

Introduction to *Linux*

Karl Stroetmann

Version of December 10, 2003

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 1.1 | Why bother learning <i>Linux</i> ? | 3 |
| 1.2 | Overview | 4 |
| 2 | The <i>XEmacs</i> Editor | 5 |
| 2.1 | Why Use <i>XEmacs</i> ? | 5 |
| 2.2 | Getting Started | 6 |
| 2.3 | The Fundamentals | 12 |
| 2.4 | Cursor Movement | 13 |
| 2.5 | Basic Editing | 14 |
| 2.6 | Searching and Replacing | 16 |
| 2.7 | Editing Multiple Files | 20 |
| 2.8 | Completion | 21 |
| 2.9 | Abbreviations | 26 |
| 2.10 | Customizing <i>XEmacs</i> | 27 |
| 2.10.1 | Customizing Key Bindings and Variables | 27 |
| 2.10.2 | Invoking Commands on Startup | 28 |
| 2.10.3 | Writing Keyboard Macros | 28 |
| 2.10.4 | Traps and Pitfalls When Writing Keyboard Macros | 30 |
| 2.10.5 | Editing Keyboard Macros | 33 |
| 2.11 | Getting Help | 34 |
| 2.12 | The <i>Info</i> System in <i>XEmacs</i> | 36 |
| 2.13 | Miscellaneous Commands | 42 |
| 2.14 | What Else is There | 44 |
| 3 | The File System | 45 |
| 3.1 | Copying Files | 49 |
| 3.2 | Deleting Files and Directories | 49 |
| 3.3 | Listing Files and Directories | 50 |
| 3.4 | Maintaining Access and Ownership | 51 |
| 3.4.1 | Changing Permissions | 52 |
| 3.4.2 | Changing Ownership | 54 |
| 3.4.3 | Changing the Group | 54 |
| 3.5 | Wildcards | 54 |
| 3.6 | Links | 56 |
| 3.7 | Adding Devices to the File System | 57 |
| 3.8 | <i>direcd</i> Mode in <i>XEmacs</i> | 60 |
| 4 | Compiling and Debugging C Programs with Linux | 62 |
| 4.1 | A Small Example | 62 |
| 4.2 | Additional Commands in <i>gdb</i> | 67 |

| | |
|---|-----------|
| 5 Getting Help | 70 |
| 6 Input and Output Redirection and Pipes | 71 |
| 6.1 Input and Output Redirection | 71 |
| 6.2 Pipes | 74 |
| 6.2.1 The “cat” command | 74 |
| 6.2.2 The “cut” command | 74 |
| 6.2.3 The “paste” Command | 75 |
| 6.2.4 The “tr” command | 76 |
| 6.2.5 The “sed” Command | 77 |
| 6.2.6 The “tee” Command | 78 |
| 6.2.7 The “fgrep” Command | 78 |
| 6.2.8 The “head” Command | 78 |
| 6.2.9 The “tail” Command | 78 |
| 6.2.10 Putting it All Together | 79 |
| 7 The Shell | 80 |
| 7.1 Shell Variables | 80 |
| 7.2 Job Control | 82 |
| 7.3 Quoting and Command Expansion | 84 |
| 7.4 Shell Scripts | 85 |
| 8 Regular Expressions | 89 |
| 9 The find command | 92 |
| 10 Conclusion | 94 |
| References | 94 |
| Index | 96 |

Warning: This script is only a draft and contains a number of errors. If you find errors, please mail them to `stroetmann@ba-stuttgart.de`.

1 Introduction

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

This script is meant as a first introduction to the operating system *Linux*. It attempts to teach the basic skills needed to work productively in the *Linux* environment. The approach is text based, i.e. we do not discuss the various graphical user interfaces like KDE and GNOME that try to resemble more or less the *Microsoft Windows*TM environment. The fact is that, for a *power user*, the text based approach yields a much greater productivity than can be gained by using graphical user interfaces.

1.1 Why bother learning *Linux*?

When selecting an operating system, there are a number of points that need to be considered.

1. How long does it take to learn the operating system?

You do not learn *Linux* in a day. *Linux* is a very powerful tool and if you know what you are doing, it is like magic. You simply can not expect to learn magic in a day.

Let us compare *Linux* to the *Microsoft Windows*TM operating systems. The latter has been designed for the typical American user. If you want to get a rough idea about the intellectual level of these users, watch the Teletubbies. Some of them are even worse, they are in the same category as their president. Users who are more advanced and more demanding are well advised to invest into learning an operating system that is, on one side, much more powerful but, as a consequence, also more challenging.

2. Will my favorite software run on the operating system?

As *Linux* is constantly growing, more and more software gets ported to *Linux*. Even better, most of this software is free.

Furthermore, even some of the software produced by Microsoft is now running under *Linux*. A prominent example is *Microsoft Office*TM. This can now be installed on a *Linux* system with the help of *Crossover Office*TM.

3. Is support readily available?

For *Linux*, there are hundreds of newsgroups offering support for various aspects. Also, *Linux* is *open software* meaning the source code for *Linux* is freely available. This has a very important consequence: If you discover a bug that really hurts, then you are able to fix it yourself. If you do not have the necessary skills to do this, you can still hire someone to fix the problem.

The above point should not be underestimated. Imagine running some variant of *Microsoft Windows*TM in a business critical application. Further, imagine you stumble over a serious bug that cripples the application you are developing. All you can do then is report this bug to Microsoft. After that you have to wait. If you are really lucky, the bug is fixed in the next version which may be available in a year or something.

4. Is it stable?

The *Linux* operating system is definitely much more stable than the *Microsoft WindowsTM* operating systems. Especially *Windows 95TM* and *Windows 98TM* are horrible when it comes to stability. If you are developing your own software you simply can not afford a system that crashes every other hour. Although *Windows XPTM* is better than its predecessors, it still isn't in the same league as *Linux*. Furthermore, the past has shown that the Microsoft operating systems is quite vulnerable to viruses, worms, Trojan horses, and the like.

To summarize, *Linux* is a very powerful operating system that is stable, open software, and a platform for lots of high quality free software. It is a variant of Unix which is the operating system in use for most workstations. Therefore, *Linux* is an affordable way to bring the power of a workstation to a desktop computer.

There is one final argument for learning *Linux* that should not be underestimated: *Linux* is part of your curriculum! So if you intent to get your diploma, you are required to learn *Linux*. Also, lots of the software that is used for teaching in various courses at this university is based on *Linux*: Therefore, if you are not sufficiently familiar with *Linux* you will have trouble mastering those courses.

1.2 Overview

Below is an overview of the topics covered in this script.

1. The *XEmacs* editor.

The most basic tool is a text editor. Whether you are developing software, maintaining a server, or just writing email, you always need an editor. This is the reason we start our tour of *Linux* with the description of an editor.

For *Linux*, there are more than a dozen editors available. By far the most powerful editor is *XEmacs*, a variant of *Emacs*. The most distinguishing feature of *XEmacs* is the fact that it is very *customizable*: *XEmacs* has its own programming language that can be used to add new features as required.

2. The file system.

An editor enables us to create files. In order to manage these files, *Linux* has a file system. We discuss the various commands that are used for copying, renaming, moving, etc., of files.

3. Getting help.

Most *Linux* commands take a large number of options and several different services. There is no way to remember them all. Instead, one constantly has to look things up. *Linux* offers various ways to get all kinds of information about a particular command.

4. Pipes and little programs.

Linux offers a bunch of little programs that are not very impressive on their own. But *Linux* also offers the possibility to combine these little programs with the help of *pipes*. The resulting programs are both short and powerful. In fact, understanding pipes is one of the keys to unleash the power of *Linux*.

5. Job control.

Like all modern operating systems, *Linux* is a multitasking operating system. This means that several processes are running virtually simultaneously. We will discuss how the user can control these processes.

6. The *bash* shell.

A *shell* can be considered as the interface of the user to the operating system. Modern shells offer a programming language of their own. We discuss the *bash*, which is the most popular shell for *Linux*.

7. Regular expressions.

A number of Unix commands make extensive use of *regular expressions*. Basically, *regular expressions* are a very powerful way to describe sets of strings. Furthermore, *regular expressions* are the foundation of modern scripting languages like *Tcl*, *Perl*, or *Python*.

8. Various utilities. We finish this script by discussing various utilities that did not fit into any of the other sections.

2 The *XEmacs* Editor

XEmacs is one of the most powerful editors that are available. This implies that *XEmacs* is a very complex editor. For lack of space, this section can only discuss the basic concepts of *XEmacs*. For those that want to learn more about *XEmacs* there are several good books available: First, there is a tutorial book by Cameron et. al. [2]. Second, there is the *Emacs* manual by Stallman [10]. Although *Emacs* and *XEmacs* are different programs, *XEmacs* is an offspring of *Emacs* and most of what is true for *Emacs* is also true for *XEmacs*. Finally, there is a nice book by Glickstein [4] showing how *Emacs* can be customized.

The remainder of this section is organized as follows. The next subsection explains why we have chosen *XEmacs* as an editor. The next two subsections describe how to start *XEmacs* and how to perform the most basic editing tasks. The following subsections describe *XEmacs* in more depth: We describe *cursor movement*, basic editing like *deleting* or *moving* large chunks of text, searching and replacing, editing several files, the help system, and various ways to customize *XEmacs* in separate subsections. Finally, there is a subsection that points to those features that, for lack of space and time, can not be described in this script.

2.1 Why Use *XEmacs*?

Since *XEmacs* is a complex editor, you might justifiably ask: Why bother learning *XEmacs*? After all, most modern programming environments offer their own editor. However, during your professional career you will most likely work with several different programming languages. Even for a single programming language there are several programming environments. Does it make sense to learn a new editor each time? Furthermore, there are other activities than just programming. If you have to maintain a system, then there are several configuration files that need to be edited. If you write a mail, you need an editor. The enumeration of activities that involve an editor could be continued for pages, but you should have gotten the point by now: Rather than learning to use a different editor each time it does make sense to get familiar with one editor that can handle all of these tasks.

You might object that you dislike *XEmacs* because you have already been accustomed to a different editor. Do not despair, *XEmacs* is highly *customizable*: You can change the key binding to match your taste. You can even write your own *XEmacs* commands and in this way you can extend the editor according to your needs. Writing your own *XEmacs* commands can be done in three different ways:

1. Via *keyboard macros*.

This is in fact rather easy. If there is a long sequence of commands that you have to type repeatedly the same way, then *XEmacs* can record this sequence as a *keyboard macro*. You can bind this macro to some key and instead of typing the whole sequence of keys again and again you just press this key.

2. Via functions written in *Emacs Lisp*.

In case you need some functionality that is too complex to be handled by a macro, you can implement this functionality in *Emacs Lisp*. This is a full-fledged programming language and, actually, you can do almost everything in this language. For example, people even have implemented a web browser in *Emacs Lisp*!

Emacs Lisp has one drawback: Since it is a full-fledged programming language it is quite complex. Therefore, there is a third option described next.

3. Via commands external to *XEmacs*.

XEmacs offers the possibility to execute arbitrary shell commands on regions of text. So to add a new functionality to *XEmacs* you can implement this functionality in a programming language of your choice. *XEmacs* can then apply the program you have written to a given region of text.

Since this script deals mostly with *Linux* we will not have the time to discuss *Emacs Lisp* in this script. If you are interested, the web page

<http://www.gnu.org/manual/emacs-lisp-intro/emacs-lisp-intro.html>

contains an excellent tutorial describing *Emacs Lisp*.

2.2 Getting Started

First, we have to start the *XEmacs* editor. This done by typing

```
xemacs
```

at the prompt of a *shell*. OK, the notion of a *shell* has not been introduced yet. Let us assume that you are working with some graphical desktop like *KDE*. Look for an icon that has a form similar to the ones depicted in Figure 1 on page 6. Move the mouse



Figure 1: Shell icons.

pointer over this icon and click the left mouse button.

After clicking this icon, a window of the form shown in Figure 2 on page 6 should appear. Of course, the text `stroetma@stroetmannpc` will be different on your screen. After all, `stroetma` is my user account and `stroetmannpc` is the name of my computer. Your screen will presumably display the name of your user account and the PC you are currently working on. Following the name `stroetmannpc` is the text `“:~>”` and after that you see a black rectangle. The black rectangle in this window is known as the *cursor*. Let us type the word `“xemacs”` on the keyboard. The screen should now look as shown in Figure 3 on page 7. Press the `Return` key. Next, you will be greeted by a screen similar to the one shown in Figure 4 on page 8. Let us edit our first file. There

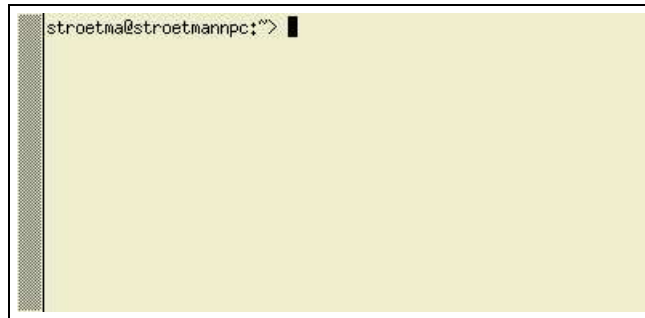


Figure 2: A shell with a prompt.

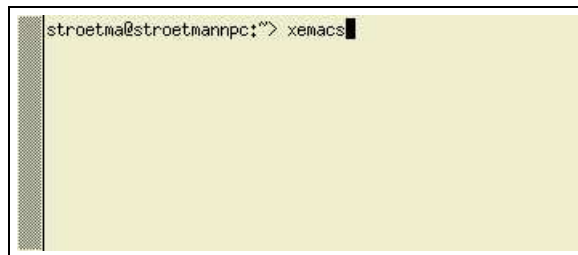


Figure 3: The shell after you have typed `xemacs`.

are two ways to do this. We could use the graphical user interface by moving the mouse over the icon labeled “Open”. Then a new window would appear asking us to either type the name of a file or just click on the name of an existing file. But **do not do** this. Using the graphical user interface (GUI) is for morons, not for power users. Using a text editor via its GUI is far too slow to be competitive. So we use a different approach by typing “Ctrl-x Ctrl-f”. If you are totally new to this you might ask “*What is this funny Ctrl-x stuff?*” Well, this is a *key combination*: You hold down the `Ctrl` key and then you press the `x` key. You release the `Ctrl` key only after you have released the `x` key.¹

In the same way, Ctrl-f means first pressing the `Ctrl` key and then pressing the `f` key. Once we have pressed these keys we will see the screen shown in Figure 5 on page 9.

In Figure 5 the cursor is on the bottom line of the screen. In front of the cursor, we see the *prompt*

```
Find file: ~/
```

This means the *XEmacs* is asking us for the name of a file that is to be edited. Let us call our first file “`story.txt`”. So we type `story.txt` and hit the *return* key. Now, the screen should look as shown in Figure 6 shown on page 10: The cursor is located on

¹ For the absolute novice: The `Ctrl` key is the key in the lower left hand corner of your keyboard. On a German keyboard this key might have the label *Strg*. Of course, the `x` key is the key that has the letter “x” printed on it. Pressing Ctrl-x involves 4 steps:

1. Press the `Ctrl` key down and do not release it.
2. While you are holding the `Ctrl` key down, press the `x` key down.
3. Release the `x` key, but continue to press the `Ctrl` key down.
4. Release the `Ctrl` key.



Figure 4: XEmacs startup screen.

the upper left hand corner of the screen that is otherwise empty. Let us start to write a short story now. After all, when discussing an editor we need some bulk of text that can be used to exercise the various functions of the editor. During typing, we can use the *arrow* keys to navigate. The arrow keys are the keys `←`, `↑`, `↓`, and `→`. To delete typing mistakes, we can use the `BS` key also known as *backspace*. This is the key above the `Return` key. Usually, it is labeled with the symbol “←”. We can also use the `Del` key also known as *delete*. On a German keyboard this key is labeled “Entf”, while on an American keyboard the label would be “Del”. The difference between the `BS` key and the `Del` key is that the `BS` key deletes the character that is left to the cursor while the `Del` key deletes the character under the cursor.

Once we have finished typing, the screen may look as shown in Figure 7 on page 11.

