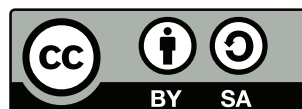
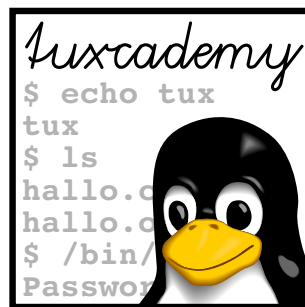




Version 4.0

Advanced Linux

The Linux Shell and Toolkit



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-102 (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.

Advanced Linux The Linux Shell and Toolkit

Revision: grd2:3c9f6dc34a335deb:2015-08-05

grd2:6eb247d0aa1863fd:2015-08-05 1–12, B–C

grd2:FcCTs1UMUHEQK5JN9VsKMJ

© 2015 Linup Front GmbH Darmstadt, Germany

© 2015 tuxcademy (Anselm Lingnau) Darmstadt, Germany

<http://www.tuxcademy.org> · info@tuxcademy.org

Linux penguin “Tux” © Larry Ewing (CC-BY licence)

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



This document is published under the “Creative Commons-BY-SA 4.0 International” licence. You may copy and distribute it and make it publically available as long as the following conditions are met:

Attribution You must make clear that this document is a product of the tuxcademy project.

Share-Alike You may alter, remix, extend, or translate this document or modify or build on it in other ways, as long as you make your contributions available under the same licence as the original.

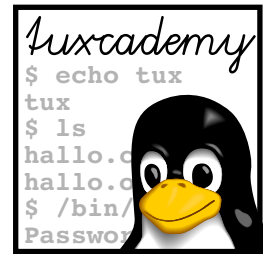
Further information and the full legal license grant may be found at <http://creativecommons.org/licenses/by-sa/4.0/>

Authors: Tobias Elsner, Anselm Lingnau

Technical Editor: Anselm Lingnau (anselm@tuxcademy.org)

English Translation: Anselm Lingnau

Typeset in Palatino, Optima and DejaVu Sans Mono

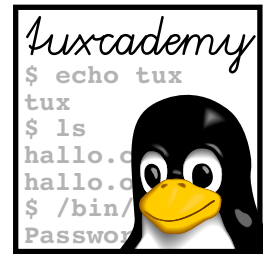


Contents

1 Shell Generalities	13
1.1 Shells and Shell Scripts	14
1.2 Shell Types	14
1.3 The Bourne-Again Shell	16
1.3.1 The Essentials	16
1.3.2 Login Shells and Interactive Shells.	17
1.3.3 Non-Interactive Shell	18
1.3.4 Permanent Configuration Changes	19
1.3.5 Keyboard Maps and Abbreviations	20
2 Shell Scripts	23
2.1 Introduction.	24
2.2 Invoking Shell Scripts	24
2.3 Shell Script Structure	26
2.4 Planning Shell Scripts	27
2.5 Error Types	28
2.6 Error Diagnosis	29
3 The Shell as a Programming Language	31
3.1 Variables	32
3.2 Arithmetic Expressions.	38
3.3 Command Execution	38
3.4 Control Structures	39
3.4.1 Overview	39
3.4.2 A Program's Return Value as a Control Parameter	40
3.4.3 Conditionals and Multi-Way Branches	42
3.4.4 Loops	46
3.4.5 Loop Interruption	49
3.5 Shell Functions.	51
3.5.1 The exec Command	52
4 Practical Shell Scripts	55
4.1 Shell Programming in Practice	56
4.2 Around the User Database	56
4.3 File Operations.	60
4.4 Log Files	62
4.5 System Administration	68
5 Interactive Shell Scripts	73
5.1 Introduction.	74
5.2 The read Command	74
5.3 Menus with select	76
5.4 "Graphical" Interfaces Using dialog	80

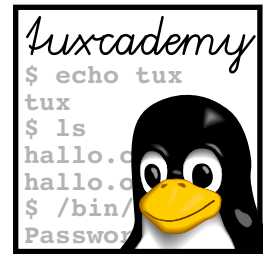
6	The sed Stream Editor	87
6.1	Introduction	88
6.2	Addressing	88
6.3	sed Commands	90
6.3.1	Printing and Deleting Lines	90
6.3.2	Inserting and Changing	91
6.3.3	Character Transformations	91
6.3.4	Searching and Replacing	92
6.4	sed in Practice	93
7	The awk Programming Language	97
7.1	What is awk?	98
7.2	awk Programs	98
7.3	Expressions and Variables	100
7.4	awk in Practice	104
8	SQL	113
8.1	Foundations of SQL	114
8.1.1	Summary	114
8.1.2	Applications of SQL	115
8.2	Defining Tables	117
8.3	Data Manipulation and Queries	118
8.4	Relations	123
8.5	Practical Examples	125
9	Time-controlled Actions—cron and at	131
9.1	Introduction	132
9.2	One-Time Execution of Commands	132
9.2.1	at and batch	132
9.2.2	at Utilities	134
9.2.3	Access Control	134
9.3	Repeated Execution of Commands	135
9.3.1	User Task Lists	135
9.3.2	System-Wide Task Lists	136
9.3.3	Access Control	137
9.3.4	The crontab Command	137
9.3.5	Anacron	138
10	Localisation and Internationalisation	141
10.1	Summary	142
10.2	Character Encodings	142
10.3	Linux Language Settings	146
10.4	Localisation Settings	147
10.5	Time Zones	151
11	The X Window System	157
11.1	Fundamentals	158
11.2	X Window System configuration	163
11.3	Display Managers	169
11.3.1	X Server Starting Fundamentals	169
11.3.2	The LightDM Display Manager	170
11.3.3	Other Display Managers	172
11.4	Displaying Information	173
11.5	The Font Server	175
11.6	Remote Access and Access Control	177

12 Linux Accessibility	181
12.1 Introduction	182
12.2 Keyboard, Mouse, and Joystick	182
12.3 Screen Display	183
A Sample Solutions	185
B Regular Expressions	199
B.1 Overview	199
B.2 Extras	200
C LPIC-1 Certification	203
C.1 Overview	203
C.2 Exam LPI-102	203
C.3 LPI Objectives In This Manual	204
D Command Index	209
Index	211



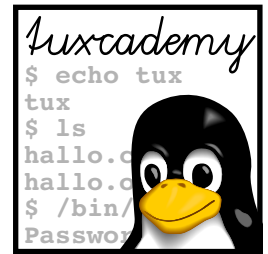
List of Tables

3.1	Reserved return values for bash	40
5.1	dialog's interaction elements	80
6.1	Regular expressions supported by sed and their meaning	89
10.1	The most common parts of ISO/IEC 8859	143
10.2	LC_* environment variables	149
10.3	Daylight Saving Time (DST) in Germany	152
B.1	Regular expression support	201



List of Figures

3.1	A simple init script	45
4.1	Which users have a particular primary group? (Improved version)	58
4.2	In which groups is user <i>x</i> ?	60
4.3	Mass file name extension changing	62
4.4	Watching multiple log files	66
4.5	df with bar graphs for disk use	70
5.1	A dialog-style menu	81
5.2	A dialog-capable version of wwtb	85
8.1	A database table: Famous spaceship commanders from films	114
8.2	Famous spaceship commanders from films (normalised)	115
8.3	The complete schema of our sample database	117
8.4	The calendar-upcoming Script	128
11.1	The X Window System as a client-server system	158



Preface

This manual is intended for advanced Linux users. It enables them to use the system in a more productive way. Building on the tuxcademy manual *Introduction to Linux* or equivalent knowledge, it provides a thorough introduction to advanced uses of the Bourne shell and shell programming. Numerous practical examples illustrate these principles. Further topics include the stream editor *sed*, the *awk* programming language, and relational database access with *SQL*.

Finally the document discusses scheduled command execution with the *at* and *cron* facilities, provides an introduction to internationalisation and localisation of Linux systems, explains the use and administration of the graphical X11 Windowing interface, and, last but not least, gives a summary of Linux's accessibility features.

This manual assumes that its readers know how to enter commands at the command line and are familiar with the most common Linux commands (roughly the material of the LPI-101 exam). Being able to use a text editor is another important prerequisite. System administration skills are a definite plus, but they are not absolutely necessary. Some parts of this document are only interesting to system administrators, though.

Finishing this manual successfully (or gathering equivalent knowledge) is a prerequisite for the tuxcademy manual *Linux System Administration II*, other manuals that build on it, and a certification with the *Linux Professional Institute*.

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

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

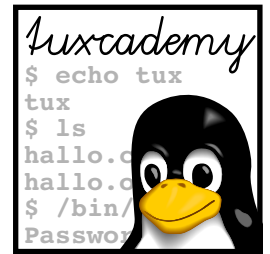
We endeavour to provide courseware that is as up-to-date, complete and error-free as possible. In spite of this, problems or inaccuracies may creep in. If you notice something that you think could be improved, please do let us know, e.g., by sending e-mail to

`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

Shell Generalities

Contents

1.1	Shells and Shell Scripts	14
1.2	Shell Types	14
1.3	The Bourne-Again Shell	16
1.3.1	The Essentials	16
1.3.2	Login Shells and Interactive Shells.	17
1.3.3	Non-Interactive Shell	18
1.3.4	Permanent Configuration Changes	19
1.3.5	Keyboard Maps and Abbreviations	20

Goals

- Learning the basics of shells and shell scripts
- Being able to distinguish login, interactive and non-interactive shells
- Knowing the methods of bash configuration

Prerequisites

- Familiarity with the Linux command line interface
- File handling and use of a text editor

1.1 Shells and Shell Scripts

shell The **shell** allows you to interact with a Linux system directly: You can state commands that are evaluated and executed by the shell—usually by means of starting external programs. The shell is also called a “command interpreter”.

command interpreter In addition, most shells include programming language features—variables, control structures such as conditionals, loops, functions, and more. Thus you can place complex sequences of shell commands and external program calls in text files and have them interpreted as **shell scripts**. This makes difficult operations repeatable and reoccurring processes effortless.

shell scripts The Linux system uses shell scripts for many internal tasks. For example, the “init scripts” in `/etc/init.d` are generally implemented as shell scripts. This also applies to many system commands. Linux makes it possible to hide the fact that a “system program” is not directly-executable machine code but a shell script from its users, at least as far the invocation syntax is concerned—if you do want to know for sure, you can of course find out the truth, for example using the `file` command:

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),>
< for GNU/Linux 2.2.0, dynamically linked (uses shared libs),>
< stripped
$ file /bin/egrep
/bin/egrep: Bourne shell script text executable
```

Exercises



1.1 [!2] What could be the advantages of implementing a command as shell script rather than a binary program? The disadvantages? Under which circumstances would you opt for a shell script rather than a binary program?



1.2 [3] How many commands in your Linux system’s `/bin` and `/usr/bin` directories are implemented as shell scripts? Give a command pipeline that will answer this question. (*Hint*: Use the `find`, `file`, `grep`, and `wc` commands.)

1.2 Shell Types

bash The canonical shell on Linux is the GNU project’s “Bourne-Again Shell” (`bash`). It is largely compatible to the first usable shell of Unix history (the Bourne shell) and to the POSIX standard. Traditionally, other shells are also used, such as the C shell (`csh`) from BSD and its improved successor, the Tenex C shell (`tcsh`), the Korn shell (`ksh`) and some others. These shells differ to a greater or lesser extent in their features and possibilities as well as their syntax.



With shells, one can identify two big incompatible schools of thought, namely the Bourne-like shells (`sh`, `ksh`, `bash`, ...) and the C-Shell-like shells (`csh`, `tcsh`, ...). The Bourne-Again Shell tries to integrate the most important C shell features.

Which shell for what? As a user, you can decide for yourself which shell to use. Either you start the shell of your dreams by means of an appropriate program invocation, or you set up that shell as your login shell, which the system will start for you when you log in. Remember that your login shell is specified within the user database (generally in `/etc/passwd`). You can change your login shell entry using the `chsh` command:

```
$ getent passwd tux
tux:x:1000:1000:Tux the Penguin:/bin/bash:/home/tux
```

```

$ chsh
Password: secret
Changing the login shell for tux
Enter the new value, or press return for the default
    Login Shell [/bin/bash]: /bin/csh
$ getent passwd tux
tux:x:1000:1000:Tux the Penguin:/bin/csh:/home/tux
$ chsh -s /bin/bash
Password: secret
$ getent passwd tux
tux:x:1000:1000:Tux the Penguin:/bin/bash:/home/tux

```

You can specify the desired shell directly using the `-s` option; otherwise you will be asked which shell you want.



With `chsh`, you may pick from all the shells mentioned in the `/etc/shells` file (provided that they are actually installed on your systems—many distributors put all shells available with the distribution into `/etc/shells`, irrespective of whether the package in question has been installed or not.)

You can find out which shell you are currently running by looking at the value of the `$0` shell variable:

```

$ echo $0
/bin/bash

```

Regardless of which shell you use interactively, you can also pick the shell(s) for your shell scripts. Several considerations influence this:

- It is of course tempting to use the interactive shell for shell scripts, too. After all, you just need to put into a file whatever you otherwise would type interactively. However, this may imply two disadvantages: Firstly, it presumes that your shell is installed wherever you want to run the shell script later on—depending on the shell in question, this is a problem or not. Secondly, some shells are useful interactively but not particularly suitable for shell scripts, and vice-versa.



The C shell has a loyal following as far as interactive use is concerned, but for shell scripts it has been deprecated due to its numerous implementation errors and syntactic inconsistencies (see [Chr96]). Although the Tenex C shell is not quite as bad, some reserve seems to be indicated even there.

- In many cases, you will do well to exercise moderation as far as the shell to be used is concerned, for example by restricting yourself in your scripts to the functionality mandated by POSIX rather than use all the (convenient) bash extensions that are available. This is, for example, important for `init` scripts—on many Linux systems, the Bourne-Again Shell is the standard shell, but there are many conceivable environments where it is left out in favour of less obese shells. A script that claims to be a “Bourne shell script” and is started as such by the system should therefore not contain “bashisms”; if you do use special bash features, you should explicitly label your script as a “bash script”. You will see how to do this in Chapter 2.

Incidentally: Often it is not the shell which restricts the portability of a shell script, but the external programs invoked by the shell. These must of course also be available on the “target system”, and behave in the same way as on your system. As long as you are only using Linux, this is not a big problem—but if you use a Linux system to develop scripts for proprietary Unix systems, you may be in for some nasty surprises: The GNU versions of the standard utilities such as `grep`, `cat`, `ls`,

portability: shell vs. programs

sed, ... not only offer more and more powerful options than the implementations you are likely to find on most proprietary Unices, but also contain fewer errors and arbitrary limits (e. g., as far as the maximum length of input lines is concerned). Moderation is the word here, too, if you are not sure that you will always be using the GNU tools.



For illustration, consider an arbitrary configure script from any free software package (configure scripts are shell scripts written for maximum portability). For your own sake, please do not do this immediately before or after a meal.


The following chapters concentrate on the Bourne-Again Shell, which does not imply that their contents would not be applicable, to a large extent, to other shells. Special bash features are, where possible, designated as such.

Exercises



1.3 [!1] Try to find out which shells are installed on your system.



1.4 [!2] Invoke—if you have it—the Tenex C shell (tcsh), enter a command with a small typo and press .—How do you return to your old shell?



1.5 [!1] Change your login shell to a different one (e. g., tcsh). Check whether it worked, for example, by logging in on a different virtual console. Reset your login shell to its original value.

1.3 The Bourne-Again Shell

1.3.1 The Essentials

The Bourne-Again Shell (or bash) was developed under the auspices of the Free Software Foundation's GNU project by Brian Fox and Chet Ramey and includes Korn shell and C shell features.



Since the Korn shell is an enhanced Bourne shell, and in a way bash represents the return of the (traditionally C-shell-using) BSD world to the Bourne concepts, the name “Bourne-Again shell”—phonetically indistinguishable from “born-again Shell”—is appropriate.

A more extensive introduction to the Bourne-Again Shell's interactive use is provided by the Linup Front training manual *Introduction to Linux for Users and Administrators*. We will reiterate only the most important points:

variables **Variables** Like most shells, the Bourne-Again Shell supports **variables** that can be set and whose values can be recalled. Variables are also used to configure many aspects of the shell's operation, e. g., the shell searches for executable programs in the directories listed in PATH, or uses the PS1 variable to output a command prompt. You will find out more about variables in Section 3.1.

Aliases The `alias` command allows you to abbreviate a longer sequence of commands. For example, by means of

```
$ alias c='cal -m; date'
$ type c
c is aliased to `cal -m; date'
```

alias you can define the new “command” `c`. Whenever you invoke “`c`”, the shell will execute the `cal` command with the `-m`, followed by the `date` command. This **alias** may also be used within other alias definitions. You may even “redefine” an existing command using an alias: The


```
$ alias rm='rm -i'
```

alias would “defang” the `rm` command. However, this is of questionable use: Once you have got used to the “safe” variant, you may count on it even in places where this alias has not been set up. Thus it is better to come up with a new name in the case of potentially dangerous commands.

By means of `alias` (without arguments) you can display all currently active aliases. The `unalias` command will delete a single alias (or all of them). `unalias`

Functions If you need more than simple textual replacement as with aliases, functions may be helpful. You will learn more about functions in Section 3.5.

The `set` Command The internal `set` command not only displays all shell variables (if invoked without a parameter), but can also set bash options. For example, with “`set -C`” (or, equivalently, “`set -o noclobber`”) you can prevent output redirection from overwriting an existing file. `set`

With “`set -x`” (or “`set -o xtrace`”) you can watch the steps that the shell is taking to reach a result:

```
$ set -x
$ echo "My $HOME is my castle"
+ echo 'My /home/tux is my castle'
My /home/tux is my castle
```

If, instead of a “-”, an option is introduced with a “+”, the option in question will be switched off.

1.3.2 Login Shells and Interactive Shells

Not all shells are created equal. Of course the Bourne-Again Shell differs from a C or Korn shell, but even one bash process’s behaviour may differ from that of another one, depending on how the shell had been started.

There are three basic forms: login shell, interactive shell, and non-interactive shell. These differ in the configuration files that they read.

Login shell You obtain this type of shell immediately after logging in to the system. The program starting the shell, i. e., `login`, “`su -`”, `ssh`, etc., passes the shell a “-” as the first character of its program name. This tells the shell that it is supposed to be a login shell. Immediately after looking in, things should look like

```
$ echo $0
-bash
$ bash
$ echo $0
bash
```

Manual bash invocation, no login

Alternatively, you can invoke `bash` using the `-l` option for it to behave like a login shell.

Every Bourne-like shell (not just the Bourne-Again Shell) executes the commands in the `/etc/profile` file first. This enables system-wide login settings for, e. g., environment variables or the `umask`. `/etc/profile`



If your installation uses the Bourne-Again Shell as well as the Bourne shell, you must make sure in `/etc/profile` to use only those instructions that are supported by the Bourne shell. Alternatively, you can check on a case-by-case basis whether the file is being processed by a Bourne shell or a bash. If it is a bash, the `BASH` environment variable will be defined.

After this, the Bourne-Again Shell looks for the `.bash_profile`, `.bash_login`, and `.profile` `.profile` files in the user's home directory. Only the *first* file found to exist will be processed.



This behaviour, too, stems from the Bourne-Again Shell's Bourne shell compatibility. If you can access your home directory from various machines, some of which support bash and others just the Bourne shell, you can put bash-specific configuration settings into the `.bash_profile` file in order not to confuse a Bourne shell, which only reads `.profile`. (You can read `.profile` from the `.bash_profile` file.)—Alternatively, you can use the `BASH` environment variable approach in the `.profile` file, as outlined above.



The `.bash_login` name derives from the C shell tradition. However, a C shell `.login` file, if it exists, will be ignored by bash.

If you quit a login shell, it will process the `.bash_logout` file in the home directory (if it exists).

Interactive Shell If you invoke the Bourne-Again Shell without file name arguments (but possibly with options) and its standard input and output channels are connected to a “terminal” (`xterm` and friends suffice), it sees itself as an interactive shell. As such, on startup it reads the `/etc/bash.bashrc` file as well as the `.bashrc` file in your home directory and executes the commands contained therein.



Whether an interactive shell reads `/etc/bash.bashrc` is, in fact, a compile-time option. The most common distributions, including the SUSE distributions and Debian GNU/Linux, enable this option.

When quitting an interactive shell that is not a login shell, no files are being processed.

1.3.3 Non-Interactive Shell

Non-interactive shells do not process any files when started or terminated. You can pass such a Bourne-Again Shell a file name to evaluate using the `BASH_ENV` environment variable, but this is normally not used.

A shell is non-interactive if it is used to execute a shell script, or if a program avails itself of a shell to run another program. This is the reason why commands like

```
$ find -exec ll {} \;
find: ll: No such file or directory
```

fail: `find` starts a non-interactive shell to execute `ll`. Even though `ll` is available on many systems, it is just an alias for “`ls -l`”. As such, it must be defined in every shell since aliases are not passed on to child processes. Non-interactive shells do not normally read configuration files containing alias definitions.

Distribution Idiosyncrasies The strict separation between login shells and “normal” interactive shells implies that you will have to set some configuration options both in `.profile` as well as `.bashrc` for them to be effective in every shell. To remove this error-prone duplication of work, many distributions have a line in the default `.profile` file reading something like

```
## ~/.profile
test -r ~/.bashrc && . ~/.bashrc
```

which results in the following: If `.bashrc` exists and is readable (`test ...`), then (`&&`) the `.bashrc` file will be processed ("`.`" is a shell command that reads the file as if its contents had been typed in at that point—see

Possibly your distribution's `bash` was compiled to read some more files as well. You can check this using `strace`; it lists all system calls posted by another command, including the `open` call to open a file. Only this lets you be sure which files the shell looks at.

Exercises



1.6 [!2] Convince yourself that your `bash` uses the `/etc/profile`, `/etc/bash.bashrc`, `~/.profile`, and `~/.bashrc` files as described above. Investigate all deviations.



1.7 [1] How does a shell process notice that it is supposed to be a login shell?



1.8 [3] How would you set up a shell as your login shell if it is not listed in `/etc/shells`?

1.3.4 Permanent Configuration Changes

Individual Customisations Individual customisations of your working environment only “keep” until the end of your shell process. If you want your changes to be re-instated when you log in again, you must take care that the shell executes them on startup. Environment variables, aliases, shell functions, the `umask`, etc. must be set in one of the files listed in Abschnitt 1.3.2—the question is just in which one?

In the case of aliases, the answer is easy. Since they are not inherited, they must be set by every shell individually. Hence you must set aliases in the `~/.bashrc` file. (For the alias to work in the login shell as well, you must enter it in `~/.profile`, *too*—unless the `~/.bashrc` file is not read from there as mentioned above.)

Other good candidates for `.bashrc` are those variables that control the shell's behaviour (`PS1`, `HISTSIZE`, etc.) but are of no further interest, i. e., not environment variables. If you want each shell to start “fresh”, you must reset these variables every time, which may be done from `.bashrc`.

Things are different with environment variables. These are generally set once, just like changes of keyboard configuration and similar settings. Therefore it suffices to define them in `.profile`. Constructions such as

```
PATH=$PATH:$HOME/bin
```

(which appends a directory to the `PATH` variable) should not go into `~/.bashrc`, since the variable's value would become bigger time and again.



If your system boots into runlevel 5, so that logins are handled by the X display manager, you do not have a proper login shell. For your settings to be taken into account nonetheless, you should put them into `~/.xsession`, or read the `.profile` file from there.

System-Wide Changes As the system administrator, you can put settings that apply to all users into the `/etc/profile` and `/etc/bash.bashrc` files. Most Linux distributors have taken precautions for these settings to survive a software update: SUSE, for example, recommends using the `/etc/profile.local` and `/etc/bash.bashrc.local`, which are read from within their respective sister files.

Another vehicle for global changes is the `/etc/skel` directory, the “skeleton” of a home directory that is copied for new users when you invoke `useradd` with the `-m` option. All files and directories contained in there become part of the default content of the new home directory.

If you put a `.bashrc` file such as

```
## System-wide settings; please do not modify:
test -r /etc/bash.local && . /etc/bash.local

## Insert individual customisations here:
```

in `/etc/skel`, you can make changes in `/etc/bash.local` that apply to all users.

Of course you can put additional pre-made configuration files for arbitrary other programs, directory hierarchies, etc. in `/etc/skel`.

Exercises



1.9 [!1] Install the `helloworld` alias for the “echo Hello world” command such that it is available within your login shell as well as all your interactive shells.



1.10 [!1] Install the `helloworld` alias for the “echo Hello world” command such that it is available within all login shells and interactive shells of all users.



1.11 [2] Ensure that you can invoke `helloworld` even from your non-interactive shells.

1.3.5 Keyboard Maps and Abbreviations

The Bourne-Again Shell uses various keyboard abbreviations to enable command line editing and access special features.

Even more extensive customisations are possible through the `.inputrc` file in your home directory as well as the `/etc/inputrc` file (for system-wide settings). This file is the configuration file for the `readline` library, which is used by the Bourne-Again Shell (among other programs). For example, the `readline` library enables searching the command line history using `Ctrl+r`.

The `readline` settings apply both to virtual terminals and the GUI. All the details are contained in `readline(3)`; we confine ourselves to some examples. If you put the lines

```
Control-t:    tab-insert
Control-e:    "\C-m"
```

into `.inputrc`, then pressing `Ctrl+t` will insert a tab character, which the shell normally does not let you use since the tab key is already spoken for. Pressing `Ctrl+e` starts a macro, in this case the characters “`cal`” followed by `Ctrl+m` (represented by “`\C-m`”), which corresponds to pressing `↵`.

Changes in this file only apply if you have set the `INPUTRC` environment variable to `$HOME/.inputrc` and start a new `bash`, or if you execute the “`bind -f ~/.inputrc`” command.

Commands in this Chapter

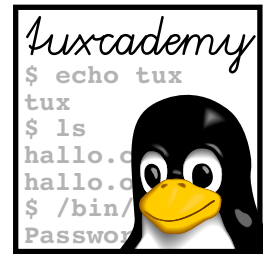
bash	The “Bourne-Again-Shell”, an interactive command interpreter	<code>bash(1)</code>	14
chsh	Changes a user’s login shell	<code>chsh(1)</code>	14
file	Guesses the type of a file’s content, according to rules	<code>file(1)</code>	14
find	Searches files matching certain given criteria	<code>find(1)</code> , <code>Info: find</code>	14
strace	Logs a process’s system calls	<code>strace(1)</code>	18

Summary

- The shell allows users to interact with the Linux system by means of text-based commands. Most shells have programming language features and allow the creation of “shell scripts”.
- A Linux system uses shell scripts in many places.
- There is a multitude of shells. The default shell on most Linux distributions is the GNU project’s “Bourne-Again Shell”, or bash.
- The Bourne-Again Shell combines features of the Bourne and Korn shells with some of the C shell.
- The Bourne-Again Shell (like most shells) behaves differently depending on whether it was started as the login shell, an interactive or a non-interactive shell.
- User-specific settings for the Bourne-Again Shell can be put in one of the `~/.bash_profile`, `~/.bash_login`, or `~/.profile` files as well as the `~/.bashrc` file.
- System-wide settings for all users can be made in the `/etc/profile` and `/etc/bash.bashrc` files.
- Customised keyboard mappings can be configured in `~/.inputrc` and `/etc/inputrc`.

Bibliography

- Chr96** Tom Christiansen. “Csh Programming Considered Harmful”, October 1996. <http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/>



2

Shell Scripts

Contents

2.1	Introduction.	24
2.2	Invoking Shell Scripts	24
2.3	Shell Script Structure	26
2.4	Planning Shell Scripts	27
2.5	Error Types	28
2.6	Error Diagnosis	29

Goals

- Knowing the purpose and basic syntax of shell scripts
- Being able to invoke shell scripts
- Understanding and specifying `#!` lines

Prerequisites

- Familiarity with the Linux command line interface
- File handling and use of a text editor
- Basic shell knowledge (e. g., from Chapter 1)

2.1 Introduction

- advantage** The unbeatable advantage of shell scripts is: If you can handle the shell, you can program! Shell scripts always come in useful for automating a task that you might as well have performed interactively “by hand”. Conversely, you can always use the shell’s “scripting” features on the command line. This not only makes testing various constructions a lot easier, but can also frequently render an actual script quite unnecessary.
- areas of use** Shell scripts are helpful in lots of places. Wherever the same commands would have to be entered over and over again, it is worth writing a shell script (for example, when repeatedly searching for and processing files that have certain properties). Shell scripts are most often used, however, to simplify complex tasks such as the automatic starting of network services. As a rule, shell scripting is less about writing sophisticated or beautiful programs, but to make one’s own work easier. Which does not mean that you should develop poor programming style—comments and documentation do not suddenly start to make sense when other people want to use and understand your scripts, but are helpful even to you if you return to your scripts after a while.
- disadvantages** However, you should not overstrain shell scripts: Wherever more complex data structures are necessary or efficiency or security are essential, “real” programming languages are usually a wiser choice. Besides the “classical” languages such as C, C++, or Java, the modern “scripting languages” like Tcl, Perl, or Python are worth considering.

2.2 Invoking Shell Scripts

A shell script can be started in different ways. Most straightforwardly, you can simply pass its name to a shell as a parameter:

```
$ cat script.sh
echo Hello World
$ bash script.sh
Hello World
```

This is quite dissatisfying, of course, since users of your script need to be aware on the one hand that it *is* a shell script (rather than an executable machine language program or a script for the Perl programming language), and on the other hand that it must be executed using `bash`. It would be nicer if the command to start your script looked like the command to start any other program. For this, you must make your script “executable” using `chmod`:

```
$ chmod u+x script.sh
```

Afterwards, you can start it directly using

```
$ ./script.sh
```



Shell script files do not just need to be executable, but also readable (as per the `r` privilege). Mere executability is sufficient for machine programs that exist as binary code.

If you want to get rid of the unaesthetic “./”, you must make sure that the shell can locate your script. For example, you might add “.”—the current working directory, whichever it is—to the command search path (the `PATH` environment variable). However, this is not a good idea for security reasons (especially not for root) as well as inconvenient, since you may well want to invoke your script from

another directory than the one containing the script file. It is best to create a directory like `$HOME/bin` to hold frequently used shell scripts, and add that directory to your `PATH` explicitly.

The third way of invoking a shell script consists of executing the script in the current shell rather than a child process. You can do this using the “source” command or its abbreviated form, “.”:

```
$ source script.sh
Hello World
$ . script.sh
Hello World
```

(Please note that, in the abbreviated form, there must be a space character after the dot). A script that was started in this way can access all of the current shell’s context. While a script started as a child process cannot, for instance, change the current directory or `umask` of your interactive shell, this is quite possible for a script started directly using `source`. (In other words: A script invoked using `source` behaves as if you were typing the script commands directly into your interactive shell.)



This method is particularly important if you want to be able to access, within a shell script, shell functions, variables or aliases defined in another shell script—such as a “library”. If that script was executed in a child process, as usual, you would not be able to profit from the definitions therein!

When naming your shell scripts, you should aim for meaningful monikers. You may also want to append a file name extension such as “.sh” or “.bash” to make it obvious that your files contain shell scripts. Or you may decide to dispense with the extensions. However, they do make sense especially when experimenting, since obvious names such as `test` or `script` are already spoken for by system commands.

naming shell scripts



As mentioned before, you should restrict the “.sh” extension to scripts that actually run with a Bourne shell (or compatible shell), i. e., that do not make use of special bash features.

Exercises



2.1 [!2] Create a text file called `myscript.sh` that might, for example, output a message using “echo”, and make this file executable. Make sure that the three invocation possibilities

```
$ bash myscript.sh
$ ./myscript.sh
$ source myscript.sh
```

work according to the description above.



2.2 [!1] What method does the login shell use to read the `/etc/profile` and `$HOME/.bash_profile` child process or “source”?



2.3 [2] A user comes to you with the following complaint: “I have written a shell script and when I start it nothing at all happens. But I’m writing a message to standard error output at the very beginning! Linux sucks!” After very close interrogation the user admits to having called his script `test`. What has happened?



2.4 [3] (Tricky.) How would you arrange for a shell script to be executed in a child process, while still being able to change the current directory of the invoking shell?

2.3 Shell Script Structure

Shell scripts are really just sequences of shell commands that have been stored in a text file. The shell can take its input either from the keyboard (standard input) or another source, such as a shell script file—there is no difference as far as executing commands is concerned. For example, line breaks serve as command separators, much like on the “real” command line.

Some readability hints: While you might, for convenience, enter something like

```
<command1> ; <command2>
```

on the command line, you should normally write one command per line in a script, for clarity’s sake:

```
<command1>
<command2>
```

There is no difference when it comes to executing the commands.

You can further increase the readability of a script by judicious use of blank lines, which are ignored by the shells—just like on the command line. The shell also ignores anything following a hash sign (“#”). This lets you comment your scripts:

```
# Command1 comes first
<command1>
<command2> # This is command2
```

Longer scripts should start with a comment block containing the name of the script, its purpose and how it works, how to invoke it, etc. The author’s name and a version history might also appear there.

Text files marked as executable using `chmod` are considered scripts for the `/bin/sh` shell—on Linux systems, this is frequently (but not always) a synonym for `bash`. To be sure, shell scripts should begin with a line starting with the “#!” character sequence followed by the name of the desired shell as an absolute path. For example:

```
$ cat script
#!/bin/bash
<<<<<<
```

This will use the specified shell to execute the script, by appending the script’s file name to the given shell name as a parameter. The bottom line in our example is that the Linux kernel executes the command

```
/bin/bash script
```



The program executing the script does not need to be a shell in the strict sense—every binary executable is eligible. That way, you could, for instance, write “awk scripts” (Refcha:grd2-awk).



On Linux, the “#!” line may be at most 127 bytes long and may also contain parameters in addition to the program name; the script name will in any case be appended to the end. Note that proprietary Unix systems often enforce much narrower limits, for example, a total length of 32 bytes and at most one option parameter. This can lead to difficulties if you want to execute a Linux shell script on a proprietary Unix system.

Exercises



2.5 [!] What output do you expect if the executable script `baz` containing the commands

```
#!/bin/echo foo bar
echo Hello World
```

is executed via the `./baz` command?



2.6 [2] What is a better choice for the first line of a shell script— `#!/bin/sh` or `#!/bin/bash`?

2.4 Planning Shell Scripts

Anyone with the least programming experience has made the saddening experience: Programs are seldom correct at the first try. The “debugging” of sizable programs takes lots of time and effort. The well-known programmer and author Brian W. Kernighan states the following:

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?

Thus, careful planning, proceeding step-by-step, and knowledge of the most common sources of errors are recommended.

Most often, you end up writing a shell script because you want to automate some particular task. In this case, the purpose of the script is fairly clear-cut, and you also know roughly which commands to execute in what order. To write the script, an evolutionary approach is often helpful. Which means: You write all the commands to a file that you would have entered on the command line, and then connect them in some sensible way. For example, you could introduce conditionals or loops, if these make the script more readable, more fault-tolerant, or more universal. Also, you should put the names of frequently-used files (such as log files) in variables and then only use the variables. Command line parameters can be inspected and used to insert elements that change between invocations; of course you should then check the completeness and plausibility of these parameters and output warnings or error messages if necessary.

You can also use the “evolutionary” approach for larger programs, by starting intuitively with the most obvious method and develop the script based on that. The big disadvantage is that it is easy to commit a conceptual error, and the ensuing changes make for a lot of unnecessary work.

Therefore, you should in every case begin with a rough outline containing all the probable steps—this makes it easier to check whether the concept contains logical errors. It is perfectly adequate to list these steps in “natural language” rather than shell code; you can worry about the actual commands and their syntax later on.

Another advantage of this planned approach is that you can decide for each single step which commands are most appropriate, for example whether you want to use simple filter commands, `awk` or even a Perl script ...

A good plan, though, does not help you much if, after a few months, you notice that something about the script needs improved, but that you no longer understand the script. If you do take the trouble of making a plan, it is best to put it in the actual script—not (just) in the shape of commands, but by means of comments. You should stop yourself from commenting every single command, preferring instead to provide a higher-level view and, in particular, concentrating on the data flow and the format of the script’s input and output: Often the required processing steps follow fairly automatically from the definition of a program’s input and

simple scripts

evolution

larger programs

outline

documentation output, while the converse is by no means as obvious. It is also never wrong to create external documentation (such as manual pages) for larger programs.

indentation A good plan has structure. There is nothing wrong with expressing this structure inside the program text, for example by using indentation. This means that commands on the same “logical” level have the same distance from the left edge of the screen (or window). You can use tabs or space characters to indent but you should try to be consistent.

Exercises



2.7 [!2] You want to create a shell script that will display the date and time of the last login of each “real” user of your system (no administrative accounts such as root or bin), as well as the disk space used by their home directories. How would you order the following steps to obtain a reasonable “plan” for a shell script?

1. Output u , t and p
2. Determine the time t of u 's last login.
3. End of the repetition.
4. Determine the amount p of disk space used by v .
5. Construct a list of all “real” users.
6. Determine the home directory v of u .
7. Repeat the following steps for each user u in the list.



2.8 [2] Design a plan for the following task: In a big installation, the home directories are distributed across various disks (imagine that the home directories are called `/home/develop/hugo` or `/home/market/susie`, for hugo from development and susie from marketing). Periodically, you want to check to what extent the disks containing the various home directories are utilised; the test script should send you e-mail if 95% or more of at least one disk's capacity is used. (We ignore the existence of LVM.)

2.5 Error Types

Fundamentally, you can distinguish two different kinds of errors:

Conceptual errors These are mistakes concerning the logical structure of the program. It can be very costly to recognise and repair these errors, and they are best avoided in the first place—by careful planning.

Syntax errors These errors occur all the time. All it takes is a simple typographical error in the program text: One character forgotten, and nothing works any longer. Many syntax errors can be avoided if you proceed from the simple to the specialised when writing the script. For example, in a parenthesised mathematical expression, you should enter *both* parentheses first, before typing their content. Then, forgotten parentheses are old hat. The same applies to control structures: Never place an `if` without also putting the `fi` on the line below. A good editor featuring “syntax highlighting” is an important tool for the early avoidance of such “structural” syntax errors.

Of course, you can still make mistakes when typing command names or options or when defining or using shell variables. (Your editor will not help you here.) This is a clear disadvantage of the shell versus “traditional” programming languages with a fixed syntax which will be painstakingly checked and, if necessary, complained about by a language compiler. Since there is no compilation and, hence, no checking of your program, you must pay particular attention to testing your script systematically, to ensure that,

if possible, all script lines, branches of conditionals, etc., are actually taken. This consideration implies that shell scripts beyond a certain “critical mass” of some hundreds or thousands of lines are no longer really manageable—you should really consider using at least a script language with better syntax checking, such as Perl or Python.

The remainder of the chapter deals mostly with a discussion of the most common syntax errors and their correction.

2.6 Error Diagnosis

As we have said, most errors only become apparent when the program is running. Therefore you should try to test your scripts as frequently as possible during development. You should avail yourself of a “testing environment”, especially if the script changes existing files. testing



For example, if you want to edit configuration files below /etc, write your script such that all references to files like /etc/... are written like \$ROOT/etc/... For testing, you can set the ROOT environment variable to an innocuous value and may even (hopefully!) be able to get by without administrator privileges. If your script is called “myscript”, you could invoke it for testing from a “scaffold” looking roughly like:

```
#!/bin/sh
# test-myscript -- Test scaffold for myscript
#
# Create the testing environment
cd $HOME/myscript-dev
rm -rf test
cp -a files test # Fresh copy of the testing files
# Call the script
ROOT=$HOME/myscript-dev/test $HOME/bin/myscript-dev $*
```

Here, the original files in ~/myscript-dev/files are copied to ~/myscript-dev/test before every run. After the test run has completed, you might compare the content of test automatically (e.g., using “diff -r”) to yet another directory containing examples of the desired output.—myscript gets passed the arguments of test-myscript; you could thus use test-myscript as a building block in an even more involved infrastructure which invokes myscript with different pre-cooked command line arguments and checks, in turn, that the program produces the desired results.

Many shell scripts invoke external commands. In these cases, you can make use of these commands’ built-in error messages to figure out any errors—especially as far as syntax is concerned.

Should these error messages not prove adequate, you can make bash much more talkative. Frequently syntax errors involve the shell’s syntax, especially if substitutions are performed in an unanticipated order, or when parenthesis mismatches occur.

With “set -x” you can see the steps that the shell takes to do its job:

```
$ set -x
$ echo "My $HOME is my castle"
+ echo 'My /home/tux is my castle'
My /home/tux is my castle
```

This is also called “tracing”.

The disadvantage of “set -x” is that the command is still executed. In the case of substitutions, it may be better if the commands would just be displayed instead

tracing

of executed. The “-n” option does exactly this, but works for shell scripts only, not for interactive shells (why not?).

Another useful option is “-v”. This executes the commands, and the shell also displays all commands as they are executed. That is, for a shell script you will obtain not just its output but also everything contained in it.

All three options can be switched off by putting a “+” instead of a “-” in the set command.

In practical shell programming, a different approach is often helpful. When developing a script, put the desired option on the first line of the script:

```
#!/bin/bash -x  
<<<<<<
```

Another rule for error diagnosis is: Before executing a command involving “doubtful” substitutions, put an echo in front of it. This causes the whole command (including all substitutions and expansions) to be output without it being executed. This is quite adequate for a quick test.

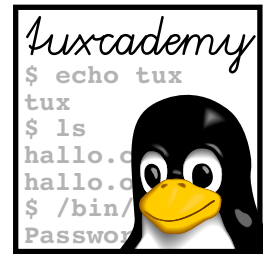
Commands in this Chapter

chmod Sets access modes for files and directories

chmod(1) 24

Summary

- Shell scripts offer a simple means to automate command sequences.
- To execute a shell script, you can pass its name to a shell as a parameter, make the script file executable and start it directly, or read its text into the current shell using `source`.
- Shell scripts are text files containing sequences of shell commands.
- The first line of an executable script can name a program (shell or otherwise) that is to be used to execute the script.
- Careful planning when programming leads to less brain-racking later.
- The Bourne-Again Shell contains various features for error diagnosis.



3

The Shell as a Programming Language

Contents

3.1	Variables	32
3.2	Arithmetic Expressions.	38
3.3	Command Execution	38
3.4	Control Structures	39
3.4.1	Overview	39
3.4.2	A Program's Return Value as a Control Parameter	40
3.4.3	Conditionals and Multi-Way Branches	42
3.4.4	Loops	46
3.4.5	Loop Interruption	49
3.5	Shell Functions.	51
3.5.1	The exec Command	52

Goals

- Knowing the shell's programming language features (variables, control structures, functions)
- Being able to create simple shell scripts using these features

Prerequisites

- Familiarity with the Linux command line interface
- File handling and use of a text editor
- Basic shell knowledge (e.g., from Chapter 1 iflabelcha:grd2-skript and Chapter 2.)

3.1 Variables

Basics The shell uses variables in various capacities: On the one hand, they are used at shell level for programming, on the other hand, certain variables govern various aspects of the shell's behaviour and (as **environment variables**) that of its child processes.

Variables in the shell are pairs consisting of a name and a (textual) value. A variable's name consists of a sequence of letters, digits, and underscores (“_”), and must start with a letter or underscore. For all practical purposes, bash variable names may be of arbitrary length. A variable's value is an arbitrary string of characters of, again, practically arbitrary length.



Modern shells such as bash also support numeric variables and “arrays”. We shall return to this later.

Unlike other programming languages, the shell does not require variables to be declared before use; you may simply assign a value to a variable and it springs into existence at that point if it had not been used before:

```
$ colour=blue
$ a123=?_/@xz
```

Ensure that there are no spaces around the equals sign “=”! The name, “=”, and value must follow each other with no intervening spaces.

Variable Substitution When processing commands, the shell replaces variable references with the value of the variable in question. You can refer to a variable by putting a “\$” in front of its name. (The shell considers the longest possible sequence of letters, digits, and underscores following the “\$” the variable name.)

variables>substitution
echo

This is also called **variable substitution**.

For instance, you can display variable values using echo:

```
$ echo $PAGER
less
```

shows you the value of `PAGER`, a variable governing which program the `man` command (among others) should use to display textual output.



Note that the shell takes care of expanding the variable reference `$PAGER`—the actual command executed is just “echo less”.

set



The internal bash command `set`, invoked without options or arguments, displays all currently defined variables (and shell functions—see below).

Variables and Quotes Variable substitution is helpful, but not always desirable. You can inhibit variable substitution in two ways:

1. You put a backslash in front of the dollar sign:

```
$ echo \$colour has the value $colour
$colour has the value blue
```

2. You put single quotes around the entire variable reference:

```
$ echo '$colour' has the value $colour
$colour has the value blue
```


Variable substitution takes place inside double quotes ("...") and backticks (`...`):

Double quotes are important when handling variables whose value may contain white space such as space characters (i. e., frequently to always). Consider the following example:

```
$ mkdir "My photographs"
$ d="My photographs"
$ cd $d
bash: cd: My: Not a directory
```

In the third line, the value of the `d` variable is substituted before the command line is split into “words”. Accordingly, `cd` tries to change to the “`My`” directory (instead of “`My photographs`”). *Moral:* Put double quotes wherever you can—“`cd "$d"`” would have been correct.

Environment Variables “Normal” variables are only visible within the shell in which they have been defined. For variables to be visible in child processes (“subshells” or other programs started from the shell), they need to be “exported” to the process environment:

process environment

```
$ colour=blue
$ sh                                     Start a subshell
$ echo $colour                           Inside the subshell
                                         The variable is not defined inside the subshell
$ exit                                    Return to the original shell
$ export colour                           Export the variable to the environment
$ sh                                     Another subshell
$ echo $colour                             Inside the subshell
blue
```

You can list several variables in a single `export` command:

```
$ export blue white red
```

You can also lump the `export` and initial assignment together (in `bash`):

```
$ export form=oval smell=musty
```

“`export -n`” revokes an `export`:

```
$ export -n blue white
```

Whenever the shell starts a child process it is passed a copy of the shell’s current environment. After that, the two environments are totally independent of each other—changes to one have no effect on the other. In particular, it is impossible to change the parent process’s environment (like the current directory) directly from the child.



If at all, this works only using tricks similar to Exercise 2.4—one program employing this method is `ssh-agent`.

You can also set environment variables for a single program invocation without influencing eponymous shell variables:

```
$ TZ=foo
$ TZ=Europe/Istanbul date                Turkish time
Mon May 10 19:23:38 EEST 2004
$ echo $TZ
foo
```

You can obtain a current list of all environment variables by means of the `export` (without arguments) or `env` commands. Some important environment variables include

PATH List of directories searched by the shell for executable programs
HOME Home directory; for the shell, the directory by which `~` is substituted
TERM Current terminal type (important for full-screen text-oriented programs)
USER Current user name
UID Current user ID; cannot be changed!

Assignment: Tricks Of The Trade When assigning values to variables, more complicated expressions are possible, such as

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.
$ PATH=$PATH:$HOME/bin
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:./home/tux/bin
```

Here the program search list in `PATH` is reconstructed from the `PATH` variable's old value, a `“:”`, and the value of `HOME` followed by `“/bin”`.

