingnau ⟨*anselm@tuxcademy.org*⟩
English Translation: Anselm Lingnau
Typeset in Palatino, Optima and DejaVu Sans Mono



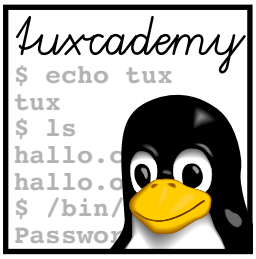
Contents



List of Tables



List of Figures



Preface

This manual is an introduction to the use of Linux. Participants will not just learn how to operate the system according to instructions, but will receive solid fundamental knowledge about Linux, to serve as a basis for further study.

The course is directed towards computer users with little or no previous exposure to Linux. Knowledge of other operating systems is helpful but not a prerequisite. Ambitious users as well as system administrators will be learning how to make the most of the Linux operating system.

Having studied this manual, participants will be able to use the Linux operating system on an elementary basis. They will be able to work on the Linux command line and be familiar with the most important tools. Completion of this course or equivalent knowledge is necessary for more advanced Linux courses and for *Linux Professional Institute* certification.

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!

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

exercises

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 are emphasized using "marginal notes" so they can be eas-

Important concepts

definitions

ily 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.

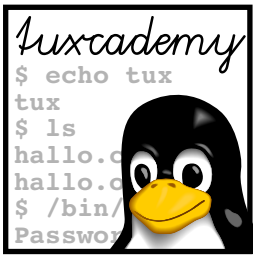
We endeavour to provide courseware that is as up-to-date, complete and error-free as possible. In spite of this, problems or inaccuracies may creep in. If you notice something that you think could be improved, please do let us know, e.g., by sending e-mail to

```
info@tuxcademy.org
```

(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.) Thank you very much!

LPIC-1 Certification

These training materials are part of a recommended curriculum for LPIC-1 preparation. Refer to Appendix C for further information.



1

Introduction

Contents

Goals

- Knowing about Linux, its properties and its history
- Differentiating between the Linux kernel and Linux distributions
- Understanding the terms “GPL”, “free software”, and “open-source software”

Prerequisites

- Knowledge of other operating systems is useful to appreciate similarities and differences

1.1 What is Linux?

Linux is an operating system. As such, it manages a computer's basic functionality. Application programs build on the operating system. It forms the interface between the hardware and application programs as well as the interface between the hardware and people (users). Without an operating system, the computer is unable to "understand" or process our input.

Various operating systems differ in the way they go about these tasks. The functions and operation of Linux are inspired by the Unix operating system.

1.2 Linux History

The history of Linux is something special in the computer world. While most other operating systems are commercial products produced by companies, Linux was started by a university student as a hobby project. In the meantime, hundreds of professionals and enthusiasts all over the world collaborate on it—from hobbyists and computer science students to operating systems experts funded by major IT corporations to do Linux development. The basis for the existence of such a project is the Internet: Linux developers make extensive use of services like electronic mail, distributed version control, and the World Wide Web and, through these, have made Linux what it is today. Hence, Linux is the result of an international collaboration across national and corporate boundaries, now as then led by Linus Torvalds, its original author.

To explain about the background of Linux, we need to digress for a bit: Unix, the operating system that inspired Linux, was begun in 1969. It was developed by
Bell Laboratories Ken Thompson and his colleagues at Bell Laboratories (the US telecommunication giant AT&T's research institute)[1]. Unix caught on rapidly especially at universities, because Bell Labs furnished source code and documentation at cost (due to an anti-trust decree, AT&T was barred from selling software). Unix was, at first, an operating system for Digital Equipment's PDP-11 line of minicomputers, but was ported to other platforms during the 1970s—a reasonably feasible endeavour, since the Unix software, including the operating system kernel, was mostly written in Dennis Ritchie's purpose-built **C** programming language. Possibly most important of all Unix ports was the one to the PDP-11's successor platform, the
C
VAX VAX, at the University of California in Berkeley, which came to be distributed as "BSD" (short for *Berkeley Software Distribution*). By and by, various computer manufacturers developed different Unix derivatives based either on AT&T code or on BSD (e. g., Sinix by Siemens, Xenix by Microsoft (!), SunOS by Sun Microsystems, HP/UX by Hewlett-Packard or AIX by IBM). Even AT&T was finally allowed to
System V market Unix—the commercial versions System III and (later) System V. This led to a fairly incomprehensible multitude of different Unix products. A real standardisation never happened, but it is possible to distinguish roughly between BSD-like and System-V-like Unix variants. The BSD and System V lines of development
SVR4 were mostly unified by "System V Release 4", which exhibited the characteristics of both factions.

The very first parts of Linux were developed in 1991 by Linus Torvalds, then a 21-year-old student in Helsinki, Finland, when he tried to fathom the capabilities of his new PC's Intel 386 processor. After a few months, the assembly language experiments had matured into a small but workable operating system ker-
Minix nel that could be used in a Minix system—Minix was a small Unix-like operating system that computer science professor Andrew S. Tanenbaum of the Free University of Amsterdam, the Netherlands, had written for his students. Early Linux had properties similar to a Unix system, but did not contain Unix source code. Linus Torvalds made the program's source code available on the Internet, and the

[1] The name "Unix" is a pun on "Multics", the operating system that Ken Thompson and his colleagues worked on previously. Early Unix was a lot simpler than Multics. How the name came to be spelled with an "x" is no longer known.

Figure 1.1: Ken Thompson (sitting) and Dennis Ritchie (standing) with a PDP-11, approx. 1972. (Photograph courtesy of Lucent Technologies.)

idea was eagerly taken up and developed further by many programmers. Version 0.12, issued in January, 1992, was already a stable operating system kernel. There was—thanks to Minix—the GNU C compiler (`gcc`), the `bash` shell, the `emacs` editor and many other GNU tools. The operating system was distributed world-wide by anonymous FTP. The number of programmers, testers and supporters grew very rapidly. This catalysed a rate of development only dreamed of by powerful software companies. Within months, the tiny kernel grew into a full-blown operating system with fairly complete (if simple) Unix functionality.

The "Linux" project is not finished even today. Linux is constantly updated and added to by hundreds of programmers throughout the world, catering to millions of satisfied private and commercial users. In fact it is inappropriate to say that the system is developed "only" by students and other amateurs—many contributors to the Linux kernel hold important posts in the computer industry and are among the most professionally reputable system developers anywhere. By now it is fair to claim that Linux is the operating system with the widest supported range of hardware ever, not just with respect to the platforms it is running on (from PDAs to mainframes) but also with respect to driver support on, e. g., the Intel PC platform. Linux also serves as a research and development platform for new operating systems ideas in academia and industry; it is without doubt one of the most innovative operating systems available today.

Exercises

1.1 [4] Use the Internet to locate the famous (notorious?) discussion between Andrew S. Tanenbaum and Linus Torvalds, in which Tanenbaum says that, with something like Linux, Linus Torvalds would have failed his (Tanenbaum's) operating systems course. What do you think of the controversy?

1.2 [2] Give the version number of the oldest version of the Linux source code that you can locate.

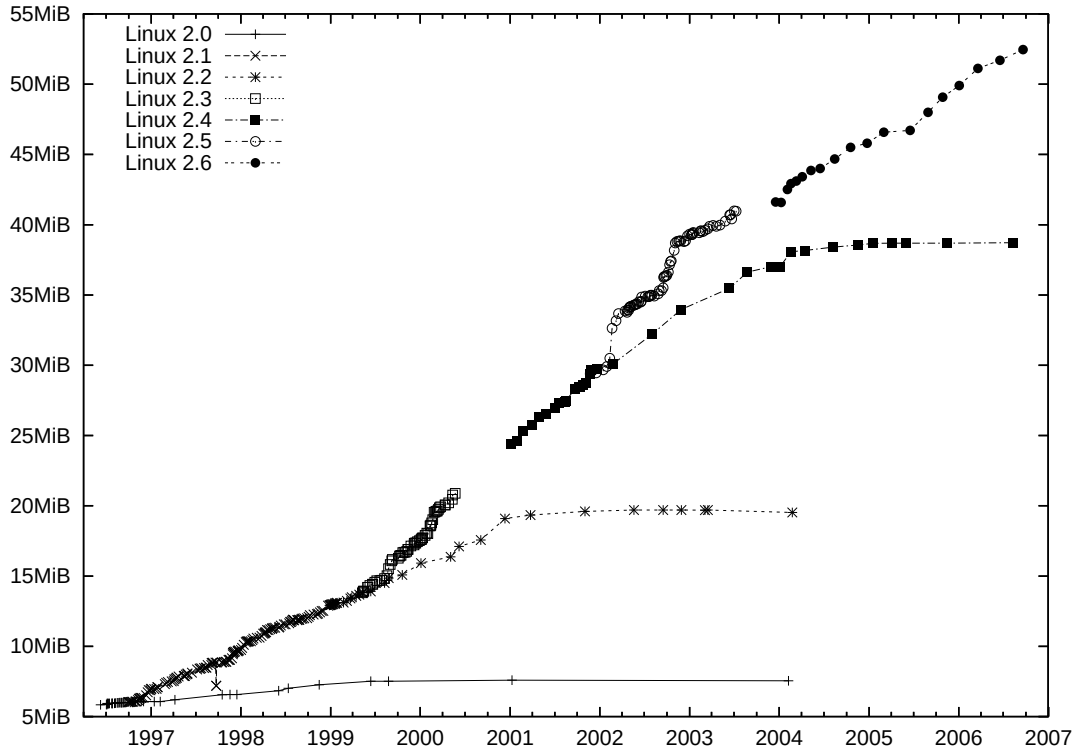


Figure 1.2: Linux development, measured by the size of `linux-*.tar.gz`. Each marker corresponds to a Linux version. During the 10 years between Linux 2.0 and Linux 2.6.18, the size of the compressed Linux source code has roughly increased tenfold.

1.3 Free Software, "Open Source" and the GPL

GPL
Free Software Foundation
Free Software
Open Source

From the very beginning of its development, Linux was placed under the *GNU General Public License* (**GPL**) promulgated by the *Free Software Foundation* (FSF). The FSF was founded by Richard M. Stallman, the author of the Emacs editor and other important programs, with the goal of making high-quality software "freely" available—in the sense that users are "free" to inspect it, to change it and to redistribute it with or without changes, not necessarily in the sense that it does not cost anything[2]. In particular, he was after a freely available Unix-like operating system, hence "GNU" as a (recursive) acronym for *"GNU's Not Unix"*. The main tenet of the GPL is that software distributed under it may be changed as well as sold at any time, but that the (possibly modified) source code must always be passed along—thus *Open Source*—and that the recipient must receive the same rights of modification and redistribution. Thus there is little point in selling GPL software "per seat", since the recipient must be allowed to copy and install the software as often as wanted. (It is of course permissible to sell support for the GPL software "per seat".) New software resulting from the extension or modification of GPL software must, as a "derived work", also be placed under the GPL.

Therefore, the GPL governs the distribution of software, not its use, and allows the recipient to do things that he would not be allowed to do otherwise—for example, the right to copy and distribute the software, which according to copyright law is the *a priori* prerogative of the copyright owner. Consequently, it differs markedly from the "end user license agreements" (EULAs) of "proprietary" software, whose owners try to *take away* a recipient's rights to do various things. (For example, some EULAs try to forbid a software recipient from talking critically—or

[2] The FSF says "free as in speech, not as in beer"

at all—about the product in public.)

The GPL is a *license*, not a contract, since it is a one-sided grant of rights to the recipient (albeit with certain conditions attached). The recipient of the software does not need to "accept" the GPL explicitly. The common EULAs, on the other hand, are *contracts*, since the recipient of the software is supposed to waive certain rights in exchange for being allowed to use the software. For this reason, EULAs must be explicitly accepted. The legal barriers for this may be quite high—in many jurisdictions (e. g., Germany), any EULA restrictions must be known to the buyer before the actual sale in order to become part of the sales contract. Since the GPL does not in any way restrict a buyer's rights (in particular as far as use of the software is concerned) compared to what they would have to expect when buying any other sort of goods, these barriers do not apply to the GPL; the additional rights that the buyer is conferred by the GPL are a kind of extra bonus.

Currently two versions of the GPL are in widespread use. The newer version 3 (also called "GPLv3") was published in July, 2007, and differs from the older version 2 (also "GPLv2") by more precise language dealing with areas such as software patents, the compatibility with other free licenses, and the introduction of restrictions on making changes to theoretically "free" devices impossible by excluding them through special hardware ("Tivoisation", after a Linux-based personal video recorder whose kernel is impossible to alter or exchange). In addition, GPLv3 allows its users to add further clauses. – Within the community, the GPLv3 was not embraced with unanimous enthusiasm, and many projects, in particular the Linux kernel, have intentionally stayed with the simpler GPLv2. Many other projects are made available under the GPLv2 "or any newer version", so you get to decide which version of the GPL you want to follow when distributing or modifying such software.

GPLv3

Neither should you confuse GPL software with "public-domain" software. The latter belongs to nobody, everybody can do with it what he wants. A GPL program's copyright still rests with its developer or developers, and the GPL states very clearly what one may do with the software and what one may not.

Public Domain

It is considered good form among free software developers to place contributions to a project under the same license that the project is already using, and in fact most projects insist on this, at least for code that is supposed to become part of the "official" version. Indeed, some projects insist on "copyright assignments", where the code author signs the copyright over to the project (or a suitable organisation). The advantage of this is that, legally, only the project is responsible for the code and that licensing violations—where only the copyright owner has legal standing—are easier to prosecute. A side effect that is either desired or else explicitly unwanted is that copyright assignment makes it easier to change the license for the complete project, as this is an act that only the copyright owner may perform.