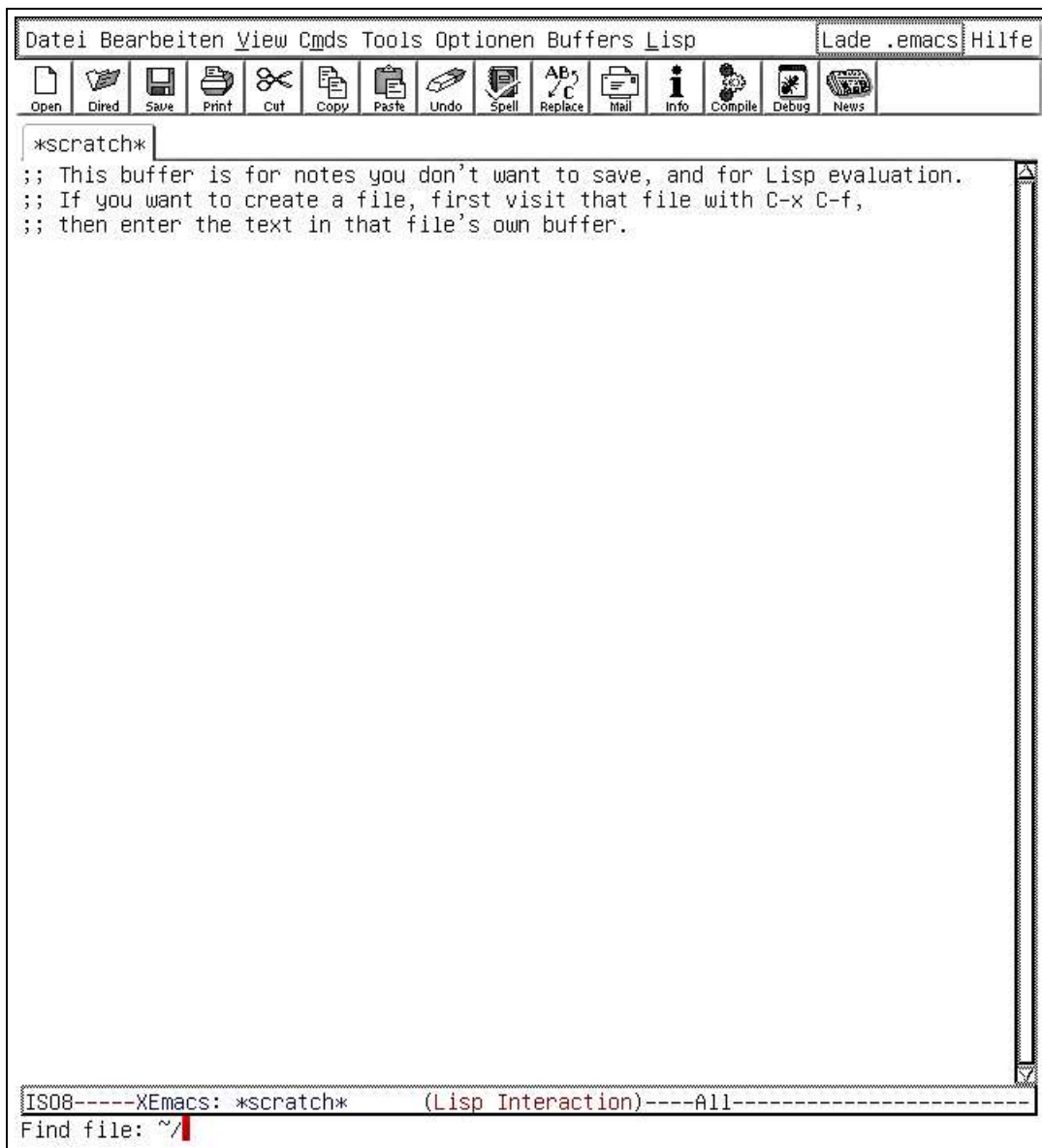


Figure 5: Screen after pressing Ctrl-x Ctrl-f.

Let us investigate this screen in more detail. The last line, that is the line at the bottom, is called the *command line*. It is also known as the *echo area* or the *mini buffer*. In the screen in Figure 7 it is empty. It was not empty in the screen shown in Figure 5 on page 9. The cursor is placed in the command line when XEmacs prompts the user for information like the name of a file. The command line is also used for messages. For example, in Figure 6 on page 10 the *echo area* displays the message

(New file)

informing us that we are starting to edit a new file.

The line above the *command line* is the *mode line*. It holds information regarding the status of XEmacs. The general form of the *mode line* is as follows:

encoding--change-info-XEmacs: file (mode)----position-----

Here the various terms set in italics have the following semantics:

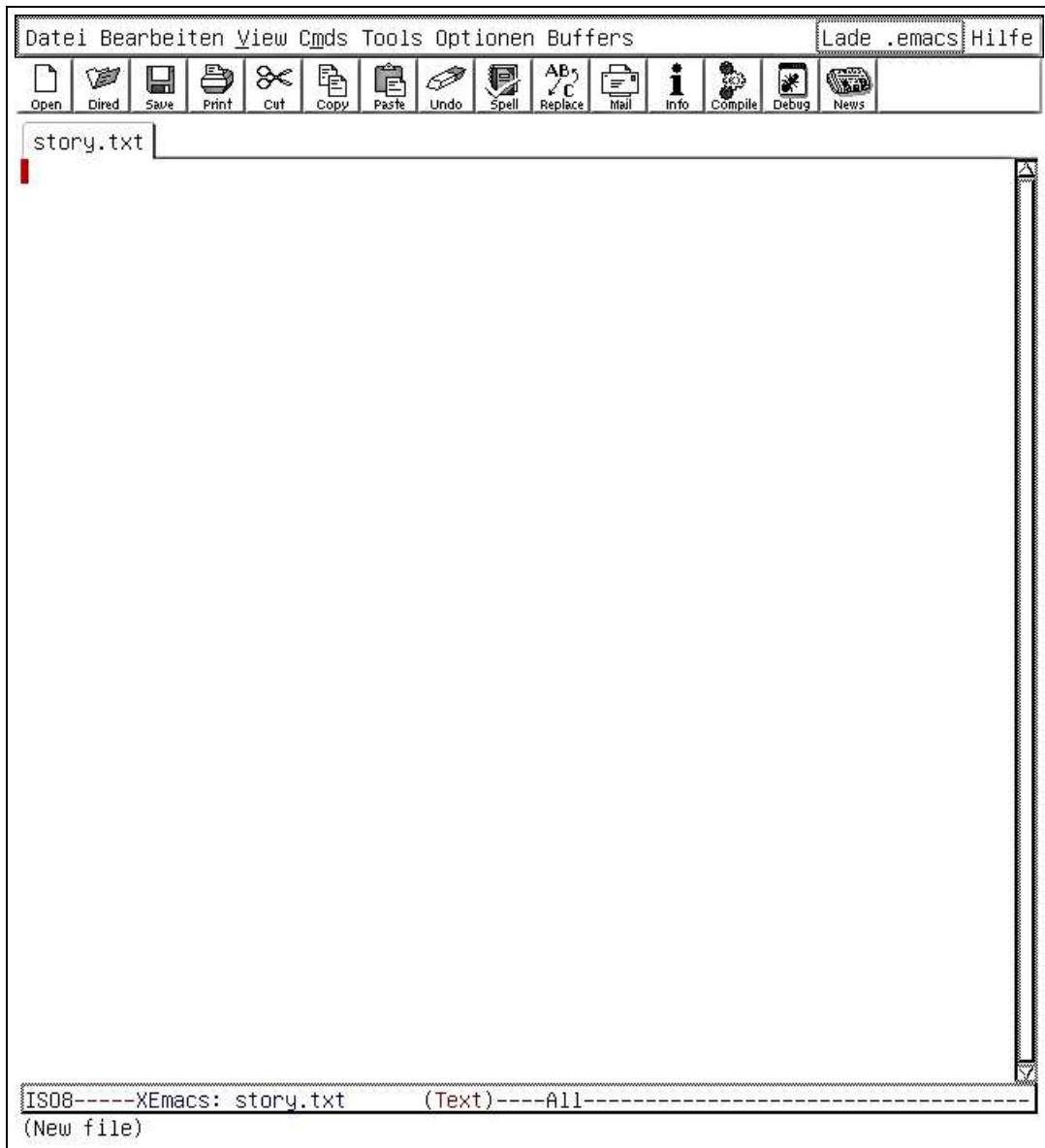


Figure 6: Screen after entering `story.txt`.

1. The *encoding* specifies the character encoding that is used. In our example *encoding* has the value “`ISO8`”. *XEmacs* is capable of editing Chinese or Japanese text. These texts are transformed using special encodings. “`ISO8`” is just one of these encodings. For the moment, the encoding information is of no interest to us. It is included here to make this description complete.
2. The *change-info* gives the information whether the file has been changed since it has last been saved. Since we have not yet saved the file the *change-info* has the value “`***`”. Let us save the file now. This is done by pressing the keys “`Ctrl-x Ctrl-s`”. Notice how the *change-info* changes to “`--`”, informing us that the file has been saved. There is another value that *change-info* can have: It is “`%%`”. This value is used to inform the user that the file loaded into the current buffer is *write protected*.

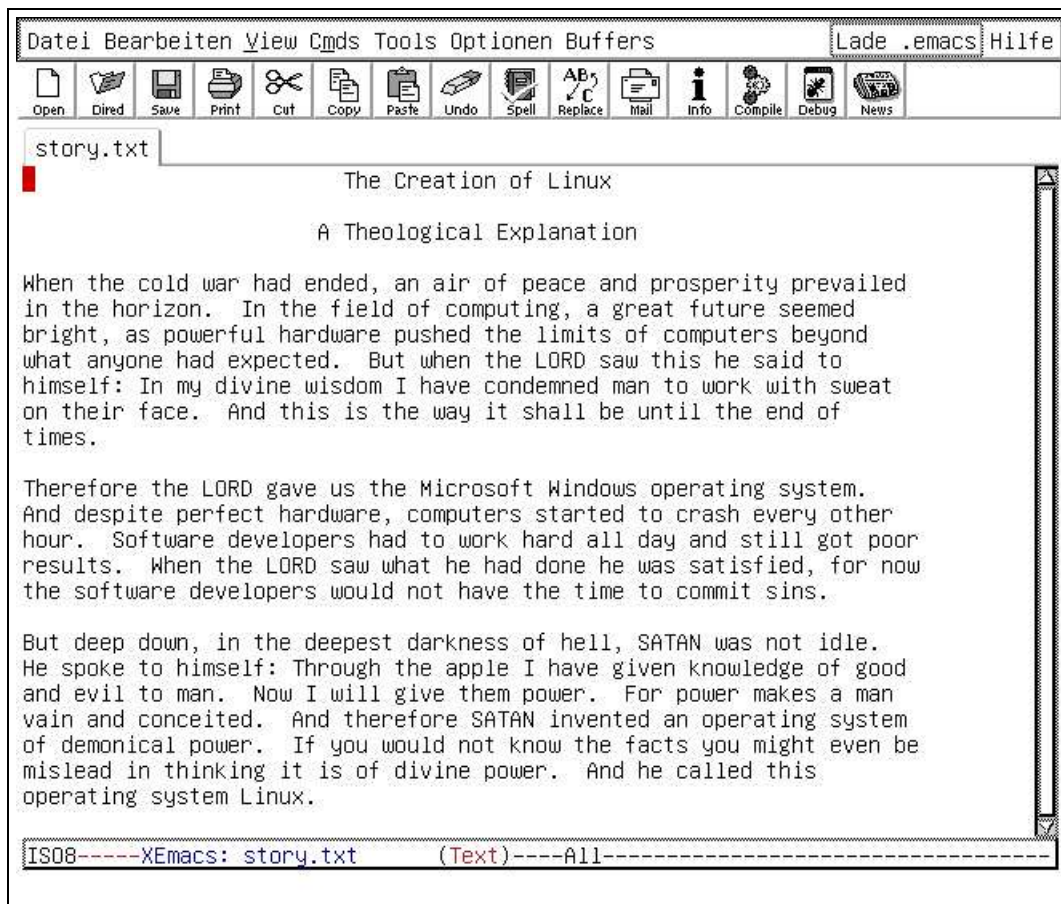


Figure 7: Screen after typing our story.

3. The *file* gives the name of the file we are currently editing. In our example, the value of *file* is “story.txt”.
4. The *mode* gives information about the *editing mode* that is active. In our example, the *mode* is given as “Text”. This is the default mode for editing text files.
5. The *position* gives us the position of the current location of the cursor. As long as the file fits onto the screen, the value is “All”. Later, when the file grows it will be something like “75%”. This would inform us that 75% of the file are above the top of the screen. Other values for *position* are “Top”, informing us that the screen displays the top of the file, or “Bot”, when we are at the bottom of a file.

Next, we take a look at the top line of the window in Figure 7 on page 11. When invoking a German version of XEmacs, this line looks as follows:

```
Datei Bearbeiten View Cmds Optionen Buffers Hilfe
```

By clicking on any of these words, a pull down menu will open and offer various options. However, in order to make full use of the menu bar, it is best to control it via the keyboard. To enable this feature, add the following lines at the end of either the file “~/ .emacs” or the file “~/ .xemacs/init.el”

```
(define-key global-map 'f10 'accelerate-menu)
(setq menu-accelerator-enabled 'menu-fallback)
(setq menu-accelerator-modifiers '(super control))
```

Normally, the file “~/ .emacs” is loaded when *XEmacs* is started. In turn, this file causes the file “~/ .xemacs/init.el” to be loaded. So if the lines above are added to either of these files, the next time you start *XEmacs* you will be able to activate the menu bar by pressing the function key `f10`. Then, pressing `f10` causes the first entry of the menu to be highlighted. To open this entry, press `return`. Use the arrow keys to move to a different menu entry. To close an opened menu entry, press `escape`. To return from the menu bar, press `escape` again. k

When a menu entry is opened, it shows several lines. To the left of every line there is an informative string describing the function that is invoked when the corresponding entry is chosen. For some entries there is also a keyboard shortcut available. In this case, this shortcut is written at the end of the line. In general, it is advantageous to remember these shortcuts and invoke the commands via these shortcuts since this is much quicker than using the menu bar.

2.3 The Fundamentals

Now that we know how to create and save a file, it is time to discuss the basic concepts of *XEmacs*. First, *XEmacs* has a number of *commands* that do various things. For example, there is a command for *opening* a file and another command for *saving* a file. Every command has a name. For example, the name of the command for saving a file is “save-buffer”. A command can be issued via its name. This is done via the Meta-x key combination. Meta-x means pressing the `Meta` key and then pressing the `x` key. The `Meta` key is the key left to the space bar. On a German keyboard it is labeled “Alt”. If you press Meta-x the command line looks as follows

```
M-x █
```

Here the box “█” represents the cursor. We can now enter any command. For example, we can enter the command “save-buffer” and press return to execute it. The commands that are used frequently have been bound to *key sequences*. A *key sequence* is just a sequence of different keys that are pressed. For example, the command “save-buffer” is bound to the key sequence “Ctrl-x Ctrl-s”. If a command requires an argument like the name of a file, a number, or something else, it will prompt for this information in the *echo area*.

There are two more fundamental commands we need to know. The first is keyboard-quit. Suppose that, in order to visit a file, you have typed Ctrl-x Ctrl-f and *XEmacs* now prompts you for the name of a file that is to be edited. At this point, you realize that you do not want to open a new file. What can you do now? The answer is the command keyboard-quit which is bound to the key Ctrl-g. Just hit this key and the prompt disappears from the command line.

Finally, you need to know how to exit *XEmacs*. This is done via the command

```
save-buffers-kill-emacs
```

which is bound to the key sequence Ctrl-x Ctrl-c. Table 1 on page 12 summarizes the most fundamental commands together with their key bindings. The commands shown this far are sufficient for the most fundamental editing tasks. However, if we would stop our treatment of *XEmacs* here, we would be missing all the goodies. The following subsections will familiarize us with some of these.

2.4 Cursor Movement

Using just the arrow keys for cursor movement is rather dull. Table 13 on page 13 lists the additional commands available for cursor movement. Looking at the first lines of

| command | key sequence | short description |
|-------------------------|---------------|--|
| previous-line | ↑ | move the cursor one line up |
| next-line | ↓ | move the cursor one line down |
| forward-char-command | → | move the cursor one character to the right |
| backward-char-command | ← | move the cursor one character to the left |
| delete-backward-char | BS | delete the character left from the cursor |
| delete-char | Del | delete the character under the cursor |
| save-buffer | Ctrl-x Ctrl-s | save the current buffer |
| find-file | Ctrl-x Ctrl-f | prompt for a file and edit it |
| keyboard-quit | Ctrl-g | abort a pending command |
| save-buffers-kill-emacs | Ctrl-x Ctrl-c | exit XEmacs |

Table 1: Fundamental commands and their key sequences.

this table you discover commands that move the cursor by just one character. This might seem redundant as the arrow keys achieve the same effect. However, to use the arrow keys you have to move your right hand away from the alpha-numeric keys of the keyboard. The idea behind offering the keys `Ctrl-f` and `Ctrl-b` is that you can keep your hands over the alpha-numeric part of the keyboard where most of the work is done.