Using backticks, you can assign a program's standard output to a variable, like

```
$ dir=`pwd`
$ echo $dir
/home/tux
```

Of course this is just a consequence of the fact that the shell replaces ``...`` expressions *anywhere* on the command line by the corresponding command's standard output. Incidentally, the “command” in question may be a complete pipeline, not merely a simple command.



The same result can be reached using `“$(...)”` (`“dir=$(pwd)”`). This construction works with modern shells such as `bash` and makes it easier to nest such calls—but makes your script `bash`-dependent, too.

escaping special characters If you want to store spaces or other special characters in a variable, you must protect them from the shell using single quotes (`'...'`), as in

```
$ PS1='% '
% _
```

This changes the `PS1` variable, which governs the appearance of your shell prompt.

unset If you are fed up with a variable, you can delete its *content* by assigning it the empty string. More drastically, the `unset` command removes a variable completely and obliterates all traces of your action:

```
$ A='Murder in the Cathedral'
$ set | grep A=
A='Murder in the Cathedral'
$ A=''
$ set | grep A=
A=
$ unset A
$ set | grep A=
$ _
```

Special Shell Variables The Bourne-Again Shell supports some special shell variables, which are mostly of interest within shell scripts. These special variables cannot be modified, only read. For example:¹

- \$? Return value (exit code) of the last command
- \$\$ Process ID (PID) of the current shell
- #! PID of the last background process to be started
- \$0 Name of the currently executing shell script (or the shell itself if running interactively)
- \$# Number of parameters passed on the shell script's command line
- \$* All of the shell script's parameters; "\$*" is equivalent to "\$1 \$2 \$3 ..."
- @\$ All of the shell script's parameters; "\$@" is equivalent to "\$1" "\$2" "\$3" ..."
- \$n The shell script's *n*-th parameter. For *n* > 9, you must write "\$*n*". Alternatively, you can use the shift command (see help shift).

The positional parameter access variables, in particular, are used frequently within shell scripts. Here is a small example to make this clearer:

```
$ cat script
#!/bin/sh
echo "The shell script was invoked as $0"
echo "$# parameters were passed altogether"
echo "All parameters together: $*"
echo "The first parameter is $1"
echo "The second parameter is $2"
echo "The third parameter is $3"
$ ./script Eat my shorts
The shell script was invoked as "./script"
3 parameters were passed altogether
All parameters together: "Eat my shorts"
The first parameter is "Eat"
The second parameter is "my"
The third parameter is "shorts"
```

Especially when referring to positional parameter variables, it is important to put variable references in quotes, since as a programmer you have no way of knowing what parameter the invoking user will pass. Stray space characters can really mess up script execution.

Special Forms of Variable Substitution When using "\$name" or "\${name}", a variable reference is simply substituted by the variable's value. The shell can do a lot more, though:

Assign a default value With \${name:=*default value*}, the name variable is assigned the *default value* if it has no value yet or its value is the empty string. Afterwards, the variable's (then-current) value is substituted into the current command line.

\$ unset colour	<i>variable has no value</i>
\$ echo Favourite colour: \${colour:=yellow}	
Favourite colour: yellow	<i>default value applies</i>
\$ echo \$colour	

¹Here and elsewhere, we will talk about "the \$? variable", even though this is not entirely accurate, since "\$?" is really the "?" variable's value. Since the special variables can only be read and not written (without special [sic] tricks, anyway), this inaccuracy is unproblematic.

```

yellow                                     was assigned on previous command
$ colour=red                               variable gets a value
$ echo Favourite colour: ${colour:=yellow}
Favourite colour: red                       existing value has priority

```

You can also omit the colon (“`${colour=yellow}`”). In this case, the assignment takes place only if the name variable has no value; an existing but empty value is left unchanged.

Use default value The expression `${name:-<default value>}` is replaced by the value of name if name’s value is different from the empty string. Otherwise it is replaced by the *<default value>*. This differs from `:=` in that the actual value of name remains unchanged:

```

$ unset colour
$ echo Favourite colour: ${colour:-yellow}
Favourite colour: yellow
$ echo $colour

```

Output: empty string

The colon may be omitted here, too.

Error message if no value Using `${name:?<message>}`, you can check whether the name variable has a non-empty value. If this is not the case—in particular if the variable has no value at all—the *<message>* is output, and if this occurs in a shell script, its execution terminates at that point.

```

$ cat script
#!/bin/sh
echo "${1:?Oops, something missing}"
echo "And on we go"
$ ./script foo
foo
And on we go
$ ./script
./script: line 2: 1: Oops, something missing

```

The colon may be omitted here as well—the message will then only appear if the variable really has no value at all.

Substrings An expression of the form `${name:p:l}` is replaced by up to *l* characters of the name variable’s value, starting from position *p*. If *l* is omitted, everything starting from *p* is returned. The first character of the value is deemed to be at position 0:

```

$ abc=ABCDEFGH
$ echo ${abc:3:2}
DE
$ echo ${abc:3}
DEFG

```

In fact, *p* and *l* are evaluated as *arithmetic expressions* (Section 3.2). *p* may be negative to refer to positions starting from the end:

```

$ echo ${abc:2*2-1:5-2}
DEF
$ echo ${abc:0-2:2}
FG

```

The `$*` and `$@` variables are treated as special cases: You will obtain *l* positional parameters of the script, starting with parameter *p*, where (confusingly, but somehow logically) the first parameter is considered to be at position 1:

```
$ set foo bar baz quux
$ echo ${*:2:2}
bar baz
```

Removing variable text from the beginning In `${name#<pattern>}`, `<pattern>` is considered a search pattern (containing “*”, “?”, and so on as in file name search patterns). The expression is substituted by the value of the `name` variable, but with everything matching `<pattern>` having been removed from its beginning:

```
$ starter="EEL SOUP"
$ echo ${starter}
EEL SOUP
$ echo ${starter#E}
EL SOUP
$ echo ${starter#E*L}
SOUP
```

If there are several possibilities, “#” tries to remove as little text as possible. If you use “##” instead, as much text as possible will be removed:

```
$ oldmacd=EIEIO
$ echo ${oldmacd#E*I}
EIO
$ echo ${oldmacd##E*I}
0
```

A typical application is the removal of the directory part from a file name:

```
$ file=/var/log/apache/access.log
$ echo ${file##*/}
access.log
```



You might of course use the `basename` command to accomplish this, but that forces an external program invocation – the “##” expression is potentially much more efficient.

Removing variable text from the end Expressions of the form `${name%<pattern>}` and `${name%%<pattern>}` work similarly, except that they apply to the end of `name`’s value rather than the beginning:

```
$ msg=00LALA
$ echo ${msg%L*A}
00LA
$ echo ${msg%%L*A}
00
```

The Bourne-Again Shell offers several additional substitution expressions which you should look up in the `bash` manual.

Text processing tricks such as these are very useful, but for more sophisticated operations you will usually have to resort to tools like `cut`, `sed` (Chapter 6), and `awk` (Chapter 7). Naturally this involves the inefficiency of another child process, which is no big deal in the individual case; for heavy-duty text processing you are generally better off using a programming language like Perl, Python, or Tcl that offers more elaborate operations in a more efficient package.

Exercises



3.1 [!2] What exactly is the difference between `$*` and `$@`? (Hint: Read up on this in the bash manual, and compare the output of the

```
$ set a "b c" d
$ for i in $*; do echo $i; done
$ for i in $@; do echo $i; done
$ for i in "$*"; do echo $i; done
$ for i in "$@"; do echo $i; done
```

commands. See Abschnitt 3.4.4 for information about `for`.)

3.2 Arithmetic Expressions

The Bourne-Again Shell supports certain mathematical features, even though these should not be overstrained: For example, it can only handle integers and does not check for overflow. The mathematical operations correspond to those of the C programming language. You can thus avail yourself of the four basic arithmetic operations as well as the usual comparison and logical operators (see the bash documentation for details). Multiplication and division have precedence over addition and subtraction, and explicit parentheses are always evaluated first.

Arithmetic expansion

“Arithmetic expansion” is applied to expressions delimited by `$(())`. Allowable operands include numbers as well as shell variables. The expression is treated as if it was contained in double quotes, thus you can also use command expansion and the “special forms of variable substitution” discussed in the previous section.

```
$ echo $((1+2*3))
7
$ echo $(((1+2)*3))
9
$ a=123
$ echo $((3+4*a))
495
```

You can refer to shell variables within arithmetic expressions without having to put a “`$`” in front of their names.



You may occasionally run into the obsolete notation “`$(...)`”. This has been deprecated and will be removed from future bash versions.

Exercises



3.2 [!1] How does bash handle division? Check the result of various division expressions using positive and negative dividends and divisors. Can you come up with some rules?



3.3 [2] (When reviewing.) What is the largest number usable in bash arithmetic? How can you find out?

3.3 Command Execution

When executing commands, the shell follows a fixed order of steps:

1. The command line is split into **words** (see also below). Every character contained in the `IFS` variable which occurs outside of quotes is considered a separator. The default value of `IFS` includes the space and tab characters and the newline character.

2. Braces are expanded; “a{b,c}d” becomes “abd acd”. All other characters remain unchanged.
3. Next, a tilde (~) at the beginning of a word is replaced by the value of the HOME environment variable; if there is a user name immediately after the tilde, this (including the tilde) will be replaced by the name of that user’s home directory. (There are a few other rules which are described in the bash documentation.)



Tilde expansion also takes place within variable assignments if a tilde occurs immediately after a “=” or “:”. This means that the Right Thing happens even for PATH and friends.

4. Afterwards, the following substitutions are performed in parallel proceeding from left to right:
 - Variable substitution
 - Command substitution
 - Arithmetic expansion

(in other words, everything that starts with a \$—if we consider the modern form of command substitution using “\$(...)”).

5. If a substitution took place during the previous steps, the command line is again split into words according to the IFS variable. “Explicit” empty words (“” and ‘’) remain unchanged, “implicit” empty words, which may, for instance, have resulted from the expansion of variables with no value, are removed.
6. At the end of this process, words containing wild card characters such as “*” or “?” are considered as search patterns; the shell tries to replace them by lists of matching file names. If this is not possible, the search pattern is (usually) passed on verbatim.
7. At the very end, all non-escaped quotes and backslashes are removed (they are no longer required because all substitutions have been performed, and the division into words can no longer change).

search patterns

The only processing steps that can change the number of words on the command line are brace expansion, word division, and search pattern expansion (or “pathname expansion”). All other processing steps replace a single word by a single word, with the exception of “\$@”.

3.4 Control Structures

3.4.1 Overview

Every programming language that is to be taken seriously (i. e., every Turing-complete programming language) needs control structures such as conditionals, loops, and multi-way branches², and thus they may not be absent from the shell, either. As usual in shell programming, everything seems a bit “roundabout” and possibly confusing to connoisseurs of “real” programming languages, but somehow it manages to follow its own perverse logic.



Stephen L. Bourne, the author of the original Bourne shell, was a big aficionado of the Algol programming language [WMPK69], and that is quite discernible in the Bourne shell’s (and its successors’) syntax. The concept of closing control structures by means of the beginning keyword spelled backwards—not quite followed through to the end in the Bourne shell—, for example, was quite *en vogue* in Algol circles.

²One can make do with just the while loop, at least if one’s name is Edsger Dijkstra, but a bit of variety does not hurt here.


Value	Description
0	Success
1	General error
2	Misuse of builtin shell functions (rarely used)
126	Command wasn't executable (no permission, or not a binary)
127	Command to be executed wasn't found
128	Invalid argument on exit – as in “exit 1.5”
129–165	Program terminated by signal $\langle Value \rangle - 128$


Table 3.1: Reserved return values for bash


3.4.2 A Program's Return Value as a Control Parameter


Here is the first peculiarity of shell control structures: Many programming languages use Boolean values (“true” or “false”) to govern conditionals or loops. Not so in bash—here the **return value** of a program is used for control.


On Linux, every process tells its parent process upon termination whether it was executed “successfully” or not. The return value in bash is an integer between 0 and 255; the value 0 always implies success, every other value (up to and including 255) implies failure. This makes it possible for a process to give more detail as to what went wrong.

 *How it actually does this is not specified. With grep, for example, a return value of 0 means “matching lines were found”, 1 stands for “no matching lines were found, but everything else was fine”, and 2 for “some error occurred”.*

 One could now get into an extended ontological discussion about whether “no matching lines were found” is, in fact, an error situation or really some type of success. As a matter of fact, it is useful to be able to distinguish the cases “lines were found” and “no lines were found” by means of grep's return value. It is also undisputed that Unix knows just one kind of success, namely a return value of 0. Whoever disagrees with this ought to look for a different operating system (such as VMS, which considers every even return value (including 0) a success and every odd one a failure).

 C programmers, who are used to 0 implying “wrong” or “no”, and “everything else” implying “true”, “yes”, or “success”, must rethink things here.

 bash, at least, uses certain conventions for exit codes (see Table 3.1). The `exit()` system call does accept values up to and including 255 (larger values will be passed on “modulo 255”), but bash uses exit codes from 128 to denote that the child process was terminated by a signal. (You can find out which signal by subtracting 128 from the exit code; to find out about signal numbers, use, e. g., `>>kill -l<<`.)

 You can, of course, generate exit codes greater than or equal to 128 in your scripts, but that is generally not a great idea, since it could lead to confusion when your script is called from another script that interprets these exit codes as “process death by signal”.

The shell makes the last command's return value available in the `$?` special variable:

```
$ ls /root
ls: /root: Permission denied
$ echo $?
1
```

Failure of ls


```
$ echo $?
0
```

Success of the first echo

The return value has nothing whatever to do with error messages that appear on the standard error output channel.



A little-known trick is that you may put an exclamation point (!) in front of a command. That command's return value is then "logically negated"—success becomes failure, failure success:

```
$ true; echo $?
0
$ ! true; echo $?
1
$ ! false; echo $?
0
```

However, this will lose information: As we mentioned before, bash knows 255 kinds of failure but just one kind of success.



C programmers and those using C-like languages such as Perl, Tcl, or PHP will remember the logical NOT operator "!".

With the Bourne-Again Shell, you can of course use logical expressions to control conditionals or loops just like you would with other programming languages. You just need to invoke a command that will evaluate a logical expression and return a suitable return value to the shell. One such command is `test`.

`test` is used to compare numbers or strings and to check file properties. Here are some examples:

test "\$x" With just one argument, `test` checks whether this argument is *non-empty*, i. e., consists of one or more characters—here, whether the `x` variable contains "something". Even if you see it again and again: Stop yourself from using the (ostensibly shorter) "`test $x`" (without quotes)—it does not work, as you can easily see for yourself using something like "`x='3 -gt 7'; test $x`".

test \$x -gt 7 Here `x` must contain a number. `test` checks whether the *numerical* value of `x` is greater than 7. `-gt` stands for "greater than"; there are also the corresponding operators `-lt` (less than), `-ge` (greater than or equal to), `-le`, `-eq` (equal) und `-ne` (unequal).

test "\$x" \> 10 Checks whether the first character string would occur after the second *in a dictionary* (the so-called "lexicographic ordering"). Thus "`test 7 \> 10`" returns success, "`test 7 -gt 10`" failure. Mind the notation: "`>`" is a shell special character; to keep it from being interpreted by the shell you must hide it. Instead of "`\>`" you might also write "'>'".

test -r "\$x" Checks whether the file whose name is contained in `x` exists and is readable. There are various other file test operators.

You can review the complete list of operators supported by `test` by looking at the `test` documentation.



Like `echo`, `test` is not just available as a program, but is also, for efficiency, implemented as an internal command by the Bourne-Again Shell (the traditional Bourne shell does not do that). Using "`help test`" you can look at the internal command's documentation, using "`man test`" that of the external command.



Beside the "long form" discussed above, `test` also supports an abbreviated notation where the expression to be evaluated is enclosed in brackets (with spaces before and after the brackets). Thus the long example "`test "$x"`" becomes "`["$x"]`" in the abbreviated form.


3.4.3 Conditionals and Multi-Way Branches

You can use `if` and `case` to implement conditionals. `if` is mostly used for simple conditionals and chains of conditionals, while `case` enables a multi-way branch according to a single value.

Conditional Execution The shell offers convenient abbreviations for the common cases “Do *A* only if *B* worked” or “Do *A* only if *B* did *not* work”. A very typical idiom in shell scripts is something like

```
test -d "$HOME/.mydir" || mkdir "$HOME/.mydir"
```


You will find this, for example, in scripts that want to use a user-specific directory to store intermediate results or configuration files. The command sequence ensures that the `$HOME/.mydir` directory exists by first checking whether the directory is already there. If the `test` command reports success, the `mkdir` is skipped. If `test` fails, however—the directory does not exist—the directory is created using `mkdir`. Thus the `||` operator executes the following command only if the preceding command has reported failure.

 C programmers know `||` as the “logical OR” operator, whose result is “true” if its left operand *or* that to its right (or both of them) are “true”. One of the more ingenious properties of the C language is that the language definition guarantees that the left-hand operand is looked at first. If this turns out “true”, the final result is already determined, and the right-hand operand is ignored. The right-hand operand is only considered if the left-hand operand returned “false”.—The shell’s `||` operator works basically similar: In “*a* `||` *b*”, we want to execute the *a* command *or* the *b* command successfully. If *a* already returns success, we are where we want to be and can ignore *b*; *b* only gets its turn when *a* could not be executed successfully.

By analogy, the `&&` operator executes the following command only if the preceding command has reported *success*. Another real-world example for this: You will find something like

```
test -e /etc/default/myprog && source /etc/default/myprog
```

within the init scripts of many Linux distributions. This construction checks whether the `/etc/default/myprog` file exists, which (presumably) contains configuration settings to be used later in the script (in the form of shell variable assignments). If this file exists, it is read using `source`, otherwise nothing happens.

 A similar analogy to the C language’s `&&` operator (logical AND) may be drawn here.

The `||` and `&&` abbreviations are very useful indeed and also behave as expected in combination (try something like

```
true && echo Wahr || echo Falsch
false && echo Wahr || echo Falsch
```

if you want). You should not overwork them, however—often explicit `if` constructions, as explained forthwith, are more readable.

if Simple Conditionals The `if` command executes a command (often `test`) and decides what to do next according to its return value. Its syntax is

```

if <testing command>
then
  <commands for "success">
[else commands for "failure"]
fi

```

If the testing command was successful, the commands following then are executed. If it was not, the commands following else are executed (if they are there). In both cases, execution continues after the fi.

This is best made clear using an example. By way of demonstration, we write the `ifdemo` program, which you can invoke using your user name as the first positional parameter. (Your “real” user name was helpfully deposited in the `LOGNAME` environment variable by the `login` program.) If the user name was entered correctly, a message like “That is correct” is displayed. If another value was entered, another message is returned:

```

#!/bin/bash
if test "$1" = "$LOGNAME"
then
  echo "That is in fact your user name!"
else
  echo "That is not your user name!"
fi
echo "End of program"

```

Of course you are not forced to use `test` as the testing command—any program obeying the return value convention is eligible. `egrep`, for example:

```

#!/bin/bash
if df | egrep '(9[0-9]%|100%)' > /dev/null 2>&1
then
  echo "A file system is overflowing" | mail -s "warning" root
fi

```

This “quick and dirty” script checks whether a mounted file system is 90% full (or more). If so, `root` is sent mail to make him aware of the fact.

In order to avoid deeply nested constructions like

```

if foo
then
  ...
else
  if bar
  then
    ...
  else
    if baz
    then
      ...
    else
      ...
    fi
  fi
fi

```

it is possible to “cascade” dependent conditionals using `elif`. An equivalent to the preceding example might be

```

if foo
then
    ...
elif bar
then
    ...
elif baz
then
    ...
else
    ...
fi

```

The else branch remains optional in any case. The conditions checked using if and the elifs do not have to have anything to do with one another, but conditions in “later” elifs are, of course, only looked at if the preceding conditions have returned values different from 0.

case **Multi-Way Branches** Unlike if, the case command allows simple multi-way branches. Its syntax is

```

case <value> in
    <pattern1>
        ...
        ;;
    <pattern2>
        ...
        ;;
    *)
        ...
        ;;
esac

```

case compares the <value> (which can derive from a variable, a program invocation, ...) to the specified patterns in turn. For the first matching pattern, the corresponding command sequence (up to the ;;) is executed. After this, the case is closed, and any further matches are not considered. The case command’s return value is that of the last command of the sequence that was executed; if no pattern matches, a null return value is assumed.

In case patterns, you may use search patterns as for file name expansion (“*”, “?”, ...). Thus you can insert “*” last as a “catch-all” pattern, for example, to output an error message. You may also specify alternatives like

```
[Yy]es|[Jj]a|[Oo]ui)
```

(in order to recognise the words “Yes”, “yes”, “Ja”, “ja”, “Oui”, or “oui”).

case is best demonstrated using a common example—an init script. As a reminder, init scripts are responsible for the starting and stopping of background services. As a rule, they work like

```
<init script> start
```

or

```
<init script> stop
```

```
#!/bin/sh


SERVICE=/usr/sbin/tcpdump
SERVICEOPTS="-w"
DUMPFILER="/tmp/tcpdump.`date +%Y-%m-%d_%H:%M`"
INTERFACE=eth0
PIDFILE=/var/run/tcpdump.pid


case $1 in
  start)
    echo "Starting $SERVICE"
    nohup "$SERVICE" "$SERVICEOPTS" "$DUMPFILER" > /dev/null 2>&1 &
    echo "$!" > "$PIDFILE"
    ;;
  stop)
    echo "Stopping $SERVICE"
    kill -15 `cat "$PIDFILE`
    rm -f "$PIDFILE"
    ;;
  status)
    if [ -f $PIDFILE ]
    then
      if ps `cat $PIDFILE` > /dev/null 2>&1
      then echo "Service $SERVICE running"
      fi
    else
      echo "Service $SERVICE NOT running"
    fi
    ;;
  clean)
    echo "Which dump files would you like to remove?"
    rm -i $DUMPFILER.`date +%Y`**
    ;;
  *)
    echo "Error: Please use one of (start|stop|status|clean)!"
    ;;
esac
```


Figure 3.1: A simple init script

where “start”, “stop”, “status”, ... are passed as the first positional parameter—a natural application for case. Figure 3.1 shows a simple init script which starts the `tcpdump` program (a packet sniffer) in order to capture all received packets in a file (for later analysis). The init scripts included with Linux distributions are usually a good deal more complicated!

Exercises

 **3.4** [!1] Give a shell script which checks whether its (only) positional parameter consists exclusively of vowels, and returns a suitable return value (0 should stand for “yes”).


 **3.5** [!1] Give a shell script which checks whether its (only) positional parameter is an absolute or relative path name, and outputs “absolute” or “relative”, respectively.

 **3.6** [2] How would you, with the least possible effort, add a “restart” action to the init script shown in Figure 3.1 which would stop the service and immediately start it again?

3.4.4 Loops

It is often convenient to be able to execute a sequence of commands several times over, especially if the number of repetitions is not known when you write your script, but depends on the conditions when the script is run. The shell supports two different approaches to looping:

- | | |
|--------------|--|
| Iteration | <ul style="list-style-type: none"> • Iteration over a predefined list of values, such as all positional parameters or all file names in a directory. The number of repetitions is determined when the loop first begins. |
| testing loop | <ul style="list-style-type: none"> • A testing loop which executes a command at the beginning of each repetition. The return value of that command determines whether the loop is repeated or not. |

 A precooked “counting loop” of the form “Start at 1 and increment the loop counter by 1 after each turn, until it reaches the value 10”, as found in many programming languages, is not part of the shell; an equivalent effect is easily obtained, however, using one of the available loop types.

for **Iteration using for** The `for` command is used to iterate over a predetermined list of values. A variable assumes the value of each list item in turn:

```
for <variable> [in <list>]
do
    <commands>
done
```

Here is a small example:

```
#!/bin/bash
# Name: fordemo
for i in one Two THREE
do
    echo $i
done
```

Its output looks like

```
$ fordemo
one
Two
THREE
```

The list that `for` iterates over does not need to be given literally within the script, but can be determined when the script executes, e. g., using file name expansion:

```
#!/bin/bash
for f in *
do
    mv "$f" "$f.txt"
done
```

The `f` variable iterates over all file names within the current directory. This re-names all files. Note that the list is constructed exactly once, namely when the shell encounters the `for` command. The fact that the file names within the current directory change during the execution of the loop is of no concern to the list being iterated over.

If you omit the *<list>* completely, the loop goes over the script's positional parameters, i. e., the loop variable assumes the values of `$1`, `$2`, ... in turn. This makes a mere "for `i`" equivalent to "for `i` in `"$@"`".



A variant of the `for` loop is based on the C programming language: In

```
for (( i=0 ; $i<10; i=i+1 ))
do
    echo $i
done
```

the variable `i` assumes the values 0, 1, ..., 9 in turn. Strictly speaking, the first arithmetic expression serves to initialise the variable, the second is checked at the start of every iteration, and the third is executed at the end of every iteration before control jumps back to the beginning of the loop (and the second expression is executed again as the test). Hence, this does not constitute a truly iterative loop like the list-based `for` presented earlier, but a "testing loop" like those in the next section – this `for` is a close relative of `while`.

Testing loops using `while` and `until` The `while` and `until` commands are used for loops whose number of iterations depends on the actual loop execution (and cannot be determined at the start of the loop as in `for`). A command is specified whose return value determines whether the loop body (and, again, the testing command) is executed once more, or whether the loop is to be terminated:

```
while <testing command>
do
    <commands>
done
```

The following example outputs the integers from 1 to 5:

```
#!/bin/bash
# Name: whiledemo
i=1
while test $i -le 5
do
    echo $i
```

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

The `$(...)` construction calculates the numerical value of the expression between the parentheses, so that this increments the value of the `i` variable by 1.

With `while`, too, you are not obliged to use `test` as the testing command, which is convenient for the following example:

```
#!/bin/bash
# Name: readline
while read LINE
do
    echo "--$LINE--"
done < /etc/passwd
```

Here `read` serves as the testing command. On each invocation, it reads a single line from its standard input and assigns it to the `LINE` variable. If `cmd` cannot read anything, for example when the input file is exhausted, its return value is different from 0. Thus the `while` loop runs until all of the input file has been consumed. Since the loop's standard input has been redirected here, the `/etc/passwd` file is read and processed line by line.



At the risk of producing a candidate for the “useless use of `cat` award”, we consider the

```
cat /etc/passwd | while read LINE
do
    ...
done
```

more readable. Note in any case that loops and conditionals do have standard input and standard output channels, and thus can occur in the middle of a pipeline.

`until` behaves like `while`, except that with `until` the loop is repeated while the testing command reports “failure”, i. e., returns a return value that is different from 0. Alternatively, the counting example might be written as

```
#!/bin/bash
# Name: untildemo
i=1
until test $i -gt 5
do
    echo $i
    i=$((i+1))
done
```

Exercises



3.7 [!1] Write a shell script that outputs all multiples of 3 up to a maximum value given as a positional parameter.



3.8 [3] Write a shell script that outputs all prime numbers up to a given upper limit. (*Hint*: Use the “modulo operator”, `%`, to check whether a number is divisible by another without a remainder.)

3.4.5 Loop Interruption

Every so often it turns out to be necessary to terminate a loop before its time, e. g., if an error occurs. Or it becomes evident that an iteration does not need to be finished, but that the next one can start immediately. The Bourne-Again Shell supports this by means of the `break` and `continue` commands.

Aborting loops using `break` The `break` command aborts the current loop iteration and arranges for execution to continue after the corresponding `done`. Consider the following script:

```
#!/bin/bash
# Name: breakdemo
for f
do
    [ -f $f ] || break
    echo $f
done
```

If you invoke this script with a number of arguments, it checks for each argument whether a file of the same name exists. If that is not the case for an argument, the loop is terminated immediately:

```
$ touch a c
$ ./breakdemo a b c
a
$ _
```



With `break`, you can “break out of” nested loops as well: State the number of “inner loops” that you want to terminate as an argument.—Try the following script:

```
#!/bin/bash
# Name: breakdemo2
for i in a b c
do
    for j in p q r
    do
        for k in x y z
        do
            break $1
        done
        echo After the inner loop
    done
    echo After the middle loop
done
echo After the outer loop
```

This lets you specify on the command line how many loops (seen from the inside out) are to be aborted.

Terminating loop iterations using `continue` The `continue` command does not abort the complete loop, but just the current iteration. Afterwards the next iteration is either started immediately (for `for`), or the testing command is evaluated to check whether another iteration is required (for `while` and `until`).

The following script shows a somewhat convoluted method of copying files only if they contain particular character sequences:

```

pattern=$1
shift
for f
do
    fgrep -q $pattern $f
    if [ $? = 1 ]
    then
        continue
    fi
    cp $f $HOME/backups
done

```

(See also Exercise 3.9.)

As with `break`, you can use a numerical argument to `continue` to determine the loop (counting from the inside out) whose next iteration is to be started.

Exercises



3.9 [!2] How would you reasonably simplify the `continue` example script?



3.10 [!1] Consider the `breakdemo2` script on page 49 and modify it such that you can use the new program to try how the `continue` command behaves when different numeric arguments are given.

Exception Handling In bash scripts, you can react to incoming signals or other unusual events. This is done using the `trap` command, which you might invoke like

```
trap "rm /tmp/script.$$" TERM
```

If you execute this command within a script, the `"rm /tmp/script.$$"` command will be stored. If the shell process is sent a `SIGTERM`, the stored command is executed. In this example, a temporary file created by the script would be removed—a typical way of “cleaning up”.

Consider the `traptest` script:

```
#!/bin/sh
trap "echo Received signal" TERM HUP
sleep 60
```

We can execute this script in the background and then send the process, e. g., a `SIGTERM`:

```

$ sh traptest &
[2] 11086
$ kill -TERM %2
Received signal

[2]+ Exit 143      sh traptest

```

The `SIGSTOP` and `SIGKILL` signals, of course, cannot be trapped. The exact behaviour of the Bourne-Again Shell with respect to signals is explained in the “Signals” section of the shell’s documentation.



The return value of a process tells you whether it was terminated by a signal. If so, the return value is 128 plus the signal number, e. g., 15 for `SIGTERM`.

With `trap`, you can react not only to (external) signals, but also to different events. Commands registered for the `EXIT` event, for example, will be executed when the process terminates no matter why (because of a signal, because of an exit, or just because the end of the script was reached). With the `ERR` event you can execute a command if a simple shell command returns a value different from 0 (this does not work if the command is part of a `while` or `until` loop, an `if` command, or a command sequence using `&&` or `||`, or if it is invoked with `!`).

Exercises



3.11 [1] Check that the `trap` command works as advertised for events such as `SIGTERM` or `EXIT`.



3.12 [2] Write a shell script that displays a digital clock inside a text terminal. When the user aborts the script using `Ctrl+C`, the screen should be cleared and the program terminated. (This is most fun if your system includes the `SysV` banner program.)

3.5 Shell Functions

Frequently used command sequences can, in principle, be implemented as “sub shell scripts” that you can invoke from a shell script. Modern shells like the Bourne-Again Shell also allow the definition of “shell functions” within the same script:

```
#!/bin/bash

function sort-num-rev () {
    sort -n -r
}

ls -l | sort-num-rev
```

From the point of view of the invoking code, shell functions behave like “normal” commands—they have standard input and output channels, can take command-line arguments and so on.



The function command may be omitted; however we do recommend to include it since it makes clear what is going on. The parentheses and braces are required.


Within a shell function, the positional parameters `$1`, `$2`, ... correspond to the shell function’s arguments, not those of the actual shell process. Accordingly, `$#` reflects the number of positional parameters of the shell function, `$*` returns all positional parameters at once and so on (after the shell function is finished everything is back to what it was before.) Other than that, you can access the shell’s and environment variables of the complete process even from a shell function. Here is an example:

```
#!/bin/bash

function panic () {
    exitcode=$1
    shift
    echo >&2 "$0: PANIC: $*"
    exit $exitcode
}
```


```
<<<<<
[ -f file.txt ] || panic 3 file.txt is not available
<<<<<
```

Here the first positional parameter serves as the process's return value, the remaining parameters as the error message.

 Even in shell functions, \$0 is the name of the shell script, not that of the function. If you want to know the name of the function: This is available in the FUNCNAME variable.

- return value The return value of a shell function is the return value of the last command executed by it.
- finding shell functions You can inspect the names and definitions of the functions currently defined within your shell using the "typeset -f" command. "typeset -F" gives you just the function names.
- function libraries It is easy to construct "function libraries" by putting the desired shell functions into a file which is then read into other shell scripts by means of the "source" command.

Exercises


 **3.13** [!1] Define a shell function toupper which translates its positional parameters to upper case and writes them to standard output.

3.5.1 The exec Command


Usually, the shell waits for an external command to finish and then reads the next command. Using the exec command you can launch an external command such that it *replaces* the shell. For example, if you'd rather use the C shell instead of bash, you can use

```
$ exec /bin/csh
% _ Here is the C shell!
```

to launch a C shell in your session without having an unused bash lying around that you must remember to exit from when you are logging out.

 exec is mostly used in shell scripts and even there not too frequently. There are more convenient messages for the C-shell-instead-of-bash deal.

Exercises

 **3.14** [!2] Assume the test1 file consists of the lines

```
echo Hello
exec bash test2
echo Goodbye
```

and the test2 file of the line

```
echo Howdy
```

What does the command "bash test1" output?

Commands in this Chapter

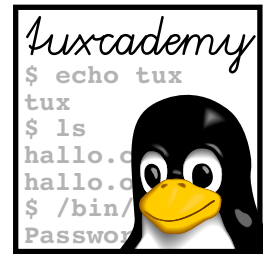
case	Shell command for pattern-based multi-way branching	bash(1)	44
env	Outputs the process environment, or starts programs with an adjusted environment	env(1)	33
exec	Starts a new program in the current shell process	bash(1)	52
export	Defines and manages environment variables	bash(1)	33
for	Shell command to loop over the elements of a list	bash(1)	46
set	Manages shell variables and options	bash(1)	32
test	Evaluates logical expressions on the command line	test(1), bash(1)	41
unset	Deletes shell or environment variables	bash(1)	34
until	Shell “=Kommando for a loop that executes “until” a condition evaluates as true	bash(1)	48
while	Shell command for a loop that executes “while” a condition evaluates to true	bash(1)	47

Summary

- Variables serve to store intermediate results, to control the shell and (as environment variables) to communicate with child processes.
- The shell defines various special variables, for example to access the shell’s positional parameters.
- There are several special types of variable substitution that insert default values or process the variable’s value in some way.
- When processing command lines, the shell follows a sequence of predefined substitution steps.
- The Bourne-Again Shell supports the usual control structures expected in programming languages.
- The shell uses the return value of subprocesses to govern control structures; a return value of 0 is considered “true”, everything else is considered “false”.
- Conditionals can be implemented using the && and || operators as well as the if and case commands.
- Loops can be defined using while, until, and for, and controlled using break and continue.
- Using trap, scripts can react to signals and other events.
- Shell functions make it possible to collect frequently-used command sequences within the same script.

Bibliography

- WMPK69** A. van Wijngarden, B. J. Mailloux, J. E. L. Peck, et al. “Report on the Algorithmic Language ALGOL 68”. *Numerische Mathematik*, 1969. **14**:79–218. This article is a classic—the first attempt to define the semantics of a programming language in a formalized way. The language itself, sadly, never gained practical importance.



4

Practical Shell Scripts

Contents

4.1	Shell Programming in Practice	56
4.2	Around the User Database	56
4.3	File Operations.	60
4.4	Log Files	62
4.5	System Administration	68

Goals

- Analyzing and understanding some shell script examples
- Knowing and using basic techniques of practical shell programming

Prerequisites

- Familiarity with the Linux command line interface
- File handling and use of a text editor
- Basic shell knowledge (e. g., from Chapter 3 and the preceding chapters)

4.1 Shell Programming in Practice

The preceding chapters have introduced large parts of the shell’s syntax. If you are anything like us, you will have started wondering what this is all about: Do we explain to you the complete content of the kitchen and larder and then expect you to cook a *Cordon Bleu* five-course menu? No sweat. Programming is not learnt from a syntax description, but by studying exemplary programs and, most of all, by experimenting. Therefore, in this chapter, we have collected some shell scripts which will teach you many common techniques and deepen your understanding of some things you have seen already. Above all, these scripts should inspire you to “get your hands dirty” and get to know the shell’s features in practice.

4.2 Around the User Database

“Normal” Linux systems store user information—user names and UIDs, primary groups, real names, home directories, and so on—in the `/etc/passwd` file.¹ Here is an example to remind you:

```
tux:x:1000:100:Tux the Penguin:/home/tux:/bin/bash
```

User `tux` has the UID 1000 and his primary group is the one with GID 100. In real life, his name is “Tux the Penguin”, his home directory is `/home/tux`, and his login shell is the Bourne-Again Shell.

The group file, `/etc/group`, contains the name, an optional password, the GID, and a list of members for each group. The membership list usually contains just those users that use the group as a supplementary group:

```
users:x:100:
penguins:x:101:tux
```

Who is in this group? Our first script will take a group name and enumerate all users who use that group as their primary group. The challenge is that `/etc/passwd` contains just the GID of the primary group, so that we need to fetch the GID for a named group from `/etc/group` first. Our script takes the group in question as a command-line parameter.

```
#!/bin/bash
# pgroup -- first version

# Fetch the GID from /etc/group
gid=$(grep "$I:" /etc/group | cut -d: -f3)

# Search users with that GID in /etc/passwd
grep "^[^:]*:[^:]*:[^:]*:$gid:" /etc/passwd | cut -d: -f1
```

Note the use of `grep` to find the correct line, and of `cut` to extract the appropriate field from that line. In the second `grep`, the search expression is more complicated, but we must watch out not to select a UID that looks like the desired GID—hence we take care to count off three colons from the left before starting to look for the GID.

The same principle applies to shell scripts as to programs in general: Most of the effort goes into catching user errors (and others). For example, our script worries neither about invocation problems—omitting the group name or specifying

¹Among not quite normal Linux systems, methods like LDAP for user data storage are spreading. If you have such a system in front of you, you can generally obtain the files necessary for the following experiments by means of commands like “`getent passwd >$HOME/passwd`” and “`getent group >$HOME/group`”.

further, extraneous parameters—nor about execution problems. It is fairly unlikely for `/etc/passwd` not to be available (you would have noticed before that), but it is quite possible for a user of the script to give the name of a group that does not in fact exist. But let us start at the beginning.

First we ought to convince ourselves that the script was invoked using the correct number of parameters (namely one). You can check this, for example, by inspecting the value of `$#` : syntax checking

```
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi
```

This snippet of shell code illustrates several important techniques at once. If the number of parameters (in `$#`) is not 1, we want to terminate the script with an error message. The message is output using `echo` , taking care that, nicely enough, it does not appear on standard output but on standard error output (the `>&2` —2 is the standard error output channel). `$0` is the name that the script was invoked with; it is customary to state this in the error message, and this way it is always correct, even if the script has been renamed. The script is terminated prematurely using `exit` , and we use 1 as the return value, meaning “generic failure”. error message



If you invoke `exit` without an argument or simply reach the end of a shell script, the shell terminates as well. In this case the shell’s return value is that of the last command that was executed (`exit` does not count). Compare

```
$ sh -c "true; exit"; echo $?
0
$ sh -c "false; exit"; echo $?
1
```

Now we need to deal with the case of the non-existing group. In Section 3.4.2 you have learned how `grep` defines its return values: 0 means “some match was found”, 1 stands for “everything basically OK, but no matching lines found”, and 2 for “something bad has happened” (possibly the regular expression wasn’t quite kosher, or something went wrong while reading the input file). This is quite useful already; we might check whether `grep` ’s return value is 1 or 2 and then terminate the script with an error message if necessary. Unfortunately there is a little problem with the critical command

```
gid=$(grep "$1:" /etc/group | cut -d: -f3)
```

—a pipeline’s return value is the return value of the *last* command, and `cut` basically works all the time, even if `grep` sends it empty input (the `cut` arguments are fundamentally all right). Thus we must think of something else. pipeline’s return value

So what happens when `grep` does not find a matching line? Right, `cut` ’s output is empty, as opposed to the case where `grep` could in fact find the group (if we assume a syntactically correct `/etc/group` file, then the matching line has a third field containing a GID). Therefore we just need to check whether our `gid` variable contains an “actual” value:

```
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi
```

(Note again `$0` as part of the error message.)

Figure 4.1 shows the “preliminary final” version of our script.

```
#!/bin/bash
# pgroup -- improved version

# Check the parameters
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi

# Fetch the GID from /etc/group
gid=$(grep "$1:" /etc/group | cut -d: -f3)
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi

# Search users with that GID in /etc/passwd
grep "[!]*:[!]*:[!]*:$gid:" /etc/passwd | cut -d: -f1
```

Figure 4.1: Which users have a particular primary group? (Improved version)

Which are a user’s groups? Our next example is a script that outputs the groups that a user is a member of—similar to the `groups` command. Note that it is not sufficient to consider `/etc/group`, since users are not normally listed in the entry of their primary group in that file. We will be using the following approach:

1. Output the name of the user’s primary group
2. Output the names of the user’s supplementary groups

The first part should be easy—it is basically the previous script “the other way round”:

```
# Primary group
gid=$(grep "$1:" /etc/passwd | cut -d: -f4)
grep "[!]*:[!]*:$gid:" /etc/group | cut -d: -f1
```

The second part seems even easier: A simple

```
grep $1 /etc/group
```

gets us near Nirwana already. Or does it not? Consider what this `grep` might turn up:

- Firstly, the user name within a group’s list of members. This is what we want.
- Additionally, user names within the member list that contain the user name in question as a substring. When searching for `john`, we also get all lines belonging to groups that user `johnboy` is a member of, but `john` not. Wrong already.
- The same problem applies to user names which are substrings of group names. A group called `staff` does not have anything to do with a user called `taf`, but matches nonetheless.

- Quite absurd, but possible: The user name in question might even be a substring of an encrypted group password that is not stored in `/etc/gshadow` but in `/etc/group` (which is absolutely permissible).

Thus care is called for here, too, as always when `grep` is involved—when designing regular expressions, you should get used to thinking as evil-mindedly and negatively as you possibly can. Then you come up with something like

```
grep "[!]*:[!]*:[!]*:*.*\<$1\>" /etc/group | cut -d: -f1
```

That is, we match the user name only within the fourth field of `/etc/group`. The “word brackets”, `\<...\>` (a GNU `grep` speciality) help prevent the `johnboy` error. All in all, this gets us to

```
#!/bin/bash
# lsgroups -- first version

# Primary group
gid=$(grep "$1:" /etc/passwd | cut -d: -f4)
grep "[!]*:[!]*:$gid:" /etc/group | cut -d: -f1

# Supplementary groups
grep "[!]*:[!]*:[!]*:*.*\<$1\>" /etc/group | cut -d: -f1
```

Let’s try this script on a Debian GNU/Linux system:

```
$ ./lsgroups tux
tux
dialout
fax
voice
cdrom
floppy
<<<<<<
src
tux
scanner
```

Two things ought to occur to us. For one, the list is unsorted, which is not nice; for the other, the `tux` group occurs twice in the list. (Debian GNU/Linux is one of those distributions that, by default, put each user into their own eponymous group.) The latter derives from the fact that `/etc/group` contains a line of the form

```
tux:x:123:tux
```

—unusual, but quite legal.

Thus we should sort the output and remove duplicates in the process (“`sort -u`”). The question remains: How? An explicit “`lsgroups | sort -u`” gives us the correct solution, but is inconvenient; sorting should be part of the script. There again, the two logically separate pipelines are a nuisance. One way of dealing with this would be by using an intermediate file:

```
grep ... >/tmp/lsgroups.$$
grep ... >>/tmp/lsgroups.$$
sort -u /tmp/lsgroups.$$
```

(the `$$` will be replaced by the shell’s PID and makes the intermediate file’s name unique). This approach is unsavoury because it tends to leave junk files around if, due to an error, the intermediate file is not removed at the end of the script (there

```
#!/bin/bash
# lsgroups -- final version

# Primary group
( gid=$(grep "$1:" /etc/passwd | cut -d: -f4)
  grep "[:]*:[:]*:$gid:" /etc/group | cut -d: -f1

# Supplementary groups
grep "[:]*:[:]*:[:]*.*\<$1\>" /etc/group \
  | cut -d: -f1 ) | sort -u
```

Figure 4.2: In which groups is user *x*?

are ways to prevent this). Besides, creating intermediate files with simple names such as these represents a possible security hole. It is much more convenient to execute both pipelines in an implicit common sub-shell and pipe that sub-shell's output to sort:

```
( grep ...
  grep ... ) | sort -u
```

Thus we arrive at our final version (Figure 4.2).

Exercises



4.1 [1] Change the `pgroup` script such that it distinguishes the error situations “input syntax error” and “group does not exist” by returning different return values.

4.3 File Operations

Automating file operations is a profitable application of shell scripts—moving, renaming, and saving files depending on various criteria is often more complex than can be expressed using simple commands. Therefore, shell scripts are a convenient way for you to define your own commands that do exactly what you need.

Renaming multiple files The `mv` command is useful to rename a file or to move several files to another directory. What it cannot do is rename several files at the same time, the way you may remember from MS-DOS:

```
C:\> REN *.TXT *.BAK
```

This works because, on DOS, the `REN` command itself is dealing with the file name search patterns—on Linux, on the other hand, it is the shell's job to deal with the search patterns, and it does not know what `mv` is about to do with the names afterwards.

It is possible to solve the general case of multiple renaming by means of a shell script, but we will consider a restricted problem, namely changing the “extension” of a file name. More precisely, we want to design a shell script called `chext`, which by

```
$ chext .bak *.txt
```

renames all specified files such that they get the extension given as the first parameter. In our example, all files whose names end in “.txt” would be renamed to end in “.bak” instead.

Obviously, the main task of the `chext` script is to construct the appropriate arguments for `mv`. We must be able to remove a file name extension and add another one—and you have already learned how to do this using bash: Remember the `${...%...}` construction for variable substitution and consider something like

```
$ f=./a/b/c.txt; echo ${f%.*}
./a/b/c
```

Everything starting from the last dot is removed.

And this leads to the first attempt at our `chext` script:

```
#!/bin/bash
# chext -- Change a file extension, first version

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.$suffix"
done
```

Note first the use of double quotes to avoid problems with whitespace inside file names. It is also interesting how the command line is used: The first argument—immediately following the script name—is the desired new extension. We put this into the `suffix` variable and then invoke the `shift` command. `shift` causes all positional parameters to “take a step to the left”: `$2` becomes `$1`, `$3` becomes `$2`, and so on. The old `$1` is discarded. Thus, after our `shift` the command line consists only of the file names to be changed (which the shell will kindly have compiled for us from file name search patterns if necessary), so that it is convenient to use “for `f`” to iterate over them.

The `mv` command might look a bit daunting, but it is really not too hard to understand. `f` contains one of our file names to be changed, and using

```
${f%.*}.$suffix
```

the old extension is removed and the new one (which is stored in the `suffix` shell variable) textually appended.