In case of the Linux operating system kernel, which explicitly does not require copyright assignment, a licensing change is very difficult to impossible in practice, since the code is a patchwork of contributions from more than a thousand authors. The issue was discussed during the GPLv3 process, and there was general agreement that it would be a giant project to ascertain the copyright provenance of every single line of the Linux source code, and to get the authors to agree to a license change. In any case, some Linux developers would be violently opposed, while others are impossible to find or even deceased, and the code in question would have to be replaced by something similar with a clear copyright. At least Linus Torvalds is still in the GPLv2 camp, so the problem does not (yet) arise in practice.

GPL and Money

The GPL does not stipulate anything about the price of the product. It is utterly legal to give away copies of GPL programs, or to sell them for money, as long as you provide source code or make it available upon request, and the software recipient gets the GPL rights as well. Therefore, GPL software is not necessarily "freeware".

You can find out more by reading the GPL [GPL91], which incidentally must accompany every GPLlicensed product (including Linux).

Other "free" licenses

There are other "free" software licenses which give similar rights to the software recipient, for example the "BSD license" which lets appropriately licensed software be included in non-free products. The GPL is considered the most thorough of the free licenses in the sense that it tries to ensure that code, once published under the GPL, *remains* free. Every so often, companies have tried to include GPL code in their own non-free products. However, after being admonished by (usually) the FSF as the copyright holder, these companies have always complied with the GPL's requirements. In various jurisdictions the GPL has been validated in courts of law—for example, in the Frankfurt (Germany) *Landgericht* (state court), a Linux kernel developer obtained a judgement against D-Link (a manufacturer of network components, in this case a Linux-based NAS device) in which the latter was sued for damages because they did not adhere to the GPL conditions when distributing the device [GPL-Urteil06].

Why does the GPL work? Some companies that thought the GPL conditions onerous have tried to declare or have it declared it invalid. For example, it was called "un-American" or "unconstitutional" in the United States; in Germany, anti-trust law was used in an attempt to prove that the GPL amounts to price fixing. The general idea seems to be that GPL-ed software can be used by anybody if something is demonstrably wrong with the GPL. All these attacks ignore one fact: Without the GPL, nobody except the original author has the right to do *anything* with the code, since actions like sharing (let alone selling) the code are the author's prerogative. So if the GPL goes away, all other interested parties are worse off than they were.

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 license.

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 he is 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.

Freedom criteria
Debian Free Software Guidelines

When is a software package considered "free" or "open source"? There are no definite criteria, but a widely-accepted set of rules are the *Debian Free Software Guidelines* [DFSG]. The FSF summarizes its criteria as the *Four Freedoms* which must hold for a free software package:

- The freedom to use the software for any purpose (freedom 0)
- The freedom to study how the software works, and to adapt it to one's requirements (freedom 1)
- The freedom to pass the software on to others, in order to help one's neighbours (freedom 2)
- The freedom to improve the software and publish the improvements, in order to benefit the general public (freedom 3)

Access to the source code is a prerequisite for freedoms 1 and 3. Of course, common free-software licenses such as the GPL or the BSD license conform to these freedoms.

Multiple licenses

In addition, the owner of a software package is free to distribute it under different licenses at the same time, e.g., the GPL and, alternatively, a "commercial" license that frees the recipient from the GPL restrictions such as the duty to make available the source code for modifications. This way, private users and free software authors can enjoy the use of a powerful programming library such as the "Qt" graphics package (published by Qt Software—formerly Troll Tech—, a Nokia subsidiary), while companies that do not want to make their own source code available may "buy themselves freedom" from the GPL.

Exercises

1.3 [!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.

1.4 [4] Some software licenses require that when a file from a software distribution is changed, it must be renamed. Is software distributed under such a license considered "free" according to the DFSG? Do you think this practice makes sense?

1.4 Linux—The Kernel

Strictly speaking, the name "Linux" only applies to the operating system "kernel", which performs the actual operating system tasks. It takes care of elementary functions like memory and process management and hardware control. Application programs must call upon the kernel to, e.g., access files on disk. The kernel validates such requests and in doing so can enforce that nobody gets to access

other users' private files. In addition, the kernel ensures that all processes in the system (and hence all users) get the appropriate fraction of the available CPU time.

Versions

Of course there is not just one Linux kernel, but there are many different versions. Until kernel version 2.6, we distinguished stable "end-user versions" and unstable "developer versions" as follows:

stable version

- In version numbers such as 1.*x*.*y* or 2.*x*.*y*, *x* denotes a stable version if it is even. There should be no radical changes in stable versions; mistakes should be corrected, and every so often drivers for new hardware components or other very important improvements are added or "back-ported" from the development kernels.

development version

- Versions with odd *x* are development versions which are unsuitable for productive use. They may contain inadequately tested code and are mostly meant for people wanting to take active part in Linux development. Since Linux is constantly being improved, there is a constant stream of new kernel versions. Changes concern mostly adaptations to new hardware or the optimization of various subsystems, sometimes even completely new extensions.

kernel 2.6

The procedure has changed in kernel 2.6: Instead of starting version 2.7 for new development after a brief stabilisation phase, Linus Torvalds and the other kernel developers decided to keep Linux development closer to the stable versions. This is supposed to avoid the divergence of developer and stable versions that grew to be an enormous problem in the run-up to Linux 2.6—most notably because corporations like SUSE and Red Hat took great pains to backport interesting properties of the developer version 2.5 to their versions of the 2.4 kernel, to an extent where, for example, a SUSE 2.4.19 kernel contained many hundreds of differences to the "vanilla" 2.4.19 kernel.

The current procedure consists of "test-driving" proposed changes and enhancements in a new kernel version which is then declared "stable" in a shorter timeframe. For example, after version 2.6.37 there is a development phase during which Linus Torvalds accepts enhancements and changes for the 2.6.38 version. Other kernel developers (or whoever else fancies it) have access to Linus' internal development version, which, once it looks reasonable enough, is made available as the "release candidate" 2.6.38-rc1. This starts the stabilisation phase, where this release candidate is tested by more people until it looks stable enough to be declared the new version 2.6.38 by Linus Torvalds. Then follows the 2.6.39 development phase and so on.

release candidate

In parallel to Linus Torvalds' "official" version, Andrew Morton maintains a more experimental version, the so-called "`-mm` tree". This is used to test larger and more sweeping changes until they are mature enough to be taken into the official kernel by Linus.

`-mm` tree

Some other developers maintain the "stable" kernels. As such, there might be kernels numbered 2.6.38.1, 2.6.38.2, …, which each contain only small and straightforward changes such as fixes for grave bugs and security issues. This gives Linux distributors the opportunity to rely on kernel versions maintained for longer periods of time.

version 3.0

On 21 July 2011, Linus Torvalds officially released version 3.0 of the Linux kernel. This was really supposed to be version 2.6.40, but he wanted to simplify the version numbering scheme. "Stable" kernels based on 3.0 are accordingly numbered 3.0.1, 3.0.2, …, and the next kernels in Linus' development series are 3.1-rc1, etc. leading up to 3.1 and so forth.

Linus Torvalds insists that there was no big difference in functionality between the 2.6.39 and 3.0 kernels—at least not more so than between any two other consecutive kernels in the 2.6 series—, but that there was just a renumbering. The idea of Linux's 20th anniversary was put forward.

You can obtain source code for "official" kernels on the Internet from `ftp.kernel.org`. However, only very few Linux distributors use the original kernel sources. Distribution kernels are usually modified more or less extensively, e. g., by integrating additional drivers or features that are desired by the distribution but not part of the standard kernel. The Linux kernel used in SUSE's *Linux Enterprise Server 8*, for example, reputedly contained approximately 800 modifications to the "vanilla" kernel source. (The changes to the Linux development process have succeeded to an extent where the difference is not as great today.)

Kernel structure

Today most kernels are *modular*. This was not always the case; former kernels consisted of a single piece of code fulfilling all necessary functions such as the support of particular devices. If you wanted to add new hardware or make use of a different feature like a new type of file system, you had to compile a new kernel from sources—a very time-consuming process. To avoid this, the kernel was endowed with the ability to integrate additional features by way of modules.

Modules

Modules are pieces of code that can be added to the kernel dynamically (at runtime) as well as removed. Today, if you want to use a new network adapter, you do not have to compile a new kernel but merely need to load a new kernel module.

hardware recognition

Modern Linux distributions support automatic hardware recognition, which can analyze a system's properties and locate and configure the correct driver modules.

Exercises

1.5 [1] What is the version number of the current stable Linux kernel? The current developer kernel? Which Linux kernel versions are still being supported?

1.5 Linux Properties

As a modern operating system kernel, Linux has a number of properties, some of which are part of the "state of the art" (i. e., exhibited by similar systems in an equivalent form) and some of which are unique to Linux.

processors

- Linux supports a large selection of processors and computer architectures, ranging from mobile phones (the very successful "Android" operating system by Google, like some other similar systems, is based on Linux) through PDAs and tablets, all sorts of new and old PC-like computers and server systems of various kinds up to the largest mainframe computers (the vast majority of the machines on the list of the fastest computers in the world is running Linux).

  A huge advantage of Linux in the mobile arena is that, unlike Microsoft Windows, it supports the energy-efficient and powerful ARM processors that most mobile devices are based upon. In 2012, Microsoft released an ARM-based, partially Intel-compatible, version of Windows 8 under the name of "Windows RT", but that did not exactly prove popular in the market.

hardware

- Of all currently available operating systems, Linux supports the broadest selection of hardware. For the very newest components there may not be drivers available immediately, but on the other hand Linux still works with devices that systems like Windows have long since left behind. Thus, your investments in printers, scanners, graphic boards, etc. are protected optimally.

multitasking

- Linux supports "preemptive multitasking", that is, several processes are running—virtually or, on systems with more than one CPU, even actually—in parallel. These processes cannot obstruct or damage one another; the kernel ensures that every process is allotted CPU time according to its priority.


This is nothing special today; when Linux was new, this was much more remarkable.

On carefully configured systems this may approach real-time behaviour, and in fact there are Linux variants that are being used to control industrial plants requiring "hard" real-time ability, as in guaranteed (quick) response times to external events.

several users
- Linux supports several users on the same system, even at the same time (via the network or serially connected terminals, or even several screens, keyboards, and mice connected to the same computer). Different access permissions may be assigned to each user.

virtualisation
- Linux can effortlessly be installed alongside other operating systems on the same computer, so you can alternately start Linux or another system. By means of "virtualisation", a Linux system can be split into independent parts that look like separate computers from the outside and can run Linux or other operating systems. Various free or proprietary solutions are available that enable this.

efficiency
- Linux uses the available hardware efficiently. The dual-core CPUs common today are as fully utilised as the 4096 CPU cores of a SGI Altix server. Linux does not leave working memory (RAM) unused, but uses it to cache data from disk; conversely, available working memory is used reasonably in order to cope with workloads that are much larger than the amount of RAM inside the computer.

POSIX, System V and BSD
- Linux is source-code compatible with POSIX, System V and BSD and hence allows the use of nearly all Unix software available in source form.

file systems
- Linux not only offers powerful "native" file systems with properties such as journaling, encryption, and logical volume management, but also allows access to the file systems of various other operating systems (such as the Microsoft Windows FAT, VFAT, and NTFS file systems), either on local disks or across the network on remote servers. Linux itself can be used as a file server in Linux, Unix, or Windows networks.

TCP/IP
- The Linux TCP/IP stack is arguably among the most powerful in the industry (which is due to the fact that a large fraction of R&D in this area is done based on Linux). It supports IPv4 and IPv6 and all important options and protocols.

graphical environments
- Linux offers powerful and elegant graphical environments for daily work and, in X11, a very popular network-transparent base graphics system. Accelerated 3D graphics is supported on most popular graphics cards.

productivity applications
- All important productivity applications are available—office-type programs, web browsers, programs to access electronic mail and other communication media, multimedia tools, development environments for a diverse selection of programming languages, and so on. Most of this software comes with the system at no cost or can be obtained effortlessly and cheaply over the Internet. The same applies to servers for all important Internet protocols as well as entertaining games.

embedded systems
The flexibility of Linux not only makes it possible to deploy the system on all sorts of PC-class computers (even "old chestnuts" that do not support current Windows can serve well in the kids' room, as a file server, router, or mail server), but also helps it make inroads in the "embedded systems" market, meaning complete appliances for network infrastructure or entertainment electronics. You will, for example, find Linux in the popular AVM FRITZ!Box and similar WLAN, DSL or telephony devices, in various set-top boxes for digital television, in PVRs, digital cameras, copiers, and many other devices. Your author has seen the bottle bank

in the neighbourhood supermarket boot Linux. This is very often not trumpeted all over the place, but, in addition to the power and convenience of Linux itself the manufacturers appreciate the fact that, unlike comparable operating systems, Linux does not require licensing fees "per unit sold".