A nice feature is the command `universal-argument` which is bound to the key `Ctrl-u`. Using this command you can repeat any other command a specified number of times. For example, if you want to move the cursor 123 lines down you can achieve this via the following key sequence:

```
Ctrl u 1 2 3 Ctrl n
```

If you just press `Ctrl u` and do not enter a number but instead enter a command, then a default value of 4 is assumed. Therefore,

```
Ctrl u Ctrl n
```

would move the cursor down by 4 lines.

There is one additional point that has to be discussed. Look at the last two lines of Table 2. These mention a *buffer*. What exactly is a *buffer*? To understand this concept, consider what happens when you edit a file with *XEmacs*: After issuing the `find-file` command you are prompted for the name of a file. If you choose the name of an already existing file, *XEmacs* loads this file into its internal memory. This internal working area is called a *buffer*. A visual representation of the buffer is what gets displayed on the screen. Now, when you start typing text, what is actually changed is the *buffer*. The file itself is not changed. It will only be changed once you issue the `save-file` command. Until this point, the buffer's content only exist in the internal memory of *XEmacs*.

2.5 Basic Editing

Suppose you have written a large chunk of text that you have to get rid of. Deleting this text one character at a time is rather dull. Therefore, *XEmacs* offers a number of functions for killing words, lines, sentences, or even a complete *region*. A *region* is a

| command | key sequence | short description |
|-----------------------|--------------|--|
| forward-char-command | Ctrl-f | move the cursor forward one character |
| backward-char-command | Ctrl-b | move the cursor backward one character |
| forward-word | Meta-f | move the cursor forward one word |
| backward-word | Meta-b | move the cursor backward one word |
| next-line | Ctrl-n | move the cursor down one line |
| previous-line | Ctrl-p | move the cursor up one line |
| backward-sentence | Meta-a | move the cursor backward one sentence |
| forward-sentence | Meta-e | move the cursor forward one sentence |
| beginning-of-line | Ctrl-a | move the cursor to the beginning of the line |
| end-of-line | Ctrl-e | move the cursor to the end of the line |
| scroll-up-command | Ctrl-v | scroll current window upward |
| scroll-down-command | Meta-e | scroll current window downward |
| forward-paragraph | Meta-} | move the cursor forward one paragraph |
| backward-paragraph | Meta-{ | move the cursor backward one paragraph |
| universal argument | Ctrl-u | repeat the following command |
| goto-line | Meta-g | move the cursor to a given line |
| beginning-of-buffer | Meta-< | move the cursor to the beginning of the buffer |
| end-of-buffer | Meta-> | move the cursor to the end of the buffer |

Table 2: Cursor movement commands.

chunk of text that is delineated by two locations known as *mark* and *point*. Now *point* is easy: This is just the position of the cursor. You can set the *mark* at any place in the text via the `set-mark-command` bound to the key `Ctrl-w`. It is also bound to the key `Ctrl-␣`. Here, the symbol “␣” denotes the space bar. Let us try this with our short story about *Linux*. If we place the cursor at the beginning of the first sentence and then move the cursor down we will see that the area between the mark we have set and the cursor is shaded grey. Figure 8 on page 15 shows this effect. Now we could issue the command `kill-region` which is bound to the key sequence `Ctrl-w` (and also to `Ctrl-x c`) and the shaded area would disappear. Furthermore, the killed text is saved in a special location in internal memory. This location is known as the *kill-ring*.

Sometimes, just being able to kill a region is not what is needed. More often a region of text needs to be moved around, that is we want to kill the region and insert it at a different place. Here the `yank` command comes to our help. It retrieves the last chunk of text that has been killed from the *kill-ring* and insert it at the cursor position. This command is bound to the key combination `Ctrl-y`.

If instead of moving a chunk of text we want to copy it, then killing is not the solution. Therefore, *XEmacs* offers the command `kill-ring-save` which is bound to the key combination `Meta-w`. This command copies the region to the *kill-ring* just as the command `kill-region` would do. However, in contrast to `kill-region` the marked

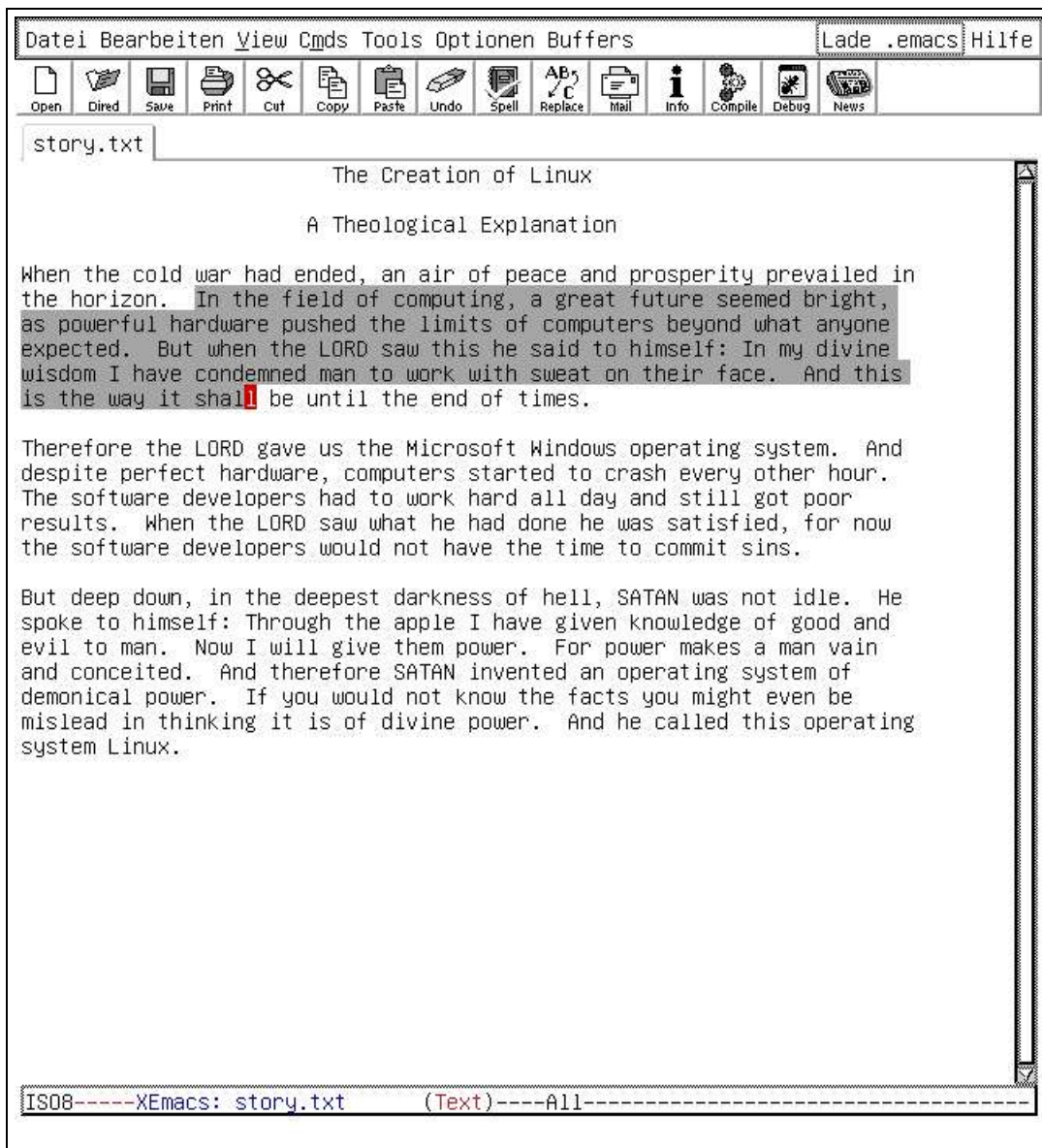


Figure 8: Effect of the `set-mark-command`.

text is not removed. So, if you want to copy a piece of text, mark it as a region and issue the command `kill-ring-save`. Then insert the text at the desired destination via the `yank` command.

Table 3 on page 14 lists the basic commands for killing and yanking text. If you look at it you realize that there two groups of commands that remove text. They either have the word “kill” or “delete” as part of their name. If they have “kill” as part of their name, then they store the text that they remove in the *kill-ring*. Those that have “delete” as part of their name just erase text, they do not store anything.

Let us discuss the structure of the *kill-ring* in more detail. Actually, the kill ring can be viewed as a stack. Every time you kill new text this new text will be put on top of this stack. The string that has been killed most recently is on top of the stack. This string can be retrieved with the “yank” command. Let us visualize this. Assume you have killed the words “first”, “second”, and “third” using separate invocations of

| command | key sequence | short description |
|----------------------|---|---|
| set-mark-command | Ctrl-w, Ctrl- <u>l</u> | set the mark |
| delete-char | Ctrl-d | delete the character under the cursor |
| delete-backward-char | backspace | delete the character left from the cursor |
| kill-word | Meta-d | kill the word following the cursor |
| backward-kill-word | Meta-backspace | kill the word left to the cursor |
| kill-line | Ctrl-k | kill the rest of the current line |
| kill-sentence | Meta-k | kill the rest of the current sentence |
| kill-region | Ctrl-w | kill the region |
| kill-ring-save | Meta-w | copy the region to the kill ring |
| yank | Ctrl-y | reinsert the last stretch of killed text |
| yank-pop | Meta-y | replace just-yanked stretch of killed text with a different stretch |
| undo | Ctrl-/ | undo last change |

Table 3: Basic commands and their key sequences.

kill-commands. Assume further that “first” has been killed first, “second” has been killed second, and “third” has been killed last. Then the kill-ring looks as is shown in Figure 9 on page 16. The word “third” is on top of the kill-ring and the *yank-pointer* points to this word. This word is retrieved when you issue the yank command.

| yank-pointer | kill-ring |
|--------------|-----------|
| → | third |
| | second |
| | first |

Figure 9: Structure of the *kill-ring*.

Now suppose you want to retrieve the second string. This string is already buried in the stack, but we can uncover it by first issuing the “yank” command. This gives us the string “third”. Now, invoke the “yank-pop” command which is bound to the key combination Meta-y. The string “third” vanishes and in its place the string “second” appears. Furthermore, this command will rotate the kill-ring, so the kill-ring will now look as shown in Figure 10 on page 16. Observe that the yank-pointer now points to the word “second”. If we would have wanted to retrieve the string “first”, then another invocation of the command *yank-pop* would have moved the yank-pointer down in the kill-ring to point to the word “first”. If we would then issue the *yank-pop* command a third time then, since the stack has reached its bottom, the yank pointer would move back to the top of the stack i.e., we would be back with the situation shown in Figure 9. This is the reason the kill-ring is called a *ring* as opposed to a *stack*.

| yank-pointer | kill-ring |
|--------------|-----------|
| → | second |
| | third |
| | first |

Figure 10: Structure of the *kill-ring* after *yank-pop*.

To summarize, the yank command retrieves the string pointed to by the yank-

pointer. The `yank-pop` command moves the `yank-pointer` down in the kill-ring. If the `yank-pointer` points to the element at the bottom of the kill ring, then the next invocation of the `yank-pop` command moves the `yank-pointer` back to the top of the kill-ring. Note that a `yank-pop` command can only be issued following a `yank` command or another `yank-pop` command.

2.6 Searching and Replacing

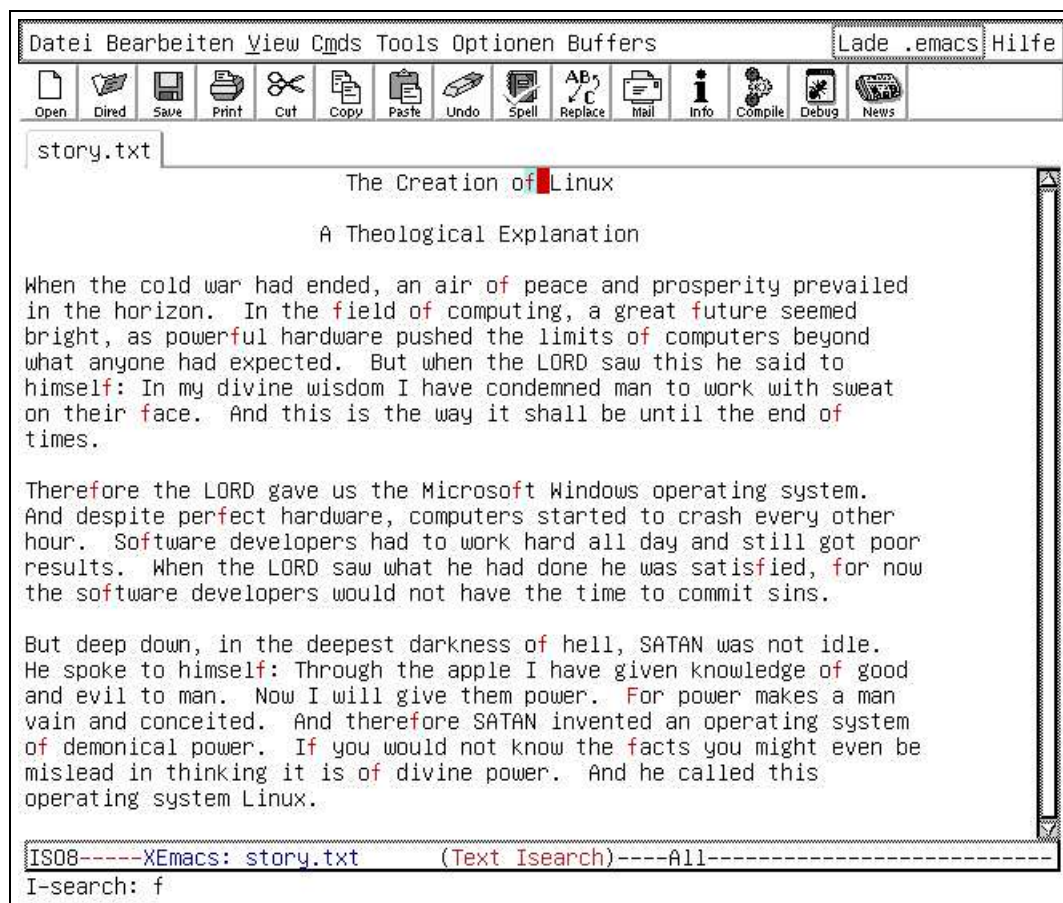


Figure 11: Effect of typing “f” in incremental search.

There are several ways to search for strings in *XEmacs*. The easiest is the command `isearch`, which is bound to the key `Ctrl-s`. The command name “`isearch`” is short for “incremental search”. Assume that we are working with a buffer containing the text shown in Figure 7. Suppose that we want to search for the string “`field`”. Assume further that the cursor is located at the upper left corner of the screen. After typing the key `Ctrl-s` to invoke `isearch` the cursor moves to the command line which now looks as follows:

```
I-search: █
```

Let us type the letter “`f`” next. At this moment, the cursor will jump to the first occurrence of this letter in the text. This is the occurrence of “`f`” in the word “`of`”. The screen will now look as shown in Figure 11 on page 17. Furthermore, notice that all other occurrences of the letter “`f`” have also been highlighted. Unfortunately, you can not observe this effect in the black-and-white printout of this script since the highlighting

uses colors. However, you can observe the highlighting if you read this script online on a color monitor. We can move to the next occurrence of “f” by typing `Ctrl-s` another time. The cursor then moves to the occurrence of “f” in the word “of” in the third non-blank line of our text. Next, let us type the letter “i”, which is the second letter in the word “field” that we are searching: The cursor moves to the word “field”. Further, notice that all the occurrences of “f” that had been highlighted previously are no longer highlighted. There is one exception: The word “satisfied” contains the letter sequence “fi” and therefore this letter sequence is highlighted also. The screen now looks as in Figure 12 on page 18. Now that we have found the word “field” our search is done. To

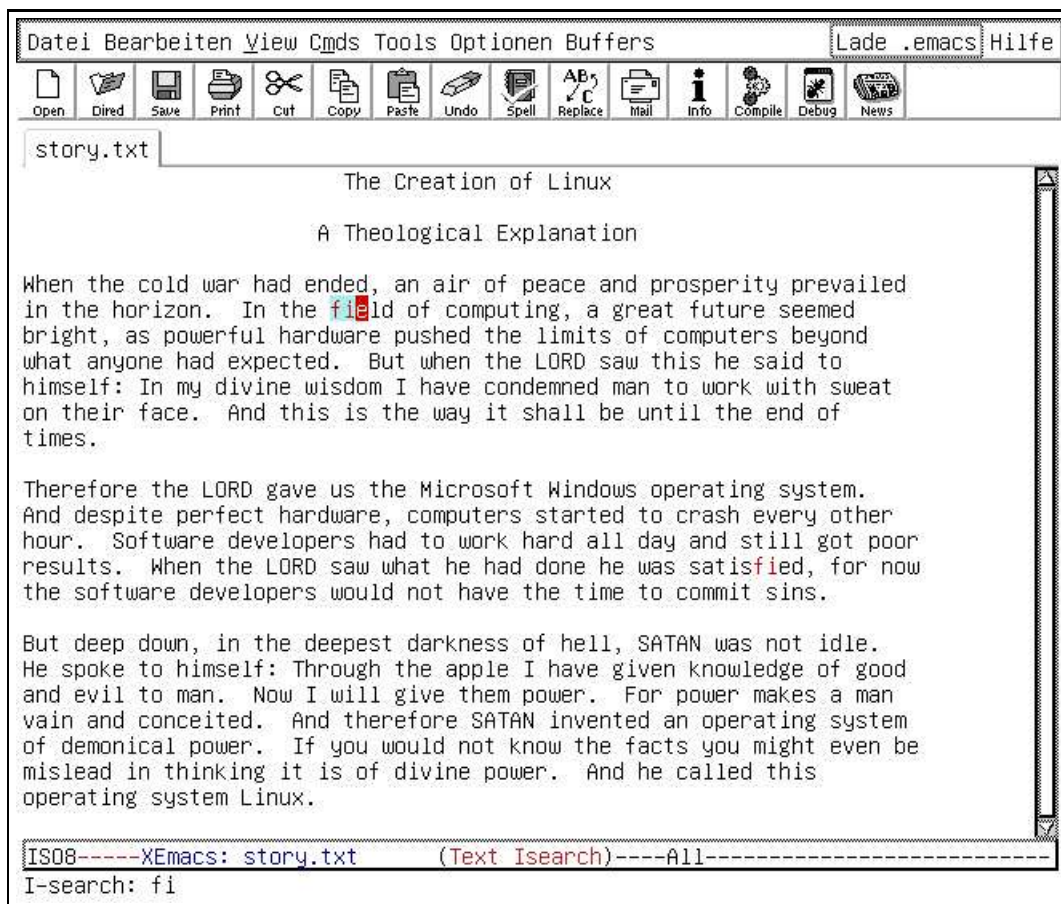


Figure 12: Effect of typing “fi” in incremental search.

quit the search we press the return key.