The whole thing is not quite safe, as you will note when you consider file names like `./a.b/c`, where the last dot is not part of the last file name component. There are various ways of solving this problem. One of them involves the “stream editor”, `sed`, which you will learn about in Chapter 6, and another uses the `basename` and `dirname` commands, which are useful in many different contexts as well. They are used to split a file name into a directory and a file component:

```
$ dirname ./a/b.c/d.txt
./a/b.c
$ basename ./a/b.c/d.txt
d.txt
```

Thus you can “defang” the shell’s `%` operator by presenting to it just the file part of the name to be changed. The `mv` command then becomes something like

```
#!/bin/bash
# chext -- Change file extension, improved version

if [ $# -lt 2 ]
then
    echo >&2 "usage: $0 SUFFIX NAME ..."
    exit 1
fi

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.$suffix"
done
```

Figure 4.3: Mass file name extension changing

```
d=$(dirname "$f")
b=$(basename "$f")
mv "$f" "$d/${b%.*}.$suffix"
```

(the `${...%...}` construction, sadly, allows just variable names and no command substitutions).

In addition, a decent shell script requires a command line syntax check. Our script needs at least two parameters—the new extension and a file name—and there is no limit to the number of file names to be passed (well almost). Figure 4.3 shows the final version.

Exercises



4.2 [!2] Write a shell script that takes a file name and produces as output the names of its superior directories, as in

```
$ hierarchy /a/b/c/d.e
/a/b/c/d.e
/a/b/c
/a/b
/a
/
```



4.3 [2] Use the script from the previous exercise to write a shell script that behaves like `mkdir -p`—the script is passed the name of a directory to be created, and should create this directory along with any possibly non-existing directories farther up the directory tree.

4.4 Log Files

Controlling log file sizes A running Linux system produces various log data that can, for example, be written to files by means of the Syslog daemon. Typical log files can grow quickly and, with time, attain considerable size. One typical system

administration task, therefore, is controlling the size of, and possibly the truncating and restarting of log files.—These days, most Linux distributions use a standardised tool called `logrotate`.

Next we shall develop a shell script called `checklog`, which checks whether a log file has reached or exceeded a certain size, and possibly renames it and creates a new log file under the old name. A basic skeleton might be something like

```
#!/bin/bash
# checklog -- Check a log file and renew it if necessary

if [ $# -ne 2 ]
then
    echo >&2 "usage: $0 FILE SIZE"
    exit 1
fi

if [ $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 )) ]
then
    mv "$1" "$1.old"
    > "$1"
fi
```

The interesting line in this script is the one containing the expression

```
$(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
```

This determines the length of the file passed as a parameter (fifth field of the output of “`ls -l`”) and compares that to the maximum length that was also passed as a parameter. The length parameter is interpreted as a number of kibibytes.

If the file is as long as, or longer than, the maximum file length, it is renamed and a new file created under the old name. In real life, this is only half the job: A program like `syslogd` opens the log file once and then continues writing into it, no matter what the file is called—our script may rename it but that by no means implies that `syslogd` will begin writing to the new file. It must be sent a `SIGHUP` first. One way of implementing this is via an (optional) third parameter:

```
<<<<<<
> "$1"
[ -n "$3" ] && killall -HUP "$3"
<<<<<<
```

Our script might then be invoked like

```
checklog /var/log/messages 1000 syslogd
```

Handling several log files at once Our script from the preceding section may be quite nice, but in real life you will have to deal with more than one log file. Of course you could invoke `checklog` *n* times with different arguments, but would it not be possible for the program to handle several files at once? Ideally, we would use a configuration file describing the work to be done, which might look like configuration file

```
SERVICES="apache syslogd"
FILES_apache="/var/log/apache/access.log /var/log/apache/error.log"
FILES_syslogd="/var/log/messages /var/log/mail.log"
MAXSIZE=100
MAXSIZE_syslogd=500
NOTIFY_apache="apachectl graceful"
```

In plain language: The program is supposed to take care of the “services” `apache` and `syslogd`. For each of these services there is a configuration variable beginning with “`FILES_`” which contains a list of the log files of interest, and optionally another one beginning with “`MAXSIZE_`” that contains the desired maximum size (the `MAXSIZE` variable specifies a default value for services without their own `MAXSIZE_` variable). Another optional variable is the “`NOTIFY_`” variable giving a command with which the service can be told about a new log file (otherwise, “`killall -HUP <service>`” will be executed by default).

This nearly fixes the operation of our new script—let us call it `multichecklog`:

1. Read the configuration file
2. For each service in the configuration file (`SERVICES`):
3. Determine the desired maximum size (`MAXSIZE`, `MAXSIZE_*`)
4. Check each log file against the maximum size
5. Notify the service if appropriate

The beginning of `multichecklog` might look like:

```
#!/bin/bash
# multichecklog -- Check several log files

conffile=/etc/multichecklog.conf
[ -e $conffile ] || exit 1
. $conffile
```

read configuration file We check whether our configuration file—here, `/etc/multichecklog.conf`—exists; if not, there is nothing for us to do. If it does exist, we read it as a shell script and thus define the `SERVICES` variable etc. *within the current shell* (the configuration file syntax was deviously specified just so that was possible).

Then we consider the individual services:

```
for s in $SERVICES
do
  maxsizevar=MAXSIZE_$s
  maxsize=${!maxsizevar:-${MAXSIZE:-100}}
  filesvar=FILES_$s
  for f in ${!filesvar}
  do
    checklonger "$f" "$maxsize" && rotate "$f"
  done
done
```

If you have followed closely, you have surely noticed the somewhat convoluted

```
maxsizevar=MAXSIZE_$s
maxsize=${!maxsizevar:-${MAXSIZE:-100}}
```

indirect substitution construction. We want to determine the value of `maxsize` as follows: First we would like to check whether “`MAXSIZE_<service>`” exists and is non-empty; if so, that variable’s value will be assumed. If not, we check whether `MAXSIZE` exists; if so, the value of that variable is assumed, otherwise (arbitrarily) 100. The problem with this is the actual name of “`MAXSIZE_<service>`”, which can only be determined within the loop body. For this we use another property of variable substitution that we have not yet explained: In a variable reference of the form “`${!<name>}`”, the *value* of `<name>` is interpreted as the name of the variable whose value will eventually be substituted, like


```
$ north>Hello
$ south=Howdy
$ brit="How do you do"
$ area=south
$ echo ${!area} world
Howdy world
```

The *<name>* still needs to be a valid variable name—in our script, we would like to say something like

```
maxsize=${!MAXSIZE_$s: -${MAXSIZE:-100}}
```

but that is not permitted, hence the extra indirection using `maxsizevar`. (The same trick is necessary for “FILES_*<service>*”.)

Note further that we did not spell out the size check and renaming within the loop body. To make our script more readable, we shall implement these two actions as shell functions: readability

```
# checklonger FILE SIZE
function checklonger () {
    test $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
}

# rotate FILE
function rotate () {
    mv "$1" "$1.old"
    > "$1"
}
```


Finally, we need the service notification (we have omitted it from the first version of our loop). We must notify the service just once, no matter how many of its log files we have “rotated”. A neat method is the following:

```
notify=0
for f in ${!filesvar}
do
    checklonger "$f" "$maxsize" && rotate "$f" \
        && notify=1
done
notifyvar=NOTIFY_$s
[ $notify -eq 1 ] && ${!notifyvar:-killall -HUP $s}
```

This, again, makes use of the indirect substitution trick. The `notify` variable has the value 1 exactly if a log file needed to be rotated.

All in all, our script now looks like the one in Figure 4.4.—The “configuration file” technique shown in this shell script is very common. Linux distributions like to use them, the SUSE distributions, for example, for the `/etc/sysconfig` files, and Debian GNU/Linux for the files in `/etc/default`. Essentially, these files may contain whatever is supported by the shell; you would, however, do best to restrict yourself to variable assignments which you might want to explain using appropriate comment lines (one of the main advantages of the approach).

Exercises

 **4.4 [1]** Change the `multichecklog` script such that the name of the configuration file can optionally also be given by means of the `MULTICHECKLOG_CONF` environment variable. Convince yourself that your change performs as specified.

```
#!/bin/bash
# multichecklog -- Checking several log files


conffile=/etc/multichecklog.conf
[ -e $conffile ] || exit 1
. $conffile


# checklonger FILE SIZE
function checklonger () {
    test $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
}


# rotate FILE
function rotate () {
    mv "$1" "$1.old"
    > "$1"
}

for s in $SERVICES
do
    maxsizevar=MAXSIZE_$s
    maxsize=${!maxsizevar:-${MAXSIZE:-100}}
    filesvar=FILES_$s
    notify=0
    for f in ${!filesvar}
    do
        checklonger "$f" "$maxsize" && rotate "$f" && notify=1
    done
    notifyvar=NOTIFY_$s
    [ $notify -eq 1 ] && ${!notifyvar:-killall -HUP $s}
done
```

Figure 4.4: Watching multiple log files

 **4.5** [2] How would you make it possible to specify the maximum length of log files conveniently like “12345” (bytes), “12345k” (kilobytes), “12345M” (megabytes)? (*Hint*: case)

 **4.6** [3] So far, the rotate function renames the current log file `$f` to `$f.old`. Define an alternate rotate function which will support, e. g., 10 old versions of the file as follows: When rotating, `$f` is renamed to `$f.0`, a possibly existing file `$f.0` is renamed to `$f.1`, and so on; a possibly existing file `$f.9` is deleted. (*Hint*: The `seq` command creates sequences of numbers.) If you want to be especially thorough, make the number of old file versions configurable.

 **4.7** [2] For many log files, the owner, group, and access mode are important. Extend the rotate function such that the newly created empty log file has the same owner, group, and access mode as the old one.

Important events Every so often, messages are written to the log which you as the system administrator would like to hear about immediately. Of course you have more important things to do than constantly observing the system logs—so what would be more obvious than getting a shell script to do it? Of course it is unwise to simply search `/var/log/messages` periodically using `grep`, since you may well be alerted about the same event several times over. It would be better to make use of a special property of the Linux `syslogd` implementation, namely that it will write to a “named pipe” if you put a vertical bar (pipe symbol) in front of its name:

```
# syslog.conf
<<<<<<
*.*;mail.none;news.none |/tmp/logwatch
```

The named pipe must naturally have been created beforehand using the `mkfifo` command.


A simple log file reader script might look like

```
#!/bin/bash


fifo=/tmp/logwatch
[ -p $fifo ] || ( rm -f $fifo; mkfifo -m 600 $fifo )

grep --line-buffered ALERT $fifo | while read LINE
do
    echo "$LINE" | mail -s ALERT root
done
```

We create the named pipe first, if it does not exist. Then the script waits for a line containing “ALERT” to appear in the stream of log messages. Such a line will be sent to the system administrator.

 Instead of e-mail, you might want to send the message using SMS, beeper, ..., depending on how urgent it is.

Essential for the functioning of the script is the `--line-buffered` extension of GNU `grep`. It causes `grep` to write its output line by line instead of buffering larger amounts of output, as it usually does to make writing more efficient. A line might otherwise take ages until the read eventually gets to see it.

 If the `expect` package by Don Libes is installed on your system, you have a program called `unbuffer`, which “unbuffers” the output of arbitrary programs. You might then write something like

```
unbuffer grep ALERT $fifo | while read LINE
```

even if `grep` did not support the `--line-buffered` option.

Why do we not simply use something like

```
grep --line-buffered ALERT $fifo | mail -s ALERT root
```

? Obviously: We do want the notification to take place as quickly as possible. With a simple pipeline, `mail` would wait for `grep` to finish its output, to be sure that it has received everything worth sending along before actually dispatching the message. The clumsier “while-read-echo” construction serves to isolate the messages.

Incidentally, instead of tediously thinking of a regular expression that will cover all of your interesting log entries, you might want to use the `-f` option to `grep`. With this, `grep` will read a number of regular expressions from a file (one per line) and search for all of them simultaneously:


```
grep --line-buffered -f /etc/logwatch.conf $fifo | ...
```

takes the search expressions from the `/etc/logwatch.conf` file. With a trick, you can include the search patterns in the `logwatch` file itself:

```
grep <<ENDE --line-buffered -f - $fifo | while read LINE
ALERT
WARNING
DISASTER
ENDE
do
    echo ...
done
```

Here the regular expressions are part of a here document which is made available to `grep` on its standard input; the special file name “-” causes the `-f` option to read the expression list from standard input.

Exercises

 **4.8** [2] What other possibility is there to have `grep` search for multiple regular expressions at the same time?

4.5 System Administration

Shell scripts are an important system administration tool—they make it possible to automate tediously repeating procedures, or to make seldom-used tasks available conveniently so you do not need to think them up again and again. It is also possible to add features that did not come with the system.

df on afterburner The `df` command determines how much space is available on the file system(s). Unfortunately, at first sight its output is fairly cryptic; it would often be nice to be able to visualise the percentage of used space as a “bar graph”. Nothing easier than that: Here is the output of `df` on a typical system:

```
$ df
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/hda3            3842408    3293172    354048   91% /
tmpfs                193308         0    193308    0% /dev/shm
/dev/hda6            10886900    7968868    2364996   78% /home
/dev/hdc              714808     714808         0 100% /cdrom
```

“Graphically” post-processed this might look like

```
$ gdf
Mounted      Use%
/             91% #####-----
/dev/shm     0% -----
/home       78% #####-----
/cdrom     100% #####
```

We must grab the 5th and 6th fields of the `df` output and insert them as the 1st and 2nd fields of our own `gdf` script’s output. The catch is that `cut`, when cutting out fields, cannot use the space character as a field separator: Two adjacent spaces lead to an empty field according to `cut`’s count (try “`df | cut -d ' ' -f5,6`”). Instead of labouriously counting the characters per line in order to be able to use `cut`’s columnar cutting mode, we take the easy way out and replace every sequence of space characters with a tab character using `tr`. Without its graphical display, our `gdf` script looks like

```
#!/bin/bash
# gdf -- "Graphical" df output (preliminary version)

df | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
done
```

There are just two new features: The `read` command reads the first field cut out by `cut` into the `pct` variable, and the second into the `fs` variable (you will hear more about `read` in Section 5.2). And the `printf` command supports the output of character strings and numbers according to a “format specification”—here “`%-12s %4s`”, or “a flush-left character string in a field that is exactly 12 characters wide (possibly truncated or padded with spaces) followed by a space and a flush-right character string in a field that is exactly 4 characters wide (ditto)”.



`printf` is available as an external program, but is also included in `bash` itself (in a slightly extended version). Documentation is available either as a manual page (`printf(1)`), an info document, or as part of the `bash` manual; you will, however, have to look up the details about the available formats in the documentation of the C library function, `printf()` (in `printf(3)`). The GNU programmers seem to assume that `printf` is ingrained so deeply within the collective subconscious of Unix users that it does no longer need to be explained at length ...

To solve the exercise, we are just missing the graphical bars, which of course derive from the percentage of used space according to field 5 (a. k. a. `pct`). For simplicity, we cut the bars themselves from predefined strings of the desired length; we just need to take care that we do not choke on the title line (“`Use%`”). After the `printf`, there must be something like

```
if [ "$pct" != "Use%" ]
then
    usedc=$(( ${#hash} * {pct%\%} / 100 )
    echo "${hash:0:$usedc}${dash:$usedc}"
else
    echo ""
fi
```

Here `hash` is a long string of “#” characters, and `dash` is an equally long string of dashes. `usedc` derives from the length of `hash`—available via the special expansion

```
#!/bin/bash
# gdf -- "Graphical" output of df (final version)

hash="#####"
dash="-----"

df "$@" | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
    if [ "$pct" != "Use%" ]
    then
        usedc=$(( ${#hash} * ${pct%}/100 )
        echo "${hash:0:$usedc}${dash:$usedc}"
    else
        echo ""
    fi
done
```

Figure 4.5: df with bar graphs for disk use

`hash`—multiplied by the use percentage divided by 100, thus gives the number of “#” characters to be displayed as part of the bar. We obtain the bar itself by outputting `usedc` characters from `hash` and appending just enough characters from `dash` to make the bar as long as `hash`. The whole script is shown in Figure 4.5; `hash` and `dash` are 60 characters each, which at a default terminal width of 80 characters, makes good use of the space available if output together with the left-hand format.

Exercises



4.9 [1] Why does Figure 4.5 contain “`df "$@"`”?



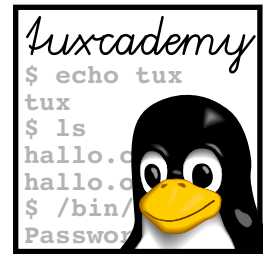
4.10 [3] Write a version of `gdf` which lets the length of each bar depend on the size of the file system. The largest file system should take up all of the original width, while other file systems should use proportionally shorter bars.

Commands in this Chapter

logrotate	Manages, truncates and “rotates” log files	<code>logrotate(8)</code>	62
mkfifo	Creates FIFOs (named pipes)	<code>mkfifo(1)</code>	67
printf	Formatted output of numbers and strings	<code>printf(1)</code> , <code>bash(1)</code>	69
seq	Writes number sequences to standard output	<code>seq(1)</code>	67
tr	Substitutes or deletes characters on its standard input	<code>tr(1)</code>	69
unbuffer	Suppresses a process’s output buffering (part of the <code>expect</code> package)	<code>unbuffer(1)</code>	67

Summary

- `grep` and `cut` are useful to extract particular lines and columns from files.
- You should handle error cases carefully—when your script is invoked as well as when it is running.
- I/O redirection for command sequence is possible by means of explicit subshells.
- The `dirname` and `basename` commands allow file name manipulations.
- Files with shell variable assignments can serve as convenient “configuration files” for shell scripts.
- Log messages can be processed individually by reading them from a named pipe using `read`.
- The `printf` command implements formatted output of textual or numeric data.



5

Interactive Shell Scripts

Contents

5.1	Introduction.	74
5.2	The read Command	74
5.3	Menus with select	76
5.4	“Graphical” Interfaces Using dialog	80

Goals

- Learning techniques for shell script control
- Knowing the Bourne-Again Shell’s methods for user interaction
- Being able to use the dialog package

Prerequisites

- Knowledge of shell programming (from the previous chapters)

5.1 Introduction

In the previous chapters you have learned how to write shell scripts that take file names and other bits of information from the command line. This is all right for people used to the command line—but “normal” users often appreciate a more “interactive” style, where a script asks questions or offers a menu interface. This chapter will show you how to implement these things using `bash`.

5.2 The `read` Command

We have made a passing acquaintance of the shell’s `read` command already: It reads lines from its standard input and assigns them to shell variables, in constructions like

```
grep ... | while read line
do
    ...
done
```

As we have seen, you can specify several variables. The input is then split into “words”, and the first variable is assigned the first word, the second variable the second, and so on:

```
$ echo Hello world | read h w
$ echo $w $h
world Hello
```

Superfluous variables remain empty:

```
$ echo 1 2 | read a b c
$ echo $c
$ _
```

Nothing

If there are more words than variables, the last variable gets all the rest:

```
$ echo 1 2 3 | read a b
$ echo $b
2 3
$ _
```

(This, of course, is the secret behind the success of “`while read line`”.)

What is a “word”? Here, too, the shell uses the content of the `IFS` variable (short for “internal field separator”), viewed per character, as the separators. The default value of `IFS` consists of the space character, the tab character, and the newline separator, but you can frequently make life easier for yourself by specifying a different value:

```
IFS=":"
cat /etc/passwd | while read login pwd uid gid gecost dir shell
do
    echo $login: $gecost
done
```

saves you from having to mess around with `cut`.

Of course you can use `read` to read from the keyboard:

```
$ read x
Hello
$ echo $x | tr a-z A-Z
HELLO
```

With bash, you can even combine this with a prompt, as in the following script to create a new user account:

```
#!/bin/bash
# newuser -- create a new user account

read -p "Login name: " login
read -p "Real name: " gecos

useradd -c "$gecos" $login
```

Especially when using `read`, it is important to enclose the “read” variables in quotes to avoid problems with spaces.

What happens if the invoker of the `newuser` script enters invalid data? We might require, for instance, that the new user’s login name consist of lowercase letters and digits only (a common convention). This could be enforced as follows: input validation

```
read -p "Benutzername: " login

test=$(echo "$login" | tr -cd 'a-z0-9')
if [ -z "$login" -o "$login" != "$test" ]
then
    echo >&2 "Invalid login name $login"
    exit 1
fi
```

We consider what remains after all invalid characters have been removed from the proposed login name. If the result does not equal the original login name, the latter contained “forbidden” characters. It may not be empty, either, which we check using `test`’s `-z` option.

With drastic operations such as the creation of new user accounts, verification is in order so that the user can abort the script if they get second thoughts (maybe because of some erroneous previous input). A convenient method to do this is via a shell function like verification

```
function confirm () {
    done=0
    until [ $done = 1 ]
    do
        read -p "Please confirm (y/n): " answer
        case $answer in
            [Yy]*) result=0; done=1 ;;
            [Nn]*) result=1; done=1 ;;
            *) echo "Please answer 'yes' oder 'no'" ;;
        esac
    done
    return $result
}
```

The safety check within the script might then be something like

```
confirm && useradd ...
```

Exercises



5.1 [!1] Change the `newuser` script such that it checks whether the new user's real name contains a colon, and possibly outputs an error message and terminates.



5.2 [2] Extend the `newuser` script such that the invoker can select the new user's login shell. Take care that only shells from `/etc/shells` will be accepted.



5.3 [2] Extend the `confirm` shell function such that it takes the prompt as a parameter. If no parameter has been passed, the default prompt "Please confirm" should be output.



5.4 [3] Write a simple guessing game: The computer selects a random number between (for example) 1 and 100 (the `RANDOM` shell variable produces random numbers). The user enters a number and the computer answers "Too big" or "Too small". This is repeated until the user has found the correct number.

5.3 Menus with select

The bash features a very powerful command for selecting options from a list, `select`, with a syntax similar to that of `for`:

```
select <variable> [in <list>]
do
    <commands>
done
```

This construct describes a loop where the value of `<variable>` is determined by a user choice from `<list>`. The loop is executed over and over again until end-of-file is reached on the standard input. Consider the following example:

```
$ select type in Hamburger Cheeseburger Fishburger
> do
>     echo $type coming up ...
> done
1) Hamburger
2) Cheeseburger
3) Fishburger
#? 2
Cheeseburger coming up ...
#? 3
Fischburger coming up ...
#?  + 
$ _
```

That is, the shell presents the entries of `<list>` with preceding numbers, and the user can make a choice by means of one of the numbers.



`select` behaves much like `for`: If the `<list>` is omitted, it presents the script's positional parameters for selection. The `select` loop, like all other shell loops, can be aborted using `break` or `continue`.

newuser revisited We can use `select` to further refine our `newuser` script. For example, you might want to support various types of users—professors, other staff, and students at a university, for example. In this case it would be useful if the `newuser` script offered a choice of various user types. Different default values might then derive from that choice, such as the primary group, home directory, or the set of default files copied to the home directory. On this occasion you will be able to learn about yet another way of storing configuration data for shell scripts.

We assume that, within the `/etc/newuser` directory, there is a file named `t` for every type of user `t`. For example, a file called `/etc/newuser/professor` for professors and another called `/etc/newuser/student` for students. The content of `/etc/newuser/professor`, for example, might look like this:

```
# /etc/newuser/professor
GROUP=profs
EXTRAGROUP=office
HOMEDIR=/home/$GROUP
SKELDIR=/etc/skel-$GROUP
```

(Professors get the Linux group `profs` as their primary group, as well as the group `office` as a supplementary group). This, of course, is our old trick “configuration file containing shell variable assignments”, with the difference that now there is one configuration file for every type of user. Creating a user once we know the user type goes approximately like

```
confirm || exit 0

. /etc/newuser/$type
useradd -c "$gecos" -g $GROUP -G $EXTRAGROUP \
-m -d $HOMEDIR/$login -k $SKELDIR $login
```

We still need to determine the proper user type. Of course we do not want to hard-code the selection list within the `newuser` script, but make it depend on the content of `/etc/newuser`:

```
echo "The following user types are available:"
PS3="User type: "
select type in $(cd /etc/newuser; ls) '[Cancel]';
do
    [ "$type" = "[Cancel]" ] && exit 0
    [ -n "$type" ] && break
done
```

The `PS3` shell variable specifies the prompt displayed by `select`.

Exercises



5.5 [!1] What happens if, at the `select` prompt, you enter something that does not correspond to the number of a menu entry?

Who wants to be a ... Our next script is loosely based on a popular television game show: Consider a file `wrtb.txt` containing questions and answers of the form

```
0?:According to the proverb, what do too many cooks do?
0-:Eat the roast
0-:Break the stove
0+:Spoil the broth
0-:Drop the cutlery
0:>:50
```

```
50?:Which of the following is edible?
50:-:Cool cat
50+:Hot dog
50:-:Lukewarm guinea pig
50:-:Tepid turtle
50:>:100
<<<<<<
```

The script—let us call it `wtb`—should, beginning at score 0, present the questions and answers. If the user selects a wrong answer, the program terminates; on a correct answer, it proceeds with the next question (whose score derives from the “>:” line).

An important basic strategy in more sophisticated programming projects is “abstraction”. In our case, we try to “hide” the actual format of the question file in a function that looks up the appropriate elements. Theoretically, we might later take the questions from a database instead of a flat text file, or change the data storage in some other way (for example, by randomly selecting a question from a pool of questions appropriate for the current score). One possible, if not exceedingly efficient, way of accessing the question data might be

```
qfile=wtb.txt

function question () {
  if [ "$1" = "get" ]
  then
    echo "$2"
    return
  fi
  case "$2" in
    display) re='?' ;;
    answers) re='[-+]' ;;
    correct) re='+' ;;
    next)   re='>' ;;
    *)      echo >&2 "$0: get: invalid field type $2"; exit 1 ;;
  esac
  grep "$1:$re:" $qfile | cut -d: -f3
}
```

The question function must first be called like

```
q=$(question get <score>)
```

This returns the unique identifier of a question with the specified score (with us, simply the score itself, since there is just one question per score in the file). We remember this identifier in a shell variable (here, `q`). Afterwards, we can use the following invocations:

<code>question \$q display</code>	<i>returns the question</i>
<code>question \$q answers</code>	<i>returns all answers, one per line</i>
<code>question \$q correct</code>	<i>returns the correct answer</i>
<code>question \$q next</code>	<i>returns the score for a correct answer</i>

The data is made available on the shell function’s standard output.



For connoisseurs: These are, of course, the beginnings of an “object-based” approach—“question get” returns a “question object” which then supports the various methods `display` etc.

Next, we require a function that displays a question and solicits the answer. Naturally, this function builds on the question function that we just discussed:

```
function present () {
  # Find and show the question
  question $1 display
  # Find the correct answer
  rightanswer=$(question $1 correct)
  # Show the answers
  PS3="Your answer: "
  IFS=$'\n'
  select answer in $(question $1 answers)
  do
    if [ -z "$answer" ]
    then
      echo "Please answer something reasonable."
    else
      test "$answer" = "$rightanswer"
      return
    fi
  done
}
```

The answers are, of course, presented using `select`. We need to take into account that `select` uses the `IFS` variable to separate the various menu entries when constructing the menu—with the `IFS` variable’s default value, `select` would show every single word in all the answers as a separate possible selection (!). Try it! For the function’s return value we exploit the fact that the return value of the last “real” command (`return` does not count) is considered the return value of the function as a whole. Instead of a tedious

```
if [ "$answer" = "$rightanswer" ]
then
  return 0
else
  return 1
fi
```

we just use the construct shown above, with a `return` right after a test.

Finally, we need the “framework” that brings the two separate parts “question management” and “user interface” together. This might look roughly like

```
score=0
while [ $score -ge 0 -a $score -lt 1000000 ]
do
  q=$(question get $score)
  if present $q
  then
    score=$(question $q next)
  else
    score=-1
  fi
done
```

The framework takes care of selecting a question (using “`question get`”) matching the player’s current score. This question is presented (using `present`), and depending on the “success” of the presenting function (thus the correctness of the answer) the score will either be increased to the next level, or the game is over. At the end just a few warm parting words from the (computerised) host:

```
if [ $score -lt 0 ]
then
```

Table 5.1: dialog's interaction elements

	Description
calendar	Displays day, month, and year in separate windows; the user may edit. Returns the value in the form "day/month/year"
checkboxlist	Displays a list of entries that can be selected or deselected individually. Returns a list of the "selected" entries
form	Displays a form. Returns the values that have been entered, one per line
fselect	Displays a file selection dialog. Returns the selected file name
gauge	Displays a progress bar
infobox	Outputs a message (without clearing the screen)
inputbox	Allows entry of a character string, returns that
inputmenu	Displays a menu where the user may change the entries
menu	Displays a menu of selections
msgbox	Displays a message, waits for confirmation
passwordbox	inputbox that does not display its input
radiolist	Displays a list of entries, of which exactly one may be selected; returns the selected entry
tailbox	Displays the content of a file, like "tail -f"
tailboxbg	Like tailbox, file is read in the background
textbox	Displays the content of a text file
timebox	Displays hour, minute, and second, with editing facilities; returns time in the format "hour:minute:second"
yesno	Displays a message and allows "yes" or "no" as an answer

```

    echo "That wasn't so hot, you lost"
else
    echo "Congratulations, you have won"
fi

```

Exercises



5.6 [!1] Extend the `wtb` script such that it displays a question's "value" (i. e., the score that the participant will have once he has answered the question correctly).



5.7 [!3] Think of an interesting extension to `wtb` and implement it (or two or three).



5.8 [3] Revise the question function of `wtb` such that it requires fewer `grep` calls.

5.4 "Graphical" Interfaces Using dialog

Instead of boring textual menus and teletype-like dialogues, you can avail yourself of a nearly "graphical" user interface for your scripts. This can be done by means of the `dialog` program, which you may have to install separately if your Linux distribution does not do it for you. `dialog` uses the facilities offered by modern terminals (or terminal emulation programs) to present full-screen menus, selection lists, text entry fields, and so on, possibly in colour.

`dialog` knows a large range of interaction elements (Table 5.1). The details of its configuration are quite complex and you should read up on them in `dialog`'s documentation; we will restrict ourselves to the bare necessities here.

For example, we might change our `wtb` program such that the questions and the final evaluation are displayed using `dialog`: The menu element is best suited to

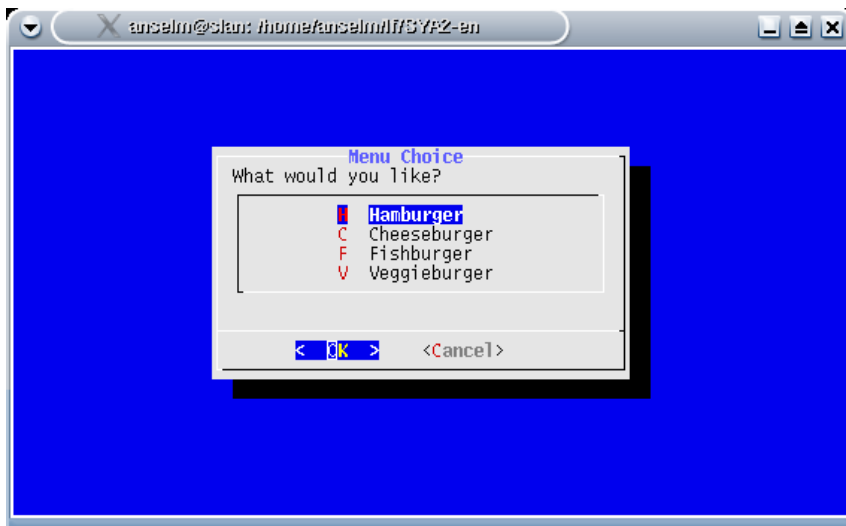


Figure 5.1: A dialog-style menu

display our questions and answers. A dialog invocation for a menu looks roughly like this:

```
$ dialog --clear --title "Menu Choice" \
  --menu "What would you like?" 12 40 4 \
    "H" "Hamburger" \
    "C" "Cheeseburger" \
    "F" "Fishburger" \
    "V" "Veggieburger"
```

You can see the result in Figure 5.1. The more important options include `--clear` (clears the screen before the menu is displayed) and `--title` (specifies a title for the menu). The `--menu` option determines this dialog to be a selection menu; this is followed by an explanatory text and the three magic numbers “height of the menu in lines”, “width of the menu in characters”, and “number of entries displayed at the same time”. At the end there are the entries themselves, all with a “short name” and their actual content. In the resulting menu, you can navigate using the arrow keys, the number keys `0` to `9`, or the initial letters of the short names; in our example, the inputs `3` or `f` would lead to the “Fishburger”.

Another consideration is important when dealing with `dialog`: The program usually produces its output on the standard error channel. This means that if, as is likely, you want to intercept and process `dialog`’s results, you must redirect `dialog`’s standard error output, not its standard output. This results from the fact that `dialog` writes to its standard output to address the terminal; thus if you redirected its standard output you would no longer see anything on the screen, while `dialog`’s actual output would drown among various terminal control characters. There are various ways of handling this: You can have `dialog` write to a temporary file via `2>`, but it is tedious to ensure that these temporary files are disposed of at the end of the script (clue: `trap`). Alternatively, you can get tricky with I/O redirection, like

```
result=$(dialog ... 2>&1 1>/dev/tty)
```

This connects standard error output (file descriptor 2) to where standard output currently goes (the `result` variable) and then connects standard output to the terminal (`/dev/tty`).



Of course this works only because the standard output is not usable on devices other than terminals—to swap a script’s standard output and standard error output, descriptors need to be juggled about like so:

```
( program 3>&1 1>output 2>&3 ) | ...
```

This command line redirects program's standard output to the output file and its standard error output to the pipeline. It is, in a certain sense, the opposite of the more common

```
program 2>output | ...
```

Who wants to be a ... with dialog Let us now look at a dialog-based version of the `wtb` script. The trouble we went to concerning "abstraction" now pays off: the changes restrict themselves mostly to the present function.

The main hurdle to overcome in order to make `wtb` dialog-capable results from the fact that, in dialog's menu syntax

```
dialog ... --menu t h w n k0 e0 k1 e1 ...
```

the program insists on being passed the short names and menu entries k_i and e_i as single words. We could use a loop like

```
items=''
i=1
question $q answers | while read line
do
    items="$items $i '$line'"
    i=$((i+1))
done
```

to construct a list of short names and answers of the form

```
1 'Eat the roast' 2 'Break the stove' ...
```

in the `items` variable, but an invocation of the form

```
dialog ... --menu 10 60 4 $items
```

`arrays` does not agree with `dialog` at all. One solution is the use of **arrays**, which `bash` supports at least in a rudimentary manner.

Arrays An array is a variable that can contain a sequence of values. These values can be addressed by means of numeric indices. You do not need to declare an array; it suffices to access a variable "indexedly":

```
$ course[0]=appetiser
$ course[1]=soup
$ course[2]=fish
$ course[4]=dessert
```

You can access these variables individually, as in

```
$ echo ${course[1]}
soup
```

(the braces are necessary to avoid confusion with file search patterns) or as a group, as in

```
$ echo ${gang[*]}
appetiser soup fish dessert
```

(Incidentally, it does not matter if not all indices are used in sequence.) The “\${<name>[*]}” and “\${<name>[@]}” expansions are similar to * and @, as the following example illustrates:

```
$ course[3]="filet mignon"
$ for i in ${course[*]}; do echo $i; done
appetiser
soup
fish
filet
mignon
dessert
$ for i in "${course[*]}"; do echo $i; done
appetiser soup fish filet mignon dessert
$ for i in "${course[@]}"; do echo $i; done
appetiser
soup
fish
filet mignon
dessert
```

The latter is what we need.



You can also assign a value to an array as a whole, like

```
$ course=(appetiser gazpacho salmon \
> "boeuf Stroganoff" "crepes Suzette")
$ for i in "${course[@]}"; do echo $i; done
appetiser
gazpacho
salmon
boeuf Stroganoff
crepes Suzette
$ course=([4]=pudding [2]=trout [1]=broth \
> [3]=fricassee [0]=appetiser)
$ for i in "${course[@]}"; do echo $i; done
appetiser
broth
trout
fricassee
pudding
```



If you want to be thorough, you can officially declare a variable an array using

```
declare -a <Name>
```

However, this is normally not necessary.

Arrays applied Our new present routine must assemble a list of short names and answers that will later be passed to dialog. This might look like

```
declare -a answers
i=0
rightanswer=$(question $1 correct)
IFS=$'\n'
for a in $(question $1 answers)
do
```

```

    answers[${(2*i)}]=${(i+1)}
    answers[${(2*i+1)}]="$a"
    [ "$a" = "$rightanswer" ] && rightshort=${(i+1)}
    i=${(i+1)}
done

```

We use *i* as an index into the `answers` array. For each answer, *i* is incremented, and the actual indices for the short name and the answer itself result from an index transformation: For example, for *i* = 1, the short name ends up in `answers[2]` and the answer text in `answers[3]`; for *i* = 3 the short name goes to `answers[6]` and the answer text to `answers[7]`. As short names, we shall be using the numbers 1, ..., 4 instead of 0, ..., 3. Additionally, we remember the correct answer's short name in `rightshort`; this is important because `dialog` returns just the short name of the selected menu entry rather than the full name (as `select` did).

With our `answers` array, we can now invoke `dialog`:

```

# Display the question
sel=$(dialog --clear --title "For $(question $1 next) points" \
    --no-cancel --menu "$(question $1 display)" 10 60 4 \
    ${answers[@]} 2>&1 1>/dev/tty)
test "$sel" = "$rightshort"

```

Again, we fetch the question text via `question`'s `display` method; the `--no-cancel` option suppresses the menu's "Cancel" button.

Finally, we can use `dialog` to pronounce the final result:

```

if [ $score -lt 0 ]
then
    msg="That wasn't so hot, you lost"
else
    msg="Congratulations, you won"
fi
dialog --title "Final Result" --msgbox "$msg" 10 60

```

This uses the much more straightforward `--msgbox` option. The main changes for the `dialog`-based script (it is accordingly called `dwtb`) are displayed more clearly in Figure 5.2.

Additional remarks `dialog` is convenient but you should not go overboard with it. It is probably most useful in the "twilight zone" where something nicer than raw text-terminal based interaction is desired, but a "real" GUI is not or not necessarily available. For instance, the installation routines of the Debian GNU/Linux distribution use `dialog`, since on the boot disks there is not really room for a full GUI environment. More complex graphical interfaces are beyond even the capabilities of `dialog`, and stay the domain of environments like Tcl/Tk or programs based on C or C++ (possibly using Qt or Gtk+, KDE or GNOME).



For displaying simple dialog boxes with buttons there is the X11 client `xmessage`. It can be used, for instance, to send messages to a user's graphical terminal. `xmessage` is part of the basic X11 package and hence should be available on virtually all Linux systems. Its looks may seem a bit old-fashioned, though.



Incidentally, KDE offers a vaguely `dialog`-ish program called `kdialog` which allows KDE-like GUIs for shell scripts. However, for anything beyond the most essential basics we would strongly urge you to use a "reasonable" basis for your GUI programs, such as Tcl/Tk or PyKDE.

```
#!/bin/bash
# dwwtb -- dialog-capable wwtb

# The question function is just as in wwtb
<<<<<<

function present () {
    declare -a answers
    i=0
    rightanswer=$(question $1 correct)
    IFS=$'\n'
    for a in $(question $1 answers)
    do
        answers[$((2*i))]=$((i+1))
        answers[$((2*i+1))]=$a
        [ "$a" = "$rightanswer" ] && rightshort=$((i+1))
        i=$((i+1))
    done

    # Display the question
    sel=$(dialog --clear --title "For $(question $1 next) points" \
        --no-cancel --menu "$(question $1 display)" 10 60 4 \
        ${answers[@]} 2>&1 1>/dev/tty)
    test "$sel" = "$rightshort"
}

# The main program is just as in wwtb
<<<<<<

if [ $score -lt 0 ]
then
    msg="That wasn't so hot, you lost"
else
    msg="Congratulations, you won"
fi
dialog --title "Final Result" --msgbox "$msg" 10 60
```

Figure 5.2: A dialog-capable version of wwtb

Exercises



5.9 [!3] Write a shell script called `seluser` which offers you a selection menu with all users from `/etc/passwd` (use the user name as the short name and the content of the GECOS field—the “real” name—as the actual menu entry). This script should produce the selected user name on the standard output. (Imagine the script to be part of a more sophisticated user management tool.)



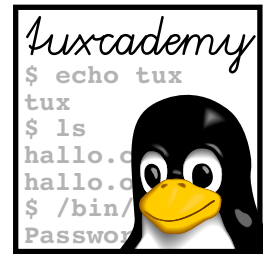
5.10 [3] Write a shell script named `show-motd` which displays the content of the `/etc/motd` using `xmessage`. Make the system run the script when a user logs in. (*Hint: Xsession.*)

Commands in this Chapter

<code>dialog</code>	Allows GUI-like interaction controls on a character screen	<code>dialog(1)</code>	80
<code>kdiallog</code>	Allows use of KDE widgets from shell scripts	<code>kdiallog(1)</code>	84
<code>xmessage</code>	Displays a message or query in an X11 window	<code>xmessage(1)</code>	84

Summary

- With `read`, you can read data from files, pipelines, or the keyboard to shell variables.
- The `select` command allows a convenient, repeated choice from a numbered list of alternatives.
- The `dialog` program makes it possible to endow text-based shell scripts with GUI-like interaction elements.



6

The sed Stream Editor

Contents

6.1	Introduction.	88
6.2	Addressing	88
6.3	sed Commands.	90
6.3.1	Printing and Deleting Lines	90
6.3.2	Inserting and Changing	91
6.3.3	Character Transformations	91
6.3.4	Searching and Replacing	92
6.4	sed in Practice	93

Goals

- Knowing the function and use of sed
- Being able to design simple sed scripts
- Using sed in shell scripts

Prerequisites

- Knowledge of shell programming (e. g., from the preceding chapters)
- Regular expressions

6.1 Introduction

Linux features a large selection of simple text processing tools—from `cut` and `grep` to `tr` and `sort` to `join` and `paste`. As the text manipulation tasks get more sophisticated, the classic tools like `cut` or `tr` may no longer be sufficient. For example, `tr` can turn an “X” into an “U”, but fails to fulfil the ancient alchemists’ dream of turning “lead” into “gold”—just as they did themselves.

Usually you would now start a suitably powerful editor to replace every occurrence of “lead” in your file by “gold”. There are two reasons, though, that might urge you to take a different route.

Firstly, it might be the case that you want to modify the file automatically, for example within a shell script. Secondly, there are cases where you do not want to change a *file* to begin with, but a (potentially infinite) stream of text, e. g., in the middle of a pipeline.

`sed` This is the domain of `sed` (for “stream editor”), which lets you process a text stream according to predefined instructions. If not even `sed` can do the trick, you can call in even more reinforcements and use `awk` (Chapter 7), or change to a more sophisticated scripting language such as Perl.

`sed`’s programming model is fairly simple: The program takes a number of instructions, reads its input line by line, and applies the instructions to the input lines where appropriate. Finally, the lines are output in possibly modified form (or not). The important point is that `sed` just reads its input file (if there is one) and never modifies it; if anything, modified data are written to standard output.

`commands` `sed` accepts commands either one after the other using the `-e` option, which may occur multiple times on the same command, or within a file whose name is passed to `sed` using the `-f` option. If you are passing just one command on the shell command line, you can even leave out the `-e`.



You can specify several `sed` commands within a single `-e` option if you separate them using semicolons.



There is nothing wrong with executable “`sed` scripts” of the form

```
#!/bin/sed -f
<sed commands>
```

`line address` Every `sed` command consists of a line address determining which lines the command applies to, and the actual command, for example

```
$ sed -e '1,15y/uU/xX/'
```

`sed` commands are exactly one character long, but can be followed by additional parameters (depending on the command).

6.2 Addressing

There are various methods of “addressing” lines in `sed`. Some commands apply to one line, others to ranges of lines. Single lines can be selected as follows:

Line numbers A number as a line address is considered a line number. The first input line is number 1, and the count continues from there (even if the input consists of several files).



You can get `sed` to consider each input file separately by means of the `-s` option. In this case, every input file starts with a line no. 1.

A line specification like `i-j` stands for “every j -th line, starting at line i ”. Thus, “2-2” would select every even-numbered line.

Table 6.1: Regular expressions supported by `sed` and their meaning

Regular expression	Meaning
[a-d]	One character from the set {a, b, c, d}
[^abc]	One character <i>except</i> a, b, or c
.	Any character (including space characters or newlines)
*	Any number of repetitions of the preceding regular expression (including none)
?	The preceding regular expression occurs once or not at all
^	Beginning of the line
\$	End of the line
\<	Beginning of a word
\>	End of a word

Regular expressions A regular expression of the form “/*expression*/” selects all lines matching the expression. “/a.*a.*a/”, for example, selects all lines containing at least three “a” characters.

Last line The dollar sign (“\$”) stands for the last line of the last input file (here again, `-s` considers every input file separately).

You can specify ranges of lines by arbitrarily combining single addresses, with a comma in-between. The range starts with a line matching the first address and extends from there to the first line matching the second address. Ranges can start in one file and finish in another, unless the `-s` option was given. Here are some examples for range addressing:

1,10 This selects the first ten input lines

1,/^{}/ This selects all lines up to the first empty line. This idiom is useful, for example, to extract the “header” of an e-mail message (which by definition always finishes with an empty line, but may not contain empty lines)

1,\$ This describes “all input lines” but may generally be omitted

/^BEGIN/,/^END/ This describes all ranges of lines starting at one beginning with “BEGIN” up to one beginning with “END” (inclusively).



If the second address is a regular expression, it is searched for beginning with the line *immediately following* the line that starts the range. If the first address is a regular expression, too, then, once a line matching the second expression was found, `sed` continues looking for another line matching the first expression—there might be another matching range of lines.



If the second address is a number describing an *earlier* line than the one matching the first address, only the first matching line is output.

You can select all lines *not* matched by an address by appending a “!” to the address:

5! addresses all input lines except for the fifth

/^BEGIN/,/^END/! addresses all input lines that are *not* part of a BEGIN-END block

Exercises



6.1 [!1] Consider the following file:

```

ABCDEFGG
123 ABC
EFG
456 ABC
123 EFG
789

```

Which lines do the following addresses describe? (a) 4; (b) 2,/ABC/; (c) /ABC/,/EFG/; (d) \$; (e) /^EFG/,2; (f) /ABC/!



6.2 [2] Study the GNU sed documentation and explain the meaning of the (GNU-specific) address “0,/⟨expression⟩/”.

6.3 sed Commands

6.3.1 Printing and Deleting Lines

Usually, sed writes every input line to its standard output. The d (“delete”) command suppresses lines so that they are not output. For example,

```
sed -e '11,$d'
```

is equivalent to the head command: Just the first 10 input lines are let through.