Another advantage of Linux and free software is the way the community deals with security issues. In practice, security issues are as unavoidable in free software as they are in proprietary code—at least nobody so far has written and published a software system of interesting size that proved completely free of them in the long run. In particular, it would be improper to claim that free software has no security issues. The differences are more likely to be found on a philosophical level:

security issues

- As a rule, a vendor of proprietary software has no interest in fixing security issues in their code—they will try to cover up problems and to talk down possible dangers for as long as they possibly can, since constantly publishing "patches" means, in the best case, terrible PR ("where there is smoke, there must be a fire"; the competition, which just happens not to be in the spotlight of scrutiny at the moment, is having a secret laugh), and, in the worst case, great expense and lots of hassle if exploits are around that make active use of the security holes. Besides, there is the usual danger of introducing three new errors while fixing one known one, which is why fixing bugs in released software is normally not an econonomically viable proposition.
- A free-software publisher does not gain anything by sitting on information about security issues, since the source code is generally available, and everybody can find the problems. It is, in fact, a matter of pride to fix known security issues as quickly as possible. The fact that the source code is publically available also implies that third parties find it easy to audit code for problems that can be repaired proactively. (A common claim is that the availability of source code exerts a very strong attraction on crackers and other unsavoury vermin. In fact, these low-lifes do not appear to have major difficulties identifying large numbers of security issues in proprietary systems such as Windows, whose source code is *not* generally available. Thus any difference, if it exists, must be minute indeed.)
- Especially as far as software dealing with cryptography (the encryption and decryption of confidential information) is concerned, there is a strong argument that availability of source code is an indispensable prerequisite for trust that a program really does what it is supposed to do, i. e., that the claimed encryption algorithm has been implemented completely and correctly. Linux does have an obvious advantage here.

Linux is used throughout the world by private and professional users—companies, research establishments, universities. It plays an important role particularly as a system for web servers (Apache), mail servers (Sendmail, Postfix), file servers (NFS, Samba), print servers (LPD, CUPS), ISDN routers, X terminals, scientific/engineering workstations etc. Linux is an essential part of industrial IT departments. Widespread adoption of Linux in public administration, such as the city of Munich, also serves as a signal. In addition, many reputable IT companies such as IBM, Hewlett-Packard, Dell, Oracle, Sybase, Informix, SAP, Lotus etc. are adapting their products to Linux or selling Linux versions already. Furthermore, ever more computers ("netbooks")— come with Linux or are at least tested for Linux compability by their vendors.

Linux in companies

Public administration

Support by IT companies

Exercises

1.6 [4] Imagine you are responsible for IT in a small company (20–30 employees). In the office there are approximately 20 desktop PCs and two servers (a file and printer server and a mail and Web proxy server). So far everything runs on Windows. Consider the following scenarios:

- The file and printer server is replaced by a Linux server using Samba (a Linux/Unix-based server for Windows clients).
- The mail and proxy server is replaced by a Linux server.
- The twenty office desktop PCs are replaced by Linux machines.

Comment on the different scenarios and draw up short lists of their advantages and disadvantages.

1.6 Linux Distributions

Distributions

Linux in the proper sense of the word only consists of the operating system kernel. To accomplish useful work, a multitude of system and application programs, libraries, documentation etc. is necessary. "Distributions" are nothing but up-to-date selections of these together with special programs (usually tools for installation and maintenance) provided by companies or other organisations, possibly together with other services such as support, documentation, or updates. Distributions differ mostly in the selection of software they offer, their administration tools, extra services, and price.

Red Hat and Fedora

"Fedora" is a freely available Linux distribution developed under the guidance of the US-based company, Red Hat. It is the successor of the "Red Hat Linux" distribution; Red Hat has withdrawn from the private end-user market and aims their "Red Hat" branded distributions at corporate customers. Red Hat was founded in 1993 and became a publically-traded corporation in August, 1999; the first Red Hat Linux was issued in 1994, the last (version 9) in late April, 2004. "Red Hat Enterprise Linux" (RHEL), the current product, appeared for the first time in March, 2002. Fedora, as mentioned, is a freely available offering and serves as a development platform for RHEL; it is, in effect, the successor of Red Hat Linux. Red Hat only makes Fedora available for download; while Red Hat Linux was sold as a "boxed set" with CDs and manuals, Red Hat now leaves this to third-party vendors.

SUSE

The SUSE company was founded 1992 under the name "Gesellschaft für Software und Systementwicklung" as a Unix consultancy and accordingly abbreviated itself as "S.u.S.E." One of its products was a version of Patrick Volkerding's Linux distribution, Slackware, that was adapted to the German market. (Slackware, in turn, derived from the first complete Linux distribution, "Softlanding Linux System" or SLS.) S.u.S.E. Linux 1.0 came out in 1994 and slowly differentiated from Slackware, for example by taking on Red Hat features such as the RPM package manager 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 were dropped at some point) soon gained market leadership in German-speaking Europe and published SuSE Linux in a "box" in two flavours, "Personal" and "Professional"; the latter was markedly more expensive and contained more server software. Like Red Hat, SuSE offered an enterprise-grade Linux distribution called "SuSE Linux Enterprise Server" (SLES), with some derivatives like a specialised server for mail and groupware ("SuSE Linux OpenExchange Server" or SLOX). In addition, SuSE endeavoured to make their distribution available on IBM's mid-range and mainframe computers.

Novell takeover

In November 2003, the US software company Novell announced their intention of taking over SuSE for 210 million dollars; the deal was concluded in January 2004. (The "U" went uppercase on that occasion). Like Red Hat, SUSE has by now taken the step to open the "private customer" distribution and make it freely available as "openSUSE" (earlier versions appeared for public download only after a delay of several months). Unlike Red Hat,

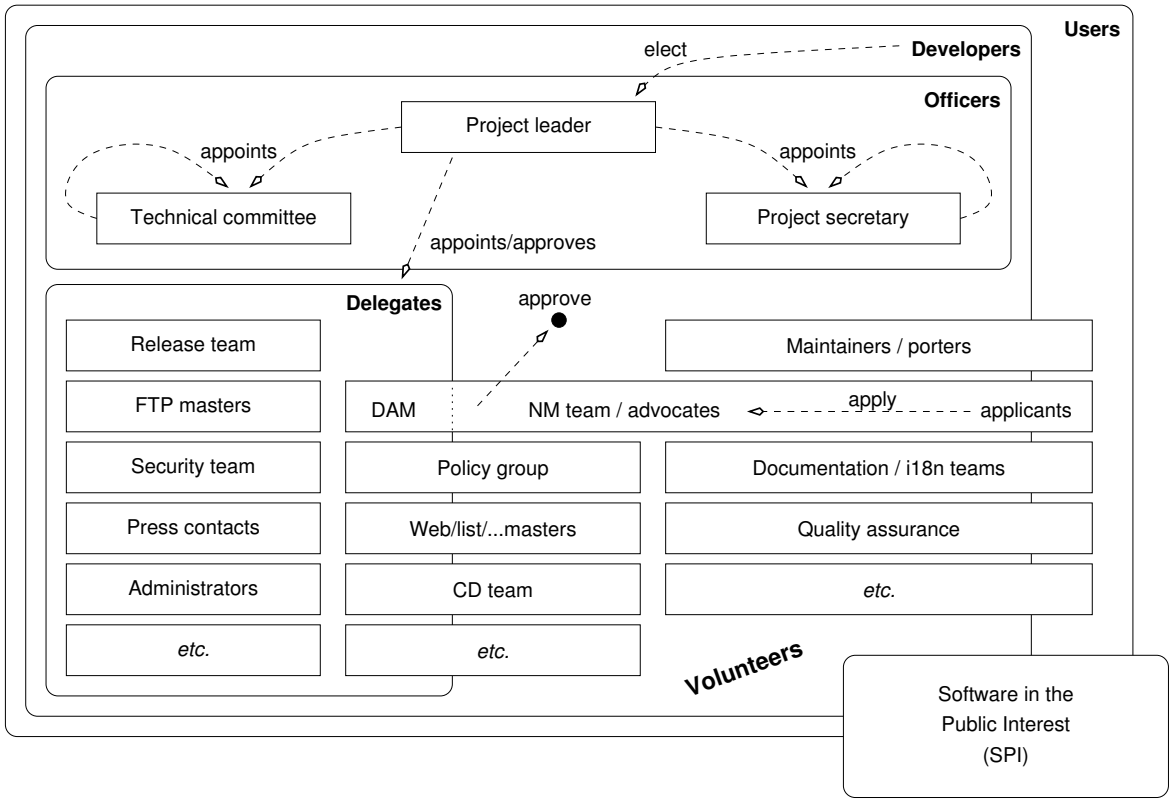


Figure 1.3: Organizational structure of the Debian project. (Graphic by Martin F. Krafft.)

Novell/SUSE still offers a "boxed" version containing additional proprietary software. Among others, SUSE still sells SLES and a corporate desktop platform called "SUSE Linux Enterprise Desktop" (SLED).

Attachmate
Micro Focus

In early 2011, Novell was acquired by Attachmate, which in turn was taken over by Micro Focus in 2014. Both are companies whose main field of business is traditional mainframe computers and which so far haven not distinguished themselves in the Linux and open-source arena. These maneuverings, however, have had fairly little influence on SUSE and its products.

YaST

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

Debian project

Unlike the two big Linux distribution companies Red Hat and Novell/SUSE, the **Debian project** 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 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.

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 between 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.

Ubuntu

goal

One of the most popular Debian derivatives is Ubuntu, which is offered 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 goal 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, 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 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 offer an attractive desktop system, but also 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 …

More distributions

In addition to these distributions there are many more, such as Mageia or Linux Mint as smaller "generally useful" distributions, various "live systems" for different uses from firewalls to gaming or multimedia platforms, or very compact systems usable as routers, firewalls, or rescue systems.

Commonalities

Even though there is a vast number of distributions, most look fairly similar in daily life. This is due, on the one hand, to the fact that they use the same basic programs—for example, the command line interpreter is nearly always `bash`. On

the other hand, there are standards that try to counter rank growth. The "Filesystem Hierarchy Standard" (FHS) or "Linux Standard Base" (LSB) must be mentioned.

Exercises

1.7 [2] Some Linux hardware platforms have been enumerated above. For which of those platforms are there actual Linux distributions available? (Hint: `http://www.distrowatch.org/`)

Summary

- Linux is a Unix-like operating system.
- The first version of Linux was developed by Linus Torvalds and made available on the Internet as "free software". Today, hundreds of developers all over the world contribute to updating and extending the system.
- The GPL is the best-known "free software" license. It tries to ensure that the recipients of software can modify and redistribute the package, and that these "freedoms" are passed on to future recipients. GPL software may also be sold.
- To the user, "open source" means approximately the same as "free software".
- There are other free licenses besides the GPL. Software may also be distributed by the copyright owner under several licenses at the same time.
- Linux is actually just the operating system kernel. We distinguish "stable" and "development kernels"; with the former, the second part of the version number is even and with the latter, odd. Stable kernels are meant for end users, while development kernels are not necessarily functional, representing interim versions of Linux development.
- There are numerous Linux distributions bringing together a Linux kernel and additional software, documentation and installation and administration tools.

Bibliography

DFSG "Debian Free Software Guidelines". `http://www.debian.org/social_contract`

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

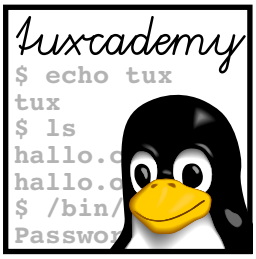
GPL91 Free Software Foundation, Inc. "GNU General Public License, Version 2", June 1991. `http://www.gnu.org/licenses/gpl.html`

LR89 Don Libes, Sandy Ressler. *Life with UNIX: A Guide for Everyone*. Prentice-Hall, 1989. ISBN 0-13-536657-7.

Rit84 Dennis M. Ritchie. "The Evolution of the Unix Time-sharing System". *AT&T Bell Laboratories Technical Journal*, October 1984. **63**(6p2):1577–93. `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`

RT74 Dennis M. Ritchie, Ken Thompson. "The Unix Time-sharing System". *Communications of the ACM*, July 1974. **17**(7):365–73. The classical paper on Unix.

TD02 Linus Torvalds, David Diamond. *Just For Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2002. ISBN 0-066-62073-2.



2

Using the Linux System

Contents

Goals

- Logging on and off the system
- Understanding the difference between normal user accounts and the system administrator's account

Prerequisites

- Basic knowledge of using computers is helpful

Figure 2.1: The login screens of some common Linux distributions

2.1 Logging In and Out

The Linux system distinguishes between different users. Consequently, it may be impossible to start working right after the computer has been switched on. First you have to tell the computer who you are—you need to "log in" (or "on"). Based on the information you provide, the system can decide what you may do (or may 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.

access rights

Modern Linux distributions want 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 in a graphical environment as well. Your computer shows a dialog

that lets you enter your user name and password (Figure 2.1 shows some representative examples.)