There is another way to quit incremental search. If we issue any of the commands for cursor movement described previously, then the incremental search is also terminated.

Let us take another look at the story in Figure 12. With a deep shock we realize that the account given in this short story is not politically correct! We have to exchange the masculine pronoun “he” with the feminine pronoun “she”. At least, as the bare minimum, this change has to be done for the occurrence of “he” that refers to the devil. Here, the command `query-replace` comes to our rescue. This command is bound to the key `Meta-%`. So we hit this key next. As a result, the cursor moves to the command line where the following prompt appears:

Query replace: █

Let us type “he” and press the return key. Now the command line looks as follows:

```
Query replace he with: █
```

so we type “she” next and press return again. Now the screen will look as shown in Figure 13 on page 19. The first occurrence of the string “he” is highlighted and the

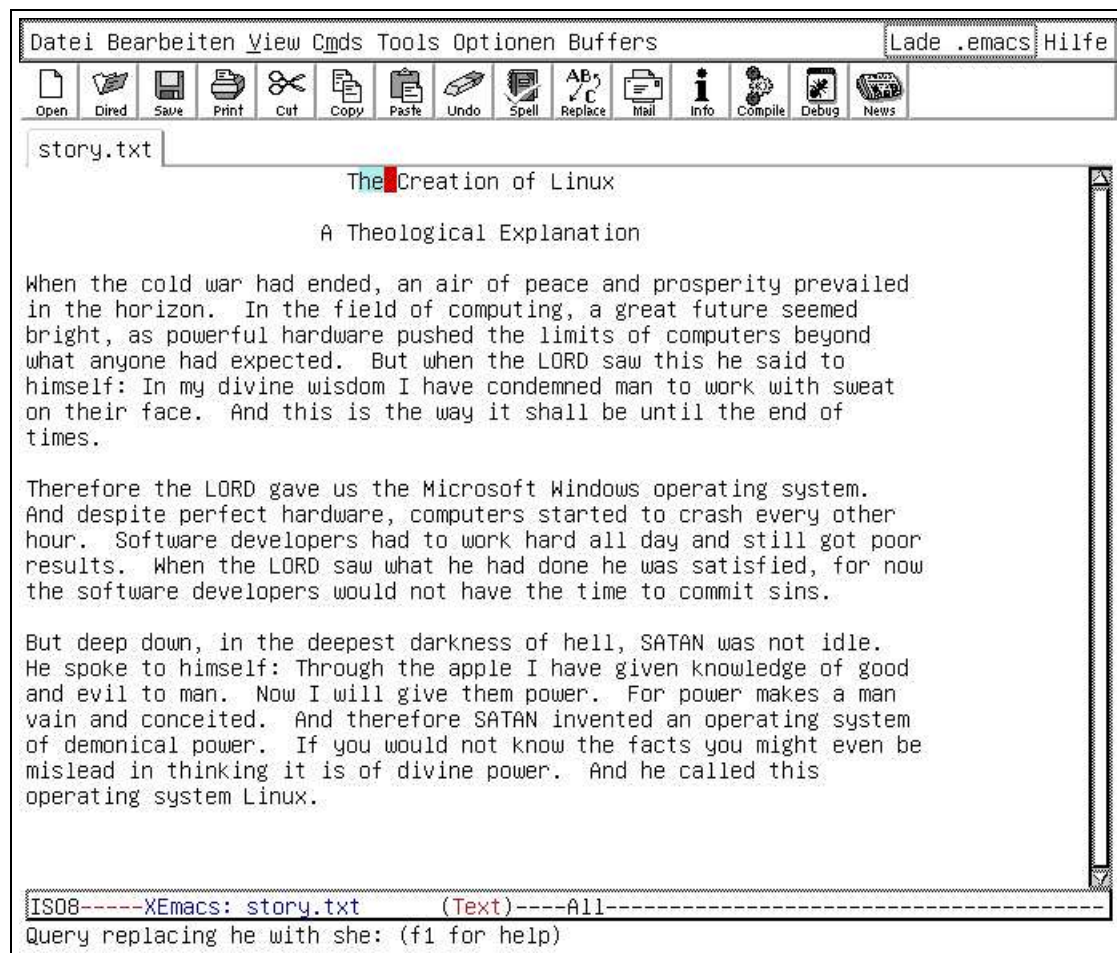


Figure 13: Effect of typing “fi” in incremental search.

cursor is placed right after this occurrence. Unfortunately, this occurrence is in the string “The” and we certainly do not want to change this string into a “Tshe”. Therefore we press the `Del` key now. This will move the cursor into the word “Theological” because there is another occurrence of “he”. We continue to press the `Del` key until we arrive at the first occurrence of “he” that is not buried in a word. To change this occurrence we press the space bar. Now the “he” is changed to a “she” and we move to the next occurrence of “he”. We continue pressing the space bar or the `Del` key as required. Occasionally, we might press the `Del` key one time to many and thereby miss an occurrence of “he”. To recover, we can press the key `^`. (On a German keyboard, the key “^” is located below the `Esc` key.) The query-replace command finishes automatically after visiting all occurrences of “he”. Upon termination it informs us about the number of occurrences of “he” that have been replaced by “she”.

Table 4 on page 18 lists the commands for searching and replacing. This table contains two commands not covered so far: `search-forward` and its pendant `search-backward`. These act more or less like `isearch-forward` and `isearch-backward`,

| command | key sequence | short description |
|------------------|--------------|--|
| isearch-forward | Ctrl-s | search forward for a given string |
| isearch-backward | Ctrl-r | search backward for a given string |
| query-replace | Meta-% | search for a given string and replace it with another string |
| search-forward | | search for a given string, not incremental |
| search-backward | | search for a given string, not incremental |

Table 4: Searching and Replacing.

but they are not incremental. Therefore, if you use these commands you have to type the entire search string in answer to the prompt of these commands. Then press return. The cursor moves to the first occurrence of the search string. If the search fails, the cursor does not move and the echo area displays the message

Search failed: (search-string)

where *search-string* is the string that you searched for. The commands `search-forward` and its pendant `search-backward` are needed when defining keyboard macros since `isearch-forward` and `isearch-backward` do not work for keyboard macros.

Table 5 on page 20 lists the keys that have a special semantics while executing a `query-replace` command. We see that instead of pressing the space to replace an

| key sequence | short description |
|------------------|---|
| <i>space bar</i> | replace one occurrence |
| y | replace one occurrence |
| , | replace one occurrence but do not move cursor |
| Del | skip to the next occurrence |
| n | skip to the next occurrence |
| ^ | move back to the previous occurrence |
| ! | replace all remaining occurrences without further questions |
| q | quit query-replace |
| <i>Return</i> | quit query-replace |
| Ctrl-r | enter recursive edit level |
| Meta-Ctrl-c | exit recursive edit level |

Table 5: Special keys active during query-replace.

occurrence we can also press the key `y`. Similarly, instead of pressing `Del` we can press the key `n`. The key `,` is quite handy: If we press it, the occurrence which is current will be replaced. However, the cursor will not be moved. If we discover that we have made a mistake we can press `q` to abort the `query-replace` command and correct the error. Normally, this will be done by issuing the undo command bound to `Ctrl-.`. Another possibility to correct any error is provided by *recursive editing*. While executing a `query-replace` command, pressing `Ctrl-r` brings us to a *recursive level* of editing where we can correct any problems that we have spotted during execution of `query-replace`. Once we have corrected these problems we finish recursive editing by pressing `Meta-Ctrl-c`. This brings us back to our `query-replace` session. The cursor

is then at the exact position where we left for recursive editing. Finally, a word of warning is in order regarding pressing `!` to replace any further matches. This should only be used with extreme care: If you have a large text file, chances are good that you replace some occurrences that you did not intend to have replaced.

2.7 Editing Multiple Files

When you edit a file with *XEmacs* the contents of this file are read from disk and put into a special place which is called a *buffer*. What you really edit is this buffer. The file which resides on disk is only updated when you save the buffer via the “save-file” command. *XEmacs* allows you to edit several files simultaneously. This happens automatically if you issue the command “find-file” several times for different files. You can list the buffers that are currently available via the command “list-buffers”. This will create a new buffer listing all available buffers. Every buffer is listed on a separate line together with information concerning it. In this list, you can search for the buffer you want to visit. Once you have found it, simply place the cursor on the line associated with the buffer and press return. If you know the name of the buffer in advance, use the “switch-to-buffer” command. This command prompts for the name of the buffer that is to be visited.

If multiple files have to be edited simultaneously, it is convenient to have two or more buffers visible at the same time. This can be achieved by splitting the window either vertically or horizontally using either of the commands “split-window-vertically” or “split-window-horizontally”. The cursor can then be moved between these windows using the command “other-window”. You can also scroll the other window up or down using the command `scroll-other-window` or

`scroll-other-window-down`. These commands are bound to the keys `Meta-Ctrl-v` and `Meta-V`, respectively. Additionally, the functions are bound to the keys `Meta-Bild↑` and `Meta-Bild↓`.

| command | key sequence | short description |
|--|----------------------------|--|
| <code>write-file</code> | <code>Ctrl-x Ctrl-w</code> | write buffer to a different file |
| <code>switch-to-buffer</code> | <code>Ctrl-x b</code> | switch to a different buffer |
| <code>switch-to-buffer-other-window</code> | <code>Ctrl-x 4 b</code> | switch to a different buffer, but display this buffer in a separate window |
| <code>list-buffers</code> | <code>Ctrl-x Ctrl-b</code> | display a list of existing buffers |
| <code>kill-buffer</code> | <code>Ctrl-x k</code> | kill a buffer |
| <code>other-window</code> | <code>Ctrl-x o</code> | move cursor to the next window |
| <code>split-window-vertically</code> | <code>Ctrl-x 2</code> | splits the window vertically |
| <code>split-window-horizontally</code> | <code>Ctrl-x 3</code> | splits the window horizontally |
| <code>delete-other-windows</code> | <code>Ctrl-x 1</code> | un-split the screen |
| <code>scroll-other-window</code> | <code>Meta-PgUp</code> | scroll the other window up |
| <code>scroll-other-window-down</code> | <code>Meta-PgDn</code> | scroll the other window down |

Table 6: Editing several files simultaneously.

Table 6 on page 21 lists the command available for editing several files simultaneously. We proceed to describe those commands that have not yet been discussed. We start with the command `write-file`. Assume that we have changed our short story

along the lines discussed in the previous subsection. Now we want to save this story, but we would like the original version of this story to be undisturbed. Therefore, we need to save the new version under a different file name. To do this, we issue the command `write-file`. This command prompts us for the new file name. After entering the name `story-correct.txt` and pressing return the new version is saved under this new name. Notice that also the mode line changes: It now displays the name `story-correct.txt`.

Let us start editing two files simultaneously by issuing the `find-file` command. In return to the prompt

```
Find file: /~
```

we type `story.txt` and press return. Now the screen changes again and we are back with our old version. In order to compare it to our new version we issue the command `split-window-vertically`, which is bound to `Ctrl-x 2`. Then the screen will look as depicted in Figure 14 on page 22. There are two windows, but both display the same

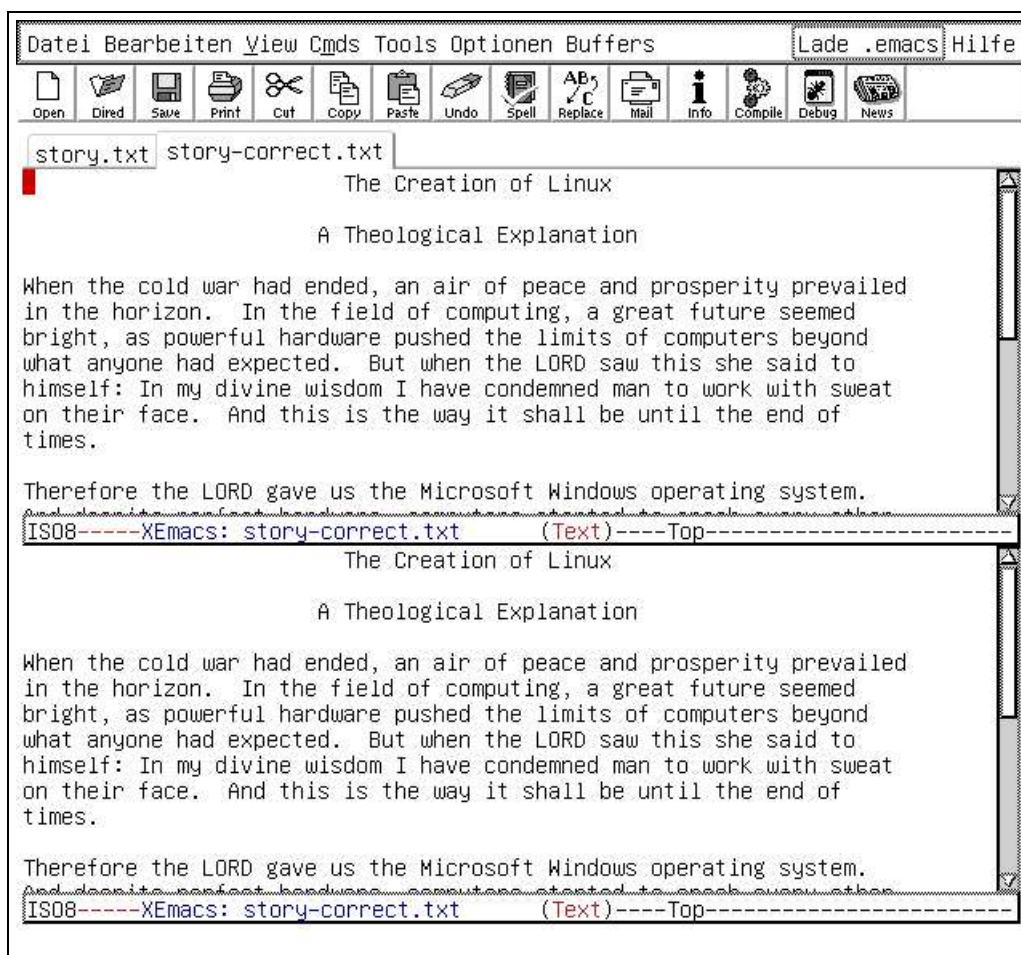


Figure 14: Effect of splitting the screen.

buffer. To get back to the buffer containing the politically correct version we issue the command `switch-to-buffer`. This will prompt us for the name of a buffer to switch to. This prompt looks as follows:

```
Switch to buffer: (default story-correct.txt)
```

The prompt already suggests the name of the buffer that we want to visit by displaying

this name in parentheses. So instead of typing the name “`story-correct.txt`” we can just type return. Now the screen will look as shown in Figure 15 on page 23.