The -n option inhibits the automatic output. sed outputs only those lines that are subject to an explicit p (“print”) command. Thus you can simulate head another way by means of

```
sed -ne '1,10p'
```

With a regular expression instead of a numbered-line range, we can imitate grep:

```
sed -ne '/[#]/p' /etc/ssh/sshd_config
```

corresponds to the grep invocation

```
grep '[#]' /etc/ssh/sshd_config
```

Back to head: Our alternatives so far have the disadvantage that they insist on reading all of the input. Considering that we are done after the tenth line, it is somewhat pointless to go on reading a hundred thousand further lines, just to delete them or not output them. The most efficient head simulation is, in fact,

```
sed -e 10q
```

—the q command (“quit”) terminates sed immediately.

Exercises



6.3 [!1] How would you delete all empty lines from sed’s input?



6.4 [!1] The popular Apache web server, in its httpd.conf file, uses blocks of the form

```

<Directory /var/www/htdocs>
...
</Directory>

```

to configure options for certain directories. Give a sed command to extract all such blocks from httpd.conf.



6.5 [2] You have seen head. How can you simulate tail using sed?

6.3.2 Inserting and Changing

sed really gets to flex its muscles once you allow it to not just filter the text stream but modify it. To do this by lines, there are the three commands “a” (“append”), i (“insert”), and c (“change”) for appending material after a line, inserting material in front of a line, or replacing one line by another:

```
$ fortune | sed -e '1 i >>>' -e '$ a <<<'
>>>
"So here's a picture of reality: (picture of circle with ▷
◁ lots of squiggles in it) As we all know, reality is a mess."

-- Larry Wall (Open Sources, 1999 O'Reilly and Associates)
<<<
```

Here, the GNU implementation of sed, which is customary on Linux, allows you to specify all of this on one line. The traditional form is somewhat more tedious and better suited to sed scripts. In the following example, the “\$” are not part of the file, but are appended to the line ends using cat in order to make them more obvious.

```
$ cat -E sed-script
li\$
First inserted line\$
Second inserted line\$
```

With the traditional syntax, sed expects a backslash after the a, i, or c command, *immediately preceding the end of line*. If more than one line is to be inserted, each of these lines *except for the last* must also be terminated with a backslash immediately preceding the end of line. This script is invoked by

```
$ sed -f sed-skript datei
```

The a and i commands are “one-address commands”: They allow only addresses matching a single line—no ranges (but there may well be several input lines which match the one address given with a and i, and which will all be duly processed). This one address may include all the bells and whistles mentioned above including regular expressions etc. With c, an address range implies that all of the range is to be replaced.

Exercises



6.6 [!2] Give a sed command that inserts a blank line after every input line that consists of capital letters and spaces only. (Imagine you want to emphasise titles).

6.3.3 Character Transformations

The y command makes sed replace single characters by others. In the contrived example

```
$ echo 'wéírd fíle nāme' | sed -e 'y/ äéí/_?/'
w??rd_f?le_n?me
```

some unusual characters and spaces are “repaired”. Unfortunately, y does not allow ranges like a-z, thus it is only a weak replacement of tr.

Exercises



6.7 [1] Give a sed command that converts all lowercase letters to capitals on each odd-numbered line.

6.3.4 Searching and Replacing

The `s` (“substitute”) command is possibly the most powerful `sed` command. It allows substitution of a regular expression by a character string whose composition may change dynamically.

The regular expression to be replaced is given like a line address, in `“/.../”`, followed by the replacement text and `“/”`, thus for example

```
$ sed -e 's/\<Lead\>/gold/'
```

Here the word brackets prevent the accidental invention of “golding lady”, “gold-ership”, or other misgolding words.

Note that `“\<”`, `“\>”`, `“^”`, and `“$”` correspond to *empty strings*. In particular, `“$”` is not the actual newline character at the end of a line but the empty string `“”` immediately preceding the newline character; therefore `“s/$/|/”` inserts a `“|”` immediately before the end of line instead of replacing the end of line with it.

The substitution text can depend on the text that is being replaced: In addition to numbered back-references to parenthesised substrings using `“\1”` etc., `“&”` stands for all of the text matched by the search expression. For example:

```
$ echo Every word quoted. | sed -e 's/\([A-Za-z]\+\)/"\1"/g'
"Every" "word" "quoted".
$ echo Every word quoted. | sed -e 's/[A-Za-z]\+/"&"/g'
"Every" "word" "quoted".
```

Normally, `s` replaces just the first “hit” on every line. If a `“g”` (as in “global”) is appended to the `s` command—like here—, it replaces every occurrence of the search pattern on each line. Another useful modifier is `“p”` (“print”), which outputs the line after replacement (like the `“p”` command).

If you append a number `n`, only the `n`-th “hit” will be replaced: An input file like

```
Column 1   Column 2   Column 3   Column 4
```

with tab characters between the columns can be converted by the `sed` command

```
s/[Tab]/\
/2
```

(where `[Tab]` is, in fact, a “real” tab character¹; the backslash at the end of the line hides a newline character) to

```
Column 1   Column 2
Column 3   Column 4
```



Sometimes the slash as a separator for search expressions and replacement strings is a nuisance, especially when dealing with file names. In fact, you can pick the separator character almost arbitrarily; you just need to be consistent and use the same one three times. Only spaces and newline characters are not allowed as separators.

```
sed 's,/var/spool/mail,/var/mail,'
```

(For regular expressions used as addresses, the slashes are, unfortunately, mandatory.)

¹Difficult to type in bash; you can type all control characters by entering `[Ctrl]+[q]` first.

Exercises



6.8 [!1] Give a sed command that replaces the word `yellow` by the word `blue` in all of its input.



6.9 [!2] State a sed command that deletes the first word from all lines beginning with `"A"`. (For the purposes of this exercise, a word is a sequence of letters.)

6.4 sed in Practice

Here are some more examples of how to use sed in more complex shell scripts:

Renaming files In Section 4.3 we worked on changing file extensions for “many” files. With sed, we have a tool that allows us to rename many files arbitrarily. A suitable command might look like this:

```
$ multi-mv 's/pqr/xyz/' abc-*.txt
```

Our (hypothetical, so far) `multi-mv` command takes as its first argument a sed command which is then applied to all of the following file names.

As a shell script, `multi-mv` might look somewhat like this:

```
#!/bin/bash
# multi-mv -- renames multiple files

sedcmd="$1"
shift

for f
do
  mv "$f" "$(echo \"$f\" | sed \"$sedcmd\")"
done
```

Overwriting files If you have paid attention during “Shell I/O Redirection 101”, you know perfectly well that commands like

```
$ sed ... file.txt >file.txt
```

do not do what one might naively expect: The `file.txt` file is ruined quite thoroughly. Thus if you edit a file using sed and want to store the result under the original file name, you have to put in extra work: Write the result to a temporary file first and then rename it, like this:

```
$ sed ... file.txt >file.tmp
$ mv file.tmp file.txt
```

This rather long-winded procedure lends itself to being automated by means of a shell script:

```
$ oversed ... file.txt
```

At first we will restrict ourselves to the simple case where the first argument of `oversed` contains all the instructions for sed. In addition, we assume that in a command like

```
$ oversed ... file1.txt file2.txt file3.txt
```

the named files should be considered and overwritten with their new content individually (everything else does not really make sense).

The script might look roughly like this:

```
#!/bin/bash
# oversed -- Edit files "in place" using sed

out=/tmp/oversed.$$
sedcmd="$1"
shift

for f
do
    sed "$sedcmd" "$f" >$out
    mv $out $f
done
```

The only thing remarkable about this script is possibly the way the name for the temporary file is constructed (in `out`). We take care to avoid a collision with some other, simultaneous `oversed` invocation, by appending the current process ID (in `$$`) to the file name.



If security is important to you, you should stay away from the `"/tmp/oversed.$$"` method, since PID-based file names can be guessed. An attacker could use this to point your program to a file that will then be overwritten. To be safe, you could use the `mktemp` program, which instead of using the PID generates a random file name that is guaranteed not to exist. It even creates the file with restrictive permissions.

```
$ TMP=$(mktemp -t oversed.XXXXXX)
```

Creates, e.g., /tmp/oversed.z19516

```
$ ls -l $TMP
```

```
-rw----- 1 anselm anselm 0 2006-12-21 17:42 /tmp/oversed.z19516
```



`mktemp`'s handy `"-t"` option places the file in a "temporary directory", namely the directory given by the `TMPDIR` environment variable, or else a directory specified using the `"-p"` option, or else `/tmp`.

We can improve the script further. For example, we could check whether `sed`, in fact, changed anything about the file, i. e., whether the input and output files are equal. In this case we can save ourselves the trouble of renaming the output file. Also, if the output file is empty, something is likely to have gone wrong and we ought not to overwrite the input file. In this case, the loop might look like

```
for f
do
    sed "$sedcmd" "$f" >$out
    if [ test -s $out ]
    then
        if cmp -s "$f" $out
        then
            echo >&2 "$0: file $f not changed, not overwriting"
        else
            mv "$f" $out
        fi
    else
        echo >&2 "$0: file $f's output empty, not overwriting"
    fi
done
rm -f $out
```

Here we use the file test operator `-s`, which reports success if the given file exists and is longer than 0 bytes. The `cmp` command compares two files byte by byte and returns success if both files are identical; the `-s` option suppresses the notice giving the position of the first differing byte (which does not help us here at all).



Do be careful with `oversed`—you should make sure that your `sed` commands do the right thing before you let them loose on important files. Also consider Exercise 6.10.

In the previous example, we have assumed that the script's first argument contains everything you want to tell `sed`. But how can you pass several `-e` options to `sed`, or even different options such as `-n` or `-r`? For this you must inspect the command line more closely:

```
sedargs=""
while [ "${1:0:1}" = "-" ]
do
  case "$1" in
    -*[ef]) sedargs="$sedargs $1 $2"
            shift 2 ;;
    -*) sedargs="$sedargs $1"
        shift ;;
    *) break ;;
  esac
done
```

This loop checks the command-line arguments: Everything starting with `"-"` and ending with `"e"` or `"f"` is presumably an option of the form `"-f file"` or `"-ne '...'"`. The option, together with the subsequent file name or `sed` command specification is stored in `sedargs` and removed from the command line (`"shift 2"`). Analogously, everything else that starts with a `"-"` is considered a `"normal"` option without a subsequent argument and also stored in `sedargs`. Thus the `sed` invocation inside the main loop body becomes

```
sed $sedargs "$f"
```

Even this is unfortunately not yet perfect (see Exercise 6.11)).



GNU `sed`, the canonical `sed` implementation for Linux, supports a non-standard option called `-i`, which works rather like `oversed`.

Exercises



6.10 [!2] Ensure that in `oversed` the original input file is saved under another name before the `sed` output file is renamed (you might, for example, append `".bak"` to the file name as an additional suffix).



6.11 [3] `sed` supports some other command-line arguments as well. For example, `"--"` (as elsewhere) says `"end of options – whatever comes now is a file name, even if it looks like an option"`, and there are `"long"` options of the form `"--expression=..."` (as a synonym for `-e`). Adapt `oversed` such that it does the Right Thing even for these arguments.



6.12 [3] With your new-found wisdom about `sed`, you can make the `wrtb` script from Chapter 5 somewhat less tedious as far as the question file format is concerned. Give an implementation of the `question` function that reads a question file formatted like

```

question 0
?According to the proverb, what do too many cooks do?
-Eat the roast
-Break the stove
+Spoil the broth
-Drop the cutlery
>50
end
question 50
?Which of the following is edible?
-Cool cat
<<<<<

```

Commands in this Chapter

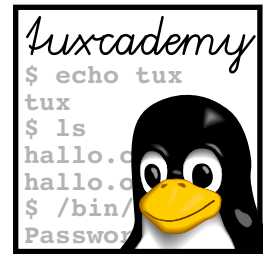
<code>cmp</code>	Byte-by-byte comparison of two files	<code>cmp(1)</code>	94
<code>mktemp</code>	Generates a unique temporary filename (securely)	<code>mktemp(1)</code>	94

Summary

- `sed` is a “stream editor” which reads its standard input and writes it (possibly modified) to its standard output.
- `sed` supports flexible addressing of input lines via their position or their content, as well as the description of line ranges in the input.
- Various text-modifying commands are available.

Bibliography

- DR97** Dale Dougherty, Arnold Robbins. *sed & awk*. Sebastopol, CA: O’Reilly & Associates, 1997, second edition. ISBN 1-56592-225-5.
<http://www.oreilly.de/catalog/sed2/>
- Rob02** Arnold Robbins. *sed & awk Pocket Reference*. Sebastopol, CA: O’Reilly & Associates, 2002, second edition. ISBN 0-596-00352-8.
<http://www.oreilly.com/catalog/sedawkrepr2/>



7

The awk Programming Language

Contents

7.1	What is awk?	98
7.2	awk Programs	98
7.3	Expressions and Variables.	100
7.4	awk in Practice	104

Goals

- Getting to know the awk programming language
- Being able to design simple awk programs
- Knowing how to use awk in shell scripts

Prerequisites

- Knowledge about shell programming (from the previous chapters)
- Programming experience in other languages is helpful

7.1 What is awk?

awk is a programming language for text file processing. The name derives from the last names of its inventors, Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan, rather than the English word “awkward”.



On the cover of the awk book by Aho, Weinberger and Kernighan [AKW88] there is a picture of an auk (*Alca torda*). This Nordic sea bird, whose name is pronounced just like awk, has nothing whatsoever to do with penguins.

awk in Linux Linux distributions usually do not contain the original AT&T awk, but compatible and more or less extended implementations such as mawk or gawk (GNU awk). For our purposes it is sufficient to talk about awk, since the extensions are not really relevant (we will point them out where appropriate).

awk as a programming language Calling awk a programming language can sound intimidating to many people who do not really see themselves as “programmers”. If you have a problem with this, then do consider awk a particularly useful tool to analyse and modify files—a kind of supercharged sed-cut-sort-paste-grep. With awk, you can, for example, create formatted reports from log files or perform various other operations on “structured data” such as “tables” with tab characters between columns. awk can, among other things, ...

- interpret text files consisting “records”, which in turn consist of “fields”;
- store data in variables and arrays;
- perform arithmetic, logical and string operations;
- evaluate loops and conditionals;
- define functions;
- post-process the output of commands.

programming model awk reads text from its standard input or files named on the command line, usually line by line. Every line is divided into fields and processed. The results are then written to standard output or a named file.

awk “programs” live in the gap between shell scripts and programs in languages such as Perl or Tcl/Tk. The main difference between awk and other programming languages like the shells, C, or Tcl/Tk consists of its “data-driven” operation, while the typical programming languages are more geared toward “functions”. In principle, an awk program works like a loop over the input records (usually lines) that is repeated until no input is left or the program is terminated. The control flow is largely given by the data. In most other languages, on the other hand, the main program is started once, and functions (that may read input) influence the progress of the calculation.

7.2 awk Programs

In the simplest case, awk works not unlike sed: You can select lines and then apply commands to them. A grep workalike in awk, for example, might look like this:

```
awk '/a.*a.*a/ { print }'
```

commands outputs all input lines containing at least three “a” characters. The braces may contain one or more commands which are applied to the lines matching the regular expression between the slashes.

awk “scripts” It is often more convenient to put awk “scripts” into their own files. You can execute such files using “awk -f <script file>”. Lines starting with “#” are considered **comments** comments, as in the shell.



Here, too, there is nothing wrong with directly executable `awk` scripts of the form

```
#!/usr/bin/awk -f
/a.*a.*a/ { print }
```

Here is an `awk` script which will output a message for each line containing a number:

```
#!/usr/bin/awk -f
/[0-9]+/ { print "This line contains a number." }
```

Lines that do not contain a number are ignored.

For every input line, `awk` checks which script lines match it, and all `awk` command sequences that match are executed. The following script, `classify`, tries to classify lines according to their content:

```
#!/usr/bin/awk -f
# classify -- classifies input lines
/[0-9]+/ { print "This line contains a number." }
/[A-Za-z]+/ { print "This line contains a word." }
/$/ { print "This line is empty." }
```

Here is an example for the `classify` script:

```
$ awk -f classify
123
This line contains a number.
foo
This line contains a word.
←
123 foo
This line contains a number.
This line contains a word.
```

You can see that the “123 foo” line matched two of the rules. It is possible to design rules such that only one matches in every case.

`awk` assumes that its input is structured and not just a stream of bytes. In the usual case, every input line is considered a “record” and split into “fields” on whitespace. input: structured



Unlike programs like `sort` and `cut`, `awk` considers sequences of spaces *one* field separator, instead of seeing an empty field between two adjacent space characters.

You can refer to the individual fields of a record using the `awk` expressions `$1`, `$2`, etc.:

```
$ ls -l *.sh | awk '{ print $9, $5 }'
dumppwd.sh 113
dwwtb.sh 1220
numbgame.sh 323
wwtb-sed.sh 1513
wwtb.sh 1132
```

Of course you need to take care that the shell does not try to expand the “\$”!



The “`$0`” expression returns the full input record (all fields).

On this occasion you might as well learn that a sequence of awk commands does not *need* to include a regular expression in front: A “{...}” without a regular expression will be applied to *every* input record.



Here, awk already comes in useful as an improved cut. Remember that cut is not able to change the order of columns in its output with respect to their order in the input.

You can change the input record delimiter using the -F option to awk:

```
$ awk -F: '{ print $1, $5 }'
root root
daemon daemon
bin bin
<<<<<<
```

Output fields are separated by blanks (you will later see how to change this).

BEGIN and END Additionally, awk lets you specify command sequences to be executed at the beginning of a run—before data has been read—and at the end—after the last record has been read. This can be used for initialisation or final results. The

```
ls -l *.txt | awk '
BEGIN { sum = 0 }
      { sum = sum + $5 }
END   { print sum }'
```

variable command, for example, adds the lengths of all files with the “.txt” extension within the current directory and outputs the result at the end. `sum` is a **variable** that contains the current total; variables in awk behave quite like shell variables, except that you can refer to their values without having to put a “\$” in front of the variable name.



For connoisseurs: awk variables may contain either strings or (floating-point) numbers. These data types are converted as required.

7.3 Expressions and Variables

We can improve the last section’s file size summation program even further. For example, we might count the files and output their number:

```
#!/usr/bin/awk -f
# filesum -- Add file sizes
  { sum += $5
    count++
  }
END { print sum, " bytes in " count " files" }
```

The “BEGIN” rule is not strictly required since new variables are set to 0 when they are first used. “sum += \$5” is equivalent to “sum = sum + \$5”, and “count++” in turn is equivalent to “count = count + 1”. (C programmers should feel right at home here.) The whole thing now works like this:

```
$ ls -l *.sh | filesum
4301 bytes in 5 files
```

This simple program is not without its problems. If you do not select a set of files from a directory (like all files with names ending in “.sh”, just now) but a complete directory—consider “ls -l .”—, the first line of the output will give the total number of “data blocks” (on Linux, usually kibibytes) occupied by files in the directory:

```
total 1234
```

This line does not have a fifth field and hence does not spoil the sum of sizes, but is counted as an input line, i. e., a file. Another problem concerns lines for subdirectories such as

```
drwxr-xr-x  3 anseIm  anseIm  4096 May 28 12:59 subdir
```

The “file size” in this entry has no meaning. Other file types (e. g., device files) add to the confusion.

To circumvent these problems, two observations are important: The number of fields of “interesting” lines is 9, and we want to consider only lines beginning with “-”. The latter is easily put into practice:

```
/-/ { ... }
```

To take the former into account, you must know that `awk` uses the `NF` to make available the number of fields found in a record. We just need to check whether this variable has the value 9. If this condition *and* the “starts with -” condition are fulfilled, the line will be considered. Thus:

```
#!/usr/bin/awk -f
# filesum2 -- Add file sizes
NF == 9 && /-/ {
    sum += $5
    count++
}
END {
    print sum, " bytes in " count " files"
}
```

The equality operator in `awk` (as in the C language) is spelled “==”, to avoid confusion with the assignment operator, “=”. As in C, “&&” is the logical AND operator; like in C, its right-hand side is only evaluated if the left-hand side evaluates to “true”, i. e., a non-zero value.



Compared to the shell this is exactly the other way round—the shell’s “if”, “while”, ... commands and its “&&” operator consider a *return value* of 0 a “success”.


`awk` expressions may contain, among others, the common basic arithmetic (“+”, “-”, “*”, and “/”) and comparison operators (“<”, “<=” (≤), “>”, “>=” (≥), “==”, and “!=” (≠)). There are also test operators for regular expressions, “~” and “!~”, which you can use to check whether a string matches (or does not match) a regular expression:

```
$1 ~ /a.*a.*a/ { ... }
```

executes the commands if and only if the first field contains at least three “a” characters.

Some other `awk` operators are not borrowed from the C language. You could view “\$” as a “field access operator”: “\$3” gives you the value of the current record’s third field (if available), but “\$NF” always returns the *last* field’s value, regardless of the number of fields in the current record. Two strings (or variable values) can be concatenated simply by writing them next to each other (separated by a space):

```
$ awk '{ print $1 "blimey" }'
Gor
Gorblimey
```

 Compared to the C language, awk is missing the bitwise logical operators “|”, “&”, and “^” as well as the shift operators “<<” and “>>”. (If you want to push bits around, you will, for better or worse, have to use C.) (Or Perl.) There *is* a “^” operator in awk, but it stands for exponentiation.

(A complete list of awk operators can be found in the awk documentation.)

awk variables As we said, awk variables are “typeless”, i. e., they can hold character strings or numbers and are interpreted as required. At least as far as possible:

```
$ awk 'BEGIN { a = "123abc"; print 2*a; exit }'
246
$ awk 'BEGIN { a = "abc"; print 2*a; exit }'
0
```

Variable names always start with a letter and may otherwise contain letters, digits, and the underscore (“_”).

system variables Besides NF, awk defines some other “system variables”: FS is the “input field separator”, which you can set using the -F option (an assignment to “FS” within a BEGIN command will do as well). RS is the “input record separator”, i. e., the character that marks the end of a record. This is usually the newline character, but nothing prevents you from selecting something else. The special value “” stands for an empty line.—This makes it easy to process files that are “block structured” rather than “line structured”, such as the following:

```
Vernon
Dursley
4 Privet Drive
Little Whinging
XY12 9PQ
(01234) 56789

Figg
Arabella
12 Wisteria Walk
Little Whinging
XY12 9PR
(01234) 98765
<<<<<<
```

You just need to set FS and RS to appropriate values:

```
#!/usr/bin/awk -f
# Output a telephone list
BEGIN { FS = "\n"; RS = "" }
{ print "$1 $2", $NF }
```

arrays In addition to “simple” variables, awk also supports arrays, i. e., indexed groups of variables sharing a name. You have already encountered arrays in the Bourne-Again shell—consider the `dwwtb` script—, but unlike the shell, awk allows arrays to be indexed using arbitrary character strings rather than just numbers. This type of array is often called an “associative array”. This is an extremely powerful tool, as the following script shows:

```
#!/usr/bin/awk -f
# shellusers -- shows who is using which shell
BEGIN { FS = ":" }
      { use[$NF] = use[$NF] " ", " $1 }
END   {
      for (i in use) {
          print i ": " use[i]
      }
  }
```

If you invoke `shellusers` with `/etc/passwd` as a parameter, it outputs a list of login shells together with their respective users:

```
# shellusers /etc/passwd
/bin/sync: sync
/bin/bash: root anselm tux
/bin/sh: daemon bin sys games man lp mail news uucp proxy>
< postgres www-data backup operator list irc gnats nobody
/bin/false: hermes identd mysql partimag sshd postfix>
< netsaint telnetd ftp bind
```

In the script, the command sequence without a regular expression serves to collect the data, while the `END` command outputs it; the `for` command introduces a loop in which the `i` variable is set to every index of the `use` array in turn (the order is nondeterministic).

`awk` expressions may also refer to functions. Some functions are predefined in `awk`, including the arithmetic functions “`int`” (determine the integer part of a number), “`sqrt`” (square root), or “`log`” (logarithm). `awk`’s capabilities are roughly equivalent to those of a scientific calculator. There are also string functions: “`length`” determines the length of a string, “`substr`” returns arbitrary substrings, and “`sub`” correspond to `sed`’s “`s`” operator.

You can also define your own functions. Consider the following example: user-defined functions

```
#!/usr/bin/awk -f
# triple -- multiply numbers by 3

function triple(n) {
    return 3*n
}

{ print $1, triple($1) }
```

This program reads a file of numbers (one per line) and outputs the original number and that number tripled:

```
$ triple
3
3 9
11
11 33
```

A function’s “body” may consist of one or more `awk` commands; the `return` command is used to return a value as the function’s result.

The variables mentioned in a function’s parameter list (here, `n`) are passed to the function and are “local” to it, i. e., they may be changed but the changes are invisible outside the function. All other variables are “global”—they are visible everywhere within the `awk` program. In particular, there is no provision in `awk` for local variables

defining extra local variables within a function. However, you can work around this by defining some extra function “parameters” which you do not use when actually calling it. By way of illustration, here is a function that sorts the elements of an array *F*, which uses numerical indices between 1 and *N*:

```
function sort(F, N, i, j, temp) {
    # Insertion sort
    for (i = 2; i <= N; i++) {
        for (j = i; F[j-1] > F[j]; j--) {
            temp = F[j]; F[j] = F[j-1]; F[j-1] = temp
        }
    }
    return
}
```

for loop The for loop first executes its first argument (*i* = 2). Then it repeats the following: It evaluates its second argument (*i* <= *N*). If the result of this is “true” (non-zero), the loop body (here a second for loop) is executed, followed by the third argument (*i*++). This is repeated until the second argument evaluates to 0.—This function would be called like

```
{
    a[1] = "Gryffindor"; a[2] = "Slytherin"
    a[3] = "Ravenclaw"; a[4] = "Hufflepuff"
    sort(a, 4)
    for (i = 1; i <= 4; i++) {
        print i ": " a[i]
    }
}
```

Note the output of the array’s elements by means of a “counting” for loop; a “for (*i* in *a*)” loop would have produced the elements in a nondeterministic order (so there would have been no point in sorting them first).

7.4 awk in Practice

Here are some more awk examples from “real life”:

Compressing shell history (From the GNU awk manual.) The Bourne-Again shell stores your commands in the `~/bash_history` file—if you execute the same command repeatedly, it will be stored multiple times. Assume that you want to shrink this file by storing every command just once. There is the `uniq` command, of course, but it only removes duplicates that occur in immediate succession and works best with pre-sorted input; however, within the shell history, the order of commands should remain essentially constant. In other words, with an input file like

```
abc
def
abc
abc
ghi
def
def
ghi
abc
```

we do not aim for `uniq`’s output


```
abc
def
abc
ghi
def
ghi
abc
```

but for something like

```
abc
def
ghi
```

awk's associative arrays make this easy: We count the number of occurrences of each line in an associative array called `data`, which is indexed by the complete text of the command. The order is preserved using a second, numerically indexed array `lines`, whose indices we later use to output the lines in their correct order:

```
#!/usr/bin/awk -f
# histsort -- Compactify a shell history file

{
    if (data[$0]++ == 0) {
        lines[++count] = $0
    }
}
END {
    for (i = 1; i < count; i++) {
        print lines[i]
    }
}
```

Note in particular that awk also supports if conditionals; their syntax is (as usual) modelled on the C language:

```
if (<condition>) {
    <commands>
} [else { <commands>
}]
```

The `data[$0]++ == 0` expression is a common idiom; it is “true” exactly if `“$0”`'s value is seen for the first time. The `“++count”` expression is equivalent to `“count++”`, except that it returns the value of `count` *after* it has been incremented (`“count++”` returns the value *before* incrementing it); this ensures that the first line seen has index 1, even though we do not set `count` to 1 explicitly.



The basic structure of this program can be profitably used for other purposes; for example, the command

```
print data[lines[i]], lines[i]
```

produces a list of lines together with the number of their occurrences. Applied to the shell history, this tells you how often you have used each command.

Duplicate words A common error in documents are accidental duplications of words, such as “The result of the the program is ...”. Sometimes the first of the two words occurs at the end of a line and the second at the beginning of the next, which makes them hard to find. Here is an `awk` script which helps with this—it considers the words of each line, and also stores the last word of every line for comparison to the first of the next.

For simplicity, we assume that the input consists of lowercase letters only and that all non-letters have been converted to spaces—not a big restriction, since this is easily done using something like the

```
tr '[:upper:]' '[:lower:]' | tr -cs '[:alpha:]' ' '
```

pipeline¹ (it could as well be done in `awk`, or GNU `awk` at any rate, but we will spare you the details). The `awk` script itself looks like

```
#!/usr/bin/awk -f
# dupwords -- find duplicate words

{
    if (NF == 0) {
        next
    }
    if ($1 == prev) {
        printf("%s:%d: %s duplicate\n", FILENAME, FNR, $1)
    }
    for (i = 2; i <= NF; i++) {
        if ($i == $(i-1)) {
            printf("%s:%d: %s duplicate\n", FILENAME, FNR, $i)
        }
    }
    prev = $NF
}
```

Here, `printf` is equivalent to the eponymous shell or C library command for formatted output: The `%s` and `%d` formatting keys will be replaced by the corresponding arguments as strings (for `%s`) or numbers (`%d`), thus the first `%s` by the value of `FILENAME` (the name of the current input file), the `%d` by the line number within the current input file (`FNR`), and the second `%s` by the word in question. Also note the reference to the “previous” word using “`$(i-1)`”: When coming from the shell, it is easy to subscribe to the fallacy that `$` is merely a prefix for variable names. In fact, `$` in `awk` is a genuine operator for input field access—whatever follows it is evaluated, and the result is the actual number of the field to be fetched.

King Soccer Here is a topic that may appeal to many of you: the (German) national soccer league (*Bundesliga*). Consider the file `bl03.txt` containing the results of all the first division’s games from the 2003/4 season:

```
1:Bayern München:Eintracht Frankfurt:3:1:6300
1:Schalke 04 Gelsenkirchen: Borussia Dortmund:2:2:61010
1:Hamburger SV:Hannover 96:0:3:53224
1:Bayer Leverkusen:SC Freiburg:4:1:22500
1:Hertha BSC Berlin:Werder Bremen:0:3:40000
1:1.FC Kaiserslautern:1860 München:0:1:35629
1:VfL Wolfsburg:VfL Bochum:3:2:20000
1:Hansa Rostock:VfB Stuttgart:0:2:23500
1: Borussia Mönchengladbach:1.FC Köln:1:0:34500
```

¹Traditionally, this would be “`tr A-Z a-z | tr -cs a-z ' '`”; however, the version using POSIX character classes also works for non-English text.

```
2:VfL Bochum:Hamburger SV:1:1:20400
2:Borussia Dortmund:VfL Wolfsburg:4:0:72500
2:1860 München:Schalke 04 Gelsenkirchen:1:1:33000
2:1.FC Köln:1.FC Kaiserslautern:1:2:33000
<<<<<
```

The first field gives the round, the second and third the opposing teams, the fourth and the fifth the number of goals scored on each side, and the sixth the number of spectators in the stadium.

Let us start with something straightforward: How many spectators have trekked to the stadiums for the first round of games? You just need to add the last columns of all the lines referring to that round:

```
#!/usr/bin/awk -f
# spectators -- Adds spectators for the first round

$1 == 1 { z += $6 }
END { print z }
```

Hence:

```
$ awk -F: -f spectators bl03.txt
296663
```

If you are interested in the number of spectators for an arbitrary round, you can pass that round's number as a parameter. The adding expression then looks a bit different: awk variables on the command line

```
$1 == round { z += $6 }
```

And the invocation is:

```
$ awk -F: -f spectators round=2 bl03.txt
257200
```

You can include assignments to `awk` variables on the command line, among the `awk` options and file name arguments. The only condition is that `awk` gets to see each assignment as a single argument—hence there may be no spaces around the “=”.

So what does the federal league's standings table look like after the n -th round? This requires some more work: For every match we must decide which side has won, and calculate the points and goals difference.



The rules say that the winning side gets 3 points and the losing side none; if a match results in a draw, each side gets 1 point. If two teams have the same number of points, their position in the standings is determined by the goals difference.

This might look like this:

```
BEGIN { FS = ":"; OFS = ":" }
$1 <= round {
  if ($4 > $5) {
    points[$2] += 3
  } else if ($4 < $5) {
    points[$3] += 3
  } else {
    points[$2]++; points[$3]++
  }
  goals[$2] += $4 - $5; goals[$3] += $5 - $4
}
```

output field separator (The `OFS` variable is the “output field separator”, i. e., the string that `awk` puts between expressions in commands such as “print a, b” (with a comma).) We tabulate the points in the `points` array and the goal differences in the `goals` array, both indexed by the team names. We can output the standings using something like

```
END {
    for (team in points) {
        print team, points[team], goals[team]
    }
}
```

For example:

```
$ awk -f bltab round=1 bl03.txt
1.FC Kaiserslautern:0:-1
Borussia Mönchengladbach:3:1
Bayer Leverkusen:3:3
Bayern München:3:2
Hansa Rostock:0:-2
Borussia Dortmund:1:0
Hertha BSC Berlin:0:-3
Hamburger SV:0:-3
Schalke 04 Gelsenkirchen:1:0
VfL Wolfsburg:3:1
```

You do not need to be a soccer buff to figure out that there is evidently something wrong with this. For one, the table is obviously not sorted correctly (which is no surprise, due to “team in points”)—but are there not in fact more than 10 teams in the federal league? Apparently our program omits some teams, and after a moment’s thought it should become obvious to you what is going on: The `points` array contains only those teams that actually did win points in the league, and after the first round of games this does not usually include all teams (if we had gone for the final standings immediately, this error might not have even occurred to us). Thus we need to ensure that the losing sides get an entry in `points` as well, and this is most easily done by means of a “useless” addition farther up:

```
<<<<<<
    if ($4 > $5) {
        points[$2] += 3; points[$3] += 0;
    } else if ($4 < $5) {
        points[$2] += 0; points[$3] += 3;
    } else {
<<<<<<
```

This results in the (at least numerically correct) standings

```
1.FC Kaiserslautern:0:-1
VfB Stuttgart:3:2
1.FC Köln:0:-1
Borussia Mönchengladbach:3:1
Bayer Leverkusen:3:3
Eintracht Frankfurt:0:-2
Bayern München:3:2
Hansa Rostock:0:-2
1860 München:3:1
Werder Bremen:3:3
SC Freiburg:0:-3
Borussia Dortmund:1:0
VfL Bochum:0:-1
```

```
Hertha BSC Berlin:0:-3
Hamburger SV:0:-3
Schalke 04 Gelsenkirchen:1:0
VfL Wolfsburg:3:1
Hannover 96:3:3
```

which only needs to be sorted.

Occupied disk space per Linux group Would you like to find out which primary group on your system contains the biggest disk space hogs? A command like “`du -s /home/*`” will tell you the disk space used by individual users, but says nothing about groups. For this you need to correlate the user-based list with the password file. Given the `du` output format

```
1234 /home/anselm
56 /home/tux
567 /home/hugo
342 /home/emil
```

a corresponding `awk` script looks roughly like this:

```
BEGIN {
    readgroups()
}
{
    sub(/\char"0302.*\//, "", $2)
    used[usergroup[$2]] += $1
}
END {
    for (g in used) {
        print used[g] " " g
    }
}
```

The “`readgroups()`” function (which we are going to show presently) constructs an array called `usergroup` giving the name of the primary group for each user name. This array is used to tabulate the disk space used by the group members’ home directories in the `used` array (all users sharing a value in `usergroup` are members of the same primary group). We obtain the user name from `du` output by using `sub` to remove everything from the first to the last slash, which does look a bit messy. At the end, the groups and their amounts of occupied disk space are output.

Now for the “`readgroups()`” function:

```
# Read the users' group names to USERGROUP (indexed by user names).
# GROUPNAME and OLDFS are local variables.

function readgroups(groupname, oldfs) {
    oldfs = FS
    FS = ":"
    while (getline <"/etc/group") {
        groupname[$3] = $1
    }
    close ("/etc/group")

    while (getline <"/etc/passwd") {
        usergroup[$1] = groupname[$4]
    }
    close ("/etc/passwd")
}
```





```

    FS = oldfs
}

```


Here you will learn about awk's file access functions: `getline` tries to read a line from the specified file and returns 1 if there was another line to read, or 0 at the end of the file—the file will be opened on the first `getline` and then just read later on. With `close`, you can close a file after use; this is seldom really required but a good habit to get into, since many awk implementations can only use a very limited number of files at the same time (10 or so). Also note that we need to redefine awk's field separator to read `/etc/group` and `/etc/passwd`; we remember the original value in `oldfs` and restore it at the end of the function so that the calling program will not be confused.

Exercises

-  **7.1** [!2] Write an awk program which counts the words in a document and outputs them together with the number of their occurrence.
-  **7.2** [!1] How can you sort the unsorted federal league standings produced by the `bltab` program in a suitable manner?
-  **7.3** [!2] Write an awk program that adds up and outputs the stadium spectators for each federal league soccer team.
-  **7.4** [3] Write a script (preferably using awk and sort) that outputs a “beautiful” table of standings like

RD	TEAM	GM	W	D	L	POINTS	GD
1	Werder Bremen	34	22	8	4	74	41
2	Bayern München	34	20	8	6	68	31
3	Bayer Leverkusen	34	19	8	7	65	34
4	VfB Stuttgart	34	18	10	6	64	28
5	VfL Bochum	34	15	11	8	56	18
6	Borussia Dortmund	34	16	7	11	55	11
7	Schalke 04 Gelsenkirchen	34	13	11	10	50	7
8	Hamburger SV	34	14	7	13	49	-13
9	Hansa Rostock	34	12	8	14	44	1
10	VfL Wolfsburg	34	13	3	18	42	-5
11	Borussia Mönchengladbach	34	10	9	15	39	-9
12	Hertha BSC Berlin	34	9	12	13	39	-17
13	1.FC Kaiserslautern	34	11	6	17	39	-23
14	SC Freiburg	34	10	8	16	38	-25
15	Hannover 96	34	9	10	15	37	-14
16	Eintracht Frankfurt	34	9	5	20	32	-17
17	1860 München	34	8	8	18	32	-23
18	1.FC Köln	34	6	5	23	23	-25

(where RD stands for “round”, GM for “games played”, W, D, L for “won”, “drawn”, and “lost”, respectively, and GD for “goals difference”)² (*Hint*: Read up on `printf` in the awk manual).

-  **7.5** [2] If you compare the output of Exercise 7.4 to the actual final standings of the 2003/4 federal league season, you will find a discrepancy: The “genuine” ranks 13 to 15 look like

13	SC Freiburg	34	10	8	16	38	-25
14	Hannover 96	34	9	10	15	37	-14
15	1.FC Kaiserslautern	34	11	6	17	37	-23

²Supporters of SC Freiburg and Hannover 96: please refer to Exercise 7.5.

since the Betzenberg braves incurred a 2-point penalty because of licensing violations. How can you take this into account in your standings software?



7.6 [2] Write an awk program which reads the output of “du -s /home/*” and displays the amount of disk space used by each home directory “graphically”, like

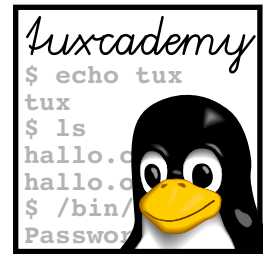
```
hugo      *****
emil      *****
tux       ***
anselm    *****
```

Commands in this Chapter

awk	Programming language for text processing and system administration	awk(1) 98
uniq	Replaces sequences of identical lines in its input by single specimens	uniq(1) 104

Bibliography

- AKW88** Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988. ISBN 0-201-07981-X. <http://cm.bell-labs.com/cm/cs/awkbook/>
- DR97** Dale Dougherty, Arnold Robbins. *sed & awk*. Sebastopol, CA: O'Reilly & Associates, 1997, second edition. ISBN 1-56592-225-5. <http://www.oreilly.de/catalog/sed2/>
- Rob02** Arnold Robbins. *sed & awk Pocket Reference*. Sebastopol, CA: O'Reilly & Associates, 2002, second edition. ISBN 0-596-00352-8. <http://www.oreilly.com/catalog/sedawkrepr2/>



8

SQL

Contents

8.1	Foundations of SQL	114
8.1.1	Summary	114
8.1.2	Applications of SQL.	115
8.2	Defining Tables	117
8.3	Data Manipulation and Queries	118
8.4	Relations	123
8.5	Practical Examples	125

Goals

- Understanding the applications of SQL
- Defining simple tables with SQL
- Manipulating data and forming queries

Prerequisites

- Knowledge of the basic Linux commands
- Basic text editing skills

8.1 Foundations of SQL

8.1.1 Summary

The “Structured Query Language” (SQL) is a standard language for defining, querying, and manipulating relational databases. Relational databases store data tuples records (also called “tuples” in tech-speak) in “tables”. You can visualise a table tables by imagining a large sheet of paper which is divided into rows and columns. The rows are the records stored in the table, and the columns describe the properties of the records (Table 8.1). The database makes it convenient to retrieve those tuples that match specific criteria (like the family names of all persons who command a starship named “USS Enterprise”).

normalisation What makes the whole thing interesting is a concept called “normalisation”. If you take a closer look at the initial example, you will notice that the names of some persons, ships, and films occur multiple times. This is not desirable, because modifications to these data would have to be applied in multiple places inside the database. There is a danger of missing one place and introducing inconsistent data. Instead, the database is “normalised”: We know that the “James T. Kirk” in the first two tuples is actually one and the same person, but the “USS Enterprise” in the first three tuples and the one in the fourth tuple are different vessels¹. So we can split our table into three, one each for the people, ships, and films. This can be seen in Table 8.2. Please note the following remarked:

- foreign key • The original table has morphed into a new table named “Person”, which no longer contains the columns holding the ship names and the films. Instead, the name of a ship is given using a “foreign key” that refers to a tuple of the “Ship” table. This foreign key us to express the fact that both Willard Decker and James T. Kirk commanded the “old” USS Enterprise, while Jean-Luc Picard commanded the “new” one. This is called a “1 : n relationship”, because the same ship may have multiple commanding officers during the course of its existence.
- The same person may appear in different films and the same film may feature multiple people from the “Person” table. This is why the relationship between people and films cannot be expressed through a simple foreign key in the “Person” table. (What we have here is called an “m : n relationship”). Instead, we introduce yet another table whose tuples represent propositions of the form “person *x* appears in film *y*”.
- We added a few columns to the “Film” table just to make things a bit more interesting.



For simplicity’s sake, we ignore the fact that one of our people might command several different spaceships during the course of their career.

¹Well, “trekkies” like us know it, but *not* knowing it may not actually make you a bad person.

First Name	Surname	Starship	Film
James T.	Kirk	USS Enterprise	Star Trek
James T.	Kirk	USS Enterprise	Star Trek: Generations
Willard	Decker	USS Enterprise	Star Trek
Jean-Luc	Picard	USS Enterprise	Star Trek: Generations
Han	Solo	Millennium Falcon	Star Wars 4
Han	Solo	Millennium Falcon	Star Wars 5
Wilhuff	Tarkin	Death Star	Star Wars 4
Malcolm	Reynolds	Serenity	Serenity

Figure 8.1: A database table: Famous spaceship commanders from films

Person	First Name	Surname	Ship	PersonFilm	Person	Film
1	James T.	Kirk	1	1	1	1
2	Willard	Decker	1	2	1	2
3	Jean-Luc	Picard	2	3	2	1
4	Han	Solo	3	4	3	2
5	Wilhuff	Tarkin	4	5	4	3
6	Malcolm	Reynolds	5	6	4	4
				7	5	3
				8	6	5

Ship	Name	Film	Title	Year	Budget (Million \$)
1	USS Enterprise	1	Star Trek	1979	46
2	USS Enterprise	2	Star Trek: Generations	1994	35
3	Millennium Falcon	3	Star Wars 4	1977	11
4	Death Star	4	Star Wars 5	1980	33
5	Serenity	5	Serenity	2005	39

Figure 8.2: Famous spaceship commanders from films (normalised)

At first glance, this “data model” may appear somewhat more complicated, but it does store every piece of information in only one place, which makes it a lot easier to keep things under control in “real life”.



Relational databases were originally proposed by Edgar F. Codd in 1970. Even today they form the backbone of computer-based data processing. Relational databases can reflect the object-oriented data structures of modern software only to a limited degree, but unlike fancy new approaches like “object-oriented databases” they have the clear advantage of being based on sound mathematical theory (“relational algebra”), as well as being amenable to reasonably efficient implementation.



The first version of SQL (then still called SEQUEL) was developed in the early 1970s by Donald D. Chamberlin and Raymond F. Boyce. It formed the basis of IBM’s first relational database system, System R. SQL was standardised for the first time in 1986, but development continued afterwards. The current version of the standard is ISO 9075:2008 (popularly known as ISO SQL:2008). Unsurprisingly, it was ratified in July 2008.



The official pronunciation of SQL is “S-Q-L”. Occasionally people will also pronounce it like the word “sequel”.

Exercises



8.1 [!2] How would you add some additional ship crew members to the data model defined in this section?




8.2 [2] Where in the data model would you place the director of a film?

8.1.2 Applications of SQL


As we said before, SQL is an essential part of today’s commercial data processing—a whole industry thrives on implementing and supporting products implementing relational database systems. Here are some SQL-based relational database products available for Linux:


MySQL and PostgreSQL These two packages are probably the first ones that come to mind when thinking of the terms “Linux” and “SQL”. Both are

freely available and quite popular and form a solid basis for common, not overly complex applications, for example in the area of the World Wide Web.

 As usual in the open-source community, there are vigorous “holy wars” between the advocates of both packages. PostgreSQL disciples decry the fact that MySQL does not implement all of the SQL standard, while MySQL proponents argue that the parts that MySQL *does* implement are entirely adequate while the speed of MySQL makes it easy to forgo the remainder. We shall not give a recommendation either way here; both products are freely available and can be evaluated on their merits as required.


Oracle, Sybase, DB2, and friends There is a whole bunch of commercially implemented and supported database systems for “mission-critical” applications that (also) run on Linux. These systems offer all the features of implementations based on Windows or traditional Unix systems and can be used without hesitation for all kinds of large-scale database applications.


 While you can run MySQL and PostgreSQL on essentially any Linux system, the commercial database manufacturers usually limit their official support to a number of specific platforms, typically the “enterprise” distributions from companies like Red Hat and Novell/SUSE, on which they “certify” their products—this means that the manufacturer tests the product thoroughly on the platform in question, proclaims that it works, and is subsequently prepared to help paying customers that run exactly that platform if they experience problems. Naturally you are free to get Oracle and friends to run on other Linux distributions than the officially certified ones, and chances are good that that will work (it’s not as if Linuxes were *that* different from one another). However, you’d be on your own in case of trouble, which does cast some doubt on why you would want to use an expensive commercial database system in the first place—since for most applications you could resort to MySQL and PostgreSQL, too.

 Incidentally, it is not a big problem to obtain “commercial” support for PostgreSQL and MySQL, too (at commercial rates).


SQLite While the other packages typically provide a “database server” in a separate process to which application programs connect over the network, SQLite is linked directly to application programs as a library, is usable without configuration, and reads and writes local files. SQLite supports most of the SQL92 standard, including features like transactions and triggers which may be problematic even with MySQL. SQLite is suitable for use on low-memory devices such as MP3 players or as a data format for application programs.