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: _
```

(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 both the user name and password were correct, the system will accept your login. It starts the command line interpreter (the *shell*), and you can 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 over 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

2.1 [!1] Try logging in to the system. After that, log out again. (You will find a user name and password in your system documentation, or—in a training centre—your instructor will tell you what to use.)

2.2 [!2] What happens if you give (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?

2.2 Switching On and Off

A Linux computer can usually be switched on by whoever is able to reach the switch (local policy may vary). On the other hand, you should not switch off a Linux machine on a whim—there might be data left in main memory that really belong on disk and will be lost, or—which would be worse—the data on the hard disk could get completely addled. Besides, other users might be logged in to the machine via the network, be surprised by the sudden shutdown, and lose valuable work. For this reason, important computers are usually only "shut down" by the system administrator. Single-user workstations, though, can usually be shut down cleanly via the graphical desktop; depending on the system's settings normal user privileges may suffice, or you may have to enter the administrator's password.

Exercises

2.3 [2] Check whether you can shut down your system cleanly as a normal (non-administrator) user, and if so, try it.

2.3 The System Administrator

As a normal user, your privileges on the system are limited. For example, you may not write to certain files (most files, actually—mostly those that do not belong to you) and not even read some files (e. g., the file containing the encrypted passwords of all users). However, there is a user account for system administration which is not subject to these restrictions—the user "`root`" may read and write all files, and do various other things normal users aren't entitled to. Having administrator (or "root") rights is a privilege as well as a danger—therefore you should only log on to the system as `root` if you actually want to exercise these rights, not just to read your mail or surf the Internet.

Simply pretend you are Spider-Man: "With great power comes great responsibility". Even Spider-Man wears his Spandex suit only if he must …

GUI as `root`: risky

In particular, you should avoid logging in as `root` via the graphical user interface, since all of the GUI will run with `root`'s privileges. This is a possible security risk—GUIs like KDE contain lots of code which is not vetted as thoroughly for security holes as the textual shell (which is, by comparison, relatively compact).

Assuming `root`'s identity

Normally you can use the command "`/bin/su -`" to assume `root`'s identity (and thus `root`'s privileges). `su` asks for `root`'s password and then starts a new shell, which lets you work as if you had logged in as `root` directly. You can leave the shell again using the `exit` command.

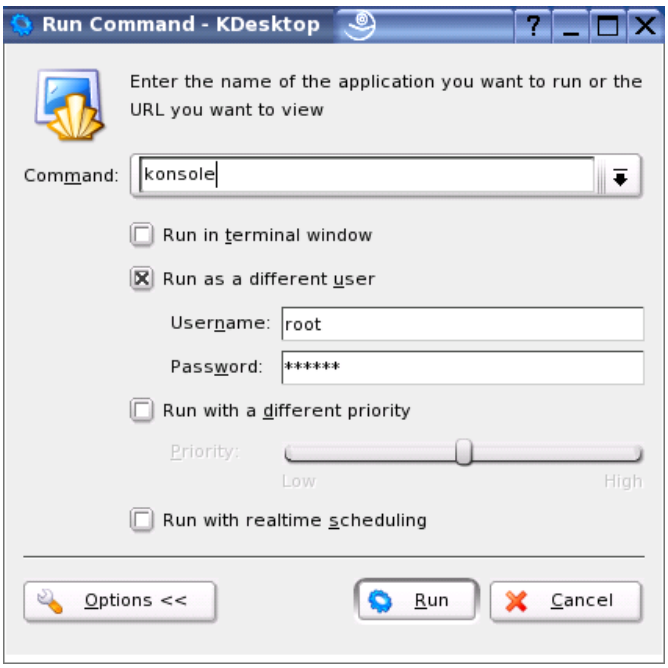


Figure 2.2: Running programs as a different user in KDE

You should get used to invoking `su` via its full path name—"`/bin/su -`". Otherwise, a user could trick you by calling you to her computer, getting you to enter "`su`" in one of her windows and to input the `root` password. What you don't realize at that moment is that the clever user wrote her own "Trojan" `su` command—which doesn't do anything except write the password to a file, output the "wrong password" error message and remove itself. When you try again (gritting your teeth) you get the correct `su`—and your user possesses the coveted administrator's privileges …

`root`'s shell prompt

You can usually tell that you actually have administrator privileges by looking at the shell prompt—for `root`, it customarily ends with the "`#`" character. (For normal users, the shell prompt usually ends in "`$`" or "`>`").

In Ubuntu you can't even log in as `root` by default. Instead, the system allows the first user created during installation to execute commands with administrator privileges by prefixing them with the `sudo` command. With

```
$ sudo chown joe file.txt
```

for example, he could sign over the `file.txt` file to user `joe` – an operation that is restricted to the system administrator.

Recent versions of Debian GNU/Linux offer a similar arrangement to Ubuntu.

`root` and KDE

Incidentally, with the KDE GUI, it is very easy to start arbitrary programs as `root`: Select "Run command" from the "KDE" menu (usually the entry at the very left of the command panel—the "Start" menu on Windows systems), and enter a command in the dialog window. Before executing the command, click on the "Settings" button; an area with additional settings appears, where you can check "As different user" (`root` is helpfully set up as the default value). You just have to enter the `root` password at the bottom (figure 2.2).

kdesu

Alternatively, you can put "kdesu" in front of the actual command in the dialog window (or indeed any shell command line in a KDE session). This will ask you for root's password before starting the command with administrator privileges.

Exercises

2.4 [!1] Use the su command to gain administrator privileges, and change back to your normal account.

2.5 [5] (For programmers.) Write a convincing "Trojan" su program. Use it to try and fool your system administrator.

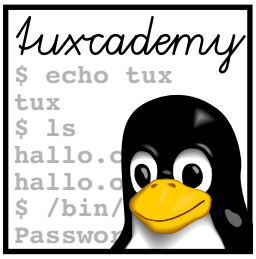
2.6 [2] Try to run the id program as root in a terminal session under KDE, using "Run command …". Check the appropriate box in the extended settings to do so.

Commands in this Chapter

exit	Quits a shell	bash(1)	32
kdesu	Starts a program as a different user on KDE	KDE: help:/kdesu	33
logout	Terminates a shell session	bash(1)	31
su	Starts a shell using a different user's identity	su(1)	32
sudo	Allows normal users to execute certain commands with administrator privileges	sudo(8)	33

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.
- Normal access rights do not apply to user root, who may do (essentially) everything. These privileges should be used as sparingly as possible.
- You should not log in to the GUI as root but use (e. g.) su to assume administrator privileges if necessary.



3

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

3.1 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.

3.1.1 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 "nutshell"—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, …

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 shell

C shell

Korn shell

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.

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.

shells: normal programs

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".

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.

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.

programming language

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

Terminating shell sessions

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

3.1 [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?

3.2 Commands

3.2.1 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.

3.2.2 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 3.2.4.

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.

3.2.3 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

3.2 [2] With bash, which of the following programs are provided externally and which are implemented within the shell itself: `alias`, `echo`, `rm`, `test`?

3.2.4 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.

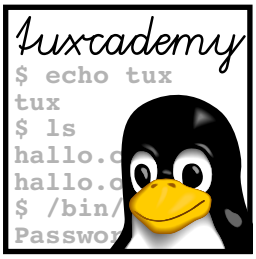
We can't explain all the other special characters here. Most of them will show up elsewhere in this manual – or else check the Bash documentation.

Commands in this Chapter

bash	The "Bourne-Again-Shell", an interactive command interpreter	bash(1)	36
date	Displays the date and time	date(1)	39
echo	Writes all its parameters to standard output, separated by spaces	bash(1), echo(1)	39
help	Displays on-line help for bash commands	bash(1)	39
type	Determines the type of command (internal, external, alias)	bash(1)	39

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.



4

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)

4.1 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.

4.2 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.

4.3 The On-Line Manual

4.3.1 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`

Table 4.1: 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.

4.3.2 Structure

Man page outline

The structure of the man pages loosely follows the outline given in table 4.1, 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 4.2.

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`.

Table 4.2: 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

4.3.3 Chapters

Chapters

Every manual page belongs to a "chapter" of the conceptual "manual" (table 4.2). 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.

4.3.4 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*⟩")

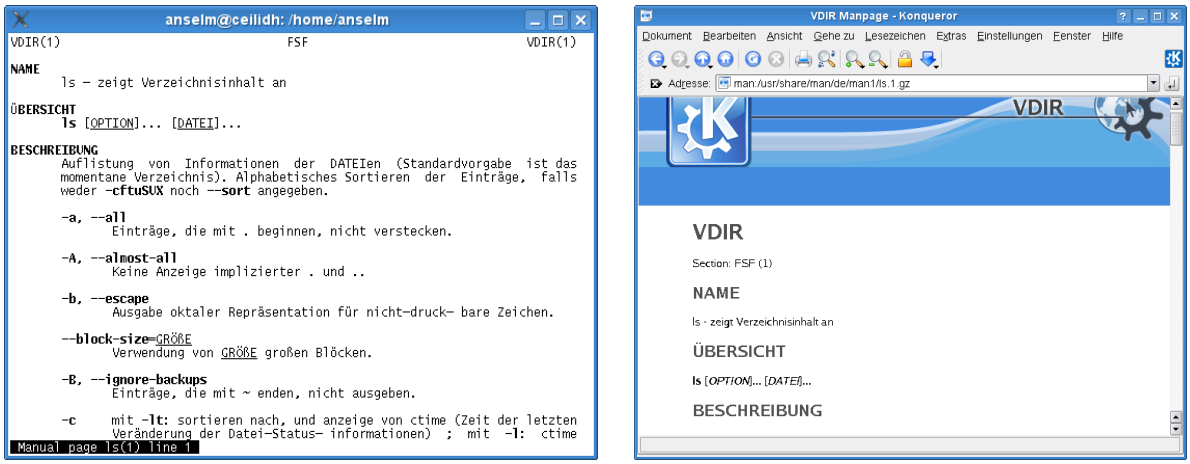


Figure 4.1: A manual page in a text terminal (left) and in Konqueror (right)

in the browser's address line. This also works on the KDE command line (figure 2.2).

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. Keyword search

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`". `whatis`

Exercises

4.1 [!1] View the manual page for the `ls` command. Use the text-based `man` command and—if available—the Konqueror browser.

4.2 [2] Which manual pages on your system deal (at least according to their "NAME" sections) with processes?

4.3 [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 -`").

4.4 Info Pages

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. hypertext

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

4.4 [!1] Look at the info page for the `ls` program. Try the text-based `info` browser and, if available, the Konqueror browser.

4.5 [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?

4.5 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

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.

HOWTOs on the Web

4.6 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.

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.

WWW
USENET

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`.

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.

search engine

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.

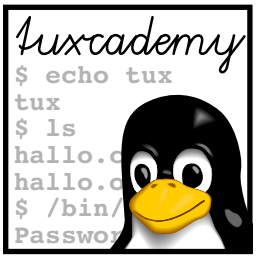
Free documentation

Commands in this Chapter

apropos	Shows all manual pages whose NAME sections contain a given keyword	apropos(1) 45
groff	Sophisticated typesetting program	groff(1) 43, 45
help	Displays on-line help for bash commands	bash(1) 42
info	Displays GNU Info pages on a character-based terminal	info(1) 45
less	Displays texts (such as manual pages) by page	less(1) 44
man	Displays system manual pages	man(1) 42
manpath	Determines the search path for system manual pages	manpath(1) 43
whatis	Locates manual pages with a given keyword in its description	whatis(1) 45

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.



5

Editors: vi and emacs

Contents

Goals

- Becoming familiar with the vi and emacs editors
- Being able to create and change text files

Prerequisites

- Basic shell operation (qv. chapter 2)

5.1 Editors

Most operating systems offer tools to create and change text documents. Such programs are commonly called "editors" (from the Latin "edire", "to work on").

Generally, text editors offer functions considerably exceeding simple text input and character-based editing. Good editors allow users to remove, copy or insert whole words or lines. For long files, it is helpful to be able to search for particular sequences of characters. By extension, "search and replace" commands can make tedious tasks like "replace every `x` by a `u`" considerably easier. Many editors contain even more powerful features for text processing.

Difference to word processors

In contrast to widespread "word processors" such as OpenOffice.org Writer or Microsoft Word, text editors usually do not offer markup elements such as various fonts (Times, Helvetica, Courier, …), type attributes (boldface, italic, underlined, …), typographical features (justified type, …) and so on—they are predominantly intended for the creation and editing of pure text files, where these things would really be a nuisance.

Of course there is nothing wrong with using a text editor to prepare input files for typesetting systems such as `groff` or LaTeX that offer all these typographic features. However, chances are you won't see much of these in your original input—which can really be an advantage: After all, much of the typography serves as a distraction when writing, and authors are tempted to fiddle with a document's appearance while inputting text, rather than concentrating on its content.

Syntax highlighting

Most text editors today support syntax highlighting, that is, identifying certain elements of a program text (comments, variable names, reserved words, strings) by colours or special fonts. This does look spiffy, even though the question of whether it really helps with programming has not yet been answered through suitable psychological studies.

In the rest of the chapter we shall introduce two common Linux editors. However, we shall restrict ourselves to the most basic functionality; it would be easy to conduct multi-day training courses for each of the two. As with the shells, the choice of text editor is up to a user's own personal preference.

Exercises

5.1 [2] Which text editors are installed on your system? How can you find out?

5.2 The Standard—`vi`

5.2.1 Overview

The only text editor that is probably part of every Linux system is called `vi` (from "visual", not Roman 6—usually pronounced "vee-i"). For practical reasons, this

`vi`: today a clone

usually doesn't mean the original `vi` (which was part of BSD and is decidedly long in the tooth today), but more modern derivatives such as `vim` (from "`vi` improved") or `elvis`; these editors are, however, sufficiently close to the original `vi`, to be all lumped together.

`vi`, originally developed by Bill Joy for BSD, was one of the first "screen-oriented" editors in widespread use for Unix. This means that it allowed users to use the whole screen for editing rather than let them edit just one line at a time. This is today considered a triviality, but used to be an innovation—which is not to say that earlier programmers were too stupid to figure it out, but that text terminals allowing free access to arbitrary points on the screen (a mandatory feature for programs like `vi`) had only just become affordable. Out of consideration for older

systems using teletypes or "glass ttys" (terminals that could only add material at the bottom of the screen), vi also supports a line-oriented editor under the name of ex.

Even with the advanced terminals of that time, one could not rely on the availability of keyboards with special keys for cursor positioning or advanced functions—today's standard PC keyboards would have been considered luxurious, if not overloaded. This justifies vi's unusual concepts of operation, which today could rightly be considered antediluvian. It cannot be taken amiss if people reject vi because of this. In spite of this, having rudimentary knowledge of vi cannot possibly hurt, even if you select a different text editor for your daily work—which you should by all means do if vi does not agree with you. It is not as if there was no choice of alternatives, and we shall not get into childish games such as "Whoever does not use vi is not a proper Linux user". Today's graphical desktops such as KDE do contain very nice and powerful text editors.

Keyboard restrictions

There is, in fact, an editor which is even cruder than vi—the ed program. The title "the only editor that is guaranteed to be available on any Unix system" rightfully belongs to ed instead of vi, but ed as a pure line editor with a teletype-style user interface is too basic for even hardcore Unix advocates. (ed can be roughly compared with the DOS program, EDLIN; ed, however, is vastly more powerful than the Redmond offering.) The reason why ed is still available in spite of the existence of dozens of more convenient text editors is unobvious, but very Unix-like: ed accepts commands on its standard input and can therefore be used in shell scripts to change files programmatically. ed allows editing operations that apply to the whole file at once and is, thus, more powerful than its colleague, the "stream editor" sed, which copies its standard input to its standard output with certain modifications; normally one would use sed and revert to ed for exceptional cases, but ed is still useful every so often.

5.2.2 Basic Functions

The Buffer Concept vi works in terms of so-called **buffers**. If you invoke vi with a file name as an argument, the content of that file will be read into a buffer. If no file exists by that name, an empty buffer is created.

buffers

All the modifications made with the editor are only applied inside the buffer. To make these modifications permanent, the buffer content must be explicitly written back to the file. If you really want to discard the modifications, simply leave vi without storing the buffer content—the file on the storage medium will remain unchanged.

In addition to a file name as an argument, you can pass options to vi as usual. Refer to the documentation for the details.

Modes As mentioned earlier, one of the characteristics of vi is its unusual manner of operation. vi supports three different working "modes":

Command mode All keyboard input consists of commands that do not appear on screen and mostly do not need to be finalized using the return key. After invoking vi, you end up in this mode. Be careful: Any key press could invoke a command.

Insert mode All keyboard input is considered text and displayed on the screen. vi behaves like a "modern" editor, albeit with restricted navigation and correction facilities.

Command-line mode This is used to enter long commands. These usually start with a colon (":") and are finished using the return key.

In insert mode, nearly all navigation or correction commands are disabled, which requires frequent alternation between insert and command modes. The fact that

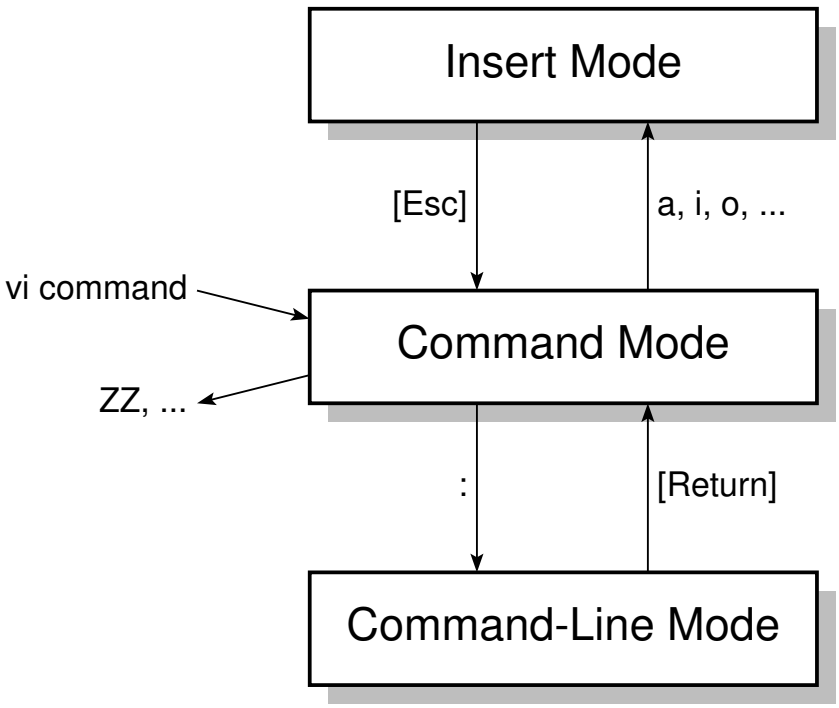


Figure 5.1: vi's modes

Table 5.1: Insert-mode commands for vi

Command	Result
a	Appends new text after the cursor
A	Appends new text at the end of the line
i	Inserts new text at the cursor position
I	Inserts new text at the beginning of the line
o	Inserts a new line below the line containing the cursor
O	Inserts a new line above the line containing the cursor

it may be difficult to find out which mode the editor is currently in (depending on the vi implementation used and its configuration) does not help to make things easier for beginners. An overview of vi modes may be found in figure 5.1.


Consider: vi started when keyboards consisting only of the "letter block" of modern keyboards were common (127 ASCII characters). There was really no way around the scheme used in the program.

command mode

After invoking vi without a file name you end up in command mode. In contrast to most other editors, direct text input is not possible. There is a cursor at the top left corner of the screen above a column filled with tildes. The last line, also called the "status line", displays the current mode (maybe), the name of the file currently being edited (if available) and the current cursor position.


If your version of vi does not display status information, try your luck with Esc :set showmode ↩.

Shortened by a few lines, this looks similar to Das sieht (um einige Zeilen verkürzt) etwa so aus:

```
_
~
~
```

Table 5.2: Cursor positioning commands in vi

Command	Cursor moves …
h or ←	one character to the left
l or →	one character to the right
k or ↑	one character up
j or ↓	one character down
0	to the beginning of the line
$	to the end of the line
w	to the next word
b	to the previous word
f ⟨*character*⟩	to the next ⟨*character*⟩ on the line
Strg + F	to the next page (screenful)
Strg + B	to the previous page
G	to the last line of the file
⟨*n*⟩ G	to line no. ⟨*n*⟩

```
~
Empty Buffer                          0,0-1
```

Only after a command such as a ("append"), i ("insert"), or o ("open") will vi change into "insert mode". The status line displays something like "-- INSERT --", and keyboard input will be accepted as text. insert mode

The possible commands to enter insert mode are listed in table 5.1; note that lower-case and upper-case commands are different. To leave insert mode and go back to command mode, press the Esc key. In command mode, enter Z Z to write the buffer contents to disk and quit vi.

If you would rather discard the modifications you made, you need to quit the editor without saving the buffer contents first. Use the command : q! ↩. The leading colon emphasises that this is a command-line mode command.

When : is entered in command mode, vi changes to command-line mode. You can recognize this by the colon appearing in front of the cursor on the bottom line of the screen. All further keyboard input is appended to that colon, until the command is finished with the return key (↩); vi executes the command and reverts to command mode. In command-line mode, vi processes the line-oriented commands of its *alter ego*, the ex line editor. command-line mode

There is an ex command to save an intermediate version of the buffer called : w ("write"). Commands : x and : wq save the buffer contents and quit the editor; both commands are therefore identical to the Z Z command.

Movement Through the Text In insert mode, newly entered characters will be put into the current line. The return key starts a new line. You can move about the text using cursor keys, but you can remove characters only on the current line using ⇐—an inheritance of vi's line-oriented predecessors. More extensive navigation is only possible in command mode (table 5.2).

Once you have directed the cursor to the proper location, you can begin correcting text in command mode.

Deleting characters The d command is used to delete characters; it is always followed by another character that specifies exactly what to delete (table 5.3). To make editing easier, you can prefix a repeat count to each of the listed commands. For example; the 3 x command will delete the next three characters. repeat count

If you have been too eager and deleted too much material, you can revert the last change (or even all changes one after the other) using the u ("undo") com- undo

Table 5.3: Editing commands in vi

Command	Result
x	Deletes the character below the cursor
X	Deletes the character to the left of the cursor
r ⟨*char*⟩	Replaces the character below the cursor by ⟨*char*⟩
d w	Deletes from cursor to end of current word
d $	Deletes from cursor to end of current line
d 0	Deletes from cursor to start of current line
d f ⟨*char*⟩	Deletes from cursor to next occurrence of ⟨*char*⟩ on the current line
d d	Deletes current line
d G	Deletes from current line to end of text
d 1 G	Deletes from current line to beginning of text

Table 5.4: Replacement commands in vi

Command	Result
c w	Replace from cursor to end of current word
c $	Replace from cursor to end of current line
c 0	Replace from cursor to start of current line
c f ⟨*char*⟩	Replace from cursor to next occurrence of ⟨*char*⟩ on the current line
c / abc	Replace from cursor to next occurrence of character sequence abc

mand. This is subject to appropriate configuration settings.

Overwriting **Replacing characters** The c command ("change") serves to overwrite a selected part of the text. c is a "combination command" similar to d, requiring an additional specification of what to overwrite. vi will remove that part of the text before changing to insert mode automatically. You can enter new material and return to command mode using Esc. (table 5.4).

5.2.3 Extended Commands

Cutting, Copying, and Pasting Text A frequent operation in text editing is to move or copy existing material elsewhere in the document. vi offers handy combination commands to do this, which take specifications similar to those for the c command. y ("yank") copies material to an interim buffer without changing the original text, whereas d moves it to the interim buffer, i. e., it is removed from its original place and only available in the interim buffer afterwards. (We have introduced this as "deletion" above.)

Of course there is a command to re-insert (or "paste") material from an interim buffer. This is done using p (to insert after the current cursor position) or P (to insert at the current cursor position).

26 buffers A peculiarity of vi is that there is not just one interim buffer but 26. This makes it easy to paste different texts (phrases, …) to different places in the file. The interim buffers are called "a" through "z" and can be invoked using a combination of double quotes and buffer names. The command sequence " c y 4 w, for instance, copies the next four words to the interim buffer called c; the command sequence " g p inserts the contents of interim buffer g after the current cursor position.

Regular-Expression Text Search Like every good editor, vi offers well-thought-out search commands. "Regular expressions" make it possible to locate character sequences that fit elaborate search patterns. To start a search, enter a slash [/] in command mode. This will appear on the bottom line of the terminal followed by the cursor. Enter the search pattern and start the search using the return key. vi will start at the current cursor position and work towards the end of the document. To search towards the top, the search must be started using [?] instead of [/] . Once vi has found a matching character sequence, it stops the search and places the cursor on the first character of the sequence. You can repeat the same search towards the end using [n] ("next") or towards the beginning using [N].


Regular expressions are explained in more detail in sections 7.1 and 7.1.1.

Searching and Replacing Since locating character sequences is often not all that is desired. Therefore, vi also allows replacing found character sequences by others. The following ex command can be used:

[:][⟨*start line*⟩,⟨*end line*⟩]s/⟨*regexp*⟩/⟨*replacement*⟩[/q]

The parts of the command within square brackets are optional. What do the different components of the command mean?

⟨*Start line*⟩ and ⟨*end line*⟩ determine the range of lines to be searched. Without these, only the current line will be looked at! Instead of line numbers, you can use a dot to specify the current line or a dollar sign to specify the last line—but do not confuse the meanings of these characters with their meanings within regular expressions (section 7.1).: range of lines

```
:5,$s/red/blue/
```

replaces the first occurrence of red on each line by blue, where the first four lines are not considered.

```
:5,$s/red/blue/g
```

replaces *every* occurrence of red in those lines by blue. (Watch out: Even Fred Flintstone will become Fblue Flintstone.)


Instead of line numbers, ".", and "$", vi also allows regular expressions within slashes as start and end markers:

```
:/^BEGIN/,/^END/s/red/blue/g
```

replaces red by blue only in lines located between a line starting with BEGIN

After the command name [s] and a slash, you must enter the desired regular expression. After another slash, ⟨*replacement*⟩ gives a character sequence by which the original text is to be replaced.

There is a special function for this argument: With a & character you can "reference back" to the text matched by the ⟨*regexp*⟩ in every actual case. That is, "[:]s/bull/& frog" changes every bull within the search range to a bull frog—a task which will probably give genetic engineers trouble for some time to come. Back reference

Command-line Mode Commands So far we have described some command-line mode (or "ex mode") commands. There are several more, all of which can be accessed from command mode by prefixing them with a colon and finishing them with the return key (table 5.5).

Table 5.5: ex commands in vi

Command	Result
: w ⟨*file name*⟩	Writes the complete buffer content to the designated file
: w! ⟨*file name*⟩	Writes to the file even if it is write-protected (if possible)
: e ⟨*file name*⟩	Reads the designated file into the buffer
: e #	Reads the last-read file again
: r ⟨*file name*⟩	Inserts the content of the designated file after the line containing the cursor
: ! ⟨*shell command*⟩	Executes the given shell command and returns to vi afterwards
: r! ⟨*shell command*⟩	Inserts the output of ⟨*shell command*⟩ after the line containing the cursor
: s/⟨*regexp*⟩/⟨*replacement*⟩	Searches for ⟨*regexp*⟩ and replaces by ⟨*replacement*⟩
: q	Quits vi
: q!	Quits vi even if the buffer contents is unsaved
: x oder :e wq	Saves the buffer contents and quits vi

Exercises

5.2 [5] (For systems with vim, e. g., the SUSE distributions.) Find out how to access the interactive vim tutorial and work through it.

5.3 The Challenger—Emacs

5.3.1 Overview

The emacs text editor ("editing macros"[1]) was originally developed by Richard M. Stallman as an extension of the antiquated TECO editor[2]. Later, Stallman wrote a completely new, eponymous program that did not require TECO. According to the GNU philosophy, Emacs source code is freely available. Thus the program can be adapted or extended according to its users' requirements. Emacs is available on nearly every computing platform.

In addition, there are dozens of editors inspired by Emacs. Usually they trade functionality and extensibility for decreased space requirements. The best-known of these is probably "MicroEmacs", which has unfortunately been unmaintained for quite some time now. Another popular Emacs derivative is, for example, jove. The Emacs epigones—or at least a few of them—are not just available for Linux but all computing platforms worth mentioning, including various flavours of Windows.

Emacs derivative

Emacs itself offers many more functions than vi and can easily be considered a complete work environment, especially because it can be extended in a dialect

[1] There are various facetious expansions, such as "Escape-Meta-Alt-Control-Shift", due to the predilection of the program for weird key combinations, or "Eight Megabytes And Constantly Swapping"; when Emacs was new, 8 megabytes of RAM were quite a lot, so today this should probably be read "Eight Hundred Megabytes"—and it cannot be too long before "Eight Thousand" will be appropriate …

[2] TECO was famous for its completely obtuse user interface. If you find vi counter-intuitive, you should not even think of looking at TECO. A favourite pastime among TECO users in the late seventies was to enter one's own first name as a TECO command and try to predict what would happen—not always a straightforward task.

Table 5.6: Possible buffer states in `emacs`

Display	Buffer state
`-:---`	Content unchanged since last write
`-:**-`	Content changed but not written
`-:%%-`	Content can only be read but not modified

of the Lisp programming language. In addition to functioning as a text editor, a complete installation of Emacs allows, for example, file operations such as copying, deleting, etc., sending and receiving e-mail, calling a calendar, diary or pocket calculator, playing Tetris or getting psychoanalyzed[3] and much more. This functionality and the wide availability of Emacs imply that this program is often considered the new standard editor on Linux systems[4]. Diverse Functions

Only `vi` is pertinent to LPI certification. Therefore, if you are just swotting up for the exam and are not interested in a wider learning experience you may skip to the end of this section.

5.3.2 Basic Functions

Buffers Just like `vi`, `emacs` is a screen-oriented editor and uses buffers, keeping the text being edited completely in RAM and making modifications to that copy only. In contrast to `vi`, `emacs` can manage multiple buffers at once, therefore it is possible to edit several files at the same time and move text between them. Even the bottom line on the screen, where commands can be input and messages shown to the user, has its own "mini buffer". Multiple buffers

Since `emacs` does not distinguish between different work modes, you do not have to change to a command mode to execute editor commands. Instead, editor functions are invoked using the Ctrl key in combination with other keys such as x or c, and using another, freely definable key (usually Esc).

In Emacs jargon, the Esc key's function is called "meta". You can also use one of the additional "shift-like" keys available on current PC keyboards. Typically, the Alt key to the left of the space bar is preconfigured accordingly.

Starting, Quitting and Help After launch, `emacs` first displays a launch screen with some noteworthy messages. This looks roughly like figure 5.2. While the manual page lists only the command line parameters for `emacs`, the Ctrl+h command inside the editor invokes the built-in help system. This explains everything one could possibly want for, from keyboard commands to search patterns and much more. There is also a step-by-step tutorial that will introduce you to the basics of `emacs` operation. launch screen

The top line on the screen shows a menu bar, which however is inaccessible by a mouse unless you use a GUI. Since it is difficult to operate via the keyboard, but easy (and not worth explaining) with a mouse in a GUI, we will not consider the menu bar further. menu bar

In the second-to-last line of the screen you find the name of the current buffer, which usually corresponds to the name of the loaded file. If you did not give a file name when launching `emacs`, the editor calls the buffer `*scratch*`. That line also gives information about the current line number (after `L`, thus line 1 here) and the position of the visible part of the text in relation to the text as a whole (here `All`, i. e., all of the text is visible—otherwise a percentage). The beginning of the line displays the state of the buffer (table 5.6). Status line

[3] Honest! Try Esc x `tetris` ↩ or Esc x `doctor` ↩.

[4] Emacs advocates claim that the only reason for the continued existence of `vi` is to edit the Emacs `Makefile`.

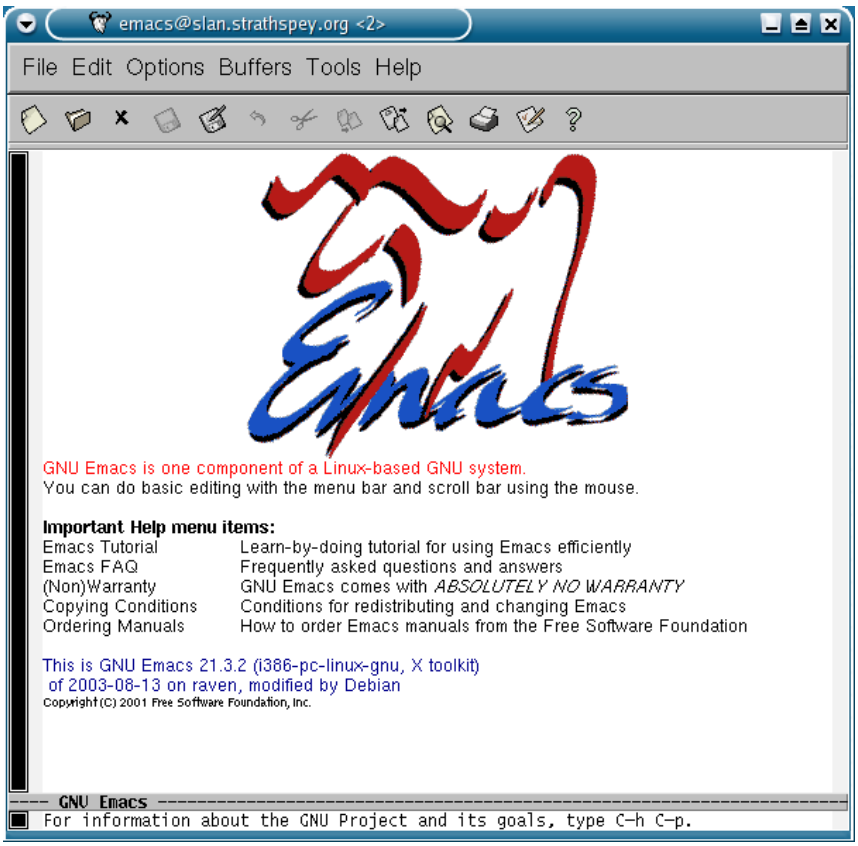


Figure 5.2: The emacs launch screen

As with most editors (excluding vi), you can enter text immediately after launching the program. The cursor can be moved freely around the text, you can delete single characters using the ⇐ key, the return key starts a new line—everything works as usual. Should characters extend past the right margin, Emacs puts a backslash at the right end of the line and continues it in a new line on screen, the actual line length can therefore by far exceed that of a screen line.

At the end the possibly most important information on the launch screen: With the Ctrl+x u key sequence, unsaved modifications can be undone. The Ctrl+x Ctrl+c key sequence quits Emacs, asking whether modified and unsaved buffers should be saved first.

Loading and Saving Files To edit a file using emacs, that file's buffer must be "current". The Ctrl+x Ctrl+f command will read the file into a buffer if it isn't already available, or make that file's buffer the current buffer if the file has already been loaded. The command will prompt for a file name; Emacs supports file name completion using Tab, similar to that of the shell.

The Ctrl+x Ctrl+v command will not create a new buffer for the designated file. Instead, the contents of the current buffer is overwritten and the buffer name changed accordingly. This will of course cause the previous contents of the buffer to be lost.

Furthermore, you can use Ctrl+x i to insert a file's contents at the current cursor position in a buffer. The contents of the current buffer can be stored using Ctrl+x Ctrl+s ("save"). If the buffer does not have a name, the program will prompt for one, otherwise the message "Wrote ⟨*file*⟩" presently appears in the mini buffer line. On the first save operation, an existing file will be made into a "backup copy" by appending a tilde to the file name.

To save the contents of a named buffer in a different file, use the command

Table 5.7: Cursor movement commands in `emacs`

Command	Result
Ctrl + f or →	One character to the right (engl. *forward*)
Ctrl + b or ←	One character to the left (engl. *back*)
Ctrl + n or ↓	One line down (engl. *next*)
Ctrl + p or ↑	One line up (engl. *previous*)
Esc f	Jump to space character in front of next word
Esc b	Jump to first character of previous word
Ctrl + a	Jump to beginning of line
Ctrl + e	Jump to end of line
Esc a	Jump to beginning of sentence
Esc e	Jump to end of sentence
Ctrl + v	Scroll up one screenful
Esc v	Scroll back one screenful
Esc <	Jump to beginning of buffer
Esc >	Jump to end of buffer

Table 5.8: Deletion commands in `emacs`

Command	Result
Ctrl + d	Deletes the character "under" the cursor
Esc d	Deletes from the character under the cursor to the end of the word containing that character
Ctrl + k	Deletes from the cursor to the end of the line (engl. *kill*). A second Ctrl + k deletes the end of the line as well.
Esc k	Deletes the sentence containing the cursor
Ctrl + w	Deletes from the "mark" (set using Ctrl + ␣) to the current cursor position

Ctrl + x Ctrl + w. The program will prompt for a new file name and leaves the original file undisturbed.

You can move between buffers using the Ctrl + x b command. The Ctrl + x Ctrl + b command will display a list of buffers.

Moving about the Text As a rule, the cursor keys can be used to move about the text character by character. Since older terminals in particular often do not feature such keys, there are equivalent Ctrl commands. In a similar way, you can also move through the text by words or sentences (table 5.7).

Deleting To remove an arbitrary character, you must first position the cursor on that character. Then the commands in table 5.8 are available.

5.3.3 Extended Functions

Cutting and Pasting Text If more than a single character has been deleted at one go, that sequence of characters is placed to the "kill buffer". In fact, the text has been "cut" instead of deleted.

The contents of the kill buffer can be inserted at the current cursor position using Ctrl + y. Older deletions are also availabe: Esc y replaces the inserted text by the previous contents of the kill buffer. This can be repeated to reach progressively older material.

Table 5.9: Text-correcting commands in emacs

Command	Result
Ctrl+t	Swaps the character at the current cursor position with the immediately preceding one (if the cursor is at the end of the line, the preceding character will be swapped with the one before that)
Esc t	Swaps the word beginning to the left of the cursor position with the one beginning to the right of the cursor. Punctuation is not moved
Ctrl+x Ctrl+t	Swaps the current and preceding lines
Esc c	Capitalizes the character at the current cursor position and forces all others in the word to lowercase
Esc u	Capitalizes all characters from the cursor position to the end of the word
Esc l	Forces all characters from the cursor position to the end of the word to lowercase

Correcting Typos Less well-practised typists will frequently enter two characters in the wrong sequence, or type lowercase characters as caps—this list could go on and on. The emacs editor offers a few commands to help with correcting these mistakes, which are described in table 5.9.

Searching Owing to the large number of search functions in emacs, we shall restrict ourselves to introducing the two basic search mechanisms: The Ctrl+s command searches from the current cursor position towards the end of the buffer, while Ctrl+r searches in the other direction (towards the beginning). The character sequence in question is displayed in the mini buffer, and the cursor jumps to the next matching place in the buffer while the search string is still being entered ("incremental search").

Additional Functions As mentioned earlier, Emacs offers an enormous number of functions, far exceeding the usual requirements for a text editor. Some of these special extensions are mentioned here:

file types

- emacs is able to recognize various file types by their file name extensions. For example, if a C source file (extension .c) is opened, the editor automatically enters "C mode". This allows automatic indentation and matching of parentheses and braces to make programming easier. Even the C compiler can be called directly from the editor. The same works for most current programming languages.
- If you receive new electronic mail while working in emacs, a notification will appear in the status line. The Esc x rmail command changes to "mail mode", which lets you read, compose and send messages.
- File management in Emacs is possible through "dired" ("directory editor"): Ctrl+x d displays a list of all files in the current directory. You can use the cursor keys to move around in the list, and pressing the return key will open the file under the cursor. You can also delete, copy or rename files.

Emacs Lisp

- The editor can be extended using a dialect of the Lisp programming language. Lisp is a list-oriented language fairly unlike other extension languages, but Emacs Lisp is a very powerful tool. Emacs comes with a manual explaining the language in great detail.

Exercises

5.3 [5] Work through the interactive Emacs tutorial. (The English language version is available via Ctrl+h t; versions in other languages can be invoked using the Emacs "Help" menu.)

5.4 Other Editors

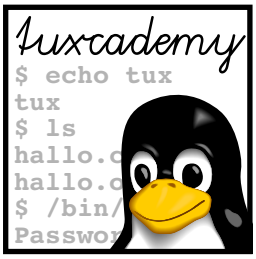
We have already alluded to the fact that your choice of editor is just as much down to your personal preferences and probably says as much about you as a user as your choice of car: Do you drive a polished BMW or are you happy with a dented Astra? Or would you rather prefer a Land Rover? As far as choice is concerned, the editor market offers no less than the vehicle market. We have presented two fairly important contenders, but of course there are many others. `kate` on KDE and `gedit` on GNOME, for example, are straightforward and easy-to-learn editors with a graphical user interface that are perfectly adequate for the requirements of a normal user. Do browse through the package lists of your distribution and check whether you will find the editor of your dreams there.

Commands in this Chapter

ed	Primitive (but useful) line-oriented text editor	ed(1)	51
elvis	Popular "clone" of the vi editor	elvis(1)	50
emacs	Powerful extensible screen-oriented text editor	emacs(1), Info: emacs	55
ex	Powerful line-oriented text editor (really vi)	vi(1)	50
jove	Text editor inspired by emacs	jove(1)	56
sed	Stream-oriented editor, copies its input to its output making changes in the process	sed(1)	51
vi	Screen-oriented text editor	vi(1)	50
vim	Popular "clone" of the vi editor	vim(1)	50

Summary

- Text editors are important for changing configuration files and programming. They often offer special features to make these tasks easier.
- vi is a traditional, very widespread and powerful text editor with an idiosyncratic user interface.
- Emacs is a freely available, modern editor with many special features.



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 (qv. chapter 2)
- Use of a text editor (qv. chapter 5)

6.1 File and Path Names

6.1.1 File Names

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.

files

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.

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 …

`convmv`

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.

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.

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`.

slash

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).

directory tree

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

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).

absolute path name

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".

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 quite as obvious sense: 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
```

```
-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.

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). Long format

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.

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: Creating directories

```
$ 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

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.