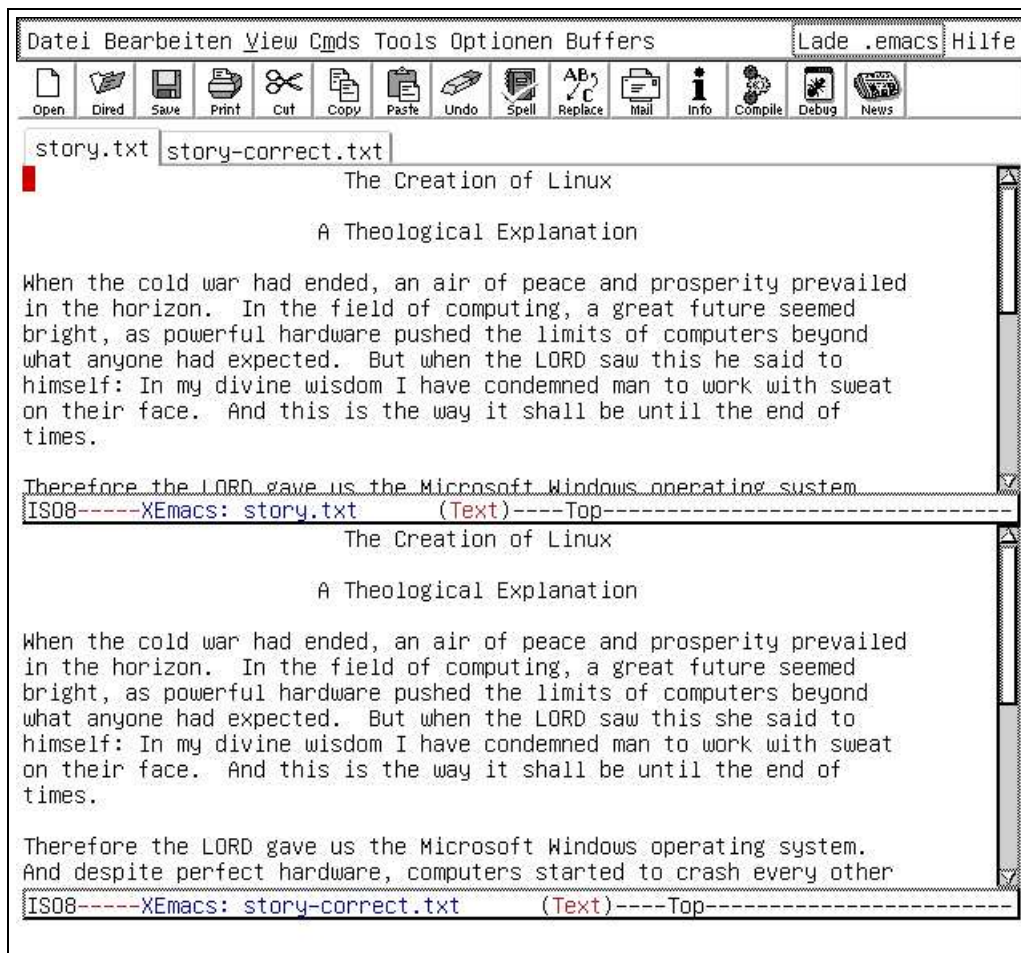


Figure 15: Effect of switching to `story-correct.txt`.

Now this is what we want: The upper half of the screen displays our original story, while the lower half displays the politically correct version. We can switch between these different versions using the command `other-window`. This command is bound to the key `Ctrl-x o`. Finally, if we decide to kill the old version, this can be done via the command `kill-buffer`. This command is bound to the key `Ctrl-x k`. This command prompts us for the buffer to kill and offers a default buffer:

```
Switch to buffer: (default story.txt) █
```

However, this time we decide to kill the buffer `story-correct.txt`, type this name, and press return. Then, this buffer vanishes but the screen is still split. It looks now as shown in Figure 16 on page 24. The lower half of the screen displays a buffer with the strange name “`*scratch*`”. We can remove this buffer from our screen by issuing the command `delete-other-windows`, which is bound to the key sequence `Ctrl-x 1`. After typing this key sequence, the screen is *un-split*.

2.8 Completion

One of the features of *XEmacs* that really excited me when I first saw them is the *completion* facility. This feature is best introduced via an example. Suppose I have already

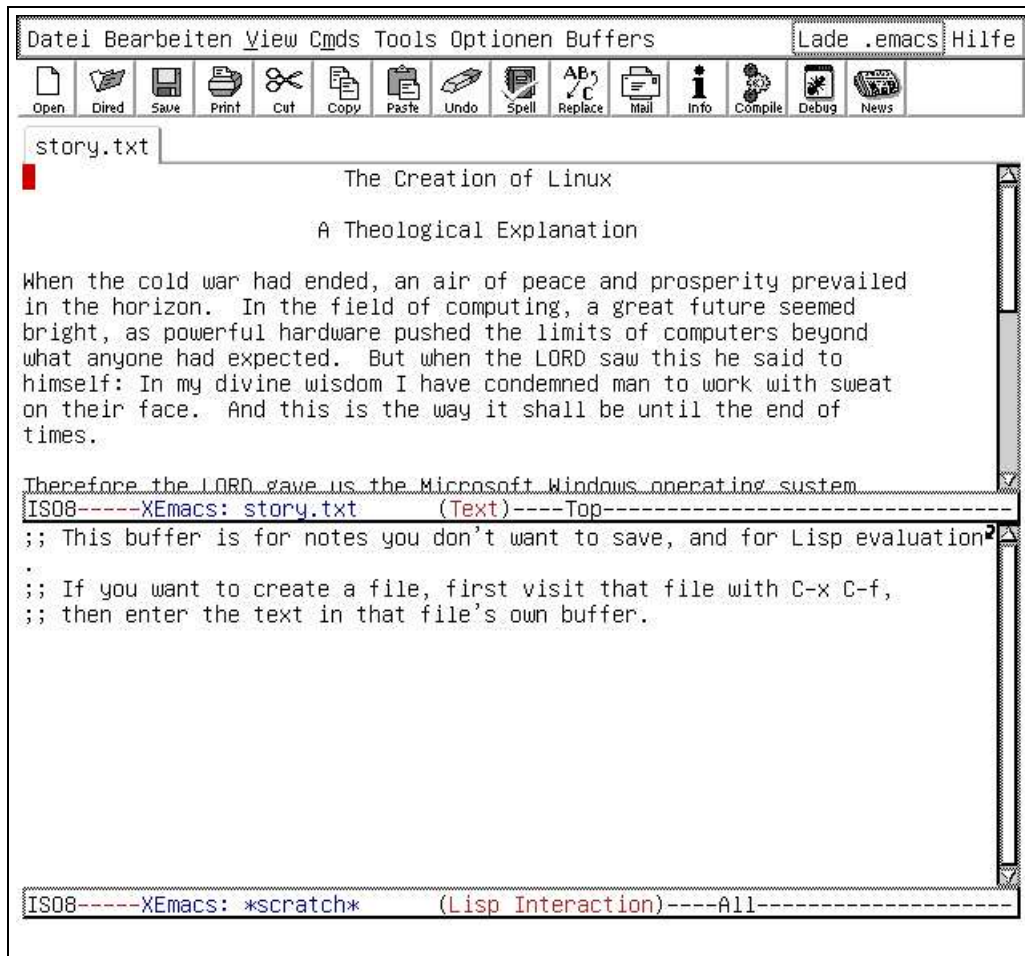


Figure 16: Effect of switching to story-correct.txt.

typed the following text into a buffer:

```
Getting Started With the Microsoft Windows Operating System
```

Ok, you have bought this new computer and it says that is has the

At this point I have to write the text “Microsoft Windows operating system”. Typing it again is rather dull. Copying it by placing its first occurrence into a region and then yanking it back is probably not worth the effort. But there is a third option. Let us write the first two character “Mi” and then press Meta-/. Like magic, the “Mi” has been changed to “Microsoft”. So far, so good. Lets us insert a space symbol and press Meta-/ again: Lo and behold, the word “Windows” appears next. Repeating the sequence of pressing the space bar and the Meta-/ key two more times the words “operating” and “system” surface like magic. XEmacs has found these words by comparing the string before the cursor with similar strings in the buffer. In the case that XEmacs guesses wrong there is no need to worry: Simply press Meta-/ again to give XEmacs another chance to guess right. You can do this several times. If XEmacs can not find any further guesses it will beep and leave you with the string that you have typed initially.

The key Meta-/ invokes the command dabbrev-expand. This command uses a substring and searches for words starting with this substring. This search is done in those

buffers that are currently loaded into XEmacs. There is another command that uses a dictionary for expansion. This command is called `ispell-complete-word`. To see it work, assume that you have to type the word

“antidisestablishmentarianism”

which is quite a mouthful. After typing “antid ” you can issue the command `ispell-complete-word`. Then, the screen will split as shown in Figure 17 on page 24. At the top, there will be a small window showing three choices:

(0) antidisestablishmentarianism (1) antidote (2) antidotes

In front of each choice is a number. Pressing the number “0” will now complete the word “antid” to “antidisestablishmentarianism”. If you had pressed “1” instead, you would have gotten “antidote”. Of course, for the function `ispell-complete-word` to be really useful we need to bind it to a key. The subsection on customizing XEmacs below will show how this is done.

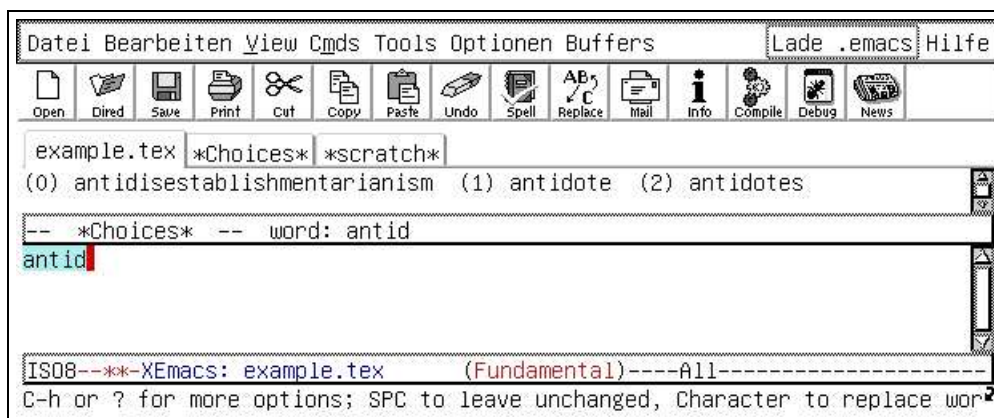


Figure 17: Effect of switching to `story-correct.txt`.

Table 7 on page 25 shows the commands available for completion.

| command | key sequence | short description |
|---|--------------|--|
| <code>dabbrev-expand</code> | Meta-/ / | expand to the most recent word |
| <code>ispell-complete-word</code> | | try to complete the word before the cursor using the ispell dictionary |
| <code>minibuffer-complete</code> | Tab | complete minibuffer |
| <code>minibuffer-complete-word</code> | space | complete at most one word |
| <code>minibuffer-complete-and-exit</code> | Return | complete and exit |
| <code>minibuffer-list-completions</code> | ? | list all completions |

Table 7: Commands for getting completion.

Of course, XEmacs has also things like *file name completion* and *buffer completion*. For example, when you issue the command `find-file` and are prompted for a file name, then, in case you want to edit an existing file you can get by via entering the first

two or three letters of this name and then hitting the `Tab` key. A similar mechanism exists when you are prompted for the name of a buffer.

There is one additional completion mechanism that is very useful: the *command completion mode*. To invoke this type of completion, we issue the command

```
icomplete-mode
```

At first, issuing this command does not produce any visible effect. However, as soon as we enter another command, we see the benefit. To illustrate command completion, let us test it with a simple command. Assume that we want to start a search using the command `search-forward`. So we type

```
Meta-x sea
```

and pause after typing the “sea”. Now the minibuffer looks as follows:

```
M-x sea(rch-){backward,backward-regexp,forward,forward-regexp}
```

This information can be interpreted as follows:

1. `M-x sea` is the string that we have typed so far.
2. Following this are characters enclosed in parentheses, in our case we have `(rch-)`. The interpretation of these characters is that any completion of the characters `sea` that form a valid *XEmacs* commands has the characters `rch` following the already typed characters `sea`. We can complete our command to search by pressing the space bar. This changes the command line to

```
M-x search-{backward,backward-regexp,forward,forward-regexp}
```

3. At the end of the command line we have a set of alternatives. The elements of this set are enclosed in the curly braces “{” and “}” and are separated by commas. In our case these elements are

- (a) `backward`
- (b) `backward-regexp`
- (c) `forward`
- (d) `forward-regexp`

To continue our example, let us type the letter “b” next. The command-line changes to

```
M-x search-b(ackward){,-regexp}
```

showing that now there are only two remaining choices, one is `search-backward` and the other one is `search-backward-regexp`. Pressing the space bar changes the minibuffer to

```
M-x search-backward{,-regexp}
```

Finally, pressing enter invokes the command `search-backward`.

The completion facility of the minibuffer is not restricted to command completion. File names and buffers can be completed in essentially the same way. For example, if you want to visit the file `story.txt` and enter the keys `Ctrl-x Ctrl-f` to invoke the command `find-file`, you are prompted for the name of a file to visit. Instead of entering the complete file name, you can restrict yourself to typing the first few letters and then pressing the space bar. This will complete the file name as much as possible. If you then press the space bar again, the screen is split horizontally and the lower half lists the alternatives for completion.

2.9 Abbreviations

Abbreviations are yet another way to reduce typing. In the following, in order to show the usefulness of the concept, the word abbreviation will be abbreviated as *abbrev*. To understand how abbrevs are used in *XEmacs*, assume that you have typed the word "Microsoft". At this point you realize that you will need to type this words quite frequently. Therefore, you decide to to define an *abbrev* for it. This is done by pressing `Ctrl-x a g` immediately after you have typed "Microsoft". You are then prompted for an abbreviation as follows

```
Global abbrev for "Microsoft":
```

Type "ms" in respond to this prompt and press return. This installs "ms" as a global abbreviation for "Microsoft". The next time you need to type the word "Microsoft", type "ms" instead and invoke the command `expand-abbrev`, which is bound to the key `Ctrl-x a e`. This will change the string "ms" into "Microsoft". But you can do better. Activate automatic expansion of abbrevs via the command `Meta-x abbrev-mode`. Now every time that you type "ms" followed by a non-alphanumeric character, the "ms" is expanded into "Microsoft". So far, so good. But assume that the text to be abbreviated consists of multiple words. For example, assume that the text "Microsoft Windows Operating System" has to be abbreviated by "mwos". How can this be done? Pressing `Ctrl-x a g` after the last word of this text would only define an abbrev for the last word, that is for "System". This problem is solved by the function `universal-argument`, which is bound to `Ctrl-u`: Just press "`Ctrl-u 4 Ctrl-x a g`". Then *XEmacs* uses the last 4 words as the text that is to be abbreviated and therefore prompts you as follows:

```
Global abbrev for "Microsoft Windows Operating System":
```

Typing "mwos" in response installs the required abbreviation.

There are some additional commands that deal with abbreviations. They are listed in Table 8 on page 26 together with those commands that have been discussed. Observe that you can edit the abbrevs via the command `edit-abbrevs` and then activate your changes to these abbrevs by pressing `Ctrl-c Ctrl-c`.

| command | key sequence | short description |
|------------------------------------|----------------------------|---|
| <code>add-global-abbrev</code> | <code>Ctrl-x a g</code> | define a new abbrevs |
| <code>expand-abbrev</code> | <code>Ctrl-x a e</code> | expand an abbrevs |
| <code>edit-abbrevs-redefine</code> | <code>Ctrl-c Ctrl-c</code> | activate changed abbrevs |
| <code>abbrev-mode</code> | | activate automatic expansion of abbrevs |
| <code>kill-all-abbrevs</code> | | delete all defined abbrevs |
| <code>expand-region-abbrevs</code> | | expand all abbrevs in region |
| <code>list-abbrevs</code> | | list all abbrevs |
| <code>edit-abbrevs</code> | | edit abbrevs |
| <code>write-abbrev-file</code> | | write all defined abbrevs to file |
| <code>read-abbrev-file</code> | | read abbrevs from file |

Table 8: Commands for abbreviations.

We have only discussed *global abbreviations*. These are those abbreviations that are defined independent of the mode of *XEmacs*. Abbreviations can also be defined specifically for one mode, i.e., it is possible to write abbreviations that are only active when editing, for example, *Html* files. However, this feature is beyond the scope of this script.