Throughout the rest of this chapter we will use SQLite to illustrate SQL.

 On Debian GNU/Linux or Ubuntu, you can easily install SQLite by using one of the following commands:

	<code># aptitude install sqlite3</code>	<i>for root</i>
	<code>\$ sudo aptitude install sqlite3</code>	<i>for other users</i>
	<code>\$ sudo apt-get install sqlite3</code>	<i>on Ubuntu</i>

Make sure to get `sqlite3`—there is also `sqlite`, which is an obsolete version which is only still offered for compatibility.

 On SUSE, SQLite (3) is part of the default installation, just like on the Red Hat distributions. So you do not need to do anything special to try the examples in this chapter—everything you need is already installed.



```

CREATE TABLE person (
  id      INTEGER PRIMARY KEY,
  firstname VARCHAR(20),
  surname  VARCHAR(20),
  ship_id  INTEGER
);

CREATE TABLE film (
  id      INTEGER PRIMARY KEY,
  title   VARCHAR(40),
  year    INTEGER,
  budget  INTEGER
);

CREATE TABLE ship (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(20)
);

CREATE TABLE personfilm (
  id      INTEGER PRIMARY KEY,
  person_id INTEGER,
  film_id  INTEGER
);

```

Figure 8.3: The complete schema of our sample database

Exercises



8.3 [2] Under which circumstances would you use a freely available SQL database product like MySQL or PostgreSQL for a web site? Does your evaluation depend on the nature of the web site, i.e. whether it is a hobby project or part of a mission-critical task?



8.4 [2] The authors of SQLite recommend SQLite to store application program data (think of the tables of a spread sheet or the configuration data of a web browser). Which advantages and disadvantages of the approach can you think of?

8.2 Defining Tables

Before you can fill an SQL database with data, you have to specify the names of the individual tables, the names of the columns, and the nature of the values to be stored in a column. SQL supports a large variety of data types like “string” or “integer” that you can resort to when defining the columns of a table. All the table definitions of a database together are incidentally called a “database schema”.



A discussion of the full SQL language standard is beyond the scope of this document. With table definition in particular there are also large differences between the various SQL databases. We limit our discussion to the absolute minimum, also because *creating* tables is not part of the LPI-102 exam.


In SQL syntax, a definition of the “Person” table from our example might look like


```

CREATE TABLE person (
  id      INTEGER PRIMARY KEY,
  firstname VARCHAR(20),
  surname  VARCHAR(20),
  ship_id  INTEGER
);

```

INTEGER and VARCHAR(20) denote the SQL data types “integer” and “string of up to 20 characters”. The “PRIMARY KEY” clause declares the id column, which corresponds to the “running count” of persons in the example table of Table 8.2, the “Rand[Primary Key]primary key”. This means that the database ensures that any value in this column occurs only once, and the values here can serve as “foreign keys” in tuples from other tables (in the case of “person”, for example, the table joining people and films).

 It is a common convention to give foreign keys the name of the table they “point to”, with a suffix of `_id`, so `ship_id` is a foreign key that refers to the `ship` table.

 If you are somewhat familiar with SQL, you may object to our defining the foreign key, `ship_id`, as a mere `INTEGER`. Please allow us this simple view of things for today.


 Incidentally, SQL does not distinguish between uppercase and lowercase characters. We follow the common convention of putting the names of tables and columns in lowercase and everything that is proper SQL in uppercase, but you can basically suit yourself. However you will do yourself and us a favour by being consistent with yourself.


Figure 8.3 shows the complete SQL schema of our sample database. If the schema is stored in the `commanders-schema.sql` file, you could initialise the actual database using SQLite as follows:

```
$ sqlite3 comm.db <commanders-schema.sql
```


This command stores the database in the `comm.db` file. SQLite always takes the name of a database file as a parameter; if the database file exists already, it will be opened, otherwise it will be created.


When you run `sqlite3` without redirecting standard input, you get an interactive session:

```
$ sqlite3 comm.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .tables
film      person   personfilm  ship
sqlite> .schema ship
CREATE TABLE ship (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(20),
);
sqlite> _
```

 SQLite features various “metacommands” whose names all begin with a dot—in the example you can see `.tables`, which lists the tables in a database, and `.schema`, which lets you inspect the database schema. There are many more, though; `.help` gets you the complete list.

Exercises

 **8.5** [!2] Create an SQLite database based on the spaceship commander database schema given in this section.

 **8.6** [3] Implement the extensions from Exercise 8.1 and Exercise 8.2 as an SQL schema.

8.3 Data Manipulation and Queries

SQL not only supports defining database schemas, but also inserting, modifying, querying, and deleting data (tuples). All of this is subject to the current database schema. For example, you might add a few ships and films to our database:

```
sqlite> INSERT INTO ship VALUES (1, 'USS Enterprise');
sqlite> INSERT INTO ship VALUES (2, 'USS Enterprise');
sqlite> INSERT INTO film VALUES (1, 'Star Trek', 1979, 46);
sqlite> INSERT INTO film VALUES (2, 'Star Trek: Generations',
  ..> 1994, 35);
```



Note that we specify explicit values for the primary keys here. In a larger database this is somewhat tedious, since you would have to figure out the right value for every new tuple—it is much more convenient to have the database itself insert the correct value, and most SQL database can in fact do this. With SQLite, the primary key must be declared as an “INTEGER PRIMARY KEY”, and you must pass the “magical” value NULL instead of an explicit value:

```
sqlite> INSERT INTO film VALUES (NULL, 'Star Wars 4', 1977, 11);
```

People and tuples specifying the person-film relation can be entered likewise:

```
sqlite> INSERT INTO person VALUES (1, 'James T.', 'Kirk', 1);
sqlite> INSERT INTO person VALUES (2, 'Willard', 'Decker', 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 2);
sqlite> INSERT INTO personfilm VALUES (NULL, 2, 1);
```

Note how we specify the primary keys of the tuples in the corresponding tables for the foreign keys `ship_id` in the `person` table and `person_id` and `film_id` in the `personfilm` table.



If you know a bit about programming, this may make you feel somewhat queasy. After all, nobody guarantees that there actually *is* a tuple with the corresponding primary key in the “other table”². This is not a fundamental problem with SQL but rather one with our simplistic examples—“good” SQL databases (not SQL, and MySQL not always) support a concept called “referential integrity” which helps solve exactly this problem. It makes it possible to specify in the schema that `ship_id` is a foreign key to the `ship` table, and the database will then ensure that the values for `ship_id` remain reasonable. Referential integrity incorporates other nice properties, too; in our simple example you yourself would have to take care, when you remove the James T. Kirk tuple from the `person` table, to also remove the tuples from `personfilm` that connect James T. Kirk to films. With a database supporting referential integrity, this could happen automatically.

referential integrity

With the data from Table 8.2 in our database we can now look at a few queries. We can obtain all tuples from a table like this:

queries
all tuples

```
sqlite> SELECT * FROM ship;
1|USS Enterprise
2|USS Enterprise
3|Millennium Falcon
4|Death Star
5|Serenity
```

The asterisk (“*”) implies “all columns”. If you want to limit your selection to specific columns, you have to enumerate them:

specific columns

```
sqlite> SELECT firstname, surname FROM person;
James T.|Kirk
```

²Unless you are a C programmer, that is; in that case there is nothing wicked about this at all.

```
Willard|Decker
Jean-Luc|Picard
Han|Solo
Wilhuff|Tarkin
Malcolm|Reynolds
```

Expressions You are not restricted to retrieving column values exactly the way they are stored in the database, but can form “expressions” based on them. The following example lists the full names of the ship commanders without the ugly vertical bar. The “||” operator concatenates two strings.

```
sqlite> SELECT surname || ', ' || firstname FROM person;
Kirk, James T.
Decker, Willard
Picard, Jean-Luc
Solo, Han
Tarkin, Wilhuff
Reynolds, Malcolm
```

Of course you can do calculations, too:

```
sqlite> SELECT title, budget * 0.755 FROM film;
Star Trek|34.73
Star Trek: Generations|26.425
Star Wars 4|8.305
Star Wars 5|24.915
Serenity|29.445
```

(Whether it makes a lot of sense to convert 1977 U. S. dollars to euros at the January 2009 exchange rate is a different question, though.)

Aggregate functions “Aggregate functions” make it possible to apply operations like sums and averages to particular columns of all tuples. For instance, you calculate the number and the average budget of all films in our database (disregarding inflation) as follows:

```
sqlite> SELECT COUNT(budget), AVG(budget) FROM film;
5|32.8
```

Of course you only get a single tuple as the result.



The “COUNT(budget)” may surprise you a little, but it stands for “the number of all tuples in a table whose budget column actually contains a value”. It might be possible for the budget of a film to be unknown—“Star Trek”, for example, is a borderline specimen—, and in this case you could enter the NULL value there (not to be confused with “0” for a film which didn’t cost anything to produce). Such films will then simply be skipped when the aggregate function is calculated. If you want to know the number of all films, no matter whether their budget is known or not, you can say “COUNT(*)”.

grouping An interesting feature, especially when dealing with aggregate functions, is “grouping”. Assume we’re interested in the average budget of the films produced in a decade, for all decades in the database. We could use something like

```
sqlite> SELECT year/10, AVG(budget) FROM film GROUP BY year/10;
197|28.5
198|33.0
199|35.0
200|39.0
```


(This may not be the greatest possible output format, but it does what it is supposed to.) The “GROUP BY” clause specifies that all tuples for which year/10 gives the same result are to be considered together, and the column specifications then refer to all the tuples in one such group. This means that `AVG(budget)` no longer calculates the average of *all* tuples, but only that of the tuples in the same group.



So far the names of the output columns derived from the column names of the input tuples. SQL does allow you to request your own names for output tuples. Especially with more complex queries this can make things clearer:

```
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade;
```

is rather less tedious to read than the original.



How your output actually looks depends mostly on your database system. By default, SQLite is fairly simple-minded, which may be mostly due to the fact that, in the spirit of the “Unix toolchest”, it tries to produce output that is easily processed by other programs and free of superfluous chatter. For interactive use, though, you can select more convenient output formats:

```
sqlite> .headers on
sqlite> .mode tabs
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade
decade avg(budget)
197    28.5
198    33.0
199    35.0
200    39.0
```

Here the individual columns are separated by tabs. With “`.mode column`” you can obtain a format where you can assign explicit column widths using `.width`:

```
sqlite> .mode column
sqlite> .width 10 12
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade
decade      avg(budget)
-----
197         28.5
198         33.0
199         35.0
200         39.0
```

Frequently you do not want to work on all tuples from a table, but only a subset matching specific criteria. You can do this with `SELECT`, too. Here, for example, is selecting tuples the list of all films in our database that were produced since 1980:

```
sqlite> SELECT title, year FROM film WHERE year >= 1980;
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005
```

You can also sort the output:

```
sqlite> SELECT title, year FROM film WHERE year >= 1980
...> ORDER BY year ASC;
```

```
Star Wars 5|1980
Star Trek: Generations|1994
Serenity|2005
```

“ASC” here means “ascending”, the opposite would be “DESC”:

```
sqlite> SELECT title FROM film ORDER BY budget DESC;
Star Trek
Serenity
Star Trek: Generations
Star Wars 5
Star Wars 4
```

Sub-SELECTs The WHERE clauses of queries may contain SELECT]other SELECT commands as long as these deliver something that fits the selection expression in question. Here is the list of all films with an above-average budget:

```
sqlite> SELECT title, year FROM film
...> WHERE budget > (SELECT AVG(budget) FROM film);
Star Trek|1979
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005
```

modifying tuples You can modify tuples by means of the UPDATE command:

```
sqlite> UPDATE person SET firstname='James Tiberius' WHERE id=1;
sqlite> SELECT firstname, surname FROM person WHERE id=1;
James Tiberius|Kirk
```

The WHERE clause is extremely important in this case so changes apply to specific tuples. One slip and the disaster is perfect:

```
sqlite> UPDATE person SET firstname='James Tiberius';
sqlite> SELECT firstname || ' ' || surname FROM person;
James Tiberius Kirk
James Tiberius Decker
James Tiberius Picard
James Tiberius Solo
James Tiberius Tarkin
James Tiberius Reynolds
```

But of course you can put this to profitable use:

```
sqlite> UPDATE film SET budget=budget * 0.755; Convert to euros
```

deleting tuples Finally, you can use the DELETE FROM command to delete tuples from a table. The WHERE warning applies here, too:

```
sqlite> DELETE FROM person WHERE surname='Tarkin';
```

You can delete all tuples from a table using

```
sqlite> DELETE FROM person; Kids, don't try this at home
```



Remember our remark above concerning “referential integrity”. Depending on the mojo of your database system, you may have to ensure by yourself that tuples containing foreign keys to a tuple will disappear along with that tuple.

Exercises



8.7 [1] Insert all tuples of Table 8.2 into the SQLite database described in Exercise 8.5. If you are into science fiction films, feel free to extend the database a bit. (For example, we are big fans of “Galaxy Quest”.)



8.8 [2] Give an SQL command that lists all films in our sample database that were produced before 1985 and had a budget of less than 40 million dollars.

8.4 Relations

SQL queries get really interesting when you combine multiple tables. Really interesting SQL queries combine multiple tables. For example, you might be interested in a list of all spaceship commanders together with the names of their ships (the tuples in person only contain the primary keys of the ships):

```
sqlite> SELECT * FROM person, ship
...> WHERE person.ship=ship.id;
1|James T.|Kirk|1|1|USS Enterprise
2|Willard|Decker|1|1|USS Enterprise
3|Jean-Luc|Picard|2|2|USS Enterprise
4|Han|Solo|3|3|Millennium Falcon
5|Wilhuff|Tarkin|4|4|Death Star
6|Malcolm|Reynolds|5|5|Serenity
```

That’s a bit thick, isn’t it? But let’s take it step by step:

- The secret to our success is, once again, the WHERE clause. It puts the foreign key of the person table in relation (Eek, the R-word!) to the primary key of the ship table and thus causes the corresponding tuples to be matched.
- The output looks a bit messy because each tuple of ship is simply appended to the matching tuple of person. In fact we do not need both ship_id from person and id from ship, if the next thing we output is the ship name, anyway. Something like

```
sqlite> SELECT firstname, surname, name FROM <<<<<<
James T.|Kirk|USS Enterprise
<<<<<<
```

would be completely adequate. However, this only works because all the columns have distinct names, and SQLite can infer which table each name refers to. If the surname column in person was simply called name, there would be a conflict with the name column of ship.

The most common method for resolving name conflicts and abbreviating long SQL commands at the same time is using aliases:

aliases

```
sqlite> SELECT * FROM person p, ship s WHERE p.ship_id=s.id;
```

This is equivalent to the original example except that, for this command, we gave the person table the alias p and the ship table the alias s. This did make the WHERE clause that much easier to read.



Aliases can be used in the column lists as well, as in

```
sqlite> SELECT p.firstname, p.surname, s.name <<<<<<
```

This also lets you handle name collisions between the columns of different tables:

```
sqlite> SELECT firstname, p.name, s.name <<<<<<
```

The example we showed for joining two tables works but is to be enjoyed with some caution (see also Exercise 8.9). When two tables are joined in this manner, the database system first constructs the Cartesian product of the tables in question and then throws out all resulting tuples that do not match the `WHERE` condition. This means that what happens is roughly this:

	<i>Condition: The fourth and fifth columns must match</i>
1 James T. Kirk 1 1 USS Enterprise	<i>OK; bingo; keep it</i>
1 James T. Kirk 1 2 USS Enterprise	<i>Doesn't match; throw away</i>
1 James T. Kirk 1 3 Millennium Falcon	<i>Oops ...</i>
<<<<<<	
4 Han Solo 3 2 USS Enterprise	<i>Not really ...</i>
4 Han Solo 3 3 Millennium Falcon	<i>OK; bingo; keep it</i>
4 Han Solo 3 4 Death Star	<i>Sigh</i>
<<<<<<	<i>Hours later</i>
6 Malcolm Reynolds 5 5 Serenity	<i>Fine, keep this (Whew.)</i>

In our toy example this isn't really a problem, but if you consider that the IRS might want to match tax payers to their employers, the dimensions are somewhat different.

This is why SQL lets you specify in advance which combinations of tuples you find interesting, instead of creating *all* possible combinations and then throwing out the uninteresting ones. This looks like

```
sqlite> SELECT *
...> FROM person JOIN ship ON person.ship_id=ship.id;
```

Result: see above

query optimisation



Whether this is a real problem in practice also depends on your database system. A large part of the development effort for a database system goes into “query optimisation”, which is the part that decides exactly how to evaluate `SELECT` commands. Clever database systems can figure out that the first example and the `JOIN` example do essentially the same thing, and handle both in the same (efficient) manner. SQLite, for example, generates the same byte code for both queries. On the other hand, both these queries are still very simple, and in real life you will have to deal with more complex queries that may tax a query optimiser to a point where it no longer notices obvious simplifications. We recommend you use the `JOIN` form just to be safe.

If you want to know which commander appeared in which film, you will have to consult the `personfilm` table:

```
sqlite> SELECT firstname, name, title
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=f.id;
```

```
James T.|Kirk|Star Trek
James T.|Kirk|Star Trek: Generations
Willard|Decker|Star Trek
Jean-Luc|Picard|Star Trek: Generations
Han|Solo|Star Wars 4
Han|Solo|Star Wars 5
Wilhuff|Tarkin|Star Wars 4
Malcolm|Reynolds|Serenity
```

So even relations between three (and more) tables are not a problem—you simply need to keep your eyes peeled!

Here are some more examples for relational queries. First the list of all commanders who appeared in films since 1980:

```
sqlite> SELECT year, title, firstname || ' ' || name
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=m.id
...> WHERE year >= 1980 ORDER BY year ASC;
1980|Star Wars 5|Han Solo
1994|Star Trek: Generations|James T. Kirk
1994|Star Trek: Generations|Jean-Luc Picard
2005|Serenity|Malcolm Reynolds
```

Here is a list of films featuring two or more commanders:

```
sqlite> SELECT title
...> FROM film f JOIN personfilm pf ON f.id=pf.film_id
...> GROUP BY film_id HAVING COUNT(*) > 1;
Star Trek
Star Trek: Generations
Star Wars 4
```

The “GROUP BY” clause causes tuples from personfilm that refer to the same film to be processed together. HAVING, (which we didn’t cover before) is similar to WHERE, but it is applied after grouping and allows the use of aggregate functions (which WHERE doesn’t); hence the COUNT(*) in HAVING clause counts the tuples in each group.

Exercises



8.9 [!1] What is the output of the SQL command

```
SELECT * FROM person, ship
```

when it is applied to our sample database?

8.5 Practical Examples

Now that you have looked into the basics of SQL, you may well wonder what all of this buys you in practice (unless you are working with databases already, in which case all of this chapter is probably old hat to you). In this section we present a few ideas of what to do with an SQL database system like SQL in “real life”.

Firefox As of version 3, Firefox (or, for Debian GNU/Linux users, “Iceweasel”) uses SQLite to manage an increasing number of its internal files. You can snoop around some of them and learn interesting things; anything in the `~/.mozilla/firefox/*.Default-User` directory (where the asterisk represents a code to make the name unique) with an extension of `.sqlite` is potentially fair game.

The `formhistory.sqlite` file, for example, contains the default values Firefox inserts into web forms. The database schema is rather obvious:

```
CREATE TABLE moz_formhistory (
  id INTEGER PRIMARY KEY,
  fieldname LONGVARCHAR,
  value LONGVARCHAR
);
```

So you can use

```
sqlite> SELECT value FROM moz_formhistory WHERE fieldname='address';
```

to find out what Firefox will propose to you if an input field in a web form has the (internal HTML) name address. Likewise, you have the possibility to systematically get rid of any default values that bug you (which Firefox itself doesn't offer)—a suitable `DELETE FROM` creates *faits accomplis*.



If you want to be on the safe side, do this when Firefox isn't running—but in principle it should work even if it is.

Also as of version 3, Firefox maintains a file named `places.sqlite`, which contains interesting things like your bookmarks (in `moz_bookmarks`), the sites you visited (in `moz_historyvisits` and `moz_places`), and much else.

Amarok Are you using the KDE music player, Amarok? If so, you can easily figure out your personal “tops of the pops” using SQL:

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \
> 'SELECT url, playcounter FROM statistics ORDER BY
> playcounter DESC LIMIT 10;'
```

The “LIMIT 10” clause at the end of the query limits the number of resulting tuples to at most 10. If you want to list the artist and title instead of the URL of the song file, the query is only a bit more complex;

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \
> 'SELECT title, playcounter FROM statistics s
> JOIN tags t ON s.url=t.url
> ORDER BY playcounter DESC LIMIT 10;'
```



Incidentally, a more convenient way to express the JOIN in the above query would be to use `JOIN tags USING(url)`, since in both tables the `url` column is used to make the connection.

Do look at the Amarok schema using `.schema`. You will surely be able to think of other interesting applications.

Poor Person's Diary If you are one of those people who tend to forget Auntie Mildred's birthday, you should attempt, for the sake of family peace, not to let this happen too often. It would be nice to be made aware of the upcoming family events and anniversaries—ideally with a little advance warning so you can still take care of gifts, cards, and so on.

Since you are familiar with the Unix shell, writing a small diary application to help you keep track should be a piece of cake. Of course we will not be able to compete with Evolution nor KOrganizer nor Google Calendar, but shall set our sights rather lower than that (but not too low).

The first thing we have to do is design the database schema. We would like to store different kinds of events (birthdays, anniversaries, etc) as well as, for each event, not just the category but also the date and an explanation (so we know what all of this is about). Here is a proposal for the category:

```
CREATE TABLE type (
  id INTEGER PRIMARY KEY,
  abbr VARCHAR(1),
  name VARCHAR(100)
);
```

*abbreviation
full text*

And here are the events themselves:

```
CREATE TABLE event (
  id          INTEGER PRIMARY KEY,
  type_id     INTEGER,
  year        INTEGER,
  date        VARCHAR(5),
  description VARCHAR(100)
);
```

foreign key

We store the date of each event separately; the year (year column) and the month and day in MM-DD format (date column), for reasons which will hopefully become clear soon. Entries are added to the diary as follows:

```
INSERT INTO type VALUES (1, 'B', 'Birthday');
INSERT INTO type VALUES (2, 'W', 'Wedding Anniversary');
INSERT INTO type VALUES (3, 'A', 'Anniversary');

INSERT INTO event VALUES (1, 1, 1926, '01-17', 'Auntie Mildred');
INSERT INTO event VALUES (2, 1, 1934, '01-21', 'Uncle Jack');
INSERT INTO event VALUES (3, 2, 2002, '02-05', 'Susie and Martin');
```

Now we can answer the pressing question: What will happen next week? In other words: If we “teleport” the dates from event into the current year, which of them fall into the range from “today” to “today plus seven days”? (We assume that one week is adequate to buy a card or a present. The stationery shop around the corner will surely have something in stock, and the likes of Amazon will, in a pinch, deliver stuff fairly quickly.)

Our task is simplified considerably by the date functions of SQLite, which we haven’t looked at before. The DATE() function, for example, expects as its first argument a date in the common “international” format (meaning first the year, then the month, and then the day, separated by hyphens—for instance, 2009-01-14) or the string now (with the obvious meaning). Any subsequent arguments “modify” the date given as the first argument. So you can obtain the point of time corresponding to “today plus seven days” simply using

```
sqlite> SELECT DATE('now', '+7 days');
2009-01-20                                Today, incidentally, is 13 January 2009
```

This pretty much settles it: We can find all “current” dates using a query like

```
sqlite> SELECT DATE('2009-' || date) AS d, description, year
...> FROM event
...> WHERE d >= DATE('now') AND d <= DATE('now', '+7 days')
2009-01-17|Auntie Mildred|1926
```



Instead of

```
d >= DATE('now') AND d <= DATE('now', '+7 days')
```

you could also write

```
d BETWEEN DATE('now') AND DATE('now', '+7 days')
```

You can now wrap the query up nicely in a shell script, which will figure out the current year (somewhat cumbersome in SQL) and formats the output nicely (*quite* cumbersome in SQL). The result might look like figure 8.4, and together with a database file in ~/.cal.db might produce output like following form:

```
#!/bin/bash
# calendar-upcoming [limit]

caldb=$HOME/.cal.db
year=$(date +%Y)
limit=${1:-14}


sqlite3 $caldb \
  "SELECT DATE('$year-' || date) AS d, name, description, year
   FROM event JOIN type ON event.type_id=type.id
   WHERE d >= DATE('now') AND d <= DATE('now', '+$limit days')
   ORDER BY d ASC;" \
| awk -F'|' '{ print $1 ": " $3 " (" year-$4 "th " $2 ")" }' year=$year
```

Figure 8.4: The calendar-upcoming Script


```
$ calendar-upcoming 60 upcoming two months
2009-01-17: Auntie Mildred (83th birthday)
2009-01-21: Uncle Jack (75th birthday)
2009-02-05: Susie and Martin (7th wedding anniversary)
2009-02-06: ROM-TOS anniversary (23th anniversary)
2009-02-17: Rabbit Keeping Club (124th anniversary)
```


There are many possible extensions. Just have a look at the exercises.


Exercises

 **8.10** [3] Add a “holiday” type to the “poor man’s diary”. When a holdiday is listed, no “age” should be added to the output:

```
2009-02-17: Rabbit Keeping Club (124th anniversary)
2009-02-23: Carnival Monday (holiday) in parts of Germany
2009-03-01: Cousin Fred (29th birthday)
```

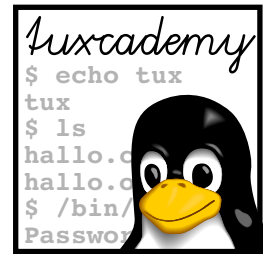
 **8.11** [3] Write a shell script named calendar-holidays which is passed the date of Easter in the usual ISO format (e.g.: 2009-04-12) and adds the holidays of the given year to the calendar database.

 **8.12** [4] (Previous exercise continued—this is a more elaborate project.) Write a program that computes Easter of the given year. Information about computing the date of Easter can be found here: <http://en.wikipedia.org/wiki/Computus>. (*Hint*: use awk.) Rewrite calendar-holidays such that it uses your new program—it will only require the year number rather than the date of Easter afterwards.

 **8.13** [2] Make the program mail you a list of the forthcoming events in each morning. (You will probably need some information from Chapter 9 to do this exercise, so feel free to come back to it at a later time.)

Summary

- SQL (“Structured Query Language”) is a standard language for defining, querying, and manipulating relational databases.
- Normalisation is an important process for maintaining the consistency of relational databases.
- There are various SQL database products for Linux systems.
- A collection of table definitions is called a “database schema”.
- SQL not only supports defining database schemas, but also inserting, modifying, querying, and deleting data (tuples).
- Aggregate functions allow you to compute values like the sum or average of all tuples of a column.
- Relations allow you to process data from multiple tables.
- Many Linux-based applications programs use SQLite for storing data.



9

Time-controlled Actions—cron and at

Contents

9.1	Introduction.	132
9.2	One-Time Execution of Commands	132
9.2.1	at and batch	132
9.2.2	at Utilities	134
9.2.3	Access Control.	134
9.3	Repeated Execution of Commands	135
9.3.1	User Task Lists.	135
9.3.2	System-Wide Task Lists	136
9.3.3	Access Control.	137
9.3.4	The crontab Command	137
9.3.5	Anacron	138

Goals

- Executing commands at some future time using at
- Executing commands periodically using cron
- Knowing and using anacron

Prerequisites

- Using Linux commands
- Editing files

9.1 Introduction

An important component of system administration consists of automating repeated procedures. One conceivable task would be for the mail server of the company network to dial in to the ISP periodically to fetch incoming messages. In addition, all members of a project group might receive a written reminder half an hour before the weekly project meeting. Administrative tasks like file system checks or system backups can profitably be executed automatically at night when system load is noticeably lower.



To facilitate this, Linux offers two services which will be discussed in the following sections.

9.2 One-Time Execution of Commands

9.2.1 at and batch

Using the at service, arbitrary shell commands may be executed once at some time in the future (time-shifted). If commands are to be executed repeatedly, the use of cron (Section 9.3) is preferable.

The idea behind at is to specify a time at which a command or command sequence will be executed. Roughly like this:

```
$ at 01:00
warning: commands will be executed using /bin/sh
at> tar cvzf /dev/st0 $HOME
at> echo "Backup done" | mail -s Backup $USER
at>  
Job 123 at 2003-11-08 01:00
```

This would write a backup copy of your home directory to the first tape drive at 1 A. M. (don't forget to insert a tape) and then mail a completion notice to you.



time specification at's argument specifies when the command(s) are to be run. Times like "*<HH>:<MM>*" denote the next possible such time: If the command "at 14:00" is given at 8 A. M., it refers to the same day; if at 4 P. M., to the next.



You can make these times unique by appending today or tomorrow: "at 14:00 today", given before 2 P. M., refers to today, "at 14:00 tomorrow", to tomorrow.

Other possibilities include Anglo-Saxon times such as 01:00am or 02:20pm as well as the symbolic names midnight (12 A. M.), noon (12 P. M.), and teatime (4 P. M. (!); the symbolic name now is mostly useful together with relative times (see below).

date specifications In addition to times, at also understands date specifications in the format "*<MM><DD><YY>*" and "*<MM>/<DD>/<YY>*" (according to American usage, with the month before the day) as well as "*<DD>.<MM>.<YY>*" (for Europeans). Besides, American-style dates like "*<month name> <day>*" and "*<month name> <day> <year>*" may also be spelled out. If you specify just a date, commands will be executed on the day in question at the current time; you can also combine a date and time specification but must give the date after the time:

```
$ at 00:00 January 1 2005
warning: commands will be executed using /bin/sh
at> echo 'Happy New Year!'
at>  
Job 124 at 2005-01-01 00:00
```

Besides "explicit" time and date specification, you can give "relative" times and dates by passing an offset from some given point in time:

```
$ at now + 5 minutes
```

executes the command(s) five minutes from now, while

```
$ at noon + 2 days
```

refers to 12 P.M. on the day after tomorrow (as long as the `at` command is given before 12 P.M. today). `at` supports the units `minutes`, `hours`, `days` and `weeks`.



A single offset by one single measurement unit must suffice: Combinations such as

```
$ at noon + 2 hours 30 minutes
```

or

```
$ at noon + 2 hours + 30 minutes
```

are, unfortunately, disallowed. Of course you can express any reasonable offset in minutes ...

`at` reads the commands from standard input, i. e., usually the keyboard; with the `“-f <file>”` option you can specify a file instead.



`at` tries to run the commands in an environment that is as like the one current when `at` was called as possible. The current working directory, the `umask`, and the current environment variables (excepting `TERM`, `DISPLAY`, and `_`) are saved and reactivated before the commands are executed.

Any output of the commands executed by `at`—standard output and standard error output—is sent to you by e-mail.



If you have assumed another user’s identity using `su` before calling `at`, the commands will be executed using that identity. The output mails will still be sent to you, however.

While you can use `at` to execute commands at some particular point in time, the (otherwise analogous) `batch` command makes it possible to execute a command sequence “as soon as possible”. When that will actually be depends on the current system load; if the system is very busy just then, `batch` jobs must wait. ASAP execution



An `at`-style time specification on `batch` is allowed but not mandatory. If it is given, the commands will be executed “some time after” the specified time, just as if they had been submitted using `batch` at that time.



`batch` is not suitable for environments in which users compete for resources such as CPU time. Other systems must be employed in these cases.

Exercises



9.1 [!1] Assume now is 1 March, 3 P.M. When will the jobs submitted using the following commands be executed?

1. `at 17:00`
2. `at 02:00pm`
3. `at teatime tomorrow`
4. `at now + 10 hours`



9.2 [1] Use the `logger` command to write a message to the system log 3 minutes from now.

9.2.2 at Utilities

The system appends at-submitted jobs to a queue. You can inspect the contents of that queue using `atq` (you will see only your own jobs unless you are root):

```
$ atq
123  2003-11-08 01:00 a hugo
124  2003-11-11 11:11 a hugo
125  2003-11-08 21:05 a hugo
```



The “a” in the list denotes the “job class”, a letter between “a” and “z”. You can specify a job class using the `-q` option to `at`; jobs in classes with “later” letters are executed with a higher nice value. The default is “a” for at jobs and “b” for batch jobs.



A job that is currently being executed belongs to the special job class “=”.

Cancelling jobs

You can use `atrm` to cancel a job. To do so you must specify its job number, which you are told on submission or can look up using `atq`. If you want to check on the commands making up the job, you can do that with “`at -c <job number>`”.

daemon

The entity in charge of actually executing at jobs is a daemon called `atd`. It is generally started on system boot and waits in the background for work. When starting `atd`, several options can be specified:

- b (“batch”) Determines the minimum interval between two batch job executions. The default is 60 seconds.
- l (“load”) Determines a limit for the system load, above which batch jobs will not be executed. The default is 0.8.
- d (“debug”) Activates “debug” mode, i. e., error messages will not be passed to `syslogd` but written to standard error output.

The `atd` daemon requires the following directories:

- at jobs are stored in `/var/spool/atjobs`. Its access mode should be 700, the owner is `at`.
- The `/var/spool/atpool` directory serves to buffer job output. Its owner should be `at` and access mode 700, too.

Exercises



9.3 [1] Submit a few jobs using `at` and display the job queue. Cancel the jobs again.



9.4 [2] How would you create a list of at jobs which is not sorted according to job number but according to execution time (and date)?

9.2.3 Access Control

`/etc/at.allow`
`/etc/at.deny`

The `/etc/at.allow` and `/etc/at.deny` files determine who may submit jobs using `at` and `batch`. If the `/etc/at.allow` file exists, only the users listed in there are entitled to submit jobs. If the `/etc/at.allow` file does not exist, the users *not* listed in `/etc/at.deny` may submit jobs. If neither one nor the other exist, `at` and `batch` are only available to root.



Debian GNU/Linux comes with a `/etc/at.deny` file containing the names of various system users (including `alias`, `backup`, `guest`, and `www-data`). This prevents these users from using `at`.



Here, too, the Ubuntu defaults correspond to the Debian GNU/Linux defaults.



Red Hat includes an empty `/etc/at.deny` file; this implies that any user may submit jobs.



The openSUSE default corresponds (interestingly) to that of Debian GNU/Linux and Ubuntu—various system users are not allowed to use `at`. (The explicitly excluded user `www-data`, for example, doesn't exist on openSUSE; Apache uses the identity of the `wwwrun` user.)

Exercises



9.5 [1] Who may use `at` and `batch` on your system?

9.3 Repeated Execution of Commands

9.3.1 User Task Lists

Unlike the `at` commands, the `cron` daemon's purpose is to execute jobs at periodic intervals. `cron`, like `atd`, should be started during system boot using an `init` script. No action is required on your side, though, because `cron` and `atd` are essential parts of a Linux system. All major distributions install them by default.

Every user has their own task list (commonly called `crontab`), which is stored in the `/var/spool/cron/crontabs` (on Debian GNU/Linux and Ubuntu; on SUSE: `/var/spool/cron/tabs`, on Red Hat: `/var/spool/cron`) directory under that user's name. The commands described there are executed with that user's permissions.



You do not have direct access to your task lists in the `cron` directory, so you will have to use the `crontab` utility instead (see below). See also: Exercise 9.6.

`crontab` files are organised by lines; every line describes a (recurring) point in time and a command that should be executed at that time. Empty lines and comments (starting with a `#`) will be ignored. The remaining lines consist of five time fields and the command to be executed; the time fields describe the minute (0–59), hour (0–23), day of month (1–31), month (1–12 or the English name), and weekday (0–7, where 0 and 7 stand for Sunday, or the English name), respectively, at which the command is to be executed. Alternatively, an asterisk (`*`) is allowed, which means “whatever”. For example,

```
58 17 * * * echo "News is coming on soon"
```

that the command will be executed daily at 5.58 P.M. (day, month and weekday are arbitrary).



The command will be executed whenever hour, minute, and month match exactly and *at least one* of the two day specifications—day of month or weekday—applies. The specification

```
1 0 13 * 5 echo "Shortly after midnight"
```

says that the message will be output on any 13th of the month *as well as* every Friday, not just every Friday the 13th.




The final line of a `crontab` file *must* end in a newline character, lest it be ignored.


In the time fields, `cron` accepts not just single numbers, but also comma-separated lists. The `"0,30"` specification in the minute field would thus lead to the command being executed every “full half” hour. Besides, ranges can be specified: `"8-11"` is equivalent to `"8,9,10,11"`, `"8-10,14-16"` corresponds to `"8,9,10,14,15,16"`.

Also allowed is a “step size” in ranges. “0-59/10” in the minute field is equivalent to “0,10,20,30,40,50”. If—like here—the full range of values is being covered, you could also write “*/10”.

month and week- The names allowed in month and weekday specifications each consist of the first three letters of the English month or weekday name (e. g., may, oct, sun, or wed). Ranges and lists of names are not permissible.

command The rest of the line denotes the command to be executed, which will be passed by cron to /bin/sh (or the shell specified in the SHELL variable, see below).

 Percent signs (%) within the command must be escaped using a backslash (as in “\%”), lest they be converted to newline characters. In that case, the command is considered to extend up to the first (unescaped) percent sign; the following “lines” will be fed to the command as its standard input.

 By the way: If you as the system administrator would rather not (as cron is wont to do) a command execution be logged using syslogd, you can suppress this by putting a “.” as the first character of the line.

assignments to environment variables Besides commands with repetition specifications, crontab lines may also include assignments to environment variables. These take the form “<variable>=<value>” (where, unlike in the shell, there may be spaces before and after the “=”). If the <value> contains spaces, it should be surrounded by quotes. The following variables are pre-set automatically:

SHELL This shell is used to execute the commands. The default is /bin/sh, but other shells are allowed as well.


LOGNAME The user name is taken from /etc/passwd and cannot be changed.


HOME The home directory is also taken from /etc/passwd. However, changing its value is allowed.

MAILTO cron sends e-mail containing command output to this address (by default, they go to the owner of the crontab file). If cron should send no messages at all, the variable must be set to a null value, i. e., MAILTO="".

9.3.2 System-Wide Task Lists

/etc/crontab In addition to the user-specific task lists, there is also a system-wide task list. This resides in /etc/crontab and belongs to root, who is the only user allowed to change it. /etc/crontab’s syntax is slightly different from that of the user-specific crontab files; between the time fields and the command to be executed there is the name of the user with whose privileges the command is supposed to be run.

 Various Linux distributions support a /etc/cron.d directory; this directory may contain files which are considered “extensions” of /etc/crontab. Software packages installed via the package management mechanism find it easier to make use of cron if they do not have to add or remove lines to /etc/crontab.

 Another popular extension are files called /etc/cron.hourly, /etc/cron.daily and so on. In these directories, software packages (or the system administrator) can deposit files whose content will be executed hourly, daily, ... These files are “normal” shell scripts rather than crontab-style files.

crontab changes and cron cron reads its task lists—from user-specific files, the system-wide /etc/crontab, and the files within /etc/cron.d, if applicable—once on starting and then keeps them in memory. However, the program checks every minute whether any crontab files have changed. The “mtime”, the last-modification time, is used for this. If cron does notice some modification, the task list is automatically reconstructed. In this case, no explicit restart of the daemon is necessary.

Exercises



9.6 [2] Why are users not allowed to directly access their task lists in `/var/spool/cron/crontabs` (or wherever your distribution keeps them)? How does `crontab` access these files?



9.7 [1] How can you arrange for a command to be executed on Friday, the 13th, *only*?



9.8 [3] How does the system ensure that the tasks in `/etc/cron.hourly`, `/etc/cron.daily`, ... are really executed once per hour, once per day, etc.?

9.3.3 Access Control

Which users may work with `cron` to begin with is specified, in a manner similar to that of `at`, in two files. The `/etc/cron.allow` file (sometimes `/var/spool/cron/allow`) lists those users who are entitled to use `cron`. If that file does not exist but the `/etc/cron.deny` (sometimes `/var/spool/cron/deny`) file does, that file lists those users who may *not* enjoy automatic job execution. If neither of the files exists, it depends on the configuration whether only `root` may avail himself of `cron`'s services or whether `cron` is “free for all”, and any user may use it.

9.3.4 The `crontab` Command

Individual users cannot change their `crontab` files manually, because the system hides these files from them. Only the system-wide task list in `/etc/crontab` is subject to `root`'s favourite text editor.

Instead of invoking an editor directly, all users should use the `crontab` command. This lets them create, inspect, modify, and remove task lists. With

```
$ crontab -e
```

you can edit your `crontab` file using the editor which is mentioned in the `VISUAL` or `EDITOR` environment variables—alternatively, the `vi` editor. After the editor terminates, the modified `crontab` file is automatically installed. Instead of the `-e` option, you may also specify the name of a file whose content will be installed as the task list. The “.” file name stands for standard input.

With the `-l` option, `crontab` outputs your `crontab` file to standard output; with the `-r` option, an existing task list is deleted with prejudice.



With the “`-u <user name>`” option, you can refer to another user (expect to be `root` to do so). This is particularly important if you are using `su`; in this case you should always use `-u` to ensure that you are working on the correct `crontab` file.

Exercises



9.9 [!1] Use the `crontab` program to register a cron job that appends the current date to the file `/tmp/date.log` once per minute. How can you make it append the date every other minute?



9.10 [1] Use `crontab` to print the content of your task list to the standard output. Then delete your task list.



9.11 [2] (For administrators:) Arrange that user `hugo` may *not* use the `cron` service. Check that your modification is effective.

9.3.5 Anacron

Using `cron` you can execute commands repeatedly at certain points in time. This obviously works only if the computer is switched on at the times in question – there is little point in configuring a 2am `cron` job on a workstation PC when that PC is switched off outside business hours to save electricity. Mobile computers, too, are often powered on or off at odd times, which makes it difficult to schedule the periodic automated clean-up tasks a Linux system needs.

The `anacron` program (originally by Itai Tzur, now maintained by Pascal Hakim), like `cron`, can execute jobs on a daily, weekly, or monthly basis. (In fact, arbitrary periods of n days are fine.) The only prerequisite is that, on the day in question, the computer be switched on long enough for the jobs to be executed—the exact time of day is immaterial. However, `anacron` is activated at most once a day; if you need a higher frequency (hours or minutes) there is no way around `cron`.



Unlike `cron`, `anacron` is fairly primitive as far as job management is concerned. With `cron`, potentially every user can create jobs; with `anacron`, this is the system administrator's privilege.

The jobs for `anacron` are specified in the `/etc/anacrontab` file. In addition to the customary comments and blank lines (which will be ignored) it may contain assignments to environment variables of the form

```
SHELL=/bin/sh
```

and job descriptions of the form

```
7 10 weekly run-parts /etc/cron.weekly
```

where the first number (here 7) stands for the period (in days) between invocations of the job. The second number (10) denotes how many minutes after the start of `anacron` the job should be launched. Next is a name for the job (here, `weekly`) and finally the command to be executed. Overlong lines can be wrapped with a “\” at the end of the line.



The job name may contain any characters except white space and the slash. It is used to identify the job in log messages, and `anacron` also uses it as the name of the file in which it logs the time the job was last executed. (These files are usually placed in `/var/spool/anacron`.)

When `anacron` is started, it reads `/etc/anacrontab` and, for each job, checks whether it was run within the last t days, where t is the period from the job definition. If not, then `anacron` waits the number of minutes given in the job definition and then launches the shell command.



You can specify a job name on `anacron`'s command line to execute only that job (if any). Alternatively, you can specify shell search patterns on the command line in order to launch groups of (skillfully named) jobs with one `anacron` invocation. Not specifying any job names at all is equivalent to the job name, “*”.



You may also specify the time period between job invocations symbolically: Valid values include `@daily`, `@weekly`, `@monthly`, `@yearly` and `@annually` (the last two are equivalent).



In the definition of an environment variable, white space to the left of the “=” is ignored. To the right of the “=”, it becomes part of the variable's value. Definitions are valid until the end of the file or until the same variable is redefined.



Some “environment variables” have special meaning to `anacron`. With `RANDOM_DELAY`, you can specify an additional random delay¹ for the job launches: When you set the variable to a number t , then a random number of minutes between 0 and t will be added to the delay given in the job description. `START_HOURS_RANGE` lets you denote a range of hours (on a clock) during which jobs will be started. Something like

```
START_HOURS_RANGE=10-12
```

allows new jobs to be started only between 10am and 12pm. Like `cron`, `anacron` sends job output to the address given by the `MAILTO` variable, otherwise to the user executing `anacron` (usually `root`).

Usually `anacron` executes the jobs independently and without attention to overlaps. Using the `-s` option, jobs are executed “serially”, such that `anacron` starts a new job only when the previous one is finished.

Unlike `cron`, `anacron` is not a background service, but is launched when the system is booted in order to execute any leftover jobs (the delay in minutes is used to postpone the jobs until the system is running properly, in order to avoid slowing down the start procedure). Later on you can execute `anacron` once a day from `cron` in order to ensure that it does its thing even if the system is running for a longer period of time than normally expected.



It is perfectly feasible to install `cron` and `anacron` on the same system. While `anacron` usually executes the jobs in `/etc/cron.daily`, `/etc/cron.weekly`, and `/etc/cron.monthly` that are really meant for `cron`, the system ensures that `anacron` does nothing while `cron` is active. (See also Exercise 9.13.)

Exercises



9.12 [!2] Convince yourself that `anacron` is working as claimed. (*Hint*: If you don’t want to wait for days, try cleverly manipulating the time stamps in `/var/spool/anacron`.)



9.13 [2] On a long-running system that has both `cron` and `anacron` installed, how do you avoid `anacron` interfering with `cron`? (*Hint*: Examine the content of `/etc/cron.daily` and `friends`.)

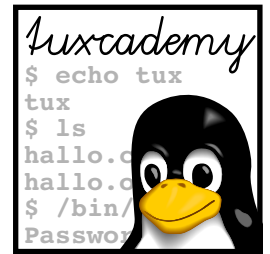
Commands in this Chapter

anacron	Executes periodic job even if the computer does not run all the time		
		<code>anacron(8)</code>	138
at	Registers commands for execution at a future point in time	<code>at(1)</code>	132
atd	Daemon to execute commands in the future using <code>at</code>	<code>atd(8)</code>	134
atq	Queries the queue of commands to be executed in the future		
		<code>atq(1)</code>	133
atrm	Cancels commands to be executed in the future	<code>atrm(1)</code>	134
batch	Executes commands as soon as the system load permits	<code>batch(1)</code>	133
crontab	Manages commands to be executed at regular intervals	<code>crontab(1)</code>	137

¹Duh!

Summary

- With `at`, you can register commands to be executed at some future (fixed) point in time.
- The `batch` command allows the execution of commands as soon as system load allows.
- `atq` and `atrm` help manage job queues. The `atd` daemon causes the actual execution of jobs.
- Access to `at` and `batch` is controlled using the `/etc/at.allow` and `/etc/at.deny` files.
- The `cron` daemon allows the periodic repetition of commands.
- Users can maintain their own task lists (crontabs).
- A system-wide task list exists in `/etc/crontab` and—on many distributions—in the `/etc/cron.d` directory.
- Access to `cron` is managed similarly to `at`, using the `/etc/cron.allow` and `/etc/cron.deny` files.
- The `crontab` command is used to manage crontab files.