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.

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

negated classes

```
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.

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

cartesian product

```
{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.

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.

Move/rename files

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.

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.

Deleting files

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.

`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.

Deleting is forever!

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
```

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
```


"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`).

You can use the "`cp -l`" command to create a "link farm". This means that instead of copying the files to the destination (as would otherwise be usual), links to the originals will be created:

link farm

```
$ mkdir prog-1.0.1                                    New directory
$ cp -l prog-1.0/* prog-1.0.1
```

The advantage of this approach is that the files still exist only once on the disk, and thus take up space only once. With today's prices for disk storage this may not be compellingly necessary—but a common application of this idea, for example, consists of making periodic backup copies of large file hierarchies which should appear on the backup medium (disk or remote computer) as separate, date-stamped file hierarchies. Experience teaches that most files only change very rarely, and if these files then need to be stored just once instead of over and over again, this tends to add up over time. In addition, the files do not need to be written to the backup medium time and again, and that can save considerable time.

Backup packages that adopt this idea include, for example, Rsnapshot (`http://www.rsnapshot.org/`) or Dirvish (`http://www.dirvish.org/`).

This approach should be taken with a certain amount of caution. Using links may let you "deduplicate" identical files, but not identical directories. This means that for every date-stamped file hierarchy on the backup medium, all directories must be created anew, even if the directories only contain links to existing files. This can lead to very complicated directory structures and, in the extreme case, to consistency checks on the backup medium failing because the computer does not have enough virtual memory to check the directory hierarchy.

You will also need to watch out if – as alluded to in the example – you make a "copy" of a program's source code as a link farm (which in the case of, e. g., the Linux source code could really pay off): Before you can modify a file in your newly-created version, you will need to ensure that it is really a separate file and not just a link to the original (which you will very probably not want to change). This means that you either need to manually replace the link to the file by an actual copy of the file, or else use an editor which writes modified versions as separate files automatically[2].

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 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).

symbolic links

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
```

[2] If you use Vim (a. k. a `vi`, you can add the "`set backupcopy=auto,breakhardlink`" command to the `.vimrc` file in your home directory.

```
-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
```

As you have seen above, "`ls -l`" will, for symbolic links, also display the file that the link is pointing to. With the `-L` and `-H` options, you can get `ls` to resolve symbolic links directly:

```
$ mkdir dir
$ echo XXXXXXXXXX >dir/file
$ ln -s file dir/symlink
$ ls -l dir
total 4
-rw-r--r-- 1 hugo users 11 Jan 21 12:29 file
lrwxrwxrwx 1 hugo users  5 Jan 21 12:29 symlink -> file
$ ls -lL dir
-rw-r--r-- 1 hugo users 11 Jan 21 12:29 file
-rw-r--r-- 1 hugo users 11 Jan 21 12:29 symlink
$ ls -lH dir
-rw-r--r-- 1 hugo users 11 Jan 21 12:29 file
lrwxrwxrwx 1 hugo users  5 Jan 21 12:29 symlink -> file
$ ls -l dir/symlink
lrwxrwxrwx 1 hugo users 5 Jan 21 12:29 dir/symlink -> file
$ ls -lH dir/symlink
-rw-r--r-- 1 hugo users 11 Jan 21 12:29 dir/symlink
```

The difference between `-L` and `-H` is that the `-L` option *always* resolves symbolic links and displays information about the actual file (the name shown is still always the one of the link, though). The `-H`, as illustrated by the last three commands in the example, does that only for links that have been directly given on the command line.

`cp` and symbolic links

By analogy to "`cp -l`", the "`cp -s`" command creates link farms based on symbolic links. These, however, are not quite as useful as the hard-link-based ones shown above. "`cp -a`" copies directory hierarchies as they are, keeping symbolic links as they are; "`cp -L`" arranges to replace symbolic links by their targets in the copy, and "`cp -P`" precludes that.

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?

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

6.17 [!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?

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 [2] How do "`ls -lH`" and "`ls -lL`" behave if a symbolic link points to a different symbolic link?

6.22 [3] What is the maximum length of a "chain" of symbolic links? (In other words, if you start with a symbolic link to a file, how often can you create a symbolic link that points to the previous symbolic link?)

6.23 [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`.

`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

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

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. `less`

As mentioned in chapter 4, `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:

```
$ 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.
`-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
./.
```

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

```
./..
./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
```

Exercises

6.24 [!2] Find all files on your system which are longer than 1 MiB, and output their names.

6.25 [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)?

6.26 [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.


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

6.27 [!1] `README` is a very popular file name. Give the absolute path names of all files on your system called `README`.

6.28 [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.

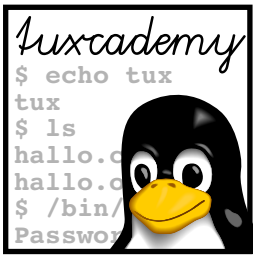
6.29 [1] Convince yourself that the `slocate` program works, by searching for files like `/etc/shadow` as normal user.

Commands in this Chapter

cd	Changes a shell's current working directory	bash(1)	67
convmv	Converts file names between character encodings	convmv(1)	64
cp	Copies files	cp(1)	74
find	Searches files matching certain given criteria	find(1), Info: find	81
less	Displays texts (such as manual pages) by page	less(1)	80
ln	Creates ("hard" or symbolic) links	ln(1)	76
locate	Finds files by name in a file name database	locate(1)	84
ls	Lists file information or directory contents	ls(1)	67
mkdir	Creates new directories	mkdir(1)	69
more	Displays text data by page	more(1)	80
mv	Moves files to different directories or renames them	mv(1)	75
pwd	Displays the name of the current working directory	pwd(1), bash(1)	67
rm	Removes files or directories	rm(1)	75
rmdir	Removes (empty) directories	rmdir(1)	70
slocate	Searches file by name in a file name database, taking file permissions into account	slocate(1)	85
updatedb	Creates the file name database for `locate`	updatedb(1)	85
xargs	Constructs command lines from its standard input	xargs(1), Info: find	83

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.



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 5)

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.

Regular expressions are often constructed "recursively" from primitives that are themselves considered regular expressions. The simplest regular expressions characters 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 classes 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— complement "`[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.)

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, how- Line start ever: A regular expression starting with a caret ("`^`") matches only at the beginning of a line, and a regular expression finishing with a dollar sign ("`$`") matches Line end only at the end. The newline character at the end of each line is ignored, so you can use "`xyz$`" 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.

Finally, you can use the asterisk ("`*`") to denote that the preceding regular ex- Repetition pression 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 precedence 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

The previous section's explanations apply to nearly all Linux programs that deal extensions 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.

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.

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).

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

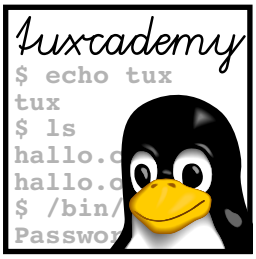
`egrep` Searches files for lines matching specific regular expressions; extended regular expressions are allowed `grep`(1) 89

`fgrep` Searches files for lines with specific content; no regular expressions allowed `fgrep`(1) 89

`grep` Searches files for lines matching a given regular expression `grep`(1) 89

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 (see chapter 2)
- Use of a text editor (see chapter 5)
- File and directory handling (see chapter 6)

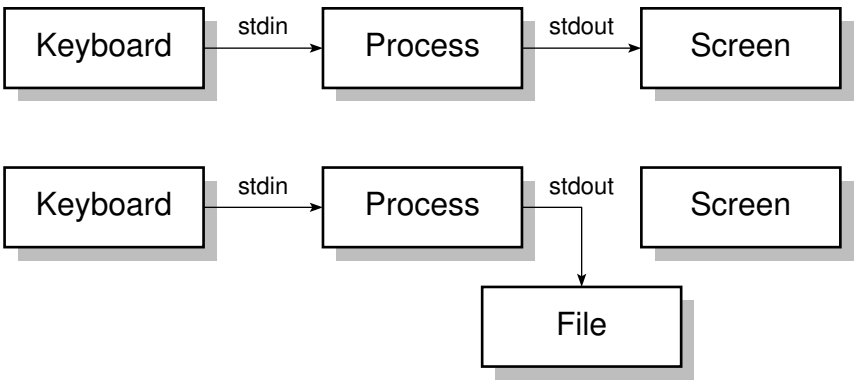


Figure 8.1: Standard channels on Linux

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.

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

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. standard channels

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. Redirection

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

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 Redirecting standard output

```
$ 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).

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. Protecting existing files

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
$ grep `date +%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:

```
$ 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?

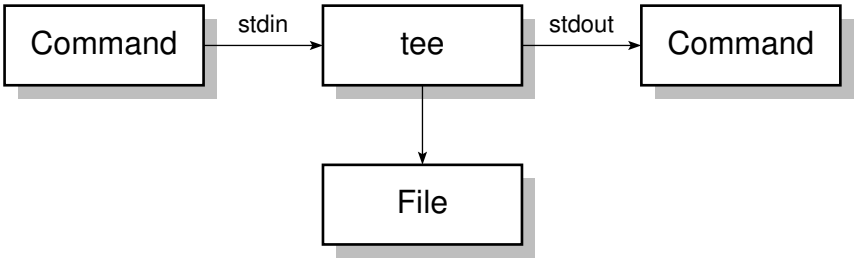


Figure 8.2: The `tee` command

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`.


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.

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.

Sorting by fields

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
```

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.

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` command

`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
⋖⋖⋖⋖⋖
```

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
◁◁◁◁◁
```

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. Relative fields

The default separator is the tab character; other separators may be specified with the `-d` option ("delimiter"): separators

```
$ 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.

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. Suppressing no-field lines

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. Joining lines of files

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. Join files "in parallel"

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).

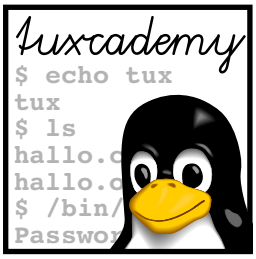
8.19 [3] How many groups are used as primary groups by users on your system? (The primary group of a user is the fourth field in `/etc/passwd`.)

Commands in this Chapter

cat	Concatenates files (among other things)	cat(1)	100
cut	Extracts fields or columns from its input	cut(1)	106
head	Displays the beginning of a file	head(1)	100
paste	Joins lines from different input files	paste(1)	107
reset	Resets a terminal's character set to a "reasonable" value	tset(1)	100
sort	Sorts its input by line	sort(1)	101
tail	Displays a file's end	tail(1)	100
uniq	Replaces sequences of identical lines in its input by single specimens	uniq(1)	105

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 3)
- File management and simple filter commands (Chapter 6, Chapter 8)
- Use of a text editor (Chapter 5)

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.

```
$ 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.

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:

Setting variables

```
$ foo=bar
$ echo foo
foo
$ echo $foo
bar
```

(note the difference).

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.

environment variables
shell variables

Using the `export` command, you can declare an existing shell variable an environment variable:

`export`

```
$ 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:

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 3, 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 tool many Linux users use most often, 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 5) 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 5 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.

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.

Completing command and file names

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.

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)

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.

return value

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 them *en bloc*. (Chapter 5 explains how to conveniently create files.) You just need to invoke the shell and pass the file name as a parameter:

```
$ bash my-commands
```

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.) shell script

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
```

After this, the

```
$ ./my-commands
```

command will suffice.

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 subshell

```
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
```

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!
```

If you omit the "`in` …", the loop iterates over the command line parameters: Loop over 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.

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 `test`

```
#!/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.

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
```

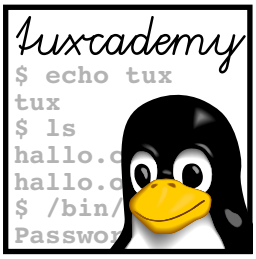
for symbolic links (instead of just »`foo: symbolic link`«)?

Commands in this Chapter

`.`	Reads a file containing shell commands as if they had been entered on the command line	bash(1) 119
`date`	Displays the date and time	date(1) 112
`env`	Outputs the process environment, or starts programs with an adjusted environment	env(1) 114
`export`	Defines and manages environment variables	bash(1) 113
`hash`	Shows and manages "'seen"' commands in `bash`	bash(1) 115
`set`	Manages shell variables and options	bash(1) 114
`source`	Reads a file containing shell commands as if they had been entered on the command line	bash(1) 119
`test`	Evaluates logical expressions on the command line	test(1), bash(1) 121
`unset`	Deletes shell or environment variables	bash(1) 114
`whereis`	Searches executable programs, manual pages, and source code for given programs	whereis(1) 115
`which`	Searches programs along `PATH`	which(1) 115

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
file contained in a file you can use the `file` command:

```
$ 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:

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

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`. FHS

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. root directory

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.