2.10 Customizing XEmacs

2.10.1 Customizing Key Bindings and Variables

XEmacs can be customized in various ways. First, and most importantly we can bind commands to different keys. There is a special file called “.emacs”. It is read when XEmacs is started up. Let us edit this file and put the following line at its end:

```
(global-set-key [ (f2) ] 'ispell-complete-word)
```

Next time, when we start XEmacs and this file is loaded the command `ispell-complete-word` will be bound to the key `[F2]`. But we do not have to wait this long. If we place the cursor at the end of this line after the closing parenthesis “)” we can then issue the command `eval-last-sexp`. This will install the new key binding immediately. We proceed by giving some examples for key bindings.

```
(global-set-key [ (control x) (v) ] 'version)
(global-set-key [ (control v) ] 'version)
(global-set-key [ (meta v) ] 'version)
(global-set-key [ (super v) ] 'version)
(global-set-key [ (meta control v) ] 'version)
```

These examples are only used to clarify the syntax of `global-set-key`, they do not serve any other purpose. The command `version` is not very significant, it just displays the version of XEmacs that you are using in the echo area. On my system it displays the message

```
XEmacs 21.4 (patch 12) "Portable Code" [Lucid] (i386-suse-linux, Mule)
```

When looking at the key bindings above you may be puzzled by the `[Super]`. This key is also known as the *Microsoft WindowsTM* key. It is not available on every keyboard. On those keyboards where it is available it is located between then `[Ctrl]` key and the `[Meta]` key. Since it is not present on all keyboards, by default no commands have been bound to this key. Therefore, this key is the ideal place for new commands.

Besides key bindings, XEmacs offers various other ways of being customized. The most important are *variables*. Many features of XEmacs are controlled via *variables*. For example, I am very annoyed by the beeping of XEmacs that occurs when I issue the `keyboard-quit` command. Fortunately, this can be changed. Issue the command `set-variable`. XEmacs will prompt you for the name of a variable:

```
Set variable: █
```

Answer this prompt with `visible-bell` and press return. Now XEmacs prompt for a value for this variable:

```
Set visible-bell to value: █
```

To this question just type “t”. In XEmacs “t” is short for “true”. (The value for “false” is “nil”.) After pressing return you can try to issue the `keyboard-quit` command. Instead of an annoying beep, the screen will flash now. We can make this change permanent by putting the line

```
(setq-default visible-bell t)
```

into our “.emacs” file. Table 9 on page 27 list the commands covered so far in this subsection.

2.10.2 Invoking Commands on Startup

Chances are that you consider the command `icomplete-mode` handy. Instead of invoking it yourself, you can have this command executed automatically by placing a line of the form

| command | key sequence | short description |
|----------------|---------------|---------------------------------------|
| eval-last-sexp | Ctrl-x Ctrl-e | evaluate expression before the cursor |
| set-variable | | set variable to a given value |

Table 9: Commands for keyboard macros.

(icomplete-mode)

at the end of your “.emacs” file. Of course, this works for all commands that should be activated on startup.

2.10.3 Writing Keyboard Macros

Let us introduce the idea of *keyboard macros* via an example. Assume you have a list of data that each consist of a name, a birth date, and an email address. The data might look as follows:

```
( 'Karl Stroetmann' '7. 4. 1961' 'stroetmann@ba-stuttgart.de' )
( 'Willi Winzig' '3. 11. 1981' 'winzig@web.de' )
( 'Daniel Dumpfbacke' '22. 5. 1945' 'Daniel.Dumpfbacke@aol.com' )
( 'Peter Hecht' '6. 8. 1967' 'phecht@freenet.de' )
```

Let us assume that for some reason we need to reformat this list: We need to put the name in the last position. For example, the first line should be reformatted to read:

```
( '7. 4. 1961' 'stroetmann@ba-stuttgart.de' 'Karl Stroetmann' )
```

Of course, if there are just four entries in this list we can easily do this reformatting by hand. But, for the arguments sake, assume that the list is much longer. Then editing it by hand would be rather boring. Therefore, let us write a macro to do the work. Now recording a macro is like writing a small program and hence is not trivial. Therefore, let us first make a plan for our macro:

1. The macro always has to start at a well-defined position. In our example, such a position is the opening parenthesis “(”. So, our first step will be to move the cursor to this position via the search-forward command.
2. Now that the cursor is positioned at the opening parenthesis, we should delete the blank following this parenthesis using the Del key.
3. Next, we kill the following two words using the kill-word command that is bound to Meta-d.
4. At this point, the cursor needs to be moved to the end of the line. In order to move it to a well-defined position we search for the closing parenthesis “)” again using the command search-forward.
5. Having the cursor positioned after the closing parenthesis, we move back one character using the command backward-char-command. This command is bound to the key combination Ctrl-b.
6. Now that the cursor is under the opening parenthesis we can insert the text just killed via the yank command bound to Ctrl-y. This inserts the name we have previously killed right before the closing parenthesis.
7. Finally, we insert one space.

| step | keys pressed | explanation |
|-------|--|-----------------------------|
| 0. | Ctrl-x (| start recording the macro |
| 1.(a) | Meta-x search-forward Return | invoke search command |
| 1.(b) | (Return | search for “(” |
| 2. | Del | delete blank |
| 3. | Meta-d Meta-d | kill name |
| 4.(a) | Meta-x search-forward Return | invoke search command |
| 4.(b) |) Return | search for “)” |
| 5. | Ctrl-b | move backward one character |
| 6. | Ctrl-y | insert killed name |
| 7. | _ | insert space |
| 8. | Ctrl-x) | finish recording the macro |

Table 10: Key strokes for recording the keyboard macro.

Now, let us record this macro. This is done as shown in Table 10. While recording this macro, we have also reformatted the first line of our data. Reformatting the rest of the lines is now easy. Just invoke the command `call-last-kbd-macro` which is bound to the key sequence `Ctrl-x e`. Every time you invoke this command, one line is changed. Cute, isn't it? But we can go further: Assume that there are 100 hundred lines that require reformatting. Then, the `universal-argument` command can speed up things considerably. Press the following keystrokes:

```
Ctrl-u 1 0 0 Ctrl-x e
```

This will invoke the macro a 100 times and thus finish our work in one sweep.

Sometimes, you write a macro that is so useful so that you want to save it. This can be done in two steps. First, we have to name the macro using the command `name-last-kbd-macro`. This command will prompt us for a name. To continue our example, let us choose the name “reformat”. Then, we visit the “.emacs” file and issue the command `insert-kbd-macro`. This command prompts for the name of the macro that is to be inserted. After choosing “reformat” the following lines are inserted into the buffer:

```
(defalias 'reformat (read-kbd-macro
  "M-x search- forward RET ( RET DEL 2*<M-d> M-x search- forward RET )
  RET C-b C-y SPC"))
```

We proceed to bind this macro to a key. This is achieved by adding the following line to our “.emacs” file:

```
(global-set-key [ (super r) ] 'reformat)
```

At this point the macro we have defined is a first class citizen of the set of commands available in XEmacs. It can now be invoked by pressing the key `Super-r`.

Occasionally, a macro needs to interact with the user. Consider the case where our database has the following form:

```
( 'Karl Stroetmann' '7. 4. 1961' 'stroetmann@ba-stuttgart.de' )
( '0711/1234567' 'Willi Winzig' '3. 11. 1981' 'winzig@web.de' )
( 'Daniel Dumpfbacke' '22. 5. 1945' 'Daniel.Dumpfbacke@aol.com' )
( 'Peter Hecht' '6. 8. 1967' 'phecht@freenet.de' )
```

Here, the second line has an additional telephone number included. Obviously, this line is to be treated differently because now the name is the second item, while our macro for

reformatting assumes that it is the first item. Here, the command `kbd-macro-query` comes to our rescue. Inserting it into our macro will query the user at the specified point. The user can then decide whether he wants to continue the macro or whether the macro should be aborted. Table 11 on page 30 shows the new sequence of keys needed to define this macro. Observe step 1.(c), which is new. When this macro is executed,

| step | keys pressed | explanation |
|-------|--|-----------------------------|
| 0. | Ctrl-x (| start recording the macro |
| 1.(a) | Meta-x search-forward Return | invoke search command |
| 1.(b) | (Return | search for "(" |
| 1.(c) | Ctrl-x q | query user |
| 2. | Del | delete blank |
| 3.(a) | Meta-d Meta-d | kill name |
| 4.(a) | Meta-x search-forward Return | invoke search command |
| 4.(b) |) Return | search for ")" |
| 5. | Ctrl-b | move backward one character |
| 6. | Ctrl-y | insert killed name |
| 7. | _ | insert space |
| 8. | Ctrl-x) | finish recording the macro |

Table 11: Key strokes for recording the 2nd keyboard macro.

it will first search for an opening parenthesis. At this point it will stop, prompting the user as follows:

```
Proceed with macro? (space, delete, q, C-l, C-r)
```

Now, the user has the same options as when doing a query replace. These options have been listed in Table 5 on page 20. Most importantly, pressing the space bar will execute the macro, pressing Del skips over one occurrence and q aborts execution of the macro.² Finally Table 12 on page 30 lists the commands that deal with macros.

2.10.4 Traps and Pitfalls When Writing Keyboard Macros

Maybe the previous paragraph sounds too good to be true. To be honest, things seldom work as smooth as described in the previous examples. Consider our first set of data to be changed again and assume now that it contains the following variation

```
( 'Karl Stroetmann' '7. 4. 1961' 'stroetmann@ba-stuttgart.de' )
( 'Willi Winzig' '3. 11. 1981' 'winzig@web.de' )
( 'Daniel Dumpfbacke' '22. 5. 1945' 'Daniel.Dumpfbacke@aol.com' )
( 'Hans Peter Hecht' '6. 8. 1967' 'phecht@freenet.de' )
```

Here, the problem is that the name of the fourth line consists of three words. However, our macro starts by killing two words. Therefore, it would change the fourth line to

²When a keyboard macro is invoked together with the `universal-argument` command, then there is a difference between aborting a macro and skipping to the next occurrence. If you have pressed `Ctrl-u 1 0 0 Ctrl-x e` to invoke an interactive keyboard macro a hundred times and press Del at the first execution of the macro, then the macro is still executed 99 times. However, if you press q, execution of the macro is aborted and there is no further invocation of the macro.

In a single invocation of a keyboard macro, i.e. in an invocation without `ctrl-u`, the keys Del and q have the same effect.

| command | key sequence | short description |
|---------------------|---------------|---|
| start-kbd-macro | Ctrl-x (| start to record a keyboard macro |
| end-kbd-macro | Ctrl-x) | finish to record a keyboard macro |
| call-last-kbd-macro | Ctrl-x e | execute the last keyboard macro that has been defined |
| name-last-kbd-macro | | associate a name with the last keyboard macro |
| insert-kbd-macro | | insert a keyboard macro into a file |
| kbd-macro-query | Ctrl-x q | suspend execution of a keyboard macro and query whether to continue |
| edit-kbd-macro | Ctrl-x Ctrl-k | edit keyboard macro |
| edmacro-finish-edit | Ctrl-c Ctrl-c | save edited keyboard macro |

Table 12: Commands for keyboard macros.

(Hecht ' '6. 8. 1967' 'phecht@freenet.de 'Hans Peter)

which is wrong. There are two ways to fix this. First, we can make the macro interactive. This approach has been described above. Secondly, we can try to generalize our macro so that it will work in the more general case also. The way to do this is that we kill all the text between the apostrophe characters “'” and later yank them at the end of the line. In order to do this, we first set a mark at the first apostrophe and then search for the second apostrophe. Once we found it, the complete text between both apostrophes is marked. The text can then be killed using the function `kill-region`. The complete macro is given in table 13.

| step | keys pressed | explanation |
|----------------|--|---|
| 0. | Ctrl-x (| start recording the macro |
| 1.(a) 1.(b) | Meta-x search-forward <input type="text" value="Return"/> (<input type="text" value="Return"/> | invoke search command search for “(” |
| 2. | <input type="text" value="Del"/> | delete blank |
| 3. | <input type="text" value="Ctrl-space"/> | set mark |
| 4. | <input type="text" value="Ctrl-f"/> | move forward one character |
| 5.(a) 5.(b) | Meta-x search-forward <input type="text" value="Return"/> ' <input type="text" value="Return"/> | invoke search command search for second apostrophe |
| 6. | Ctrl-w | kill region |
| 7.(a) 7.(b) | Meta-x search-forward <input type="text" value="Return"/>) <input type="text" value="Return"/> | invoke search command search for “)” |
| 8. | Ctrl-b | move backward one character |
| 9. | Ctrl-y | insert killed name |
| 10. | <input type="text" value="␣"/> | insert space |
| 11. | Ctrl-x) | finish recording the macro |

Table 13: Key strokes for recording the 3rd keyboard macro.

Unfortunately, as soon as we invoke the command `kill-region` the screen flashes and we are informed in the echo area that

The region is not active now

We did set the mark command with `Ctrl-space`, so the region should be active, but apparently it isn't. What happened? The answer is that the search command that was invoked later deactivated the mark. Therefore, we have to reactivate the mark. This is done with the command `exchange-point-and-mark` which is bound to `Ctrl-x Ctrl-x`. This command does two things:

1. As its name states, it exchange the position of the cursor and the position of the mark. In our case, the cursor is set back to the first apostrophe, because this is the place where the old mark was located. Further, the mark is set to the position of the second apostrophe because this is the place where the cursor has been.
2. More importantly, the mark and therefore also the region, is *reactivated* .

Using `exchange-point-and-mark`, we can write our macro as shown in Table 14. This macro solves our problem.

| step | keys pressed | explanation |
|----------------|--|---|
| 0. | <code>Ctrl-x (</code> | start recording the macro |
| 1.(a) 1.(b) | <code>Meta-x search-forward Return</code> <code>(Return</code> | invoke search command search for "(" |
| 2. | <code>Del</code> | delete blank |
| 3. | <code>Ctrl-space</code> | set mark |
| 4. | <code>Ctrl-f</code> | move forward one character |
| 5.(a) 5.(b) | <code>Meta-x search-forward Return</code> <code>' Return</code> | invoke search command search for second apostrophe |
| 6. | <code>Ctrl-x Ctrl-x</code> | reactivate the mark |
| 7. | <code>Ctrl-w</code> | kill region |
| 8.(a) 8.(b) | <code>Meta-x search-forward Return</code> <code>) Return</code> | invoke search command search for ")" |
| 9. | <code>Ctrl-b</code> | move backward one character |
| 10. | <code>Ctrl-y</code> | insert killed name |
| 11. | <code>␣</code> | insert space |
| 12. | <code>Ctrl-x)</code> | finish recording the macro |

Table 14: Key strokes for recording the 4th keyboard macro.

The above example illustrates, that getting macros right can be tricky. In practice, when a macro fails, this is often due to one of the following reasons:

1. The macro is not general enough. This was illustrated by the previous macros. There are two simple rules for making a macro more general.

- (a) Avoid making assumption about the number of characters that make up a particular field. Therefore, avoid movement of the cursor using simple cursor movement commands like

`forward-char-command` or `backward-char-command`.

Instead, use search commands to position the cursor.

In our first macro, we have already followed this advice. If we had used the cursor keys to get to the second apostrophe, then we would have moved the cursor by 16 characters to the right. This would have worked for the first line, but it would have failed on the second line because there the name has a different length.

- (b) Avoid making assumption about the number of words that make up a particular field. Again, try to use search commands.

This point has been illustrated by the previous example.

When using search commands, remember to reactivate the mark afterwards.

- The macro is intended to edit a text on a line by line basis and assumes that the cursor is positioned at the beginning of the line. In this case, after performing its work, the macro has to take care to position the cursor to the beginning of the next line so that its subsequent invocation is able to work.

2.10.5 Editing Keyboard Macros

The previous subsection shows that it is quite easy to introduce a bug into a keyboard macro. The good news is that it is not always necessary to start from scratch when a keyboard macro has been found to be buggy. We illustrate this with our first macro and show how this macro can be made interactive without rewriting it from scratch. The command that comes to our help is `edit-kbd-macro`. According to Table 12 on page 30 it is bound to the key `Ctrl-x Ctrl-k`. After pressing `Ctrl-x Ctrl-k` the minibuffer prompts us as follows:

Keyboard macro to edit (C-x e, M-x, C-h l, or keys):

To edit the recently recorded keyboard macro, we respond with `Ctrl-x e`. This displays a screen as in Figure 18 on page 33.