10

Localisation and Internationalisation

Contents

10.1 Summary.	142
10.2 Character Encodings.	142
10.3 Linux Language Settings	146
10.4 Localisation Settings.	147
10.5 Time Zones	151

Goals

- Knowing about the most common character encodings
- Converting text files between encodings
- Knowing the language-related environment variables
- Knowing the Linux infrastructure for localisation
- Knowing how Linux handles time zones

Prerequisites

- Linux commands
- Text editing

10.1 Summary

“Internationalisation”, or I18N for short (because there are 18 letters between the first and the last character of the word), is the preparation of a software system such that “localisation” becomes possible. Localisation, or L10N (you get the idea), is the adaptation of a software system to the local customs of different countries or culture groups. The primary aspect of localisation is, of course, the language of the user interface including the messages printed by the system. Another important aspect is the data that is being processed by the system. Such data may require special character encodings and input facilities. Finally, aspects like notations for dates, times, and currencies, the collating order of alphabetic characters, and other minor details are also covered by localisation. In graphical programs, even colors may need to be localised: In the Western world, the color “red” indicates danger, but this is not the case everywhere else.

For the Linux operating system kernel, internationalisation is not a very pressing issue since most of the areas requiring internationalisation are not really its concern. (There is widespread consensus that the kernel should not be loaded with error messages in all sorts of languages; the expectation is that anybody who actually gets to see these messages has enough English to understand them.) Linux *distributions*, on the other hand, contain vast amounts of application software that stands to benefit from localisation. Accordingly, the major distributions are available in a wide variety of localised versions. There are also diverse special Linux distributions that concentrate on specific culture groups and attempt to support these particularly well.



While commercial software manufacturers let local subsidiaries or paid contractors do the localisation work, the localisation of open-source software like most Linux applications is mostly done by volunteers. The advantage of this approach is that usually volunteers who will translate and adapt a program can be found for even the most exotic languages—languages that, for business reasons, no commercial vendor would consider supporting. On the other hand, using volunteers presents special requirements to quality control. A legendary anecdote tells the story that a KDE developer from the Arab world once replaced all occurrences of the word “Israel” by “Occupied Palestine”; a step that not all of the KDE community approved of.

10.2 Character Encodings

The most important prerequisite for the internationalisation and localisation of programs in foreign languages (in effect, “anything but English”) is that the system needs to be able to display the script of the language in question. The traditional character encoding for computers is ASCII or the “American Standard Code for Information Interchange”, which, as its name suggests, was meant for American English—in the early days of electronic data processing this was adequate, but fairly soon it proved necessary to take into account the requirements of “foreign languages”.

character set control characters **ASCII** ASCII represents 128 different characters, of which 33 (positions 0–31 and position 127) are reserved for control characters like “line feed”, “horizontal tabulation”, and “bell”. The remaining 95 characters include uppercase and lowercase letters, digits and a selection of special characters, mostly punctuation marks.

DIN 66003



Germany used to use the DIN 66003 character encoding, which for the most part corresponded to ASCII, except that the square brackets, curly braces, the vertical bar and backslash were replaced by the German “umlauts” (letters with diacritical double dots above them). The tilde was replaced by the “sharp s” (a ligature of the “s” and the “z”), and the “commercial” at sign

Table 10.1: The most common parts of ISO/IEC 8859

Part	Common name	Scope
ISO 8859-1	Latin-1 (Western European)	Most Western European languages: Danish, Dutch*, English, Faroese, Finnish*, French*, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, and Swedish. It also contains characters of some other languages, including Afrikaans, Albanian, Indonesian, and Swahili. (* = partial support)
ISO 8859-2	Latin-2 (Central European)	Central and Eastern European languages using the Latin alphabet, like Bosnian, Croatian, Czech, Hungarian, Polish, Serbian, Slovak, and Slovenian.
ISO 8859-3	Latin-3 (Southern European)	Esperanto, Maltese, and Turkish.
ISO 8859-4	Latin-4 (Northern European)	Estonian, Greenlandic, Latvian, Lithuanian, and Sami.
ISO 8859-5	Latin/Cyrillic	Most Slovenian languages that use the Cyrillic alphabet, like Belorussian, Bulgarian, Macedonian, Russian, Serbian, and Ukrainian (partly).
ISO 8859-6	Latin/Arabic	Contains the most common Arabic characters, but is not suitable for languages other than Arabic itself. Note that Arabic is written from right to left!
ISO 8859-7	Latin/Greek	Modern and ancient Greek without diacritics.
ISO 8859-8	Latin/Hebrew	Modern Hebrew.
ISO 8859-9	Latin-5 (Turkish)	Mostly equal to ISO 8859-1, but with additional Turkish characters in place of the Icelandic ones. Also used for Kurdish.
ISO 8859-10	Latin-6 (Northern)	A variation of Latin-4.
ISO 8859-11	Latin/Thai	Thai language.
ISO 8859-12	Latin/Devanagari	Never finished.
ISO 8859-13	Latin-7 (Baltic)	Contains even more characters of the Baltic languages that do not occur in Latin-4 and Latin-6.
ISO 8859-14	Latin-8 (Celtic)	Celtic languages like Breton and (Irish) Gaelic.
ISO 8859-15	Latin-9	Mostly Latin-1, but includes the Euro sign and the missing characters for Estonian, Finnish, and French in place of some very rarely used characters.
ISO 8859-16	Latin-10	Albanian, Croatian, Hungarian, Italian, Romanian, Slovenian. Also: Finnish, French, (Irish) Gaelic, German. Includes a Euro sign. Practically unused.

by the “paragraph” sign. This was acceptable for German-language text but not necessarily for C programs. (Your author remembers the first printer he had to deal with, a Centronics 737-2, which came with a row of tiny switches that were used to switch the printer from ASCII to DIN 66003 and back—a tedious method.)

ISO/IEC 8859 Later, as pressure from international computer users mounted, a transition from ASCII with its 128 characters to extended character sets took place, which were able to use all 256 possible values of a byte to encode characters. The most widely used extended character sets are those described in the ISO/IEC 8859 standard, which includes character sets for many different languages. In fact, ISO/IEC 8859 consists of a set of numbered, separately published parts which are often considered separate standards. Table 10.1 provides a summary.

256 characters


The focus of the ISO/IEC-8859 standard is on information interchange rather than elegant typography, so various characters necessary for beautiful output—such as ligatures or “typographic” quotes—are missing from the encodings. When creating the code tables the emphasis was on characters that were already present on computer keyboards, and also made some compromises. For example,

information interchange

the French “œ” and “ø” ligatures were not included in Latin-1, as they can be written as “OE” and “oe”, respectively, and the space in the table was required for other characters.

limitations ISO/IEC 8859 does not address Oriental languages like Chinese or Japanese because the character set of these languages by far exceeds the 256 characters that fit into a single ISO/IEC 8859 code table. There are other scripts which are not the subject of an ISO/IEC 8859 standard, either.


two for all **Unicode and ISO 10646** Unicode and ISO 10646 (the “Universal Character Set”s) are parallel efforts to create one single character set to cover *all* alphabets of the world. Initially both standards were developed separately, but were merged after the world’s software vendors rebelled fiercely against the complexity of ISO 10646. Today Unicode and ISO 10646 standardise the same characters with identical codes; the difference between the two is that ISO 10646 is a pure character table (basically an extended ISO 8859), while Unicode contains additional rules for details like lexical sorting order, normalisation, and bidirectional output. In this sense ISO 10646 is a “subset” of Unicode; with Unicode, characters also have various extra properties which indicate, for example, the ways in which a character can be combined with others (which is, for instance, important for Arabic, where the rendition of a character depends on whether it occurs at the beginning, in the middle, or at the end of a word).

ISO 10646 without Unicode  The Unix/Linux program, `xterm`, is an example of a program that does support ISO 10646 but not Unicode. It can display all ISO-10646 characters that derive directly from the character encoding table and are written in a single direction, as well as some “combined” characters that consist of several others (such as a “base character” and diacritics), but not Hebrew, Arabic, or Devanagari. Implementing Unicode correctly is by no means a piece of cake.

ISO 10646 character set code points The ISO 10646 character set contains not just letters, digits, and punctuation marks, but also ideographs (like Chinese and Japanese characters), mathematical characters, and much more. Each of these characters is identified by a unique name and an integer number that is called a “code point”. There are over 1.1 million code points in the character set, of which only the first 65,536 of them are in common use. These are also called the “basic multilingual plane”, or BMP. Unicode and ISO-10646 code points are written in the form `U+0040`, where the four digits represent a hexadecimal number.

Encodings **UCS-2 and UTF-8** Unicode and ISO 10646 specify code points, i. e., integer numbers for the characters in the character set, but do not specify how to handle these code points. Encodings are defined to explain how to represent the code points inside a computer.

UCS-2 The simplest encoding is UCS-2, in which a single “code value” between 0 and 65,535 is used for each character, which is represented by two bytes. This implies that UCS-2 is limited to characters in the BMP. Furthermore, UCS-2 implies that Western-world data, which would otherwise be represented by an 8-bit encoding such as ASCII or ISO Latin-1, require twice the storage space, since suddenly two bytes are used per character instead of one.

UTF-16  Instead of UCS-2, systems like Windows have changed to an encoding called UTF-16, which does represent code points beyond the BMP. The storage space problem remains, though.

UTF-8 UTF-8 is capable of representing any character in ISO 10646 while maintaining backward compatibility with ASCII and ISO-8859-1. It encodes the code points `U+0000` to `U+10FFFF` (i. e., 32 times as many as UCS-2) using one to four bytes, where the ASCII characters occupy a single byte only. Here are the design goals of UTF-8:

ASCII characters represent themselves This makes UTF-8 compatible with all programs that deal with byte strings, i. e., arbitrary sequences of 8-bit bytes, but assign special meaning to some ASCII characters. Migrating a system from ASCII to UTF-8 is easy.

No first byte appears in the middle of a character If one or more complete bytes are lost or mutilated, it is still possible to locate the beginning of the next character.

The first byte of each character determines its number of bytes This ensures that a byte sequence representing a specific character cannot be part of a longer sequence representing a different character. This makes it efficient to search strings for substrings at the byte level.

The byte values FE and FF are not used These bytes are used at the beginning of UCS-2 texts in order to identify the byte ordering inside of the text. Because these characters are not valid UTF-8 data, UTF-8 documents and UCS-2 documents cannot be confused.

By now, UTF-8 is the encoding of choice for representing Unicode data on a Linux system.



If you want to know how exactly UTF-8 works, you should have a look at the `utf-8(7)` man page, which explains the encoding in detail and provides lots of examples.

The `iconv` command converts between character encodings. In the simplest case it converts the contents of the files specified on the command line from a given encoding to the encoding that is currently being used. The result is written to the standard output:

```
$ iconv -f LATIN9 test.txt >test-utf8.txt
```

You can also specify a different target encoding:

```
$ iconv -f UTF-8 -t LATIN9 test-utf8.txt >test-l9.txt
```

The `-o` (or `--output`) option can be used to write the output directly to a file:

```
$ iconv -f LATIN9 -o test-utf8.txt test.txt
```

When no input file is given, `iconv` reads its standard input:

```
$ grep bla test.txt | iconv -f LATIN9 -o grep.out
```



The `-l` option lists all character encoding that `iconv` supports (which does not necessarily mean that it can convert successfully between arbitrary pairs of these encodings).



When `iconv` encounters an invalid character in its input, it reports an error and bails out. To counter this, you can append one of the suffixes `//TRANSLIT` or `//IGNORE` to the target encoding. `//IGNORE` simply drops any characters that do not exist in the target encoding, while `//TRANSLIT` attempts to approximate them using one or more similar characters:

```
$ echo xäöüy | iconv -f UTF-8 -t ASCII//IGNORE
xy
iconv: (stdin):1:1: cannot convert
$ echo xäöüy | iconv -f UTF-8 -t ASCII//TRANSLIT
xaeoeuey
```

The `-c` option drops invalid characters silently:

```
$ echo xääüü | iconv -c -f UTF-8 -t ASCII
xy
```

10.3 Linux Language Settings

The language that a Linux system uses to communicate with its users is normally selected from a menu when the system is being installed. It is rarely changed at a later time. Desktop environments like KDE and GNOME allow individual users to change the interface language in a convenient way. Linux users typically do not change the language settings on the command line, but this can be done, too.

Language is per session First we have to recognise that the “system language” is not really a property of the entire system, but a parameter of each individual session. In the normal flow of operation, the login shell or graphical desktop environment is initialised with a specific language setting, and the subprocesses of this shell “inherit” this setting just like the current working directory, resource limits of processes, etc. So there is nothing to keep you from using the system with an English-language setting while at the same time someone is logged on over the network or at another terminal who is using a German or French session.



To be precise, there is nothing to keep you from setting a different language in one or more windows inside your own session.

LANG The controlling factor for the language of a session is the value of the LANG environment variable. In the simplest case, it consists of three parts: a language code according to ISO 639, followed by an underscore character, followed by a country code as per ISO 3166, for example something like

en_GB	<i>English in Great Britain</i>
en_US	<i>English in the United States</i>

The country code is important because the languages in two countries may well differ even though they use the same language in principle. For example, American English uses words like “elevator” instead of “lift” or “gas” instead of “petrol”. Spelling differs, too, so American English uses “color” instead of “colour” and “catalog” instead of “catalog”. This means that a word processor may flag the word “color” as wrong in a `en_GB` text, just as it might complain about “colour” in a `en_US` text¹.



If the difference between `en_GB` and `en_US` isn’t obvious enough for you, then consider `de_DE` versus `de_AT`, or even `pt_PT` versus `pt_BR`.

extensions Some extensions may follow this plain specification, such as a character encoding (separated by a period) or a “variant” (separated by @). This means you can use values such as

de_DE.ISO-8859-15	<i>German German, according to ISO Latin-9</i>
de_AT.UTF-8	<i>Austrian German, Unicode/UTF-8-based</i>
de_DE@euro	<i>German German, including the Euro sign (ISO Latin-9)</i>

This is how different LANG settings affect the output:

¹According to George Bernard Shaw, “England and America are two countries divided by a common language”.

```
$ for i in en_US de_DE de_AT fi_FI fr_FR; do
> LANG=$i.UTF-8 date +"%B %Y"
> done
January 2009
Januar 2009
Jänner 2009
tammikuu 2009
janvier 2009
```

(With date, the format designator %B denotes the name of the month according to the current language setting.)



Of course this presupposes that the system in question actually *does* provide support for the given language. Debian GNU/Linux, for example, lets you pick which settings should be supported and which shouldn't. If you select a setting that your system does not support, the system falls back to a built-in default, which is usually English.

The LANGUAGE environment variable (which is not to be confused with LANG) is only evaluated by programs that use the GNU gettext infrastructure to translate their messages into different languages (which, on a Linux system, means most of them). The most obvious difference between LANGUAGE and LANG is that LANGUAGE allows you to enumerate multiple languages (separated by colons). This lets you specify a preference list:

```
LANGUAGE=de_DE.UTF-8:en_US.UTF-8:fr_FR.UTF-8
```

means “German, or else English, or else French”. The first language that a program actually features messages for wins. LANGUAGE is preferred over LANG (for programs that use GNU gettext, anyway).

Exercises



10.1 [1] What does the command

```
$ LANG=ja_JP.UTF-8 date +"%B %Y"
```

output (assuming support for the language in question is installed)?

10.4 Localisation Settings

In fact, the value of the LANG variable not only influences the interface language but all of the “cultural setup” of a Linux system. This includes things like

Time and date formatting In the United States, for instance, it is common to specify a date in the form “month/day/year”:

\$ date +"%x"	<i>Locale-specific time format</i>
01/14/2009	
\$ LANG=de_DE.UTF-8 date +"%x"	<i>German-style</i>
14.01.2009	


The Americans (and British) also give the time of the day using a 12-hour clock while elsewhere a 24-hour clock is the norm: What is called “3 p.m.” in Great Britain and the USA equals “15 Uhr” in Germany.

Number and currency formatting In the United Kingdom and the USA a period is used as a decimal separator, while commas serve to make large numbers more readable:

```
299,792,458.0
```


Other countries (and an insignificant multinational body called ISO) use these characters the other way round:

```
299.792.458,0
```


 This feature is mostly used by programs that make use of the `printf()` and `scanf()` C functions. Other programs have to query the variable themselves and format their output accordingly.

With monetary amounts, there are additional complications. For example, negative balances are sometimes denoted by a leading minus sign and sometimes put in parentheses (among others).

Character classification The classification of a character as a letter, a special character, or whatever depends on the language. In the age of Unicode, this problem has mostly gone away, as the code points as a whole are classified, but things are not that simple in ISO 8859 or even ASCII environments. The ASCII character “`l`”, for instance, is obviously a special character, but the character “`Ä`”, which occupies the same location in the DIN 66003 table, is as obviously a letter. This also influences the conversion between uppercase and lowercase letters and similar operations.

 The German “sharp s” (“`ß`”) does not have a graphic equivalent in uppercase—a word like “Fuß”, in uppercase, is spelled “FUSS”. (In ambiguous cases “SZ” used to be recommended as a replacement, as in “MASSE” (mass) versus “MASZE” (measurements), but this was abolished during the recent German orthography reform.) Amendment 4:2008 to ISO 10646, which was promulgated on 23 June 2008, defines a code point for a “capital ß” (U+1E9E) so there is nothing major to keep this problem from being fixed for good. We Germans just need to agree about what that character should actually look like. (“SS” would be a strong contender.)

Character collating order This, too, is not quite as unambiguous as one may believe. In Germany there are two different methods for sorting words, according to DIN 5007: In dictionaries and similar publications like encyclopedias, umlauts (letters with diacritical marks) are considered equivalent to their “base characters” (thus “`ä`”, for the purposes of sorting, is interpreted as “`a`”), while in name lists such as phone books, umlauts are sorted according to their transliteration (“`ä`” is treated like “`ae`”, etc.). In both cases “`ß`” is equivalent to “`ss`”.

 For name lists, one apparently wishes that the difference between the homophones, “Müller” and “Mueller”, not complicate the actual search. Otherwise you would have to look for Herr Müller in between Frau Muktedir and Herr Muminovic, while Frau Mueller would fit in between Herr Mueders and Frau Muffert—a first-degree inconvenience. In the encyclopedia, though, the spelling of a search term and hence its collation should be clear.

In Sweden, on the other hand, the characters “`å`”, “`ä`”, and “`ö`” are located at the end of the alphabet (after “`z`”). In the United Kingdom, “`ä`” comes right

Table 10.2: LC_* environment variables

Variable	Description
LC_ADDRESS	Formatting of addresses and locations
LC_COLLATE	Collating (sorting) order
LC_CTYPE	Character classification and uppercase/lower-case conversion
LC_MONETARY	Formatting of monetary amounts
LC_MEASUREMENT	Units of measurement (metric and others)
LC_MESSAGES	Language for system messages and the form of positive and negative responses
LC_NAME	Formatting of proper names
LC_NUMERIC	Formatting of numbers
LC_PAPER	Paper formats (controversial)
LC_TELEPHONE	Formatting of phone numbers
LC_TIME	Formatting of time and date specifications
LC_ALL	All settings

after “a” and is inserted between “az” and “b”, and, nastily, the name component “Mc” is considered to be equal to “Mac” (so the correct sorting order is “Macbeth, McDonald, MacKenzie”). Ideograph-based languages like Japanese and Chinese are even more difficult to collate; dictionaries usually go by the structure of the ideographs and their number of strokes, while computers conveniently sort according to Latin transliterations. (We shall stop here before your head explodes.)

Besides the language, the LANG variable changes all of this in one fell swoop to the values suitable for a specific culture group (a “locale”). However, it is also possible to set various aspects of the localisation separately. The system supports a number of environment variables, all of which start with the LC_ prefix (see table 10.2).



In case you want to know what the values of these parameters actually mean, try the locale command:

```
$ locale -k LC_PAPER
height=297
width=210
paper-codeset="UTF-8"
```

So you can find out that sheets of paper in the United Kingdom are typically 297 mm high and 210 mm wide. We know this as “A4”.



You will find the actual *definitions* that these settings are based on in the /usr/share/i18n/locales directory. In principle nothing will stop you from designing your own locale definition (other than the scant documentation, perhaps). The localedef program does the actual work.

With LC_ALL, as with LANG, you can set all locale parameters at once. The system uses the following approach to figure out which setting is authoritative: LC_ALL

1. If LANG is set, its value counts.
2. If LANG is not set but the LC_* variable for the topic in question (such as LC_COLLATE) is, its value counts.
3. If neither LANG nor the appropriate LC_* variable are set, but LC_ALL is set, then its value counts.

4. When none of these variables are defined at all, a compiled-in default value is used.

Note that if (as usual) the `LANG` variable is set, you can do whatever you please with the `LC_*` variables—without any consequences.



If you scratch your head now in amazement, you are completely right—why bother about `LC_*` variables at all if `LANG`, the environment variable that the system sets on your behalf when you log in, overrides everything anyway? This is as much of an enigma to us than it is to you, but in a pinch there is always `.bash_profile` and “unset `LANG`”.

The “`locale -a`” command provides a list of values that your system supports for `LANG` and the `LC_*` variables:

```
$ locale -a
C
de_AT.utf8
de_DE
de_DE@euro
de_DE.utf8
deutsch
<<<<<<
POSIX
```



Things like `LANG=deutsch` may look tempting at first glance, but they are too unspecific to be useful. Besides, they are officially deprecated but are kept on for compatibility (for the time being). Give them a wide berth.

C The magic values `C` and `POSIX` (which are equivalent) describe the built-in default that programs use if they cannot find another valid setting. This is useful if you want programs like `ls` to deliver a predictable output format. Compare, for example,

```
$ LANG=de_DE.UTF-8 ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 4. Apr 2008 /bin/ls
$ LANG=ja_JP.UTF-8 ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 2008-04-04 16:22 /bin/ls
$ LANG=fi_FI.UTF-8 ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 4.4.2008 /bin/ls
$ LANG=C ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 Apr 4 2008 /bin/ls
```

We run the same command with four different `LANG` settings and obtain four different results, all of which differ in the date stamp. Unfortunately this date stamp can, depending on the language setting, appear to programs like `awk` or “`cut -d ' '`” to consist of one, two, or three fields—which is fatal if a script is to parse this output! So it is best, in such ambiguous cases, to fix the output of programs such as `ls`, whose output depends on the language setting, to a standard that will definitely exist. Use an explicit `LANG=C` (you cannot be sure about any other settings).

Exercises



10.2 [2] The `printf(1)` program (not to be mixed up with `bash`’s built-in `printf` command) formats its input data according to the `LC_NUMERIC` variable. In particular, a command like

```
$ /usr/bin/printf "%'g\n" 31415.92
```

formats a decimal number using the decimal separator and “readability character” appropriate to the current locale. Experiment with the program using different LANG settings.



10.3 [2] Find programs other than `date`, `ls`, and `printf` that change their output according to the language and cultural parameters. (Translated error messages are too trivial and do not count, it must be something interesting.)

10.5 Time Zones

Finally we shall have to say a few things about time zones and how Linux handles them. If you have ever taken a long flight to the east or to the west, you will have noticed that local times vary among the areas of the Earth—when it is high noon here in Europe, it may still be dark in America while in eastern Asia the day is drawing to a close. This wouldn’t be a big deal (you do set your clock according to the time signal on the radio) if there wasn’t the Internet, which makes it possible to exchange data at high speed among arbitrary computers anywhere in the world. And whether an e-mail message was written at 12 o’clock European, American, or Australian time does make a difference of several hours that one would like to take into account.

Hence current computer systems allow you to specify in which time zone the computer resides—typically you will be asked about this during installation, and unless you emigrate and take your computer along, you are unlikely to have to change the value again once it has been set. time zone



The time zones of the Earth are loosely based on the fact that a difference of 15 degrees of longitude equals to one hour on the clock (which makes sense, since the complete circumference of the Earth, 24 hours’ worth, corresponds to 360 degrees, and $360/24$ just happens to be 15.) In former times there were no time zones, but each town simply defined its own time, where the main criterion was that at noon the sun was supposed to be as exactly South as possible (presumably so that sun dials would be accurate). The introduction of railroad travel and the mounting difficulties of taking “local time” into account when preparing timetables made this more and more impractical, which led to the introduction of time zones, at the price that the time on the clock no longer corresponds to “astronomical” time. In any case, time zone boundaries do not (exclusively) derive from lines of longitude but really from political boundaries.



That this can be quite noticeable in practice is evident in Europe. Spain, for instance, follows “Central European Time” (CET), which is appropriate for 15 degrees of Eastern longitude. This corresponds to, e. g., the town of Görlitz on the German-Polish border. When the clock strikes 12 noon in A Coruña on the Spanish Atlantic coast (nearly 8.5 degrees of *Western* longitude), according to the sun it is only approximately half past ten. (Portugal, incidentally, uses the same time zone as the United Kingdom, so it will be about half past eleven there already.)



Things get even more complicated through “daylight saving time” (DST), where the clocks in a time zone are artificially advanced by one hour in spring and put back again in autumn. DST is a purely political phenomenon which still needs to be taken into account—its history in Germany was quite eventful (Table 10.3), which illustrates why you cannot simply set “CET” as the timezone for Germany but must select “Europe/Berlin”.

Table 10.3: Daylight Saving Time (DST) in Germany

Period of Time	Situation
prior to 1916	No DST
1916	DST from May 1st to October 1st
1917–1918	DST from mid-April to mid-September
1919–1939	No DST
1940–1942	The clock was advanced by one hour on April 1st 1940 and remained so until November 2nd 1942 (!)
1943–1944	DST from end of March/beginning of April to beginning of October
1945	DST from April 2nd to November 18th plus “double” DST from May 24th to September 24th; in “double” DST, the clock was advanced by another hour
1946–1949	DST from mid-April to beginning of October
1947	“Double” DST from May 11th to June 29th
1950–1979	No DST
1980–1995	DST from the last weekend in March to the last Sunday in October (this had been decided in the FRG already in 1978, but the GDR did not go along until 1980)
since 1996	From the last Sunday in March to the last Sunday in October (EU standard)

Like the settings related to languages and culture groups, the time zone on a Linux system is not a unique, system-wide setting but belongs to the inheritable properties of a process. The Linux kernel itself measures time in seconds since 1 January 1970, midnight UTC, so the “time zone” issue is merely a question of formatting this (by now fairly large) number of seconds². This elegantly sidesteps all the difficulties that other operating systems had and still have, and there is no problem with your mate from Sydney logging in on your computer via the Internet and seeing Australian time while you yourself, of course, use CET. This is how it ought to be.

The default time zone that is selected when the system is installed is saved to the `/etc/timezone` file:

```
$ cat /etc/timezone
Europe/Berlin
```

You can find all valid time zones by inspecting the names of the files below `/usr/share/zoneinfo` directory:

```
$ ls /usr/share/zoneinfo
Africa/      Chile/      Factory    Iceland    MET         Portugal   Turkey
America/    CST6CDT    GB         Indian/    Mexico/     posix/     UCT
Antarctica/ Cuba        GB-Eire    Iran        Mideast/    posixrules Universal
Arctic/     EET        GMT        iso3166.tab MST         PRC        US/
Asia/       Egypt      GMT0       Israel     MST7MDT     PST8PDT    UTC
Atlantic/   Eire       GMT-0      Jamaica    Navajo      right/     WET
Australia/  EST        GMT+0      Japan      NZ          ROC        W-SU
Brazil/     EST5EDT    Greenwich Kwajalein NZ-CHAT     ROK        zone.tab
Canada/     Etc/       Hongkong   Libya      Pacific/    Singapore  Zulu
CET         Europe/    HST        localtime@ Poland      SystemV/
```

Most of these files are subdirectories:

```
$ ls /usr/share/zoneinfo/Europe
Amsterdam  Chisinau   Kiev       Moscow     Sarajevo   Vatican
Andorra    Copenhagen Lisbon     Nicosia    Simferopol Vienna
```

²On 14 February 2009 at 0:31:30 CET, exactly 1234567890 seconds will have passed since the beginning of Linux time. Happy Valentine’s Day!

Athens	Dublin	Ljubljana	Oslo	Skopje	Vilnius
Belfast	Gibraltar	London	Paris	Sofia	Volgograd
Belgrade	Guernsey	Luxembourg	Podgorica	Stockholm	Warsaw
Berlin	Helsinki	Madrid	Prague	Tallinn	Zagreb
Bratislava	Isle_of_Man	Malta	Riga	Tirane	Zaporozhye
Brussels	Istanbul	Mariehamn	Rome	Tiraspol	Zurich
Bucharest	Jersey	Minsk	Samara	Uzhgorod	
Budapest	Kaliningrad	Monaco	San_Marino	Vaduz	



The rule is that the time zone isn't named after the capital of the country in question (which would have been too obvious) but after the most populous city in the part of the country in question that the time zone in question applies to. Switzerland is thus covered by Europe/Zurich (rather than Europe/Berne), and Russia uses 11 time zones in total, not all of which are counted under Europe. Europe/Kaliningrad, for example, is one hour ahead of Europe/Moscow, which in turn is two hours ahead of Asia/Yekaterinburg.



`/usr/share/zoneinfo` also contains some "convenience time zones" like Poland or Hongkong. Zulu is nothing to do with South Africa, but, as readers of Tom Clancy novels probably know, refers to universal time (UTC), which is often given as 12:00Z, where NATO spells "Z" as "Zulu".

`/etc/localtime` is a copy of the file of `/usr/share/zoneinfo` which contains the information for the time zone specified in `/etc/timezone`—for instance, `/usr/share/zoneinfo/Europe/Berlin`.



In principle, `/usr/share/zoneinfo` should cater to all tastes in time zones. Should you ever feel the urge to define a new time zone yourself, you can do this using the "time zone compiler", *zic*. `zdump` lets you find the time in any arbitrary time zone, or the "prehistory" of Daylight Saving Time in every time zone. (Guess where we got the information for Table 10.3 from.)

You can change the default time zone of the system manually by adjusting the content of the `/etc/timezone` and `/etc/localtime` files:

```
# echo Asia/Tokyo >/etc/timezone
# cp /usr/share/zoneinfo/$(cat /etc/timezone) /etc/localtime
```

On top of this, distributions often provide more comfortable tools for setting a new time zone.



The `tzselect` utility lets you select a time zone interactively. The program first presents a selection of continents and then a selection of existing time zones on that continent. It finally writes the name of the time zone to the standard output while the user interaction is done via the standard error stream, so you can use a command like

```
$ TZ=$(tzselect)
```

to put the result into an environment variable.—To change the default system time zone you would use the `debconf` mechanism instead of `tzselect`. Just run

```
# dpkg-reconfigure tzdata
```

The `tzconfig` program, which various documents keep talking about, is deprecated.



Users of SUSE Linux can change the default system time zone using YaST. There is no obvious convenient tool to change your "personal" time zone.



The default system time zone is located in the file `/etc/sysconfig/clock`. Besides, there is a program called `timeconfig`, and the `tzselect` tool discussed in the Debian GNU/Linux paragraph is available, too.

Irrespective of the system-wide default time zone you can put the name of a time zone into the `TZ` environment variable, which will subsequently be used for the process in question (and, like other environment variables, is passed to child processes). With something like

```
$ export TZ=America/New_York
```

you might set the time zone `America/New_York` for your shell and all programs launched by it.

You may also change the time zone for just a single command:

```
$ TZ=America/New_York date
```

will show you the current time in New York.



The `TZ` variable can even be used to describe time zones without having to use the data stored in `/usr/share/zoneinfo`. In the simplest case you specify the (abbreviated) name of the desired time zone and the offset from UTC. The offset must have an explicit sign (“+” for time zones west of the zero meridian, “-” for east), followed by a time span in the format `HH:MM:SS` (the minutes and seconds parts are optional, and currently there are no time zones with a seconds offset). Something like

```
$ export TZ=CET-1
```

would select “central European time”, but without considering daylight saving time, let alone the German DST history. To specify DST, too, you have to give the name of the DST time zone, its offset from “normal” time (in case it is not “plus one hour”), and a rule for switching to and from DST. The DST rule consists of a day specification and an optional time specification (separated by a slash), where the day specification may take one of three forms: three forms:

Yn The day number within the year, counted from 1 to 365. 29 February is ignored.

n The day number within the year, counted from 0 to 365. In leap years, 29 February is counted.

Mm.w.d Day *d* of week *w* in month *m*. *d* is between 0 (Sunday) and 6 (Saturday), *w* is between 1 and 5, where 1 is the first week in the month and 5 the last one, and *m* is a value between 1 and 12.

The rule for German DST that is currently in force would look like

```
$ export TZ=CET-1CEST,M3.5.0/2,M10.5.0/2
```

but once more the “history” from Table 10.3 is not taken into account.

Exercises



10.4 [1] Why is `/etc/localtime` a copy of the corresponding file in `/usr/share/zoneinfo`? Why does the system not use a symbolic link instead? Or it could just look up the original file in `/usr/share/zoneinfo`. What do you think?



10.5 [!2] Imagine you are a stockbroker and need a quick overview of the current times in Tokyo, Frankfurt, and New York. How can you implement this in Linux?



10.6 [2] Specify the TZ daylight saving time rules for the hypothetical country of Nowheristan. The following ground rules apply:

- In Nowheristan, Nowheristianian Normal Time (NNT) applies. 12:00 UTC corresponds to 13:15 NNT.
- From the second Wednesday in April at 3 a. m. until 10 October (in leap years, 11 October) at 9 p. m., Nowheristan Daylight Saving Time (NDT) is in force, during which all clocks in Nowheristan are advanced by 25 minutes.

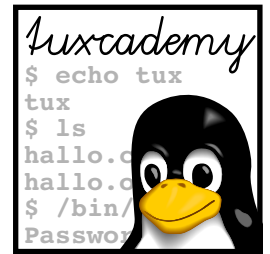
How can you test your rule?

Commands in this Chapter

iconv	Converts between character encodings	<code>iconv(1)</code>	145
locale	Displays information pertaining to locales	<code>locale(1)</code>	149, 150
localedef	Compiles locale definition files	<code>localedef(1)</code>	149
timeconfig	[Red Hat] Allows the convenient configuration of the system-wide time zone	<code>timeconfig(8)</code>	153
tzselect	Allows convenient interactive selection of a time zone	<code>tzselect(1)</code>	153
zdump	Outputs the current time or time zone definitions for various time zones	<code>zdump(1)</code>	153
zic	Compiler for time zone data files	<code>zic(8)</code>	153

Summary

- Internationalisation is the preparation of a software system for localisation. Localisation is the adaptation of a software system to the local customs of different countries or culture groups.
- Common character encodings on Linux systems are ASCII, ISO 8859, and ISO 10646 (Unicode).
- UCS-2, UTF-16, and UTF-8 are character encodings of ISO 10646 and Unicode.
- The `iconv` command converts between different character encodings.
- The language of a Linux process is specified by the `LANG` environment variable.
- The environment variables `LC_*` and `LANG` control the localisation of Linux processes.
- The `locale` command provides access to more detailed localisation information.
- Use `LANG=C` in shell scripts to make sure that locale-sensitive commands deliver predictable output.
- Linux fetches the system-wide time zone from the file `/etc/timezone`.
- The time zone of an individual process can be changed by setting its TZ environment variable.



11

The X Window System

Contents

11.1	Fundamentals	158
11.2	X Window System configuration	163
11.3	Display Managers.	169
11.3.1	X Server Starting Fundamentals	169
11.3.2	The LightDM Display Manager.	170
11.3.3	Other Display Managers	172
11.4	Displaying Information.	173
11.5	The Font Server	175
11.6	Remote Access and Access Control	177

Goals

- Understanding the structure and operation of X11
- Handling display names and the DISPLAY variable
- Being aware of various methods of starting X11
- Handling a display manager
- Knowing about font management and the font server
- Being able to configure remote access to an X server

Prerequisites

- Knowledge of other graphical user interfaces is helpful

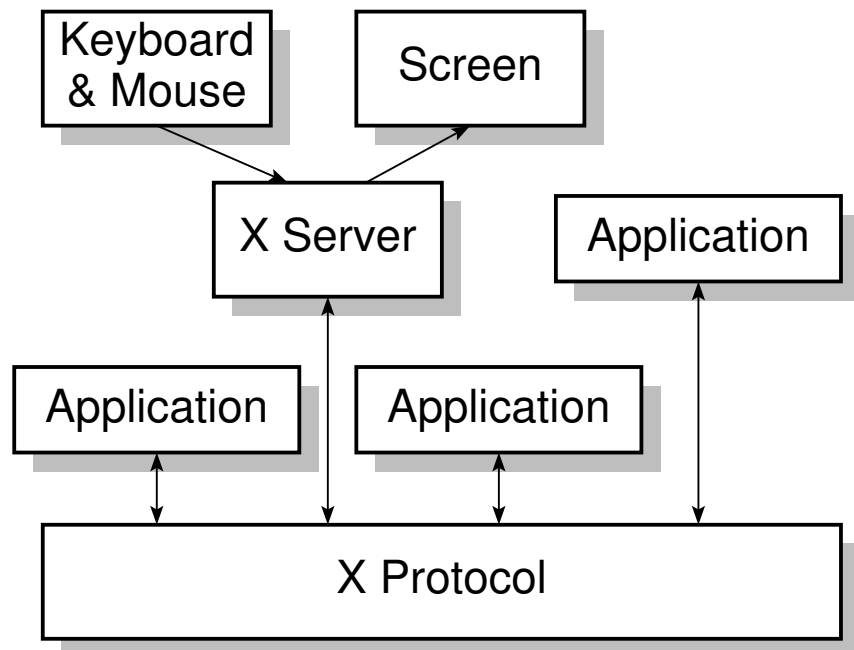




Figure 11.1: The X Window System as a client-server system


11.1 Fundamentals


The X Window System, “X11” or “X” for short (but never “X Windows”), is a graphics environment developed at MIT (the Massachusetts Institute of Technology) from 1985 to 1987.

 X11 originally derives from the MIT project, “Athena”, and was later continued under the auspices of the “X Consortium”, which eventually became part of the Open Group. The software was freely available from the start and contained, apart from various (portable) user programs, a server with support for most graphical workstations of the time.

X.org  The canonical implementation of X11 for Linux is called **X.org** (there are others, but they are not of practical concern).


X protocol The basis of X11 is the **X protocol**, a method of transmitting basic graphics operations across a network connection. X11 is a client-server system, albeit with a difference: The **X server** runs on a desktop computer with a graphics card, mouse, graphics tablet, or similar peripherals, and **X clients**—application programs—send graphics commands to it via the X protocol. Conversely, the X server sends events such as key presses and mouse movements to the X clients.


 Usually on Linux, X clients and the X server exchange X protocol messages using Unix-domain sockets, i. e., fast connections between programs on the same computer. There is, however, nothing to prevent you from transporting X protocol messages via TCP/IP. Therefore, X clients can absolutely run on computers that (in the extreme case) do not contain any graphics hardware of their own, as long as they can talk to the X server on another computer via the network. Of which more anon.


 The X protocol operations are fairly trivial—it supports mostly simple graphics operations such as drawing dots, lines, circles, rectangles or character strings. In addition, there are functions to manage windows (rectangular screen areas that can be the target of events like mouse clicks or key presses) and for internal organization. This implies that, if an application presents a pop-up menu, the X server must report a click on the menu’s


“button” to the application; the application then sends the graphics commands necessary to render the menu and takes over the user interaction—being fed with a constant stream of mouse movement messages by the X server—until the user decides on a menu item. Of course this produces considerable network traffic. This low-level interaction between server and application has often been criticised. It would be quite possible to move more of this interaction into the X server—one might, for example, invent a method with which the application tells the server about the menu entries, so that the server can handle all of the user interaction and return the index of the selected menu entry to the application at the end. However, this would mean that the server needs to know exactly how the menu should be rendered on screen and how the interaction is supposed to work. So far, the X11 philosophy is that the server offers graphics operations but does not regulate their use (“mechanism, not policy”)—and, especially with today’s fast networking, this is not unendurable enough to force such a “paradigm shift”. Various other graphics systems—Sun Microsystems’s “NeWS” in the 1980s or, more currently, the “Berlin” project—did try to implement this idea more consistently but were unsuccessful for various reasons; NeWS failed because of the inadequate (at that time) performance of computer workstations, and Berlin because X11 is quite “good enough”.

There are certain minimal system requirements for computers which are to support an X11 server. We shall not mention these here for fear of ridicule, since in the 21st century they are regularly surpassed by mobile phones and small netbook-style computers (not that many mobile phones actually use X11, but it wouldn’t be a problem in theory). The only interesting question these days is whether there are appropriate drivers for the computer’s graphics chip (either on the same die as the CPU or on an additional plug-in card). This is usually not a problem, although there may be (temporary) issues with exotic—i. e., not manufactured by one of the three “800-pound gorillas” of the business, Intel, AMD, or nVidia—or extremely new hardware.


 Serious computer graphics today is 3D graphics, even if what is displayed doesn’t really look three-dimensional at all. Ideally, applications can—with a little help from X.org—talk directly to the specialised 3D hardware on the graphics chip, and put strange and wonderful things on the screen at amazing speed. Less than ideally, parts of the 3D graphics are computed by the CPU, and that is still plenty fast enough for “office application”. Formerly, graphics chips had hardware acceleration for 2D graphics, but since no operating system today still uses 2D graphics, the manufacturers have stopped bothering.

 For the ideal case, you need a kernel driver for your graphics chip and another driver for X.org. The former takes care of basic graphics configuration and low-level operations, the latter of the operations in the X protocol.

 Generally speaking, Intel graphics hardware is best supported by Linux and X.org (which may have to do with the fact that various important X developers are working for Intel). However, as 3D performance is concerned, it does not quite play in the same league as the best graphics chips by the large manufactures, AMD and nVidia. For these there are both freely available drivers as well as “proprietary” drivers distributed by the manufacturers themselves, which have better hardware support but are not available as source code.

 X11 does not include 3D graphics operations, and so the “little help” X.org can provide to 3D clients generally amounts to making some graphics memory available to the client in which it can draw its output using direct 3D operations—typically using a graphics language such as OpenGL. A special X11 client, the “compositor”, then takes care of stacking the graphics


output of various clients in the correct order and adorning it with effects such as transparency, drop shadows, etc.


 The newest trend in Linux graphics is based on the observation that the X11 server tends to do little of a useful nature beyond providing “help”: The compositor and the clients are doing all of the real work. So what would be more obvious than getting rid of the X server altogether? The future infrastructure—Wayland—basically does exactly that. Wayland implements a compositor that can talk directly to the graphics chip. This makes the X server essentially superfluous. The prerequisite is that clients generate graphics output as 3D operations (in OpenGL) rather than X11 operations, but that is reasonable to enforce by adapting the “toolkits”, i. e., the programming environments for X11 clients. Today the popular toolkits already contain Wayland support of a more or less experimental nature.


The logical separation between X server and clients by way of the X protocol allows the server and clients to run on separate machines. On the same computer, you can theoretically start various clients that in turn communicate with different X servers¹. The interesting question that arises here is how a client figures out which server to talk to, and the answer is “By means of the DISPLAY environment variable”. To address the X server on `red.example.com`, you must set


```
DISPLAY=red.example.com:0
```


The “:0” at the end denotes the first X server on that computer. Something like display name `red.example.com:0` is also called a **display name**.

 In principle, nothing prevents you from running more than one X server on the same computer. Today there are relatively cheap “port extenders” with connections for a keyboard, a mouse, and a monitor, which are connected to a PC via USB. This allows you to have more than one user working on the same PC, which for the typical office application is not a problem at all. In such cases there will be one X server per “head”.

 The additional X servers on `red.example.com` will be called, respectively, `red.example.com:1`, `red.example.com:2` and so on.

 If you address an X server this way, that means you want to communicate with it using TCP/IP. (If the number after the colon is n , the client tries connecting to the TCP port $6000 + n$ on the computer in question. The X server listens to connections on that port.) If you would rather connect using a Unix-domain socket—which is vastly preferable when the server and client are running on the same machine—, then use the names `unix:0`, `unix:1`, etc.

 You can simply use `:0`, `:1`, ... This is equivalent to “pick the fastest local connection method”. On Linux, this typically amounts to a Unix-domain socket, but other Unix operating systems may support additional transport mechanisms such as shared memory.

 In principle, on top of addressing an X server on a computer you can even address a “screen” that is controlled by that X server, by adding a dot and the screen number—for example, `red.example:0.0` for the first screen of the first X server on `red`. We mention this mostly for completeness, because even if you connect several monitors to a Linux computer, these usually work as

¹In the 1990s, there was the idea of “X terminals”—basically specialised computers that ran little besides an X server, and whose sole purpose was to take care of the input and output of clients on one (or several) centralised computer(s), just like one used to have text-based terminals that took care of the input and output of programs on a centralised computer. At some point it became obvious, however, that PCs with Linux and X11 were typically much more powerful and flexible (to say nothing of “cheaper”), and X terminals (usually in the shape of Linux-based PCs) are now relegated to niche applications.

one large “logical” monitor, which is way more convenient if you’re actually sitting in front of them, but does not allow addressing the individual screens.



The advantage of the huge logical monitor is that you can move windows from one actual monitor to another. Windows can also be placed partly on one monitor and partly on the one next to it. This is usually what we want today. With separate screens, it is possible to drive the screens differently (e. g., one as a colour monitor and the other as a black-and-white monitor, way back when black-and-white monitors were still a thing), but then you would have to decide when launching a program on which screen it should appear, because moving it from one to the other after the fact is not allowed.

Being able to address the X server on arbitrary remote computers by means of their display names does not mean at all that these X servers actually want to talk to your clients—in fact, it would be a serious security hole to make your X server accessible to arbitrary clients (not just your own)². This means on the one hand that there is rudimentary access control (see Section 11.6), and on the other hand that many X servers do not actually bother listening for direct TCP connections at all anymore. Since otherwise anyone who can listen in to the data traffic between the client and server would be able to visualise what the client draws on the screen (they would simply need to interpret the X11 protocol messages), the preferred method today is to allow local connections only and enable remote access by means of “X11 forwarding” via the “Secure Shell”. This means that the X protocol traffic is encrypted by the Secure Shell, and eavesdroppers can no longer access the graphics output (and the user’s mouse and keyboard input).

security hole

X11 forwarding



You can find out more about the Secure Shell and X11 forwarding in the Linup Front training manual, *Linux Administration II*.

Besides the DISPLAY environment variable, most clients let you select the server on their command line using an option such as “-display red.example.com:0”. The environment variable does make it more convenient, though:

```
$ xclock -display red.example.com:0           Display on red
$ DISPLAY=red.example.com:0 xclock           The same
$ export DISPLAY=red.example.com:0
                                           All X clients started from now on will display on red
```

If you log in using a graphical environment, this variable should be set correctly on your behalf. Therefore you will not have to worry about it except in special cases.