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.

"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.

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`.

log files

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

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 "/".

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.

partitioning

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.

server systems

More information and strategies for partitioning are presented in the Linup Front training manual, *Linux Administration I*.

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.

`/etc/fstab`

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:

```
# 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''
```

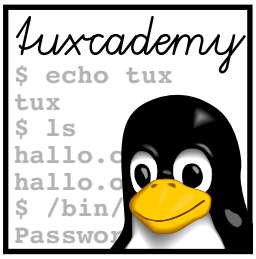
the administrator can also unmount the medium again.

Commands in this Chapter

dmesg	Outputs the content of the kernel message buffer	dmesg(8)	133
file	Guesses the type of a file's content, according to rules	file(1)	126
free	Displays main memory and swap space usage	free(1)	132
klogd	Accepts kernel log messages	klogd(8)	133
mkfifo	Creates FIFOs (named pipes)	mkfifo(1)	127
mknod	Creates device files	mknod(1)	127
syslogd	Handles system log messages	syslogd(8)	133
uptime	Outputs the time since the last system boot as well as the system load averages	uptime(1)	131

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 3)
- 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?

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
◁◁◁◁◁
```

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.)

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

```
$ 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.

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

The .zip suffix may be omitted

```
$ unzip -v test
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?

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

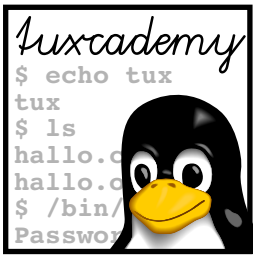
`bunzip2`	File decompression program for `.bz2` files	`bzip2`(1)	144
`bzip2`	File compression program	`bzip2`(1)	139
`gzip`	File compression utility	`gzip`(1)	139
`split`	Splits a file into pieces up to a given maximum size	`split`(1)	139
`tar`	File archive manager	`tar`(1)	138
`unzip`	Decompression software for (Windows-style) ZIP archives	`unzip`(1)	145
`vimtutor`	Interactive introduction to `vim`	`vimtutor`(1)	150
`zip`	Archival and compression software like PKZIP	`zip`(1)	144

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`



A

Sample Solutions

This appendix contains sample solutions for selected exercises.

1.2 There is a copy of `linux-0.01.tar.gz` on `ftp.kernel.org`.

1.3

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).

1.5 This exercise can of course not be answered correctly in printed courseware. Look around—on `ftp.kernel.org` or in the weekly edition of `http://lwn.net/`.

2.2 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.

2.5 Decency forbids us from printing a sample program here. It is reasonably simple using the (deprecated) C function `getpass`(3).

3.1 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.

3.2 `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.

4.2 Try "`apropos process`" or "`man -k process`".

4.5 The format and tools for info files were written in the mid-1980s. HTML wasn't even invented then.

5.1 In theory, you could start every program on the system and check whether it behaves like a text editor … but that might take more time than this exercise is worth. You could, for example, begin with a command like "`apropos edit`" and see which of the man pages in the output correspond with an actual text editor (rather than an editor for graphics, icons, X resources or some such). Text editors from graphical desktop environments such as KDE or GNOME frequently do not actually have man pages, but are documented within the desktop environment, so it can be useful to check the desktop's menus for a submenu such as "Editors" or "Office". The third possibility is to use a package management command—such as "`rpm -qa`" or "`dpkg -l`"—to obtain a list of installed software packages and check for text editors there.

5.2 The program is called `vimtutor`.

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.

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 The chain of symbolic links will be followed until you reach something that is not a symbolic link. However, the maximum length of such chains is usually bounded (see Exercise 6.22).

6.22 Examining this question becomes easier if you can use shell loops (see, e. g., the *Advanced Linux* training manual). Something like

```
$ touch d
$ ln -s d L1
$ i=1
$ while ls -lH L$i >/dev/null
> do
>   ln -s L$i L$((i+1))
```

```
>   i=$((i+1))
> done
```

creates a "chain" of symbolic links where every link points to the previous one. This is continued until the "`ls -lH`" command fails. The error message will tell you which length is still allowed. (On the author's computer, the result is "40", which in real life should not unduly cramp anybody's style.)

6.23 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, usually 4 KiB), but there is a special exception in the `ext` 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. Other 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.

6.24 One possible command could be "`find / -size +1024k -print`".

6.25 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.

6.26 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.)

6.27 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.

6.28 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.

6.29 `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`".

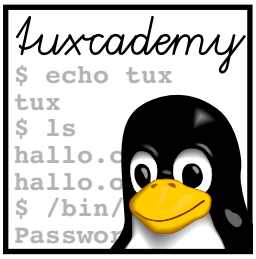
11.8 Take care: To handle `gzip`-compressed `tar` archives, `tar` requires the `-z` option: "`tar -tzf contents.tar.gz`". To restore the original *archive*, you need the "`gunzip contents.tar.gz`" command.

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"
```



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.«

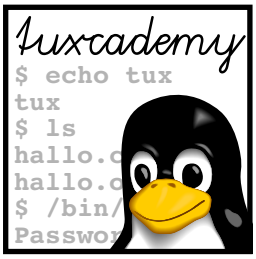
In the meantime it knocked a second time, and cried, »Princess, youngest princess, open the door for me, do you not know what you said to me yesterday by the cool waters of the well. Princess, youngest princess, open the door for me.«

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

```
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.)



C

LPIC-1 Certification

C.1 Overview

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.

Through LPIC-1 certification you can demonstrate basic Linux skills, as required, e. g., for system administrators, developers, consultants, or user support professionals. The certification is targeted towards Linux users with 1 to 3 years of experience and consists of two exams, LPI-101 and LPI-102. These are offered as computer-based multiple-choice and fill-in-the-blanks tests in all Pearson VUE and Thomson Prometric test centres. On its web pages at `http://www.lpi.org/`, the LPI publishes **objectives** outlining the content of the exams.

objectives

This training manual is part of Linup Front GmbH's curriculum for preparation of the LPI-101 exam and covers part of the official examination objectives. Refer to the tables below for details. An important observation in this context is that the LPIC-1 objectives are not suitable or intended to serve as a didactic outline for an introductory course for Linux. For this reason, our curriculum is not strictly geared towards the exams or objectives as in "Take classes x and y, sit exam p, then take classes a and b and sit exam q." This approach leads many prospective students to the assumption that, being complete Linux novices, they could book n days of training and then be prepared for the LPIC-1 exams. Experience shows that this does not work in practice, since the LPI exams are deviously constructed such that intensive courses and exam-centred "swotting" do not really help.

Accordingly, our curriculum is meant to give you a solid basic knowledge of Linux by means of a didactically reasonable course structure, and to enable you as a participant to work independently with the system. LPIC-1 certification is not a primary goal or a goal in itself, but a natural consequence of your newly-obtained knowledge and experience.

C.2 Exam LPI-101

The following table displays the objectives for the LPI-101 exam and the materials covering these objectives. The numbers in the columns for the individual manuals refer to the chapters containing the material in question.

No	Wt	Title	GRD1	ADM1
101.1	2	Determine and configure hardware settings	–	5–6
101.2	3	Boot the system	–	8–10
101.3	3	Change runlevels/boot targets and shutdown or reboot system	–	9–10
102.1	2	Design hard disk layout	–	6
102.2	2	Install a boot manager	–	8
102.3	1	Manage shared libraries	–	11
102.4	3	Use Debian package management	–	12
102.5	3	Use RPM and YUM package management	–	13
103.1	4	Work on the command line	3–4	–
103.2	3	Process text streams using filters	8	–
103.3	4	Perform basic file management	6, 11	7.3
103.4	4	Use streams, pipes and redirects	8	–
103.5	4	Create, monitor and kill processes	–	4
103.6	2	Modify process execution priorities	–	4
103.7	2	Search text files using regular expressions	7–8	–
103.8	3	Perform basic file editing operations using vi	5, 7	–
104.1	2	Create partitions and filesystems	–	6–7
104.2	2	Maintain the integrity of filesystems	–	7
104.3	3	Control mounting and unmounting of filesystems	–	7
104.4	1	Manage disk quotas	–	7.4
104.5	3	Manage file permissions and ownership	–	3
104.6	2	Create and change hard and symbolic links	6	–
104.7	2	Find system files and place files in the correct location	6, 10	–

C.3 LPI Objectives In This Manual

103.1 Work on the command line

Weight 4

Description Candidates should be able to interact with shells and commands using the command line. The objective assumes the Bash shell.

Key Knowledge Areas

- Use single shell commands and one line command sequences to perform basic tasks on the command line
- Use and modify the shell environment including defining, referencing and exporting environment variables
- Use and edit command history
- Invoke commands inside and outside the defined path

The following is a partial list of the used files, terms and utilities:

- `bash`
- `echo`
- `env`
- `export`
- `pwd`
- `set`
- `unset`
- `man`
- `uname`
- `history`
- `.bash_history`

103.2 Process text streams using filters

Weight 3
Description Candidates should should be able to apply filters to text streams.
Key Knowledge Areas

- Send text files and output streams through text utility filters to modify the output using standard UNIX commands found in the GNU textutils package

The following is a partial list of the used files, terms and utilities:

- `cat`
- `cut`
- `expand`
- `fmt`
- `head`
- `join`
- `less`
- `nl`
- `od`
- `paste`
- `pr`
- `sed`
- `sort`
- `split`
- `tail`
- `tr`
- `unexpand`
- `uniq`
- `wc`

103.3 Perform basic file management

Weight 4
Description Candidates should be able to use the basic Linux commands to manage files and directories.
Key Knowledge Areas

- Copy, move and remove files and directories individually
- Copy multiple files and directories recursively
- Remove files and directories recursively
- Use simple and advanced wildcard specifications in commands
- Using find to locate and act on files based on type, size, or time
- Usage of tar, cpio and dd

The following is a partial list of the used files, terms and utilities:

- `cp`
- `find`
- `mkdir`
- `mv`
- `ls`
- `rm`
- `rmdir`
- `touch`
- `tar`
- `cpio`
- `dd`

- `file`
- `gzip`
- `gunzip`
- `bzip2`
- `xz`
- file globbing

103.4 Use streams, pipes and redirects

Weight 4

Description Candidates should be able to redirect streams and connect them in order to efficiently process textual data. Tasks include redirecting standard input, standard output and standard error, piping the output of one command to the input of another command, using the output of one command as arguments to another command and sending output to both stdout and a file.

Key Knowledge Areas

- Redirecting standard input, standard output and standard error
- Pipe the output of one command to the input of another command
- Use the output of one command as arguments to another command
- Send output to both stdout and a file

The following is a partial list of the used files, terms and utilities:

- `tee`
- `xargs`

103.7 Search text files using regular expressions

Weight 2

Description Candidates should be able to manipulate files and text data using regular expressions. This objective includes creating simple regular expressions containing several notational elements. It also includes using regular expression tools to perform searches through a filesystem or file content.

Key Knowledge Areas

- Create simple regular expressions containing several notational elements
- Use regular expression tools to perform searches through a filesystem or file content

The following is a partial list of the used files, terms and utilities:

- `grep`
- `egrep`
- `fgrep`
- `sed`
- `regex`(7)

103.8 Perform basic file editing operations using vi

Weight 3

Description Candidates should be able to edit text files using `vi`. This objective includes `vi` navigation, basic `vi` modes, inserting, editing, deleting, copying and finding text.

Key Knowledge Areas

- Navigate a document using `vi`
- Use basic `vi` modes

- Insert, edit, delete, copy and find text

The following is a partial list of the used files, terms and utilities:

- `vi`
- `/`, `?`
- `h`, `j`, `k`, `l`
- `i`, `o`, `a`
- `c`, `d`, `p`, `y`, `dd`, `yy`
- `ZZ`, `:w!`, `:q!`, `:e!`

104.6 Create and change hard and symbolic links

Weight 2
Description Candidates should be able to create and manage hard and symbolic links to a file.
Key Knowledge Areas

- Create links
- Identify hard and/or soft links
- Copying versus linking files
- Use links to support system administration tasks

The following is a partial list of the used files, terms and utilities:

- `ln`
- `ls`

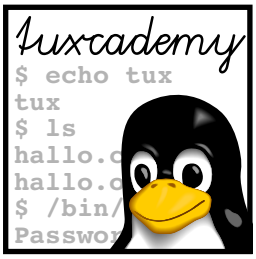
104.7 Find system files and place files in the correct location

Weight 2
Description Candidates should be thoroughly familiar with the Filesystem Hierarchy Standard (FHS), including typical file locations and directory classifications.
Key Knowledge Areas

- Understand the correct locations of files under the FHS
- Find files and commands on a Linux system
- Know the location and purpose of important file and directories as defined in the FHS

The following is a partial list of the used files, terms and utilities:

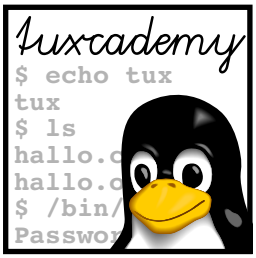
- `find`
- `locate`
- `updatedb`
- `whereis`
- `which`
- `type`
- `/etc/updatedb.conf`



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.



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".