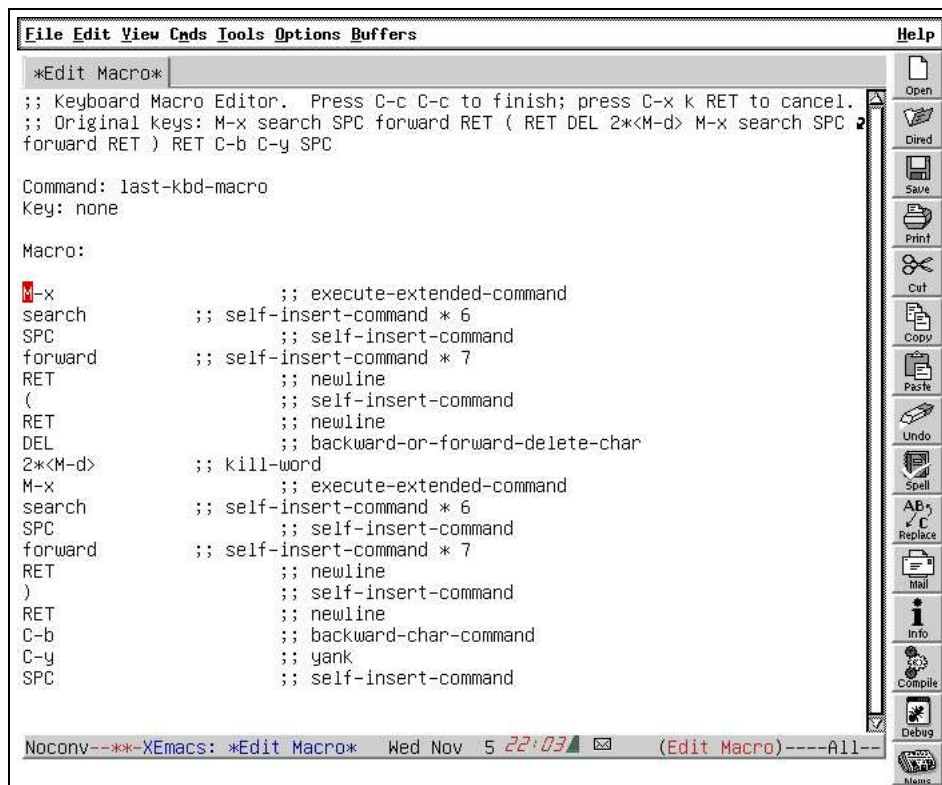


Figure 18: Editing a keyboard macro with `Ctrl-x Ctrl-k`.

Next, we enter the three lines shown below next to the `RET` statement that finishes our search for the opening parenthesis:

```

M-x
kbd-macro-query
RET

```

The screen now looks as displayed in Figure 19 on page 34. To finish recording of the

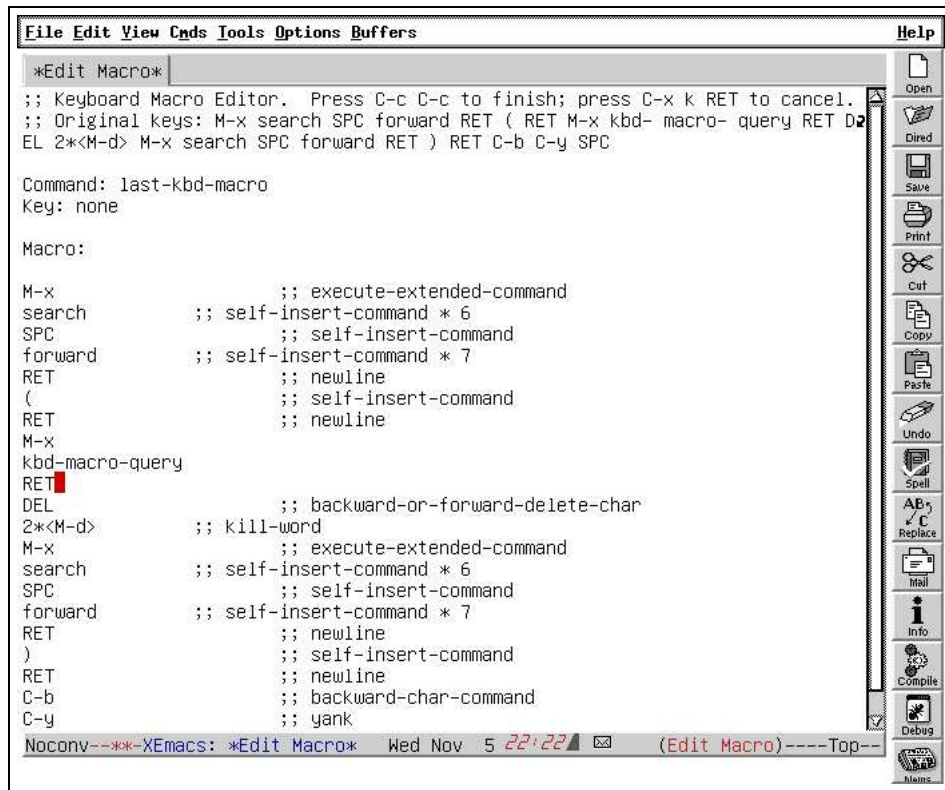


Figure 19: Macro after inserting three lines.

macro, we press `Ctrl-c Ctrl-c`. This key executes the command `edmacro-finish-edit` and saves our changes to the macro. We can now test this macro. If it works, we are done. Otherwise, we can edit the macro again until we get it right.

Keyboard macros are both powerful and complex. For the most times, these two properties go hand in hand in computer science. Therefore, it is necessary to spend some time to master keyboard macros. Given the power of keyboard macros this time is well invested.

2.11 Getting Help

Since *XEmacs* offers such a wealth of commands it is quite easy to forget either the name of a command or its key binding. Then the function “`describe-bindings`” comes to help. It describes the bindings of all keys in a separate buffer. If you are just curious which function is invoked by a particular key, the function “`describe-key`” is useful. Simply press “`Ctrl-h k`”. You are then prompted for a key. Press the key you are curious in. Next, the function invoked by this key is described in a separate buffer. For example, when pressing `Ctrl-h k Meta-%` a buffer of the form shown in Figure 20 on page 35 will appear. The text first says that the key `Meta-%` invokes the command `query-replace`. Furthermore, it gives a detailed account of the command `query-replace`. In particular, the description mentions a number of variables:

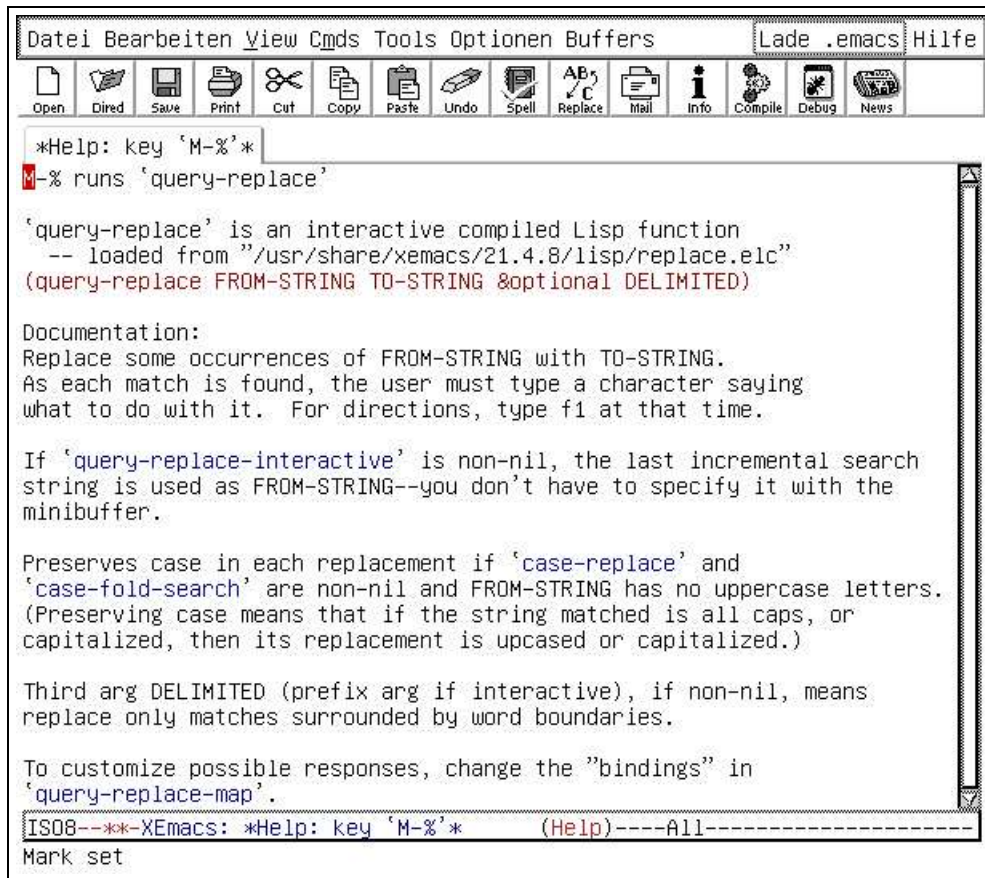


Figure 20: Effect of pressing Ctrl-h k Meta-%.

1. query-replace-interactive,
2. case-replace,
3. case-fold-search,
4. query-replace-map.

On a color printout of this script you will notice that these variables are highlighted. The command `query-replace` can be customized via these variables. If we want to get more information on any of these variables we can use the command `describe-variable`, which is bound to the key Ctrl-h v. Let us try this command with the variable `query-replace-interactive`. This will bring up the screen shown in Figure 21 on page 35. This screen informs us that if we change the value of the variable `query-replace-interactive` to "t" then the command `query-replace` will be changed. It will no longer prompt for a string to replace. Rather, it will use the last search string as the string to replace. To test it, add the following line at the end of your ".emacs" file and issue the command `eval-last-sexp` at the end of this line:

```
(setq-default query-replace-interactive t)
```

Now you can try first to do an incremental search using the `isearch-forward` command followed by a `query-replace` command. You will notice how the behavior of the `query-replace` command has changed. OK, I think this change is not very beneficial. So let us change the line setting the variable `query-replace-interactive` to "t" back as follows:

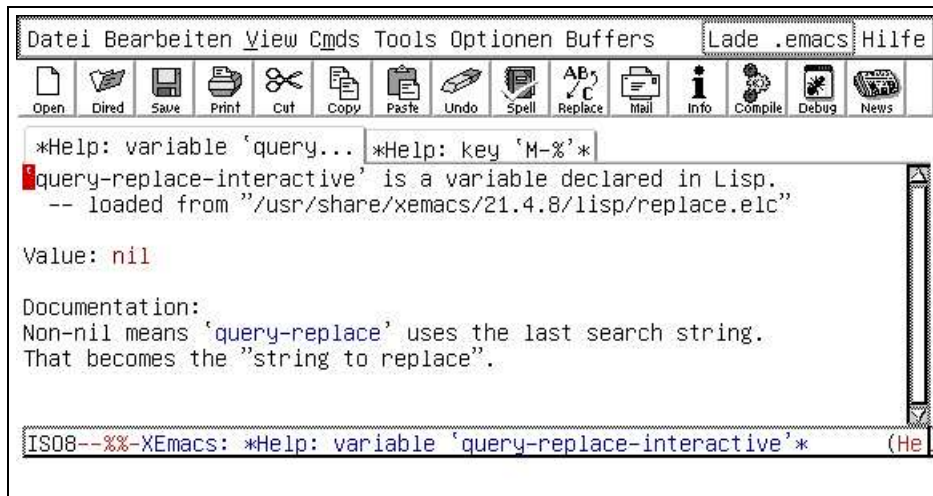


Figure 21: Effect of pressing Ctrl-h k Meta-%.

```
(setq-default query-replace-interactive nil)
```

and invoke the command `eval-last-sexp` at the end of it. Now we have reset the variable

`query-replace-interactive` to “nil” which is the default value.

Table 15 on page 36 list the various commands available for help in *XEmacs*. We

| command | key sequence | short description |
|---------------------------------|--------------|--|
| <code>describe-bindings</code> | Ctrl-h b | show a list of all defined keys and their definitions |
| <code>describe-key</code> | Ctrl-h k | display documentation of the function invoked by a particular key. |
| <code>describe-function</code> | Ctrl-h f | display the full documentation of a function |
| <code>describe-variable</code> | Ctrl-h v | display the full documentation of a variable |
| <code>apropos-command</code> | Ctrl-h A | display list of functions matching a given string |
| <code>describe-mode</code> | Ctrl-h m | describe the current mode |
| <code>where-is</code> | Ctrl-h w | print all key sequences that invoke a given command |
| <code>help-with-tutorial</code> | Ctrl-h t | present a tutorial |

Table 15: Commands for getting help.

have not yet mentioned the command `apropos-command` bound to the key Ctrl-h A. It prompts for a string and lists all commands that have this string as part of their name. Let us try this command using the string “macro”. This will bring up the screen shown in Figure 22 on page 37. This screen lists all commands that have the string “macro” as part of their name. For each command, a short description is given. We can get the full documentation via the `describe-function` command.

Then, there is the command `describe-mode`. An editor for editing a *Java* program needs to be different from an editor editing a text file. Therefore *XEmacs* has the concept of different *modes*: When you edit a *Java* program the editor will be in `java-mode`, while it will be in text mode when editing a text file. The main point about modes is that different modes have different key bindings. Furthermore, many of *XEmacs* internal variables have different values for different modes. This allows *XEmacs* to be very

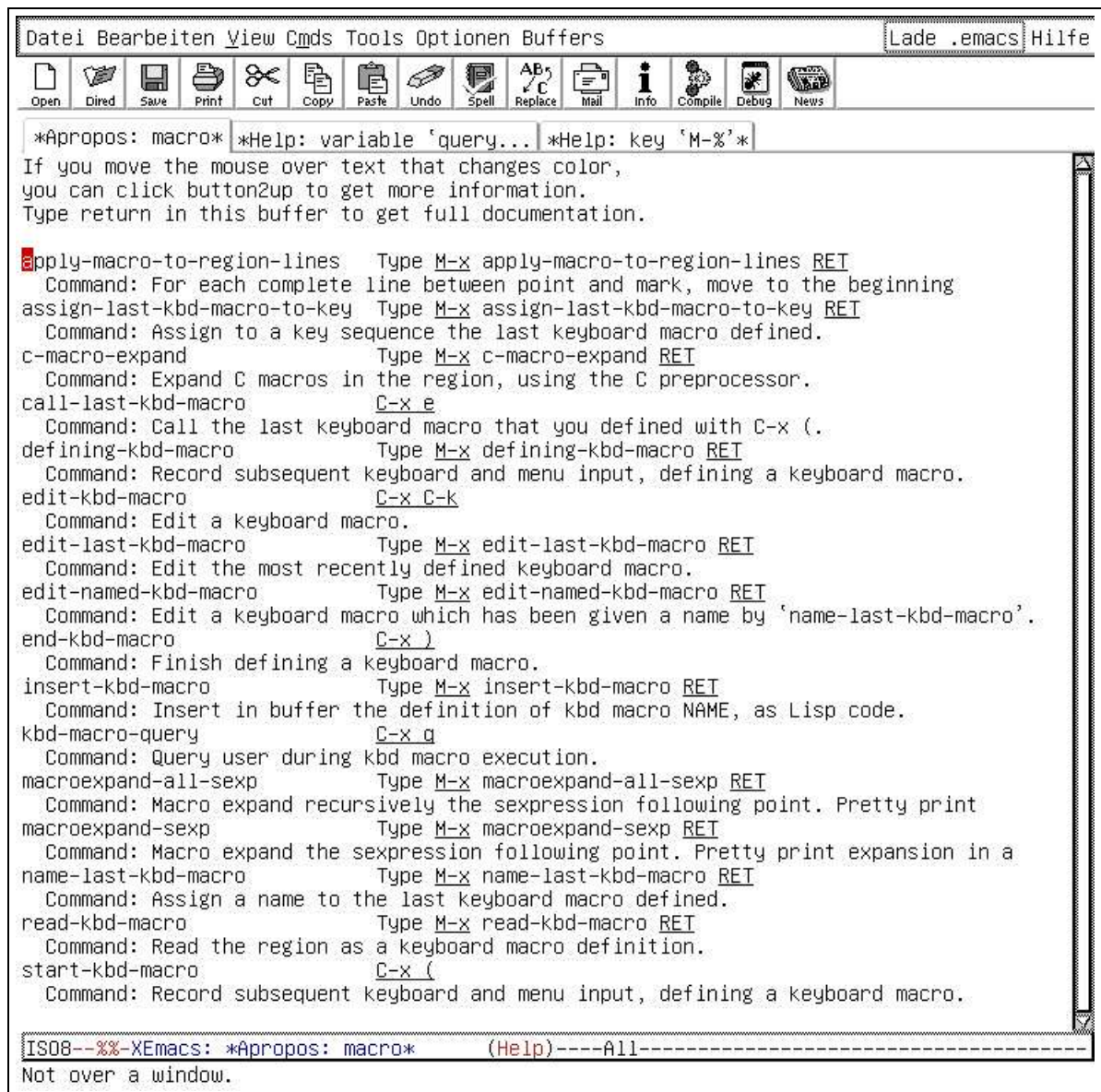


Figure 22: Running apropos-command with argument “macro”.

flexible. The command describe-mode describes the mode that is active in the current buffer.