Here are a few quick definitions of X11 terms:

Window manager A special X11 client whose job it is to control the placement (position and foreground/background) of windows on the screen. Clients may make suggestions, but the window manager (and thus the user) has the last word. There are various window managers with differing philosophies, for example concerning overlapping windows—some users prefer window managers that “tile” the screen and avoid overlap or outlaw it entirely. Many window managers today double as compositors for 3D graphics.

Display manager A program that manages the X server (or X servers) on a computer. The display manager allows users to log in using a graphical environment and subsequently constructs a graphical session for them. Many display managers also support session management, that is, they try to save the current state of the session when the user logs out and then to reconstruct it when the user logs in again.

session management

²A malicious X client could, for example, cover your complete screen with a transparent window, read all your key presses and look for passwords, credit card numbers, and the like. Or it could open thousands of windows and keep beeping obnoxiously.



How well session management works in practice depends on how thoroughly the X11 clients go along with it. Not all clients manage really well, and in fairness we should mention that many clients need to deal with very complex internal state that is not easy to save and restore.

Toolkit A programming environment for X clients. Toolkits provide programmers with the means to describe the graphical output of a program and how it deals with events like mouse clicks, key presses, and so on. This functionality is then mapped to X11 protocol messages by the toolkit. The main advantage of toolkits is that, as a programmer, you do not need to deal with the very primitive X11 operations, but can write code which is convenient in a high-level programming language. There are various toolkits, some of which are optimised for particular programming languages (such as C or C++), and some of which allow programming in other languages (like Python).

KDE
GNOME

Desktop environment When X11 was new, the developers were mostly about providing the technical means to implement graphical applications. They didn't really care about defining rules that governed how such applications should look and behave. Over the years this began to become a liability because almost every large program had its own conventions³. Desktop environments like KDE or GNOME sit on top of X11 and try to enforce (separate) uniform standards for the appearance and behaviour of a multitude of useful programs, as well as offer programs that actually implement these standards in order to provide a comfortable and consistent “user experience”.



The desktop environments do not preclude each other. If you are working in desktop environment X but would like to use a nice program that was written for environment Y, nothing keeps you from doing so—you may have to live with the fact that Y's runtime libraries must be loaded and that these will occupy additional memory. There are common standards for various basic functions such as cutting and pasting pieces of text which are supported by all desktop environments and facilitate “mixing and matching”.



Most desktop environments rely on specific toolkits—KDE, for example, on Qt, and GNOME on Gtk+. This means that if you plan to develop software that is meant for a particular desktop environment, you should use the toolkit in question to enable the best possible integration.

Exercises



11.1 [!1] If you are working in a graphical environment: What is your current display name?



11.2 [1] Try connecting a client to your X server via TCP/IP, by using a display name such as `red.example.com:0`. Does that work?




11.3 [1] What does the display name, `bla.example.com:1.1`, stand for? Give a command line that starts the `xterm` program such that its output appears on that display.

³The competition also didn't stay still—Apple and Microsoft were much more adamant in requiring a consistent “look and feel” for applications on their platforms.

11.2 X Window System configuration

If you want to find out whether your graphics system is supported by Linux and X.org, the simplest method is to boot the computer with a suitable (as up-to-date as possible) “live” Linux such as Knoppix and to see what happens. If you end up with a graphical environment then everything is fine.


 In former times, getting X11 to run on a Linux machine could border on black magic. It was not unusual to have to enter details of the graphics card in use, or detailed control parameters for the monitor, into a text file by hand (and at least for the once-ubiquitous fixed-frequency monitors it was quite possible to damage the screen beyond repair by getting this wrong). Fortunately, current versions of X.org can figure out for themselves what hardware they need to deal with both inside the computer and as the monitor, and what the optimal parameters are like. You can still override these manually if you like, but this is only necessary in exceptional cases.

If the X server does not start correctly, your first step should be to look at its log file, which is usually found in `/var/log/Xorg.0.log`. The X server uses this for a detailed report on the hardware it recognised, and the drivers and settings it allocated.

In principle, as the system administrator you can start the X server using the


```
# Xorg -configure
```

command. The X server will then try to detect the available hardware and write its findings to the `/etc/X11/xorg.conf` file as a rudimentary configuration. You can then use that file as the starting point for your own configuration.

 Instead of `Xorg`, you can also use the `X` command. According to convention, this is an abbreviation for “the appropriate X11 server for this system”.

`xorg.conf` is the central configuration file for the X server. It consists of separate sections, each of which starts with the `Section` keyword and ends with the `EndSection` keyword. Some sections, in particular those describing input and output devices, can occur several times. This allows you to, for example, use a mouse and touchpad simultaneously. Inside the configuration file, case is irrelevant except when specifying file names.

Syntax
sections

 As usual, blank lines are ignored, as are comment lines that start with the traditional hash sign (“#”). Lines with actual settings may be indented in order to make the file structure clearer.

Here is an overview of the most important sections in `xorg.conf`:

The `Files` section defines paths, namely:


Files


FontPath denotes directories that the X server searches for fonts, or a font server (Section 11.5). Usually there are many font directories and hence many `FontPath` directives. The order is important!


ModulePath describes directories containing extension modules for X.org. This is typically `/usr/lib/xorg/modules`.

RGBPath used to be used to name a file containing all the colour names known to the X server together with the corresponding RGB (red/green/blue) values. The conventional name is `/etc/X11/rgb.txt`, and this file is, confusingly, still part of many Linux distributions, presumably to let you look up valid colour names. You may use the colour names wherever X11 clients let you specify colors:

```
$ xterm -fg GoldenRod -bg NavyBlue
```

 If your desired colour is not mentioned in the file by name, you can also specify it as hexadecimal numbers. GoldenRod, for example, is #daa520 (the values for red, green, and blue are, respectively, 218, 165, and 32). The leading “#” denotes an RGB colour.

 The fanciful names like GoldenRod, PeachPuff, or MistyRose derive from the 72-colour Crayola set that the early X11 developer John C. Thomas happened to have to hand.

 In principle, nothing keeps you from adding your own favourite colours to the file. You should not remove any of the existing colours, nor change them too radically, since some programs may rely on their existence and appearance. In addition, it is fairly likely that your X server will not look at the file in the first place, since support for RGBPath is no longer included in X.org by default.

Here is a heavily abridged example:


```
Section "Files"
  ModulePath  "/usr/lib/xorg/modules"
  # RGBPath   "/usr/share/X11/rgb.txt" # no longer supported
  FontPath    "/usr/share/fonts/X11/misc:unscaled"
  <<<<<<
EndSection
```


Both FontPath and ModulePath may occur several times in the file; their values will then be concatenated in the order of appearance.

Module The Module section lists the hardware-independent X server modules that should be loaded, e.g., to support specific font types (Type 1, TTF, ...) or to enable particular features such as direct video rendering.

```
Section "Module"
  Load "dri"
  Load "v4l"
EndSection
```

The specific selection of modules depends on the X server; usually the list does not need to be changed. It can also be omitted entirely, in which case the X server will fetch whichever modules it needs.

 Modules will be looked for in the directories mentioned in ModulePath, as well as their subdirectories drivers, extensions, input, internal, and multimedia (if available).

 The extmod, dbe, dri, dri2, glx, and record modules will be loaded automatically if they exist. If that is not desired, you need to disable them using directives such as

```
Disable "dbe"
```

At least extmod, however, should be loaded in every case.

Extensions The Extensions section lets you specify which X11 protocol extensions should be enabled or disabled:

```
Section "Extensions"
  Option "MIT-SHM" "Enable"
  Option "Xinerama" "Disable"
EndSection
```

The names of extensions must be given using the correct capitalisation. You can generate a list of extensions using a command like

```
$ sudo Xorg -extension ?
```

There is no particular reason to disable specific extensions (except perhaps if you are doing compatibility tests). The server uses the ones that clients ask for and disregards the others.

The `ServerFlags` section influences the X server's behaviour. Individual options are set using the `Option` directive, and may be overridden within the `ServerLayout` section or on the command line when the server is started. This looks roughly like

```
Section "ServerFlags"
    Option "BlankTime" "10"                Screen saver after 10 minutes
EndSection
```

Some important server flags include:

AutoAddDevices Specifies whether keyboards, mice, and similar input devices should be recognised automatically by means of `udev`. Enabled by default.

DefaultServerLayout Specifies which arrangement of graphics cards, monitors, keyboards, mice, ... should be used by default. Points to a `ServerLayout` section. Other server layouts may be selected from the command line.

DontZap If this option is enabled (the default case), the server *cannot* be terminated using the `Ctrl`+`Alt`+`←` key combination.

Most options are switches that can assume values like 1, true, yes, or on or else 0, false, no, or off. If you specify no value at all, then true is assumed. You can also prepend a "No" to the option name, which means "no":

```
Option "AutoAddDevices" "1"                Enable option
Option "AutoAddDevices" "off"             Disable option
Option "AutoAddDevices"                  Enable option
Option "NoAutoAddDevices"                 Disable option
```

Other options have values that could be integers or strings. All values must be placed inside quotes.

Every `InputDevice` section configures one input device such as a mouse or keyboard. The section may occur several times. Here is an annotated example:

```
Section "InputDevice"
    Identifier "Keyboard1"
    Driver "Keyboard"
    Option "XkbLayout" "de"
    Option "XkbModel" "pc105"
    Option "XkbVariant" "nodeadkeys"
EndSection
```





This is a typical entry for a modern PC keyboard. The individual options have the following meanings:

Driver loads a module ("driver") from the `ModulePath`.

Identifier gives the section a name, so that it can be mentioned in a `ServerLayout` section.

XkbLayout enables a German-language layout.

XkbModel defines a standard (105-key) "international" PC keyboard.

XkbVariant The value `deadkeys` makes it possible to compose accented characters from several inputs, i. e., to input “ñ” as  . With `nodeadkeys`,   will produce “~n”.



If the `AutoAddDevices` server flag is set (the normal case), you really need no `InputDevice` sections at all since the X server will recognise the input devices automatically. A more detailed configuration is only required if, for example, the X server is not supposed to actually use all the available input devices.



In older configuration files you may sometimes still find the `Keyboard` or `Pointer` sections. These names are deprecated; use `InputDevice` instead.

Monitor The `Monitor` section describes the properties of the display device in use. This section may occur several times, too.

```
Section "Monitor"
  Identifier   "Monitor0"
  HorizSync   30-90
  VertRefresh 50-85
EndSection
```

Like `InputDevice`, this section needs an `Identifier` to be able to be referenced in `ServerLayout` sections. The horizontal and vertical frequencies may be found in the monitor’s documentation.

Modes



The optional `Modes` section (which may also occur several times) lets you specify your monitor’s display parameters in great detail. Our recommendation is to use the requisite time and energy for more profitable aims if you can manage this at all, since contemporary hardware can figure out the required settings all on its own. Having said that, here’s an example:

```
Section "Monitor"
  <<<<<<
  UseModes   "Model"
  <<<<<<
EndSection

Section "Modes"
  Identifier "Model"
  Modeline  "800x600" 48.67 800 816 928 1072 600 600 610 626
  Modeline  "640x480" <<<<<<
  <<<<<<
EndSection
```

(The `UseModes` directive in `Monitor` points to the `Modes` section that is to be used. You may also place `Modeline` entries directly within the `Monitor` section, or use the somewhat less compact `Mode` subsections either there or within a `Modes` section.



If you’re desperate to find out what the magic numbers in the mode definitions mean, then by all means consult `xorg.conf(5)`.

Device The `Device` section determines which graphics card the server should use. This section, too, may occur several times. Here’s an example for a minimal configuration using the VGA driver:

```
Section "Device"
  Identifier  "Standard VGA"
  Driver      "vga"
EndSection
```

Here's an example for an nVidia graphics card using the proprietary driver:

```
Section "Device"
  Identifier "Device0"
  Driver     "nvidia"
  VendorName "NVIDIA Corporation"
  BoardName  "NVS 3100M"
EndSection
```



If the system contains several graphics cards, you should use the `BusID` directive to specify the desired card's PCI address in order to avoid confusion. You can find the correct PCI address using the `lspci` command (for example):

```
# lspci | grep "VGA compatible"
01:00.0 VGA compatible controller: NVIDIA Corporation GT218M
```

The `Screen` section connects a graphics card and a monitor:

Screen

```
Section "Screen"
  Identifier "Screen0"
  Device     "Device0"
  Monitor    "Monitor0"
  DefaultDepth 24
  SubSection "Display"
    Depth     24
    Modes     "1280x720"
  EndSubSection
  SubSection "Display"
    Depth     24
    Modes     "1600x900"
  EndSubSection
EndSection
```

The subsections called `Display` determine various combinations of colour depths and resolutions, between which you can switch at runtime using `Ctrl` + `Alt` + `+` or `Ctrl` + `Alt` + `-`.

You may possibly have a `DRI` section which can contain settings for direct access to the graphics hardware by the X server.


The `ServerLayout` section describes the total configuration of the server including input and output devices. This is what you would use to specify the arrangement of multiple monitors:

ServerLayout

```
Section "ServerLayout"
  Identifier "Layout0"
  Screen     0 "Screen0" 0 0
  InputDevice "Keyboard0" "CoreKeyboard"
  InputDevice "Maus0" "CorePointer"
  Option     "Xinerama" "0"
EndSection
```





You need one `Screen` directive for every monitor you're using. The first zero is the screen number, which must be assigned contiguously starting from zero (it can be omitted, in which case the screens will be numbered in the order of their appearance). After the name of a `Screen` section which must occur elsewhere in the file there is a position specification, where "0 0" merely means that the upper left corner of this screen should correspond to the (0,0) coordinate. X11 coordinates increase to the right and downwards.

 If the Xinerama option is enabled, all Screens will be considered as fragments of one large logical screen whose size is adequate to fit all Screens. The individual Screens must be configured to have the same colour depth (today, usually 24 bits); the resolution must not be identical. For example, you could have a primary monitor with 1920 by 1080 pixels and also connect an LCD projector with 1024 by 768 pixels. With something like


```
Screen 0 "LaptopDisplay" 0 0
Screen 1 "Projector" RightOf "LaptopDisplay"
Option "Xinerama" "Enable"
```

you will then have a “logical” screen with a width of 2944 pixels and a height of 1080 pixels, where the top edges of both screens are aligned with each other.

 In the example from the preceding paragraph there is a “dead” strip of 1024 by 312 pixels which is theoretically present (X11 can only handle rectangular screens, whether physical or logical) but cannot really be used. In particular, new windows may not be automatically placed entirely within the dead strip, because you can’t drag them from there to a visible part of the screen. In such a situation you should make sure to use a window manager that supports Xinerama and ensures that such things won’t happen.

 Of course the monitors within the Screen lines may overlap (for example, it is often useful if a projector for a presentation shows the upper left corner of the laptop display). To do so, it is best to specify the position of the extra screen using absolute numbers:


```
Screen 0 "LaptopDisplay" 0 0
Screen 1 "Projector" 64 0           Leave room for left-edge control panel
Option "Xinerama" "Enable"
```


 You may include several ServerLayout sections in your configuration file and pick one of those by means of the -layout option when starting the X server (directly). This may be useful in order to have several configurations for the external video connector on a laptop computer.

Within a ServerLayout section you may also include options from the ServerFlags section, which will then only apply to that particular configuration.

Here is a brief Randsummary of the configuration file: At the highest level are the ServerLayout sections. These name input and output devices belonging to a configuration; these refer to InputDevice and Screen sections elsewhere in the configuration file. A Screen consists of a graphics card (Device) and an associated monitor (Monitor). The Files, ServerFlags, Module, and DRI sections apply to the X server as a whole.

Exercises

 **11.4** [!1] Look at the X.org configuration file on your system (if you have one at all⁴). Was it created manually or by X.org? Which devices are defined in it? Which server flags have been set?

 **11.5** [2] The X protocol transports graphics commands and events that allow screen display on an arbitrary X server connected to the X client via the network. Compare this approach to the similarly popular method of directly copying screen contents, as used by VNC and comparable products. What are the advantages and disadvantages of both approaches?

⁴X.org configuration file, that is.

11.3 Display Managers

11.3.1 X Server Starting Fundamentals

In principle, you can start the X server on your computer by simply using a text console to enter the

```
$ X
```

command. (In many cases the X server needs to run as, shock horror, `root`, but X will take care of that.) Next you can use the text console again to start a graphical terminal emulator by means of a command like

```
$ xterm -display :1
```

You can then use this `xterm` to launch further X clients as the shell running within the `xterm` will have its `DISPLAY` variable set appropriately.



You shouldn't use this method in real life—on the one hand it is terribly inconvenient, and on the other hand the approaches shown next will avoid a lot of hassle as far as ease of use and security go.

A more convenient way to start X from a text-based session is to use the `startx` command. `startx` makes a few initialisations and then invokes another program called `xinit`, which does the actual work—it arranges for the launch of the X server and initial X clients.

You can use the `~/.xinitrc` and `/etc/X11/xinit/xinitrc` files to start X clients, such as a clock, a terminal emulator, or a window manager. All X clients must be launched to the background by putting an “&” at the end of the line. Only the final X client—usually the window manager—must be started in the foreground. If this final process is terminated, `xinit` stops the X server. Here is a simple example for a `~/.xinitrc` file:

```
# A terminal
xterm -geometry 80x40+10+10 &
# A clock
xclock -update 1 -geometry -5-5 &
# The window manager
fvwm2
```

You can use the `-geometry` option to specify beforehand where the windows should appear.



The value of `-geometry` consists of an optional size specification (usually in pixels, but for some programs, such as `xterm`, in characters) followed by an optional position specification (they're both optional but you should really have at least one of the two). The position specification consists of an *x* and a *y* coordinate, where positive numbers count from the left or top edge of the screen while negative numbers count from the right or bottom edge.

You can also specify a server number when invoking `startx`:

```
$ startx -- :1
```

This would let you start a second X server.

Exercises



11.6 [2] How would you start an `xclock` such that it is 150 by 150 pixels in size and appears in the lower left corner of the screen, 50 pixels away from the screen's edges?



11.7 [2] Try to start an additional X server using the “`startx -- :1`” command. (This should manifest itself on the `tty8` console, thus should be reachable using `Ctrl` + `Alt` + `F8`.)

11.3.2 The LightDM Display Manager

On modern workstation computers it is usual to start the graphical environment when the system is booted, by means of a display manager. The common distributions offer several display managers; the system picks one to start according to a distribution-specific selection mechanism.



Since version 4.0 of the LPIC-1 certification, particular attention has been given to the LightDM display manager, which we shall explain in more detail. You should be aware that other display managers such as `xm`, `kdm`, or `gdm` exist.

LightDM is a popular display manager which is largely independent from specific desktop environments. As its name suggests, it is relatively parsimonious with the system's resources, but can do all that is required from a display manager, and can be installed (at least optionally) on all important Linux distributions.

One important function of a display manager is letting users log into the system in a graphical environment. LightDM does not do this itself, but delegates this to so-called “greeters”. There are various greeters that are usually written to blend with different desktop environments. The greeters control the appearance of the login screen.

Configuration The configuration of LightDM is contained in files whose names end in `.conf` within the `/usr/share/lightdm/lightdm.conf.d` and `/etc/lightdm/lightdm.conf.d` directories, as well as the `/etc/lightdm/lightdm.conf` file. The configuration files are read in that order. As the system administrator, you should ideally make changes by placing files in `/etc/lightdm/lightdm.conf.d`. The configuration files are composed of sections that have titles within square brackets and contain key-value pairs. You could, for example, change the X server layout by creating a file called `/etc/lightdm/lightdm.conf.d/99local.conf` containing the lines

```
[Seat:*]
xserver-layout=presentation
```

The most important sections of the LightDM configuration include:

[LightDM] Configuration for LightDM as a whole. This includes parameters like the user identity used for executing greeters, the directories for log files, runtime data, and session information, and similar settings.

[Seat:*] (or, for older versions, `[SeatDefaults]`) Configuration for a single “seat” (combination of graphics card, monitor(s), keyboard, mouse, ..., that is being controlled by a single X server). This lets you specify whether the seat is locally connected or remote (“X terminal”), how the X server is invoked, how the greeter should work and be launched, how the session should be constructed, and whether a user is logged in automatically. All of these settings apply to all seats connected to this computer unless they are specifically overwritten.

Some common settings include the following:

```
[Seat:*]
greeter-hide-users=true
greeter-show-manual-login=true
```

Hides the clickable list of users in the greeter and allows the textual entry of user names. This can be useful for security purposes or because you have too many users and the list would therefore be unwieldy.

```
[Seat:*]
autologin-user=hugo
autologin-user-timeout=10
```


When the greeter is executed, it waits 10 seconds for user interaction before automatically logging in the user `hugo`. Of course you should only use something like this when there is no chance of random people switching on your computer.

```
[Seat:*]
user-session=mysession
```


Establishes `mysession` as the session name. This presupposes the existence of a file called `/usr/share/xsessions/mysession.desktop` describing the desired session. This could look roughly like

```
[Desktop Entry]
Name=My Session
Comment=My very own graphical session
Exec=/usr/local/bin/startmysession
Type=Application
```

where `/usr/local/bin/startmysession` would typically be a shell script that established the session.

 The actual details here would be somewhat involved; do take your inspiration from a file called `/etc/X11/xsession` or `/usr/bin/startlxde` (depending on what desktop environment you have installed).

[Seat:0] (and **[Seat:1]** and so on) Configurations for individual seats that differ from the basic settings in **[Seat:*]**.


 If you want to control more than one seat, you must list the desired seats within the **[LightDM]** section:


```
[LightDM]
seats = Seat:0, Seat:1, Seat:2
```

[XDMCPServer] Remote seats (“X terminals”) use XDMCP (the “X Display Manager Control Protocol”) to contact the display manager. This section includes settings for XDMCP, including by default

```
[XDMCPServer]
enabled = false
```

[VNCServer] This section lets you configure an X server which is accessible via VNC (the program is called `xvnc`). This enables remote access from computers which do not support X at all—there are VNC clients for many different operating systems.

 The LPI's exam objectives mention that you should be able to “change the display manager greeting”. It turns out that LightDM's standard greeter does not even support a greeting message in the first place, so we must take a pass here. This may well be different for other, less common, greeters—check whether there is a configuration file for the greeter in question within `/etc/lightdm`, and if so, what you can put in there.

 What you *can* set up even with the standard greeter is a background image (preferably in the SVG format). You can go wild here using an SVG editor such as Inkscape, and you will only need to ensure that the `/etc/lightdm/lightdm-gtk-greeter` contains something like

```
[greeter]
background=/usr/local/share/images/lightdm/my-greeter-bg.svg
```


Starting and Stopping The display manager is launched by the init system. This means that something like

```
# service lightdm start
```

should let you start LightDM, regardless of whether your system is based on System-V init or systemd. Accordingly, something like

```
# service lightdm stop
```

should also work.

 Alternatively, you can invoke the init script directly (with System-V init) or say

```
# systemctl start lightdm or stop
```

To activate the display manager on boot for System-V init, you should ensure that the LightDM init script is active within the desired run level (typically 5). (Of course you will also want to deactivate any other display manager(s). Display managers operate on the Highlander principle, at least per individual X server.) On systemd-based systems, something like

```
# systemctl enable lightdm
```

should suffice to activate, and something like

```
# systemctl disable lightdm
```

to deactivate LightDM on boot. By rights, your Linux distribution should take care of details like that.

11.3.3 Other Display Managers

Here are a few remarks about the more traditional display managers mentioned in the LPI exam objectives (there are more).

xdm `xdm` is the default display manager of the X11 system. Like many X11 sample programs, it is very plain—it offers just a simple graphical login window. It is configured using files within the `/etc/X11/xdm` directory:

Xresources This lets you set up—among others—the greeting message (`xlogin*greeting`), the font used for that (`xlogin*login.greetFont`), or the logo displayed by `xdm` (`xlogin*logoFileName`).

Xsetup This is a shell script that will be executed when `xdm` is started. Among other things, this lets you start a program that puts a background image on the login screen.

Xservers This determines which X servers will be started for which displays.

Xsession Plays a similar role for `xdm` as `~/xinitrc` for `startx` or `xinit`, as far as the initialisation of a user session is concerned; here, too, there is a user-specific analogue, namely `~/xsession`.

kdm `kdm` derives from the KDE project and is basically an extension of `xdm`. Its configuration corresponds to that of `xdm` (the configuration files may be placed elsewhere). You can also configure `kdm` by means of the KDE control center, `kcontrol`, whose settings are placed in a file like `/etc/X11/kdm/kdmrc`. `kcontrol`

gdm The GNOME display manager, `gdm`, is part of the GNOME desktop environment. It was developed from scratch, but offers approximately the same features as `kdm`. It is configured using the `gdm.conf` file, which can often be found in the `/etc/X11/gdm` directory. For `gdm`, too, there is a convenient configuration program by the name of `gdmconfig`. `gdm.conf`
`gdmconfig`

Exercises



11.8 [3] Does the display manager on your system (if there is one at all) allow a choice between various desktop environments or “session types”? If so, which ones? Try a few of them (including, if available, the “failsafe” session).

11.4 Displaying Information

Once your X session is running, you may use various programs to display interesting information.

`xdpyinfo` shows information about your current X display. We shall highlight only a few interesting elements: `xdpyinfo`

```
$ xdpyinfo
name of display:      :0                               On the local computer
version number:      11.0                             Not a big surprise
vendor string:       The X.Org Foundation
vendor release number: 11702000
X.Org version:       1.17.2                           The server's version number
<<<<<<
number of extensions: 30                               Loaded extensions
  BIG-REQUESTS
  Composite
  DAMAGE
  <<<<<<
  XVideo
default screen number: 0                               Standard screen ...
number of screens:   1                               ... just one there!

screen #0:
  dimensions:        1680x1050 pixels (442x276 millimeters)
  resolution:        97x97 dots per inch
  depths (7):        24, 1, 4, 8, 15, 16, 32
  root window id:    0x8f
  depth of root window: 24 planes
```

```

number of colormaps:   minimum 1, maximum 1
<<<<<<
options:   backing-store WHEN MAPPED, save-unders NO
largest cursor:   64x64
current input event mask:   0xfac033
  KeyPressMask  EnterWindowMask  LeaveWindowMask
  <<<<<<
number of visuals:   204
default visual id:  0x21
visual:
  visual id:   0x21
  class:   TrueColor
  depth:   24 planes
  available colormap entries:   256 per subfield
  red, green, blue masks:   0xff0000, 0xff00, 0xff
  significant bits in color specification:   8 bits
  <<<<<<

```

We will skip 203 other visuals

This describes fairly accurately the graphical possibilities offered by this X server. 24 bits of colour depth are what one would like to see today—this allows a possible 16 million different colours, while the visual system of normal humans only lets us distinguish around 100,000. A “visual” describes how to put pixels of specific colours onto the screen, where once more TrueColor is the holy grail⁵. There are other types of visuals that, these days, are of any concern to hard-core X developers only.



You can find out more about `xdpinfo` by consulting `xdpinfo(1)`, which describes the program in detail.

`xwininfo` You can obtain information about individual windows by means of the `xwininfo` command. After starting it in a terminal emulator, it prompts you to click on the desired window using the mouse:

```

$ xwininfo

xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.

xwininfo: Window id: 0x500007d "emacs@red.example.com"

Absolute upper-left X: 1071
Absolute upper-left Y: 27
Relative upper-left X: 0
Relative upper-left Y: 0
Width: 604
Height: 980
Depth: 24
Visual Class: TrueColor
Border width: 0
Class: InputOutput
Colormap: 0x20 (installed)
Bit Gravity State: NorthWestGravity
Window Gravity State: NorthWestGravity
Backing Store State: NotUseful
Save Under State: no
Map State: IsViewable

```

⁵Users of older graphics cards may remember, in principle, visuals of type `PseudoColor`—typically one got to pick 256 colours out of the proverbial 16 million. Not nice.

```
Override Redirect State: no
Corners: +1071+27 -5+27 -5-43 +1071-43
-geometry 80x73-1+0
```

Here you can see, for example, the coordinates of the window on the screen, the visual in use, and similar data. The “backing store state” and the “save under state” determine what happens to window content that is obscured by other windows (typically pop-up menus)—in our case, they will not be stored but redrawn as required, which on today’s computers is often faster—, and the “map state” specifies whether the window is actually visible on screen.

You can find the window information without a mouse click if you know the window ID or the window’s name. The sample output shown above mentions both of them—the window name is on the same line as the window ID.

`xwininfo` supports various options that control the type and extent of the data being output. Most of that is only interesting if you know about X’s inner workings. Play around with this for a bit. The details are in `xwininfo(1)`.

Exercises



11.9 [!1] What is the resolution of your screen in pixels? Are the metrical sizes output by X11 correct (and hence the resolution in dots per inch)?



11.10 [2] Use `xwininfo` to display information about a window on your screen. Move and/or resize the window and make sure that `xwininfo` outputs different coordinates.

11.5 The Font Server

The X11 protocol contains not just operations for drawing points, lines, and other geometrical shapes, but also for displaying text. Traditionally, the X server offers a selection of fonts; the client can query which fonts exist, and then specify which font should be used to display text at what position on the screen. Actually obtaining the font data and displaying the text are the server’s job.





It turns out that today practically no modern X client still uses this mechanism. X.org and the common toolkits support the `XRENDER` extension, which among other things can handle transparency to allow “antialiasing”. This in turn enables the display of scalable fonts with smooth edges and is necessary for high-quality text output. With `XRENDER`, the X11 operations for text display are avoided entirely; instead, the client can upload “glyphs” (letters, digits, and other characters) and use these for rendering text. The fonts offered by the X server are unimportant for `XRENDER`, since the fonts are installed on the client and will be made available to the X server on a glyph-by-glyph basis [Pac01].



The remainder of this section is only interesting if you want to pass the LPI-102 exam. It is no longer of any conceivable relevance to real life. Save your time and do something useful instead—clean your bathroom or walk the dog.

Local Fonts on the Server If you do indeed want to (or need to) use the X11 text operations and therefore the fonts provided by the X server, you must first make sure that the X server can find the fonts in question. The first port of call for configuring fonts in X.org is the `Files` section of the configuration file, with the directories to be set up there (`FontPath` entries) that the X server will search for fonts. Typically, one `FontPath` entry per directory will be added.

 In practice this means that, as far as font installation is concerned, you must place the fonts in an appropriate directory and make that directory known to the X server by means of a `FontPath` entry. The order of the entries in the configuration file is very important because it determines the X server's search order. The first matching font will always be used.


`xset`  Instead of adding a font directory permanently to the configuration file, you can add it to the X server temporarily using the `xset` command. The

```
$ xset +fp /usr/share/fonts/X11/truetype
```

command adds a font directory temporarily (until the next restart of the X server).

```
$ xset -fp /usr/share/fonts/X11/truetype
```

lets the X server forget about it again.

 The “`xset q`” command lets you query the X server's current configuration, and therefore check that it knows about the correct font directories.

Copying the fonts and announcing the font directory are often not enough. Apart from the fonts themselves, the directory may contain the following files:

`fonts.dir` The `fonts.dir` file contains a list of all the fonts contained in the directory, including the file name, manufacturer, font name, font weight, slant, width, style, pixel and point sizes, *x* resolution, *y* resolution, font encoding, and various other data. A (one-line) entry for a font could, for example, look like

```
luBIS12-IS08859-4.pcf.gz-b&h-lucida-bold-i-normal-sans-▷
◁ 12-120-75-75-p-79-iso8859-4
```

The first line of the file gives the total number of fonts in the directory.

`mkfontdir` The `fonts.dir` file must exist. Of course, though, you will not have to maintain it by hand, but can use the `mkfontdir` command to do so. Assuming you have added a font file to the `/usr/local/share/X11/fonts/truetype` directory, a call to

```
# mkfontdir /usr/local/share/X11/fonts/truetype
```


will suffice to update the corresponding `fonts.dir` file. If you add fonts to an existing font directory, you must issue the

```
$ xset fp rehash
```

command within your running session for the X server to be able to find these fonts.

`fonts.scale` The `fonts.scale` file is useful for directories containing scalable (vector-based rather than bitmap-based) fonts and gives a list of the fonts in question, while the `fonts.alias` file lets you set up alias names for individual fonts.

The Font Server The font server, `xfst` (not to be confused with the XFS file system), makes it possible to centrally manage fonts on a network. With today's prices for hard-disk storage, a complete centralisation of X11 fonts is no longer necessary (nor desirable), but, for specialised fonts which are not part of the X11 or Linux distribution and should be available within the local network, this can make font management considerably easier.

 The font server has yet another advantage: Without a font server, problems may arise if a client queries the X server for a list of available fonts. The X server will then drop everything it does and search the system for fonts, which can lead to considerable delays because during that time no X11 operations for graphics display will be executed.

`xfst` is a free-standing daemon. It offers its services on the TCP port 7100 and is configured using the `/etc/X11/fs/config` or `/etc/X11/xfst.conf` files. After any changes to these files, the server must be informed of the new situation by means of a `SIGHUP` signal:

```
# pkill -1 xfst
```

`xfst` accesses files that are installed as described for the X11 server, and can thus also be used by the local X server. The font directories are entered into the configuration file by means of the `catalogue` parameter, for example as follows:

```
catalogue = /usr/share/fonts/X11/misc:unscaled,
            /usr/share/fonts/X11/75dpi:unscaled,
            /usr/share/fonts/X11/100dpi:unscaled
            <<<<<<
```


The individual paths must be separated by commas. Further options can be found in the documentation.


To connect an X server to the font server, you just need to add an entry to the `Files` section of the `xorg.conf` file:


```
FontPath  "tcp/<host name>:<port number>"
```

If you want to prefer fonts from the font server, you should add this entry in front of all the other `FontPath` sections. (For experimentation, do it using `xset`.)

Exercises

 **11.11** [1] Use the `xlsfonts` command to list the fonts your X server knows about.

 **11.12** [1] Use the `xfontsel` command to conveniently explore the selection of fonts. Find a font you like and remember it for the next exercise.

 **11.13** [2] Start a sufficiently antique X client (`xman`, `xterm`, or `xedit` would come to mind) using the font from the previous exercise. The canonical command line option to do so is `-fn` (as in “font”). What happens?

11.6 Remote Access and Access Control

In principle, as mentioned above, the X server can be reached by remote clients via a TCP port (6000 + “server number”). These just need to be started with the correct display setting—via a command-line option à la `-display` or the `DISPLAY` environment variable—and can display their output on the server and accept input, but theoretically also disrupt or spy on the session as desired.

There are two “native” methods to control access to the X server, namely `xhost` and `xauth`.

`xhost` offers host-based access control. Using

`xhost`

```
$ xhost red.example.com
```

or

```
$ xhost +red.example.com
```

you allow access to your server to clients running on `red.example.com`. The command

```
$ xhost -red.example.com
```

withdraws that permission again. “xhost” on its own outputs a list of computers authorised for remote clients. Since *any* user on the remote host can have access to your X session, you should *not* use xhost in real life!!



Sometimes—usually within installation instructions for third-party proprietary software packages—you will be asked to execute the “xhost +” command. This opens your X server to clients from arbitrary other computers (theoretically the complete Internet). Don’t fall for that kind of thing.

xauth With xauth, a random key or “magic cookie” is created and passed to the X server when the server is started (usually by the display manager or startx).
 ~/.Xauthority The key is also stored in the ~/.Xauthority file of the current user, which other users cannot read. The X server only accepts connections from clients which can present the correct magic cookie. Using the xauth program, magic cookies can also be transferred to other hosts or removed from them. More details are in xauth(1).



A much more secure method to start X clients on remote hosts consists of using the X-forwarding feature of the Secure Shell. We describe this in *Linux Administration II*.

The “-nolisten tcp” option lets you make your X server completely inaccessible from the outside. This is a sensible setting and many Linux distributions today default to it.

Exercises



11.14 [2] Make sure that your X server is *not* started using “-nolisten tcp” (or start another X server using something like “startx -- :1”), and try to connect a client to your server by means of a suitable DISPLAY setting (extra credit, if your client runs on a different host). (*Hint:* If you have trouble connecting from a different host, check whether your computer uses a hyperactive packet filter which shields your X server from outside connections. The SUSE distributions, in particular, like to do this.)

Commands in this Chapter

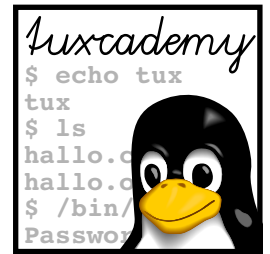
x	Starts the appropriate X server for the system	x(1)	169
lspci	Displays information about devices on the PCI bus	lspci(8)	167
xauth	X server access control via “magic cookies”	xauth(1)	178
xdpyinfo	Shows information about the current X display	xdpyinfo(1)	173
xhost	Allows clients on other hosts to access the X server via TCP	xhost(1)	177
xwininfo	Displays information about an X window	xwininfo(1)	174

Summary

- X11 is a client-server network-transparent graphics system.
- The X server manages a computer's graphics screen, keyboard and mouse; X clients access the X server via the X protocol.
- On workstations, X11 is often installed such that a graphical login is possible. On other computers, the graphical environment can be started by hand if needed.
- Apart from the simple `xdm` display manager, most desktop environments furnish their own display manager.

Bibliography

- Pac01** Keith Packard. "Design and Implementation of the X Rendering Extension". *Proc. FREENIX Track, 2001 Usenix Annual Technical Conference*. The USENIX Association, 2001 pp. 213–224.
<http://keithp.com/~keithp/talks/usenix2001/>



12

Linux Accessibility

Contents

12.1 Introduction	182
12.2 Keyboard, Mouse, and Joystick	182
12.3 Screen Display	183

Goals

- Learning about Linux accessibility facilities

Prerequisites

- Basic knowledge of Linux, X11, and graphical environments like KDE or GNOME

12.1 Introduction

Computers and the Internet extended the range of activities that disabled people can attend and so increases their quality of life a lot. Visually impaired and blind people have access to much more information than ever before, and people with other special needs also benefit greatly from the ways in which the computer allows them to express themselves, make friends, work, and dig up information.

facilities This chapter provides a brief summary of the facilities offered by Linux and the software packages distributed with it to make life a little easier for people with special needs. We will not dive into detailed technical discussions at this point, but we will focus on the big picture and tell you where to get more information when needed.

12.2 Keyboard, Mouse, and Joystick

People who are not in a position to use an ordinary keyboard or mouse—be it because they are missing the requisite limbs or because these cannot be controlled accurately enough—can resort to various aids available on Linux. Typical assistive tools include:

Sticky keys arrange for modifier keys like the shift and control keys not to have to be held down while you are pressing another key. A press (and subsequent release) of the modifier key is enough for the next key to be evaluated as if the modifier key was still held down. This helps, for example, paraplegics who can only move their head type with the aid of a stick.

Slow keys let the system ignore unwanted key presses that arise when you press other keys on the way to the key that you really want.

Bounce keys arrange for the system to ignore extraneous presses of the same key.

Repeat keys allow you to specify whether held-down keys should be repeated or just reported once.

Mouse keys allow the mouse to be controlled using the numeric key pad on the keyboard.

XKEYBOARD With X11 on Linux, these tools are provided by the XKEYBOARD extension, which by default is part of the X server. Therefore the challenge is only how to activate it. This is done using the `xkbsset` utility. The graphical desktop environments also offer user interfaces like the KDE control center (see the “Accessibility” dialog below “Regional & Accessibility”).

screen keyboard For GNOME, at least, there is a “screen keyboard” called GOK, which allows users to “type” by means of a mouse, a joy stick, or even a single key. This is a tool for people who cannot handle a keyboard but can use the mouse. For the time being KDE does not offer a corresponding facility.

In many cases the mouse can be replaced by the keyboard. At least in theory, all features of the graphical environments should also be available via the keyboard.

RSI People who cannot use the mouse, for example due to repetitive strain injury (RSI), may prefer to use a stationary “track ball”. For people with motor difficulties it may also help to increase the delay for double clicks.



In the KDE control center you will find the settings for using the numeric key pad as a “mouse substitute” under “Peripherals/Mouse”. With GNOME, the equivalent is part of the keyboard configuration.

12.3 Screen Display

For the visually impaired and blind, Linux has the considerable advantage that screen display can be very finely controlled. Since the system does not depend on a graphical interface, it is much easier to operate for blind people using a Braille display or screen reader than purely graphical systems.

People with some remaining vision can configure Linux in such a way that it magnifies parts of the display or the mouse cursor, so they can be spotted more easily. On KDE, for instance, the looks of the mouse cursor can be altered in the “pointer design” tab under “Peripherals/Mouse”. (You may have to install a special “accessible” mouse cursor theme, though.) On a GNOME desktop, the appearance of the mouse cursor is also set up in the mouse configuration section.

pointer design

Both KDE and GNOME allow you to change the size of the display fonts. On high-resolution displays, fonts can be tiny and hard to read even for people with excellent vision, so visually impaired people benefit greatly from a generously “oversized” text representation. A high-contrast color scheme also helps to make the desktop easier to view.

font size

Another common tool are “screen magnifiers”, which display a greatly magnified copy of the area around the mouse cursor. The KMagnifier program of KDE and the GNOME equivalent that can be found under “Accessibility” in the system setup dialog both offer this useful feature.

screen magnifiers

For blind people, Linux supports various Braille displays as well as voice output. A Braille display can show a line of text using Braille dot representation, which the blind can feel using their fingertips; Braille devices, however, are fairly expensive and mechanically intricate. BrlTTY is a program that runs on the Linux console and controls a Braille display. Orca is the same for GNOME. Then there is Emacspeak, which is essentially a screen reader for GNU Emacs that reads screen contents aloud; because many other programs can be launched inside Emacs this is nearly as good as a graphical desktop.

Braille displays

voice output

BrlTTY

Orca

Emacspeak



KDE has some difficulties with some of the facilities described here. There is a reasonably well-established protocol called AT-SPI (Assistive Technologies Service Provider Interface) which is used on Unix and Linux to facilitate the communication between accessible software and technical aids like Braille displays. Unfortunately (from the point of view of KDE), AT-SPI was invented by GNOME developers and is based on the GTK2+ (a graphics library that GNOME is based on, but which is no use to KDE). It also uses CORBA for communication, which does not fit KDE either. How to solve these problems is still unclear.

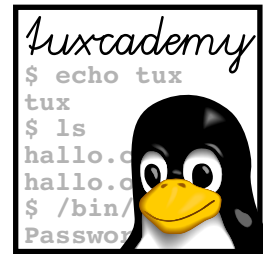
Commands in this Chapter

xkbset Controls keyboard setup options for X11

xkbset(1) 182

Summary

- Computers and the Internet can help disabled people increase their quality of life.
- Linux provides a variety of aids for people who cannot operate an ordinary mouse or keyboard.
- Blind and otherwise visually impaired people benefit from facilities like screen magnification, high-contrast color schemes, screen readers with voice output and Braille displays that Linux can control.



A

Sample Solutions

This appendix contains sample solutions for selected exercises.

1.1 Some conceivable advantages include: Shell scripts are generally much quicker to develop, they are independent of the computer architecture (Intel, PowerPC, SPARC, ...) and shorter than equivalent programs in a language like C. Shell programming, at least at an elementary level, is much easier to learn than programming in a language like C. Disadvantages include the fact that shell scripts often make less efficient use of the system than compiled programs, that the number of available data structures is very limited, and that shell scripts do not lend themselves to the implementation of software that needs to fulfil strong security requirements.—Shell scripts are most useful for “ad hoc” or “throw-away” programming where not much time is available, for the creation of prototypes (where there is a considerable chance that the prototype will prove “good enough”), and for automating tasks that would otherwise be done from the command line. Programming languages like C always imply a larger development effort, which must be worthwhile; therefore, a common approach consists of implementing a program as a shell script first and later replacing, e. g., performance-critical parts by C programs. The optimum is often a mixture of shell and binary code.

1.2 One possible approach might be:

```
find /bin /usr/bin -type f -exec file \; \  
| grep "shell script" | wc -l
```

The `find` command enumerates all files in `/bin` and `/usr/bin` and applies `file` to each file name in turn. `grep` picks up those lines that contain “shell script”, and `wc` determines their number.—How could you make this pipeline execute more quickly?

1.3 A “quick and dirty” method would be the command

```
ls $(cat /etc/shells) 2>/dev/null
```

which lists those shells that are approved as login shells (error messages about shells listed in the file but not installed on the system as programs are suppressed by redirecting standard error output to `/dev/null`). Also note that `/etc/shells` allows comment lines starting with a hash mark, which cause additional error messages that are also suppressed.

1.4 The `tcsh` helpfully tries to correct your erroneous command and suggest a “correct” version that you can accept, reject, or edit, or cancel the command altogether.—You can leave the `tcsh` by means of the `exit` command; virtually all shells also support `Ctrl+d` as “end of file on standard input”.

1.7 A “-” is passed as the first character of the program name.

1.8 Arrange for the shell to believe it was called as a login shell: Use `bash` as the actual login shell, with a `.profile` file like

```
PATH=$HOME/bin:$PATH
exec -mysHELL
```

where `\$HOME/bin/-mysHELL` is a (symbolic) link to `/usr/local/bin/mysHELL` (or wherever your desired shell ended up). This shell sees “-mysHELL” as its program name and initialises itself as a login shell—at least if it plays by the rules. The “exec” is not strictly necessary; it causes the login `bash` process to replace itself by `mysHELL` instead of invoking `mysHELL` as a child process. A simple “-mysHELL; exit” would suffice (why the exit?).

2.2 As these files can be used, e. g., to set environment variables the method used must apparently be `source` ...

2.3 Instead of the user’s script, the `/bin/test` program (or the internal shell command `test`, e. g., with `bash`) was started. This happens when the directory containing the script is not part of the user’s `PATH`, or `/bin` occurs before that directory. Perfidiously, the `test` command, when invoked without parameters, does not produce an error message—in our view, a grave omission on the part of both the external as well as the shell’s implementation.

2.4 The shell script must write the name of the desired new directory to its standard output, and be invoked by the shell using something like

```
$ cd `myscript.sh`
```

This somewhat tedious call is best hidden away in an alias name:

```
$ alias myscript='cd `myscript.sh`'
```

(For extra credit: Why are the single quotes important?)

2.5 The invocation command “./baz” is appended to the first line and this is executed. Therefore, the output

```
foo bar ./baz
```

appears. Since the `echo` program does not process its parameters as file names, the actual content of the file is irrelevant; thus the “echo Hello World” is pure obfuscation.

2.6 It depends on the script. As mentioned earlier in this document, “#!/bin/sh” is basically a promise by the shell script saying something like “I can be executed using any Bourne-like shell”. Such a script should therefore not contain any `bash`-specific constructions. Accordingly, the “#!/bin/bash” line says “I do, in fact, need `bash`”. Whoever wants to integrate such a script in their own software system is warned that they might have to take along the rather hefty Bourne-again shell, when they might otherwise have been able to use one of the slimmer shells (`dash`, `busybox`, ...). Thus, always writing “#!/bin/bash” does not really make sense; always writing “#!/bin/sh”, on the other hand, is more dangerous.

2.7 One workable solution might be:

1. Construct a list of all “real” users.
2. Repeat the following steps for each user u in the list.
3. Determine the time t of u 's last login.
4. Determine the home directory v of u .
5. Determine the amount p of disk space used by v .
6. Output u , t and p
7. End of the repetition.

(Of course there are others.)

2.8 One possible approach would be:

1. Obtain a sorted list of all users' home directories
2. Create from that a list of all directories containing home directories (`/home/develop` and `/home/market`, in our case)—any duplicates should be removed.
3. For each of these directories d , check the used space b_d :
4. If $b_d > 95\%$, include d in the warning list
5. Send the warning list to the system administrator

Here, too, countless other possibilities are conceivable.

3.1 `$*` and `$@` behave equivalently, except for one special case—the expansion of `"$@"`. Here every positional parameter becomes a single “word”, while `"$*"` results in a single word containing all parameters.

3.2 The shell always rounds towards the nearest integer whose absolute value is less than that of the result. Since it does not support floating-point numbers, this is an obvious (though not necessarily optimal) approach.

3.3 On “standard” Linux systems, `bash` uses 64-bit arithmetic, thus the largest number that can be represented is $2^{63} - 1$ or 9.223.372.036.854.775.807. If you do not want to check the source code, you can execute a command such as

```
$ a=1; while true; do a=$((2*a)); echo $a; done
```

and interpret the result.

3.4 Use something along the lines of

```
#!/bin/sh
echo $1 | grep -i '[âeiou]'
```

(parameter checking *ad libitum*). The return value of a script is the return value of the last command, and a pipeline's return value is the return value of the pipeline's last command. What `grep` does is exactly right—bingo!

3.5 Try something like

```
#!/bin/sh
# absrelpath -- check path names for absoluteness

if [ "$1:0:1" = "/" ]
then
    echo absolute
else
    echo relative
fi
```

Other versions—like

```
[ "$1:0:1" = "/" ] && echo absolute || echo relative
```

—are conceivable but possibly too cryptic for serious use.

3.6 According to general usage, the easiest method to do this is another case alternative containing something like

```
<<<<<<
    restart)
        $0 stop
        $0 start
        ;;
<<<<<<
```

3.8 We allow ourselves to use a Bash-specific notational convenience for readability: The `((...))` “command” (with no dollar sign!) evaluates the expression between the parentheses and returns a return value of 0 if its value is different from 0, 1 otherwise.

```
#!/bin/bash
# prim -- Determines prime numbers up to a limit
#         Horribly inefficient method.

echo 2
i=3
while ((i < $1))
do
    prime=1
    j=2
    while ((prime && j < i/2))
    do
        if ((i % j == 0))
        then
            prime=0
        fi
        j=$((j+1))
    done
    ((prime)) && echo $i
    i=$((i+2))
done
```

(The mathematicians in our audience will wince; this method is much more inefficient than approaches like the “sieve of Eratosthenes”, and anyway we would only have to consider numbers up to \sqrt{i} rather than $i/2$. Unfortunately, bash cannot calculate square roots, so that the looser upper bound must do ...)

3.9 At first, the if could depend directly on the fgrep program rather than the \$? variable:

```
if fgrep -q $pattern $f
then
    cp $f $HOME/backups
fi
```

Instead of inverting the command's return value using ! and then possibly calling continue, we pull the cp invocation into the then branch and thus manage to get rid of the continue altogether. Another observation is that, in our new version, copying is skipped if something else unexpected happens during the fgrep (i. e., if the specified file does not exist). The original script would have tried to copy the file even so.

In this simple case, you might also use conditional evaluation and write something like

```
fgrep -q $pattern $f && cp $f $HOME/backups
```

This is the shortest possible form.

3.12 One possibility:

```
#!/bin/bash
# tclock -- Display a clock in a text terminal
trap "clear; exit" INT
while true
do
    clear
    banner $(date +%X)
    sleep 1
done
```

3.13 For example:

```
function toupper () {
    echo $* | tr '[:lower:]' '[:upper:]'
}
```

3.14 The exec command irrevocably ends the execution of test1. Hence, the output is

```
Hello
Howdy
```

4.1 By analogy to grep, the exit after the first error message probably ought to have a 2 as its argument. The return value 1 then signals a non-existing group.

4.2 A possible approach:

```
#!/bin/bash
# hierarchy -- Follow a file name hierarchy to the root

name="$1"
until [ "$name" = "/" ]
do
```

```

    echo $name
    name="$(dirname $name)"
done

```

4.3 The hierarchy script gets us the correct names but in reverse order. We must also check whether the directory in question already exists:

```

#!/bin/bash
# mkdirp -- Poor person's "mkdir -p"

for dir in $(hierarchy "$1" | tac)
do
    [ -d "$dir" ] || mkdir "$dir"
done

```

4.4 Replace the line

```
conffile=/etc/multichecklog.conf
```

by something like

```
conffile=${MULTICHECKLOG_CONF:-/etc/multichecklog.conf}
```

4.5 One possibility (within the checklonger function):

```

function checklonger () {
    case "$1" in
        *k) max=$((($1*k*1000)) ;;
        *M) max=$((($1*M*1000000)) ;;
        *) max=$1 ;;
    esac
    test ...
}

```

4.6 The trick consists of having seq count backwards:

```

function rotate () {
    rm -f "$1.9"
    for i in $(seq 9 -1 1)
    do
        mv -f "$1.$((i-1))" "$1.$i"
    done
    mv "$1" "$1.0"
    > "$1"
}

```

Instead of the hard-coded 9, you might want to insert a variable.

4.7 The most convenient method uses the --reference option of the chmod and chown commands. For example:

```

<<<<<<
mv "$1" "$1.0"
> "$1"
chmod --reference="$1.0" "$1"
chown --reference="$1.0" "$1"
<<<<<<

```

4.8 `grep` supports the `-e` option, which introduces a regular expression to be searched. This option may occur several times on the same command, and `grep` will search for all expressions thus specified, simultaneously.

4.9 `"$@"` arranges for the arguments of `gdf` to be passed to `df`. This makes invocations such as `"gdf / /home"` work. You had better stay away from options that radically change `df`'s output format.

5.1 The technique for this is quite like that for validating the login name.

5.2 You could use something like

```
read -p "Login shell: " shell
if ! [ grep "$shell$" /etc/shells ]
then
    echo >&2 "$shell is not a valid login shell"
    exit 1
fi
```

5.3 You might replace the `read` command by something like

```
prompt=${1:-"Please confirm"}
read -p "$prompt (y/n): " answer
```

5.4 One possible example:

```
#!/bin/sh
# numbergame -- simple guessing game

max=100
number=$(( RANDOM % max ))

echo Guess a number between 0 and $max.
while true; do
    read -p "Number? " guess
    d=$(( guess - number ))
    if [ $d = 0 ]; then
        echo "Congratulations, that was correct"
        break
    elif [ $d -gt 0 ]; then
        echo "Too big"
    else
        echo "Too small"
    fi
done
```

5.5 The `select` loop variable is assigned an empty string. For this reason, the `newuser` script says `"[-n "$type"] && break"` so the loop is finished only if the user entered something valid.

5.6 The obvious solution (output `"score"`) is not correct, since that variable contains the *current* score. The future score can be found using the next option to question. Thus, in present:

```
# Display and show the question
echo "For $(question $1 next) points:"
question $1 display
```

5.7 Some obvious extensions might be (roughly ordered by effort required):

- More than one question per score level (with random choice?)
- “Folding”: Whoever can’t answer the question may leave the game with their current score
- “Safety” levels (whoever has between 500 and 16000 points when giving a wrong answer goes back to 500, whoever has 16000 points or more gets to keep 16000)
- “50/50 lifeline”: The participant gets another menu selection that will allow him to remove two wrong answers (once)

What else can you think of?

5.8 Actually, one call to `grep` per question should be enough if you store the different lines in shell variables and output them if required. Since there is at most one question in memory at any one time, there are no problems with difficult data structures.

6.1 The corresponding line numbers are: (a) 4 (who would have thought?); (b) 2–4 (the ABC on line 2 does not count); (c) 2–3 and 4–5 (address ranges with a regular expression as the first address may match multiple times); (d) 6; (e) 3 (line 2 is already “through”); (f) 3 and 5–6 (the lines not containing ABC).

6.2 Regular expressions as the second address of a range match the first line after the range start, at the earliest. With the “1,/*expression*/" range, *expression* could never match the first input line. “0,/*expression*/" allows exactly that.

6.3 Possibly the most straightforward method is

```
sed '/$/d'
```

6.4 For example:

```
sed -ne '/<Directory>/,/<\Directory>/p' httpd.conf
```

6.5 You may be surprised to hear that it actually works, but it is by no means as trivial as `head`. Read GNU `sed`’s info documentation for the details.

6.6 Try something like

```
sed '/[^A-Z]\+$\a\
\'
```

6.7 The *i-j* addresses are useful here:

```
sed '1~2y/abcdefghijklmnopqrstuvwxy/ABCDEFGHIJKLMNOPQRSTUVWXYZ/'
```

6.8 For example: “`sed 's/\<yellow\>/blue/g'`”. Remember the `g` modifier, in order to replace all yellows on each line, and the word brackets, to prevent accidental hyperactivity.

6.9 Try “`sed -ne '/^[A]/p; /^A/s/[A-Za-z]\+//p'`”.

6.11 Fortunately, there is nothing to be done for the “long” options, since these are adequately covered by the existing code (how?). As far as “-” is concerned, you will need to add a branch for this in the case, which needs to terminate the loop (similar to the “*” case). Why does the “*” branch not suffice?

6.12 One possibility that does not change much else might be

```
function question () {
  if [ "$1" = "get" ]
  then
    echo $2
    return
  fi
  case "$2" in
    display) re='?' ;;
    correct) re='+' ;;
    answers) re='[-+]';;
    next)    re='>' ;;
    *)      echo >&2 "$0: get: invalid field type $2"; exit 1 ;;
  esac
  sed -ne "/question $1/,/end/p" $qfile | sed -ne "/$fe/s///p"
}
```

7.1 This basically combines techniques from the “shell history” and “duplicate words” examples. In the easiest case, something like

```
#!/usr/bin/awk -f
# countwords -- count words in a document

{
  for (i = 1; i <= NF; i++) {
    count[$i]++
  }
}
END {
  for (w in count) {
    print count[w], w
  }
}
```

may suffice. This simple approach, however, ignores capitalisation and separates words by whitespace, so that punctuation is considered part of a word. To avoid this, you can preprocess the input using a `tr` pipeline similar to that in the “duplicate words” example, or use the GNU `awk` functions “`tolower`” and “`gsub`” (see the GNU `awk` manual for details).

7.2 The second field of each line is the team’s point score, the third is the goal difference. Since the point score is more important than the goal difference, a `sort` invocation like

```
sort -t: -k2,2nr -k3,3nr
```

recommends itself (the entries should be sorted as numbers, and the largest value should come first). See `sort(1)` for details.—You could also handle this within `awk` (you have already seen a sorting function, and GNU `awk`, at least, contains an efficient built-in sorting function called `asort`), but `sort` is often more convenient, especially if complex criteria are involved.

7.3 Here is a suggested solution (which you should have been able to come up with yourself):

```
#!/usr/bin/awk -f

BEGIN { FS = ":"; OFS = ":" }

{
    spectators[$2] += $6
    spectators[$3] += $6
}

END {
    for (team in spectators) {
        print team, spectators[team]
    }
}
```

7.4 A three-stage approach is useful here. We use an `awk` program similar to the one already discussed to construct the unsorted table, sort it using `sort`, and then use another `awk` program to format it nicely. The first `awk` program, `bltab2`, differs from the former mostly because it determine the number of games that were won or lost:

```
$1 <= tag {
    games[$2]++; games[$3]++
    goals[$2] += $4 - $5; goals[$3] += $5 - $4
    if ($4 > $5) {
        won[$2]++; lost[$3]++
    } else if ($4 < $5) {
        lost[$2]++; won[$3]++
    }
}
```

The point scores are easily calculated on output:

```
END {
    for (team in games) {
        d = games[team] - won[team] - lost[team]
        points = 3*won[team] + d
        print team, games[team], won[team], d,
            lost[team], points, goals[team]
    }
}
```

The output `awk` script—let's call it `blfmt`—might look like this:

```
#!/usr/bin/awk -f
# blfmt -- Bundesliga-Tabelle formatiert ausgeben

BEGIN {
    FS = ":"
    print "RD TEAM"
    print "RD TEAM"
    print "-----"
}

{
    printf "%2d %-25.25s %2d %2d %2d %2d %3d %3d\n",
```

```

    ++i, $1, $2, $3, $4, $5, $6, $7
}

```

The lot is invoked using a pipeline like

```
bltab2 round=34 bl03.txt | sort -t: -k6,6rn -k7,7rn | blfmt
```

which you can of course put into a shell script to solve the exercise perfectly.

7.5 Ensure that the 1. FC Kaiserslautern’s score is reduced appropriately, for example between the calculation and sorting:

```
bltab2 round=34 bl03.txt | awk -f '{
    if ($1 == "1.FC Kaiserslautern") {
        $6 -= 2
    }
    print
}' | sort -t: -k6,6rn -k7,7rn | blfmt

```

7.6 Like many of the other programs shown here, `gdu` consists of a “data collection phase” and an “output phase”. Here is the data collection phase: We read the input and remember the space used per user as well as the greatest amount of space used so far—the latter is used to scale the output.

```
{
    sub(/:*/, "", $2)
    space[$2] = $1
    if ($1 > max) {
        max = $1
    }
}

```

For testing, an output phase producing numerical results is useful:

```
END {
    for (u in space) {
        printf "%-10.10s %f\n", u, 60*space[u]/max
    }
}

```

Once you have convinced yourself that the numerical results are sensible, you can try your hand at the graphical display:

```
END {
    stars = "*****"
    stars = stars stars
    for (u in space) {
        n = int(60*space[u]/max + 0.5)
        printf "%-10.10s %-60s\n", u, substr(stars, 0, n)
    }
}

```

8.1 You could add crew members to the “Person” table and add a “Position” column or something like that. That column would contain entries like “Commanding Officer” or “Executive Officer”. Of course, if you wanted to work cleanly, this column in the “Person” table would be a foreign key referring to a “Position” table, which should make it straightforward later to retrieve all executive officers.

8.2 Directors can be added in the same way as crew members in the previous exercise, but you should add another table to avoid mixing up film parts and existing people. Here, too, you would probably do well not to confine yourself to directors, but to add outright a table “Function” (or something) to cater not just for directors but also script writers, gaffers, and all the other people one encounters in a film crew. If you’re really devious, think of the film crew function “actor”, and add a foreign key that refers to the “Person” table and whose value is NULL for non-actors.

8.8 Try something like

```
sqlite> SELECT title FROM film
...> WHERE year < 1985 AND budget < 40
```

(You can connect several expressions in a WHERE clause using AND, OR, and NOT.)

8.9 If there is no WHERE and no explicit JOIN, a SELECT spanning multiple tables yields the Cartesian product of all tuples of all the tables in question, e. g.:

```
1|James T. |Kirk|1|1|USS Enterprise
1|James T. |Kirk|1|2|USS Enterprise
1|James T. |Kirk|1|3|Millennium Falcon
<<<<<<
2|Willard|Decker|1|1|USS Enterprise
2|Willard|Decker|1|2|USS Enterprise
2|Willard|Decker|1|3|Millennium Falcon
<<<<<<
```

9.1 (a) On 1 March, 5 P. M.; (b) On 2 March, 2 P. M.; (c) On 2 March, 4 P. M.; (d) On 2 March, 1 A. M.

9.2 Use, e. g., “at now + 3 minutes”.

9.4 One possibility might be “atq | sort -bk 2”.

9.6 Your task list itself is owned by you, but you do not have permission to write to the crontabs directory. Debian GNU/Linux, for example, uses the following permission bits:

```
$ ls -ld /var/spool/cron/crontabs
drwx-wx--T 2 root crontab 4096 Aug 31 01:03 /var/spool/cron/crontabs
```

As usual, root has full access to the file (in fact regardless of the permission bits) and members of the crontab group can write to files in the directory. Note that members of that group have to know the file names in advance, because the directory is not searchable by them (ls will not work). The crontab utility is a set-GID program owned by the crontab group:

```
$ ls -l $(which crontab)
-rwxr-sr-x 1 root crontab 27724 Sep 28 11:33 /usr/bin/crontab
```

So it is executed with the access permissions of the crontab group, no matter which users invokes the program. (The set-GID mechanism is explained in detail in the document *Linux System Administration I*.)

9.7 Register the job for the 13th of every month and check within the script (e. g., by inspecting the result of “date +%u”) if the current day is a Friday.

9.8 The details depend on the distribution.

9.9 Use something like

```
* * * * logger -p local0.info "cron test"
```

To write the date to the file every other minute, you could use the following line:

```
0,2,4, <<<<<<,56,58 * * * * /bin/date >>/tmp/date.log
```

But this one is more convenient:

```
*/2 * * * * /bin/date >>/tmp/date.log
```

9.10 The commands to accomplish this are »crontab -l« and »crontab -r«.

9.11 You should add hugo to the /etc/cron.deny file (on SUSE distributions, /var/spool/cron/deny) or delete him from /etc/cron.allow.

9.13 /etc/cron.daily contains a script called 0anacron which is executed as the first job. This script invokes “anacron -u”; this option causes anacron to update the time stamps without actually executing jobs (which is the next thing that cron will do). When the system is restarted, this will prevent anacron from running jobs unnecessarily, at least if the re-boot occurs after cron has done its thing.

10.4 There are systems where /usr is on a separate partition or (with “thin clients”) on a different machine. However, system time is so important that it should be available correctly very early when the system is booted, even if there are horrible problems. A copy of the file in question (and as a rule time zone definitions aren’t very big) is therefore the safest choice.

10.5 A simple solution involves zdump in combination with watch. Just try something like this:

```
$ ZONES=Asia/Tokyo Europe/Berlin America/New_York
$ watch -t zdump $ZONES
```

Interrupt the program with **Ctrl**+**C** when you do not need any further output. In a graphical environment, of course, you could do something like this:

```
$ for z in $ZONES; do
> TZ=$z xclock -title $z#/ -update 1 &
> done
```

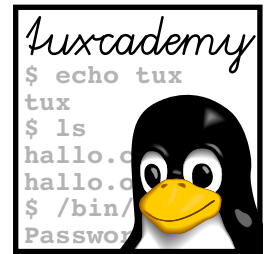
11.3 The display name addresses the X server no. 1 (presumably the second one) on the computer called bla.example.com, and in particular the second screen controlled by that server. One possible command line would be

```
# xterm -display bla.example.com:1.1
```

11.5 The VNC method is a lot easier to implement, since the X protocol is quite featureful and extensive. A “remote framebuffer protocol” as used by VNC has, by comparison, very few operations and is a lot more straightforward, which is proven by the existence of VNC clients written in a few hundreds of lines of a programming language like Tcl. This results in a *de facto* much larger number of platforms supporting VNC, including PDAs and other devices with very scarce resources.—On the other hand, the X protocol enables various optimisations that can only be exploited if one knows *what* is currently being rendered. The efficiency question is difficult to answer in the general case, since it depends a lot on what is currently being done: Image processing, for example, is fairly costly in X, since it mostly takes place within the client, and large amounts of data must be transferred to the server (special communication mechanisms make this bearable in the case where server and client run on the same host). For VNC, there is no *a priori* difference to normal system usage, since pixel data is transferred in any case. X11 has the advantage where large changes of the display can be described by a few X protocol commands.

11.6 Try something like

```
$ xclock -geometry 150x150+50-50
```



B

Regular Expressions

B.1 Overview

Regular expressions are an essential prerequisite for shell programming and the use of programs like sed and awk. By way of illustration, here is an adapted introduction to the topic from *Introduction to Linux for Users and Administrators*:

Regular expressions are often constructed “recursively” from primitives that are themselves considered regular expressions. The simplest regular expressions are letters, digits and many other characters from the usual character set, which stand for themselves. “a”, for example, is a regular expression matching the “a” character; the regular expression “abc” matches the string “abc”. Character classes can be defined in a manner similar to shell wildcard patterns; therefore, the regular expression “[a-e]” matches exactly one character out of “a” to “e”, and “a[xy]b” matches either “axb” or “ayb”. As in the shell, ranges can be concatenated—“[A-Za-z]” matches all uppercase and lowercase letters—but the complement of a range is constructed slightly differently: “[^abc]” matches all characters *except* “a”, “b”, and “c”. (In the shell, that was “[!abc]”.) The dot, “.”, corresponds to the question mark in shell wildcard 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

characters

Character classes

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

Line start

Line end



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 expression may be repeated arbitrarily many times (including not at all). The as-

Repetition

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

B.2 Extras

extensions The explanation from the previous section applies to nearly all Linux programs that deal with regular expressions. Various programs support different extensions providing 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”. 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.

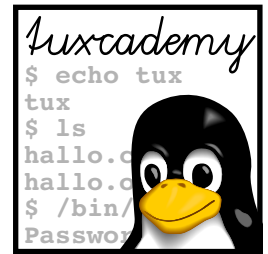
Not every program supports every extension. Table B.1 shows an overview of the most important programs. Perl and Tcl in particular support lots of extensions that have not been discussed here.

Table B.1: Regular expression support

Extension	GNU grep	GNU egrep	trad egrep	sed	awk	Perl	Tcl
Word brackets	•	•	•	•1	•1	•2	•2
Grouping	•1	•	•	•1	•	•	•
Alternative	•1	•	•	•	•	•	•
Option	•1	•	•	•1	•	•	•
At-least-once	•1	•	•	•1	•	•	•
Limits	•1	•	◦	•1	•1	•	•
Back-Reference	◦	•	•	•	◦	•	•
Non-Greedy	◦	◦	◦	◦	◦	•	•

•: supported; ◦: not supported

Notes: 1. Requires a preceding backslash (“\”), e.g. “ab\+” instead of “ab+”. 2. Completely different syntax (see documentation).



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-102

The following table displays the objectives for the LPI-102 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	ADM1	GRD2	ADM2
105.1	4	Customize and use the shell environment	–	1–2	–
105.2	4	Customize or write simple scripts	–	2–5	–
105.3	2	SQL data management	–	8	–
106.1	2	Install and configure X11	–	11	–
106.2	1	Setup a display manager	–	11	–
106.3	1	Accessibility	–	12	–
107.1	5	Manage user and group accounts and related system files	2	–	–
107.2	4	Automate system administration tasks by scheduling jobs	–	9	–
107.3	3	Localisation and internationalisation	–	10	–
108.1	3	Maintain system time	–	–	8
108.2	3	System logging	–	–	1–2
108.3	3	Mail Transfer Agent (MTA) basics	–	–	11
108.4	2	Manage printers and printing	–	–	9
109.1	4	Fundamentals of internet protocols	–	–	3–4
109.2	4	Basic network configuration	–	–	4–5, 7
109.3	4	Basic network troubleshooting	–	–	4–5, 7
109.4	2	Configure client side DNS	–	–	4
110.1	3	Perform security administration tasks	2	–	4–5, 13
110.2	3	Setup host security	2	–	4, 6–7, 13
110.3	3	Securing data with encryption	–	–	10, 12

C.3 LPI Objectives In This Manual

105.1 Customize and use the shell environment

Weight 4

Description Candidates should be able to customize shell environments to meet users' needs. Candidates should be able to modify global and user profiles.

Key Knowledge Areas

- Set environment variables (e.g. PATH) at login or when spawning a new shell
- Write Bash functions for frequently used sequences of commands
- Maintain skeleton directories for new user accounts
- Set command search path with the proper directory

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

- .
- source
- /etc/bash.bashrc
- /etc/profile
- env
- export
- set
- unset
- ~/.bash_profile
- ~/.bash_login
- ~/.profile
- ~/.bashrc
- ~/.bash_logout
- function
- alias
- lists

105.2 Customize or write simple scripts

Weight 4

Description Candidates should be able to customize existing scripts, or write simple new Bash scripts.

Key Knowledge Areas

- Use standard sh syntax (loops, tests)
- Use command substitution
- Test return values for success or failure or other information provided by a command
- Perform conditional mailing to the superuser
- Correctly select the script interpreter through the shebang (!) line
- Manage the location, ownership, execution and suid-rights of scripts

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

- for
- while
- test
- if
- read
- seq
- exec

105.3 SQL data management

Weight 2

Description Candidates should be able to query databases and manipulate data using basic SQL commands. This objective includes performing queries involving joining of 2 tables and/or subselects.

Key Knowledge Areas

- Use of basic SQL commands
- Perform basic data manipulation

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

- insert
- update
- select
- delete
- from
- where
- group by
- order by
- join

106.1 Install and configure X11

Weight 2

Description Candidates should be able to install and configure X11.

Key Knowledge Areas

- Verify that the video card and monitor are supported by an X server
- Awareness of the X font server
- Basic understanding and knowledge of the X Window configuration file

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

- /etc/X11/xorg.conf
- xhost
- DISPLAY
- xwininfo
- xdpinfo
- X

106.2 Setup a display manager

Weight 1

Description Candidates should be able to describe the basic features and configuration of the LightDM display manager. This objective covers awareness of the display managers XDM (X Display Manger), GDM (Gnome Display Manager) and KDM (KDE Display Manager).

Key Knowledge Areas

- Basic configuration of LightDM
- Turn the display manager on or off
- Change the display manager greeting
- Awareness of XDM, KDM and GDM

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

- lightdm
- /etc/lightdm/

106.3 Accessibility

Weight 1

Description Demonstrate knowledge and awareness of accessibility technologies.

Key Knowledge Areas

- Basic knowledge of keyboard accessibility settings (AccessX)
- Basic knowledge of visual settings and themes
- Basic knowledge of assistive technology (ATs)

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

- Sticky/Repeat Keys
- Slow/Bounce/Toggle Keys
- Mouse Keys
- High Contrast/Large Print Desktop Themes
- Screen Reader
- Braille Display
- Screen Magnifier
- On-Screen Keyboard
- Gestures (used at login, for example GDM)
- Orca
- GOK
- emacspeak

107.2 Automate system administration tasks by scheduling jobs

Weight 4

Description Candidates should be able to use cron or anacron to run jobs at regular intervals and to use at to run jobs at a specific time.

Key Knowledge Areas

- Manage cron and at jobs
- Configure user access to cron and at services
- Configure anacron

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

- /etc/cron.{d,daily,hourly,monthly,weekly}/
- /etc/at.deny
- /etc/at.allow
- /etc/crontab
- /etc/cron.allow
- /etc/cron.deny
- /var/spool/cron/
- crontab
- at
- atq
- atrm
- anacron
- /etc/anacrontab

107.3 Localisation and internationalisation

Weight 3

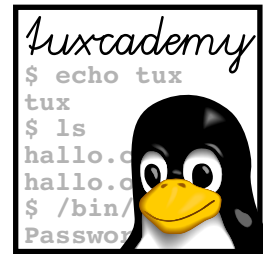
Description Candidates should be able to localize a system in a different language than English. As well, an understanding of why LANG=C is useful when scripting.

Key Knowledge Areas

- Configure locale settings and environment variables
- Configure timezone settings and environment variables

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

- /etc/timezone
- /etc/localtime
- /usr/share/zoneinfo/
- LC_*
- LC_ALL
- LANG
- TZ
- /usr/bin/locale
- tzselect
- timedatectl
- date
- iconv
- UTF-8
- ISO-8859
- ASCII
- Unicode



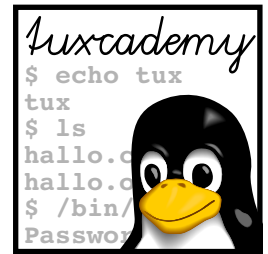
D

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.

X	Starts the appropriate X server for the system	X(1)	169
anacron	Executes periodic job even if the computer does not run all the time	anacron(8)	138
at	Registers commands for execution at a future point in time	at(1)	132
atd	Daemon to execute commands in the future using at	atd(8)	134
atq	Queries the queue of commands to be executed in the future	atq(1)	133
atrm	Cancels commands to be executed in the future	atrm(1)	134
awk	Programming language for text processing and system administration	awk(1)	98
bash	The “Bourne-Again-Shell”, an interactive command interpreter	bash(1)	14
batch	Executes commands as soon as the system load permits	batch(1)	133
case	Shell command for pattern-based multi-way branching	bash(1)	44
chmod	Sets access modes for files and directories	chmod(1)	24
chsh	Changes a user’s login shell	chsh(1)	14
cmp	Byte-by-byte comparison of two files	cmp(1)	94
crontab	Manages commands to be executed at regular intervals	crontab(1)	137
dialog	Allows GUI-like interaction controls on a character screen	dialog(1)	80
env	Outputs the process environment, or starts programs with an adjusted environment	env(1)	33
exec	Starts a new program in the current shell process	bash(1)	52
export	Defines and manages environment variables	bash(1)	33
file	Guesses the type of a file’s content, according to rules	file(1)	14
find	Searches files matching certain given criteria	find(1), Info: find	14
for	Shell command to loop over the elements of a list	bash(1)	46
iconv	Converts between character encodings	iconv(1)	145
kdiallog	Allows use of KDE widgets from shell scripts	kdiallog(1)	84
locale	Displays information pertaining to locales	locale(1)	149, 150
localedef	Compiles locale definition files	localedef(1)	149
logrotate	Manages, truncates and “rotates” log files	logrotate(8)	62
lspci	Displays information about devices on the PCI bus	lspci(8)	167
mkfifo	Creates FIFOs (named pipes)	mkfifo(1)	67
mktemp	Generates a unique temporary filename (securely)	mktemp(1)	94
printf	Formatted output of numbers and strings	printf(1), bash(1)	69

seq	Writes number sequences to standard output	seq(1)	67
set	Manages shell variables and options	bash(1)	32
strace	Logs a process's system calls	strace(1)	18
test	Evaluates logical expressions on the command line	test(1), bash(1)	41
timeconfig	[Red Hat] Allows the convenient configuration of the system-wide time zone	timeconfig(8)	153
tr	Substitutes or deletes characters on its standard input	tr(1)	69
tzselect	Allows convenient interactive selection of a time zone	tzselect(1)	153
unbuffer	Suppresses a process's output buffering (part of the expect package)	unbuffer(1)	67
uniq	Replaces sequences of identical lines in its input by single specimens	uniq(1)	104
unset	Deletes shell or environment variables	bash(1)	34
until	Shell "Kommando" for a loop that executes "until" a condition evaluates as true	bash(1)	48
while	Shell command for a loop that executes "while" a condition evaluates to true	bash(1)	47
xauth	X server access control via "magic cookies"	xauth(1)	178
xdpyinfo	Shows information about the current X display	xdpyinfo(1)	173
xhost	Allows clients on other hosts to access the X server via TCP	xhost(1)	177
xkbset	Controls keyboard setup options for X11	xkbset(1)	182
xmessage	Displays a message or query in an X11 window	xmessage(1)	84
xwininfo	Displays information about an X window	xwininfo(1)	174
zdump	Outputs the current time or time zone definitions for various time zones	zdump(1)	153
zic	Compiler for time zone data files	zic(8)	153



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”.

- != (awk operator), 101
- !~ (awk operator), 101
- # (shell variable), 57
- \$ (awk operator), 99, 101, 106
- \$ (shell variable), 59, 94
- && (awk operator), 101
- * (awk operator), 101
- * (shell variable), 36–37, 82, 187
- + (awk operator), 101
- (awk operator), 101
- . (shell command), 24
- / (awk operator), 101
- < (awk operator), 101
- <= (awk operator), 101
- = (awk operator), 101
- == (awk operator), 101
- > (awk operator), 101
- >= (awk operator), 101
- ? (shell variable), 34, 40, 188
- @ (shell variable), 36–37, 39, 82, 187
- ^ (awk operator), 101
- _ (environment variable), 133
- ~ (awk operator), 101
- Ø (shell variable), 57
- 1 (shell variable), 47
- 2 (shell variable), 47
- \$3 (awk operator), 101

- Aho, Alfred V., 98
- alias, **16**
- alias (shell command), 16–17
- anacron, 138–139, 197
 - s (option), 139
 - u (option), 197
- arrays, **82**
 - associative, **102**
- asort (awk Function), 193
- at, 11, 132–135, 137
 - c (option), 134
 - f (option), 133
 - q (option), 134
- atd, 134–135
 - b (option), 134
 - d (option), 134
 - l (option), 134
- atq, 133–134
 - q (option), 134
- atrm, 134
- aук, 98
- awk
 - !=, 101
 - !~, 101
 - \$, 99, 101, 106
 - &&, 101
 - *, 101
 - +, 101
 - , 101
 - /, 101
 - <, 101
 - <=, 101
 - =, 101
 - ==, 101
 - >, 101
 - >=, 101
 - ^, 101
 - ~, 101
 - \$3, 101
- asort, 193
- close, 110
- FILENAME, 106
- FNR, 106
- for, 103–104
- FS, 102
- getline, 110
- gsub, 193
- if, 105
- int, 103
- length, 103
- log, 103
- NF, 101–102
- OFS, 107
- print, 107

- printf, 106, 110
- return, 103
- RS, 102
- sqrt, 103
- sub, 103, 109
- substr, 103
- tolower, 193
- awk, 11, 26–27, 37, 88, 97–107, 109–111, 128, 150, 193–194, 199–200
 - F (option), 100, 102
 - f (option), 98
- banner, 51
- basename, 37, 61
- BASH (environment variable), 17–18
- bash, 14–20, 24–26, 32–34, 37–41, 50, 52, 60, 69, 74–76, 82, 92, 150, 186–188
 - l (option), 17
- BASH_ENV (environment variable), 18
- ~/.bash_history, 104
- .bash_login, 17–18, 21
- .bash_logout, 18
- .bash_profile, 17–18, 21, 150
- .bashrc, 18–19, 21
- batch, 133–135
- /bin/sh, 136
- bind, 20
 - f (option), 20
- Bourne, Stephen L., 39
- Boyce, Raymond F., 115
- break (shell command), 48–50, 76
- C, 47
- cal, 16
 - m (option), 16
- case (shell command), 41, 44, 65, 188
- cat, 15, 48, 91
- cd (shell command), 33
- Chamberlin, Donald D., 115
- chmod, 24, 26, 190
 - reference (option), 190
- chown, 190
 - reference (option), 190
- chsh, 14–15
 - s (option), 15
- Clancy, Tom, 153
- close (awk Function), 110
- cmd (shell command), 48
- cmp, 94
 - s (option), 94
- Codd, Edgar F., 115
- continue (shell command), 48–50, 76, 189
- cp, 189
- cron, 11, 132, 135–139, 197
- crontab, 135–137, 196–197
 - e (option), 137
 - l (option), 137, 197
 - r (option), 137, 197
 - u (option), 137
- csh, 14
- cut, 37, 56–57, 69, 74, 88, 98–99, 150
 - d (option), 150
- dash, 186
- date, 16, 147, 150, 196
- debconf, 153
- definitions, **12**
- /dev/tty, 81
- df, 68–70
- dialog, 80–84
 - clear (option), 81
 - menu (option), 81
 - msgbox (option), 84
 - no-cancel (option), 84
 - title (option), 81
- diff, 29
 - r (option), 29
- Dijkstra, Edsger, 39
- dirname, 61
- DISPLAY (environment variable), 133, 160–161, 169, 177–178
- display name, **160**
- done (shell command), 49
- du, 109, 111
 - s (option), 111
- echo (shell command), 25, 30, 32, 40–41, 57, 68, 186
- EDITOR (environment variable), 137
- egrep, 43, 200
- elif (shell command), 43–44
- else (shell command), 42, 44
- env, 33
- environment variable
 - _ , 133
 - BASH, 17–18
 - BASH_ENV, 18
 - DISPLAY, 133, 160–161, 169, 177–178
 - EDITOR, 137
 - HOME, 34, 39, 136
 - INPUTRC, 20
 - LANG, 146–147, 149–150
 - LANGUAGE, 147
 - LC_*, 149–150
 - LC_ADDRESS, 149
 - LC_ALL, 149
 - LC_COLLATE, 149
 - LC_CTYPE, 149
 - LC_MEASUREMENT, 149
 - LC_MESSAGES, 149
 - LC_MONETARY, 149
 - LC_NAME, 149
 - LC_NUMERIC, 149–150
 - LC_PAPER, 149
 - LC_TELEPHONE, 149
 - LC_TIME, 149

- LOGNAME, 43, 136
- MAILTO, 136
- PAGER, 32
- PATH, 16, 19, 24, 34, 39, 186
- ROOT, 29
- SHELL, 136
- TERM, 133
- TMPDIR, 94
- TZ, 153–154
- VISUAL, 137
- environment variables, 32
- /etc/anacrontab, 138
- /etc/at.allow, 134
- /etc/at.deny, 134
- /etc/at.deny, 134
- /etc/bash.bashrc, 18–19
- /etc/bash.bashrc.local, 19
- /etc/bash.local, 20
- /etc/cron.allow, 137, 197
- /etc/cron.d, 136
- /etc/cron.daily, 136–137
- /etc/cron.deny, 137, 197
- /etc/cron.hourly, 136–137
- /etc/crontab, 136–137
- /etc/default, 65
- /etc/group, 56–59, 110
- /etc/gshadow, 58
- /etc/init.d, 14
- /etc/inputrc, 20
- /etc/lightdm/lightdm.conf, 170
- /etc/lightdm/lightdm.conf.d, 170
- /etc/localtime, 153–154
- /etc/motd, 86
- /etc/passwd, 48, 56, 84, 103, 110, 136
- /etc/profile, 17, 19, 25
- /etc/profile.local, 19
- /etc/shells, 15, 19, 76, 185
- /etc/skel, 19–20
- /etc/sysconfig, 65
- /etc/sysconfig/clock, 153
- /etc/timezone, 152–153
- /etc/X11/gdm, 173
- /etc/X11/rgb.txt, 163
- /etc/X11/xdm, 172
- /etc/X11/xinit/xinitrc, 169
- /etc/X11/xorg.conf, 163
- exec, 52, 189
- exit, 39
- exit (shell command), 50, 57, 186, 189
- expect, 67
- export
 - n (option), 33
- export (shell command), 33
- fgrep, 188–189
- fi (shell command), 42
- file, 14, 185
- FILENAME (awk variable), 106
- find, 14, 18, 185
- FNR (awk variable), 106
- fonts.alias, 176
- fonts.dir, 176
- fonts.scale, 176
- for (awk command), 103–104
- for (shell command), 38, 46–47, 49, 76
- Fox, Brian, 16
- FS (awk variable), 102
- FUNCNAME (shell variable), 52
- gawk, 98
- gdm, 170, 173
- gdm.conf, 173
- gdmconfig, 173
- getline (awk Function), 110
- grep, 14–15, 40, 56–59, 67–68, 80, 88, 90,
 - 98, 185, 187, 189–192, 200
 - e (option), 191
 - f (option), 68
 - line-buffered (option), 67
- groups, 57
- gsub (awk Function), 193
- Hakim, Pascal, 138
- #hash (shell variable), 69
- head, 90, 192
- HISTSIZE (shell variable), 19
- HOME (environment variable), 34, 39, 136
- \$HOME/.bash_profile, 25
- httpd.conf, 90
- iconv, 145
 - c (option), 145
 - l (option), 145
 - o (option), 145
 - output (option), 145
- if (awk command), 105
- if (shell command), 41–42, 44, 50, 101, 188
- IFS (shell variable), 38–39, 74, 79
- Inkscape, 172
- INPUTRC (environment variable), 20
- .inputrc, 20
- int (awk Function), 103
- Iteration, 46
- join, 88
- kcontrol, 173
- kdiallog, 84
- kdm, 170, 173
- Kernighan, Brian W., 27, 98
- kill, 40
 - l (option), 40
- killall, 63
- ksh, 14
- LANG (environment variable), 146–147, 149–150
- LANGUAGE (environment variable), 147
- LC_* (environment variable), 149–150

- LC_ADDRESS (environment variable), 149
- LC_ALL (environment variable), 149
- LC_COLLATE (environment variable), 149
- LC_CTYPE (environment variable), 149
- LC_MEASUREMENT (environment variable), 149
- LC_MESSAGES (environment variable), 149
- LC_MONETARY (environment variable), 149
- LC_NAME (environment variable), 149
- LC_NUMERIC (environment variable), 149–150
- LC_PAPER (environment variable), 149
- LC_TELEPHONE (environment variable), 149
- LC_TIME (environment variable), 149
- length (awk Function), 103
- Libes, Don, 67
- ll, 18
- locale, 149–150, 155
 - a (option), 150
- localedef, 149
- log (awk Function), 103
- logger, 133
- .login, 18
- login, 17, 43
- LOGNAME (environment variable), 43, 136
- logrotate, 62
- ls, 15, 40, 63, 150
- lspci, 167

- mail, 68
- MAILTO (environment variable), 136
- man, 32
- mawk, 98
- mkdir, 42, 62
 - p (option), 62
- mkfifo, 67
- mkfontdir, 176
- mktemp, 94
 - p (option), 94
 - t (option), 94
- mv, 60–61

- n (shell variable), 35
- name (shell variable), 35
- newuser (shell command), 75, 191
- NF (awk variable), 101–102
- nice, 134

- objectives, **203**
- OFS (awk variable), 107
- Open Group, 158
- oversed, 95

- PAGER (environment variable), 32
- PAGER (shell variable), 32
- paste, 88, 98
- PATH (environment variable), 16, 19, 24, 34, 39, 186

- Perl, 200
- positional parameters, 61
- present (shell command), 82
- print (awk command), 107
- printf (awk command), 106, 110
- printf, 150
- printf (shell command), 69
- .profile, 17–19, 21, 186
- PS1 (shell variable), 16, 19, 34
- PS3 (shell variable), 77
- Python, 200

- Ramey, Chet, 16
- RANDOM (shell variable), 76
- read (shell command), 48, 67–69, 71, 74–75, 191
- return (awk command), 103
- return (shell command), 79
- return value, **40**
- rm, 16
- ROOT (environment variable), 29
- \$ROOT/etc/, 29
- RS (awk variable), 102

- sed, 11, 15, 37, 61, 87–96, 98, 103, 192, 199–200
 - e (option), 88, 95
 - expression= (option), 95
 - f (option), 88
 - i (option), 95
 - n (option), 90
 - s (option), 88–89
- select (shell command), 76–77, 79, 84, 191
- seq, 67, 190
- set
 - C (option), 17
 - n (option), 29
 - o noclobber (option), 17
 - o xtrace (option), 17
 - v (option), 30
 - x (option), 17, 29
- set (shell command), 17, 29–30, 32
- sh, 14
- Shaw, George Bernard, 146
- shell, **14**
 - #, 57
 - \$, 59, 94
 - *, 36–37, 82, 187
 - ., 24
 - ?, 34, 40, 188
 - @, 36–37, 39, 82, 187
 - 0, 57
 - 1, 47
 - 2, 47
 - alias, 16–17
 - break, 48–50, 76
 - case, 41, 44, 65, 188
 - cd, 33

- cmd, 48
- continue, 48–50, 76, 189
- done, 49
- echo, 25, 30, 32, 40–41, 57, 68, 186
- elif, 43–44
- else, 42, 44
- exit, 50, 57, 186, 189
- export, 33
- fi, 42
- for, 38, 46–47, 49, 76
- FUNCNAME, 52
- #hash, 69
- HISTSIZE, 19
- if, 41–42, 44, 50, 101, 188
- IFS, 38–39, 74, 79
- n*, 35
- name, 35
- newuser, 75, 191
- PAGER, 32
- present, 82
- printf, 69
- PS1, 16, 19, 34
- PS3, 77
- RANDOM, 76
- read, 48, 67–69, 71, 74–75, 191
- return, 79
- select, 76–77, 79, 84, 191
- set, 17, 29–30, 32
- shift, 35, 61, 95
- source, 24–25, 42, 52, 186
- suffix, 61
- test, 41–43, 48, 75, 79, 186
- then, 42, 189
- trap, 50–51, 81
- typeset, 52
- unalias, 17
- unset, 34
- until, 47–50
- while, 39, 47–50, 68, 101
- SHELL (environment variable), 136
- shell scripts, **14**
- shift (shell command), 35, 61, 95
- signals, 50
- sort, 59, 88, 98–99, 110, 193–194
- source (shell command), 24–25, 42, 52, 186
- sqlite3, 118
- sqr (awk Function), 103
- ssh, 17
- ssh-agent, 33
- startx, 169, 173, 178
- strace, 18
- su, 133, 137
- sub (awk Function), 103, 109
- substr (awk Function), 103
- suffix (shell variable), 61
- syslogd, 63, 67, 134, 136
- system load, 133
- tail, 90
- Tcl, 197, 200
- tcpdump, 44
- tcsh, 14, 16, 185
- TERM (environment variable), 133
- test
 - eq (option), 41
 - ge (option), 41
 - gt (option), 41
 - le (option), 41
 - lt (option), 41
 - ne (option), 41
 - s (option), 94
 - z (option), 75
- test (shell command), 41–43, 48, 75, 79, 186
- testing loop, **46**
- then (shell command), 42, 189
- Thomas, John C., 164
- timeconfig, 153
- /tmp, 94
- /tmp/oversed.z19516, 94
- TMPDIR (environment variable), 94
- tolower (awk Function), 193
- tr, 69, 88, 91, 193
- trap (shell command), 50–51, 81
- typeset
 - F (option), 52
 - f (option), 52
- typeset (shell command), 52
- TZ (environment variable), 153–154
- tzconfig, 153
- tzselect, 153
- Tzur, Itai, 138
- unalias (shell command), 17
- unbuffer, 67
- uniq, 104
- unset, 150
- unset (shell command), 34
- until (shell command), 47–50
- useradd, 19
 - m (option), 19
- /usr/lib/xorg/modules, 163
- /usr/share/i18n/locales, 149
- /usr/share/lightdm/lightdm.conf.d, 170
- /usr/share/zoneinfo, 152–154
- /usr/share/zoneinfo/Europe/Berlin, 153
- /var/log/messages, 67
- /var/spool/atjobs, 134
- /var/spool/atspool, 134
- /var/spool/cron/allow, 137
- /var/spool/cron/crontabs, 135–136
- /var/spool/cron/deny, 137, 197
- variable, **100**
- variables
 - references to, **32**
 - substitution, **32**

vi, 137
VISUAL (environment variable), 137
VNC, 171

watch, 197
wc, 14, 185
Weinberger, Peter J., 98
while (shell command), 39, 47–50, 68,
101
words, **38**

X, 163, 169
-layout (option), 168
X clients, **158**
X protocol, **158**
X server, **158**
X.org, **158**
xauth, 178
~/.Xauthority, 178
xclock, 169
xdm, 170, 172–173, 178
xdpyinfo, 173–174
xedit, 177
xfontsel, 177
xfs, 176–177
xhost, 177–178
xinit, 169, 173
~/.xinitrc, 169, 173
xkbset, 182
xlsfonts, 177
xman, 177
xmessage, 84, 86
Xorg, 163
-nolisten tcp (option), 178
xorg.conf, 163, 177
Xresources, 172
Xservers, 172
Xsession, 173
~/.xsession, 19, 173
xset, 176–177
q (option), 176
Xsetup, 172
xterm, 18, 144, 162, 169, 177
Xvnc, 171
xwininfo, 174–175

zdump, 153, 197
zic, 153