2.12 The Info System in XEmacs

Finally, the most powerful help function is the “Info” system. To invoke it from XEmacs, issue the command info. This command is also bound to the key sequence Ctrl-h i. To get started, type the key `[h]` after invoking Info. This causes a screen of the form shown in Figure 23 on page 38 to appear. I recommend to exercise this tutorial in front of a computer. Nevertheless, the most important concepts and commands are discussed in the following.

The Info system is a system to enable efficient browsing of a certain kind of manual pages. The manual pages that can be browsed are the so called *info* pages. These

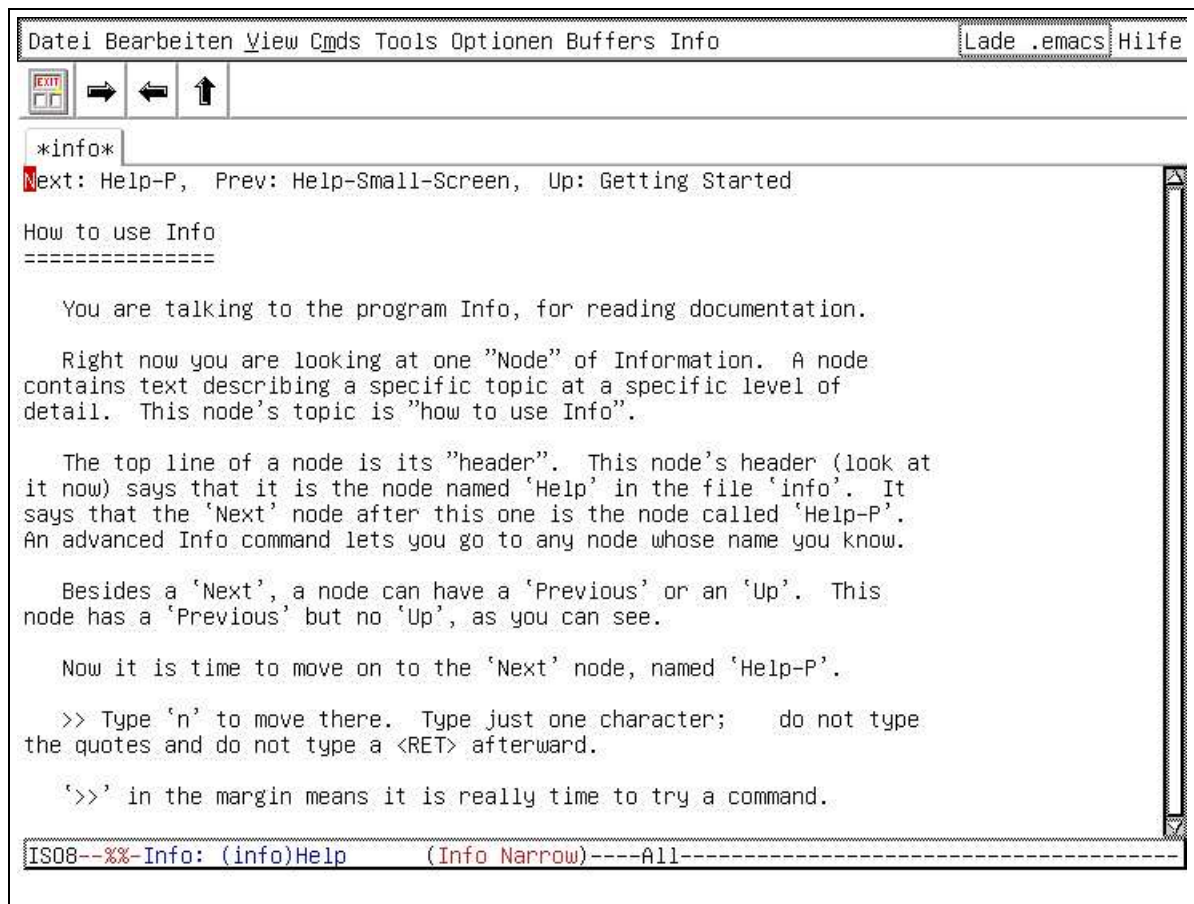


Figure 23: Invoking the *Info* tutorial.

pages have some resemblance to *Html* pages: They also contain *links*. However, to distinguish these links from *Html* links, they are called *cross references*. When viewing an *Info* page via *XEmacs*, the cross references are highlighted. Figure 24 on page 39 displays a typical *Info* page. The *Info* pages are also known as *nodes*. The first line of the node displayed contains three cross references.

1. "Next: **Buffers**" is the first cross reference. It points to a page named **Buffers**. This cross reference is marked with the label *Next*.
2. "Prev: **Fixit**" is the second cross reference. It points to a page named **Fixit**. This cross reference is marked with the label *Prev*.
3. "Up: **Top**" is the third cross reference. It points to a page named **Top**. This cross reference is marked with the label *Up*.

Next, the name of the node is given as "**File Handling**". This name is underlined with "*" characters and can be used to jump to this node directly. Then, there are two cross references in the text of the node:

1. *Note **EFS**: (efs)Top and
2. *Note **TRAMP**: (tramp)Top.

Finally, there is a menu containing cross references to various other sections.

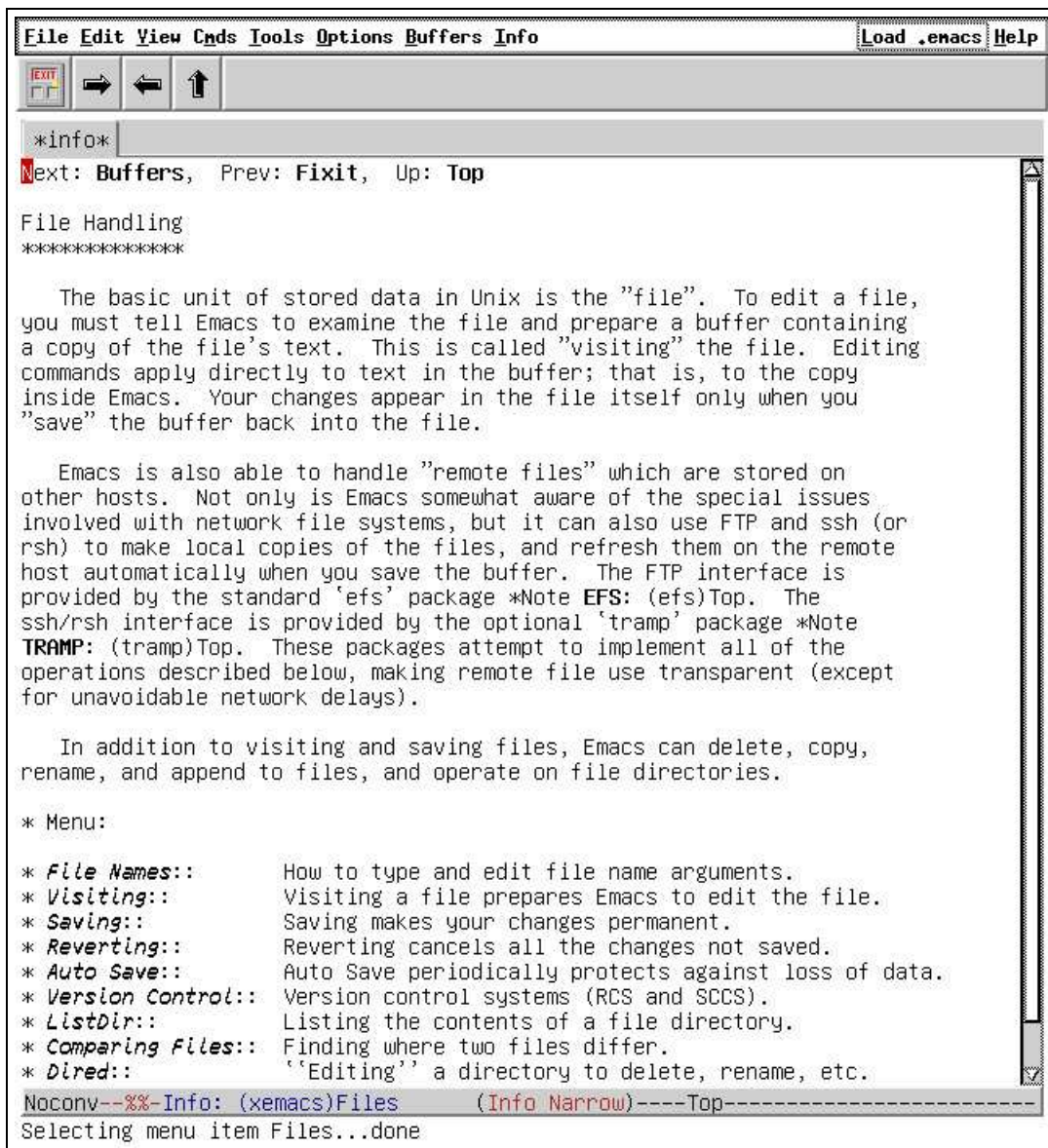


Figure 24: Viewing an info page under XEmacs.

This *Info* node shown in Figure 24 on page 39 is typical. The first line of most *Info* pages contains three cross references that are marked with the labels *Next*, *Prev*, and *Up*. These labels reflect the structure of an *Info* document: Conceptually, every *Info* document can be viewed as a book that is structured in chapters, sections, subsections, subsubsection, and so on. To make this concrete, let us assume that a book with the following table of contents is given:

- Chapter 1: Introduction to Something.
- Chapter 2: More on Something.
 - Section (a): The History of Something.
 - Section (b): The Future of Something.
- Chapter 3: Something Else.

When this book is cast into an *Info* document, it is split up into 6 nodes. The first node, also known as the *top node*, will contain a menu with an entry for each of the three chapters. There will be a node for every chapter. The node corresponding to the second chapter will have a menu that has entries for the two sections of this Chapter. Then there will be nodes for each of these sections. Let us label these 6 nodes as *top*, *chap-1*, *chap-2*, *chap-3*, *sec-1*, and *sec-2*, respectively. If somebody wants to read these nodes in the same way as he would read the corresponding book, he would read them in the sequence *top*, *chap-1*, *chap-2*, *sec-1*, and *sec-2*, *chap-3*. In order to read an *Info* document as a book, *XEmacs* makes the following commands available.

1. `Info-scroll-next` scrolls the screen of the currently displayed node down. If the screen already displays the end of the node, then the command `Info-scroll-next` switches to the following node. For example, if the screen displays the end of node *chap-2*, then the node *sec-1* is displayed next.

The command `Info-scroll-next` is bound to the key `space`.

2. `Info-scroll-prev` is the opposite of `Info-scroll-next`. It scrolls the screen back. If the screen already displays the beginning of the node, then the command `Info-scroll-prev` switches to the preceding node. For example, if the screen displays the top of node *sec-1*, then the node *chap-2* is displayed next.

The command `Info-scroll-prev` is bound to the key `backspace`.

Instead of reading a book cover to cover you might want to skim through it and just look at the beginning of each chapter. Of course, this is also possible with an *Info* document. The *Info* system offers several commands that deal with this approach:

1. `Info-next` moves to the next node that is on the same level as the node you are currently visiting. For example, when visiting node *chap-2*, the next node on the same level is *chap-3*, while when visiting node *sec-1* the next node on the same level is *sec-2*. The name of this node is also shown on the first line of the current node. It is marked with the label "Next:". In Figure 24 on page 39 this name is "**Buffers**".

The command `Info-next` is bound to the key `n`.

2. `Info-prev` moves to the previous node that is on the same level as the node you are currently visiting. For example, when visiting node *chap-3*, the previous node on the same level is *chap-2*, while when visiting node *sec-2* the previous node on the same level is *sec-1*. The name of this node is also shown on the first line of the current node. It is marked with the label "Prev:". In Figure 24 on page 39 this name is "**Fixit**".

The command `Info-prev` is bound to the key `p`.

3. `Info-up` moves to the node that is one level above the node you are currently visiting. For example, when visiting node *sec-2*, the node one level above is the node *chap-2*, while when visiting node *chap-2* the node one level above is the node *top*. The name of this node is also shown on the first line of the current node. It is marked with the label "Up:". In Figure 24 on page 39 this name is "**Top**".

The command `Info-up` is bound to the key `u`.

4. `Info-top-node` moves to the topmost node in the hierarchy. In the example, this is the node *top*.

The command `Info-top` is bound to the key `t`.

5. `Info-directory` moves to a node showing a menu that has entries for all *Info* documents installed on the system.

The command `Info-directory` is bound to the key `d`.

The *Info* system offers some more commands that are useful but that did not fit above. These are listed below.

1. `Info-follow-nearest-node` is the function that can be used to visit a cross reference. Conceptually, invoking this function is the same as clicking the mouse on a hyperlink in an HTML document. To use this command, move the cursor to a highlighted cross reference and invoke the command by pressing return.

The command `Info-follow-nearest-node` is bound to the `return` key.

2. `Info-last` moves you back to the last node you have visited. It is useful when you have followed a cross reference and you want to go back to the place where you came from. If you have followed a chain of several cross references, invoking `Info-last` repeatedly takes you back to the start of your journey. Conceptually, it is similar to the *back* button of a web browser.

The command `Info-last` is bound to the key `l`.

3. `Info-search` searches the whole *Info* document for a string you specify. It is important to understand that the search is not confined to the current node but rather extends to all the nodes that make up the document.

The command `Info-search` is bound to the key `s`.

4. `Info-index` searches the *index* of an *Info* document for a string you specify. In an *Info* document, certain nodes can be specified as *index nodes*. The search invoked by this function is confined to these nodes.

The command `Info-index` is bound to the key `i`.

5. `Info-menu`

When a node contains a menu, there is a shortcut to visit the nodes associated with the menu entries. The function `Info-menu` asks for the name of a menu entry and then visits the corresponding node. When entering the name of the node, the completion facility of the minibuffer is available, i.e. pressing the space bar completes the name as far as possible.

The command `Info-menu` is bound to the key `m`.

6. `Info-follow-reference` is similar to `Info-menu`, but asks for the name of a cross reference instead of a menu entry. When entering the name, the completion facility of the minibuffer is available.

The command `Info-follow-reference` is bound to the key `f`.

7. `Info-next-reference` moves the cursor to the next cross reference. When the last cross reference is reached, the command `Info-next-reference` moves the cursor back to the first cross reference in the node. This way, you can easily scroll to all cross references in a node.

The command `Info-next-reference` is bound to the `tab` key.

8. `Info-prev-reference` is the opposite of `Info-next-reference`. It is bound to the key `Meta-tab`. However, depending on your window manager, this key might not be available for *XEmacs*. In this case you should rebind the command `Info-prev-reference` to a different key. A convenient key is `Ctrl-tab`.

9. `Info-bookmark` allows to set a bookmark. The command asks for a name which can later be used to reference the bookmark. The bookmarks set are saved to the file `~/xemacs/info.notes`. Therefore, bookmarks are persistent: if you terminate your *XEmacs* session and later start *XEmacs* again, the bookmarks that you have set in the first session will still be available.

Bookmarks are not confined to a single word, you can enter more or less arbitrary text. However, this text may not contain newlines. This way, bookmarks can be used to annotate an *Info* document. If there really is a need to write multi-line annotations, annotate the same place with several bookmarks.

The command `Info-bookmark` is bound to the key `[x]`.

10. `Info-goto-bookmark` asks for the name of a bookmark and then moves to the specified bookmark. When the name of the bookmark is entered, the completion facility of the minibuffer can be used.

The command `Info-goto-bookmark` is bound to the key `[j]`.

11. `Info-visit-file` asks for the name of a file containing an *Info* document and loads the nodes associated with the specified document.

The command `Info-visit-file` is bound to the key `[v]`.

12. `Info-quit` exits the *Info* system.

Table 16 on page 42 summarizes the commands available when using the *Info* system.

The *info* system offers help on various topics, not only on *XEmacs*.

2.13 Miscellaneous Commands

Finally, there are some commands that did not quite fit into any of the other sections but that are nevertheless quite useful. These are listed in Table 17 on page 43. Let us discuss these commands one by one. I frequently make typos of the form “`taht`” instead of “`that`”, i.e. I exchange the order of letters. In the example, I have exchanged the letters “`a`” and “`h`”. If I realize this error after typing the “`h`” I can then press `Ctrl-t`, which invokes the command `transpose-chars` and the typo is corrected. There is a similar command for transposing word: It is called `transpose-words` and is bound to the key `Meta-t`. As an aside, note that the following is a principle in *XEmacs*: If there is a command that is used for characters and that is bound to `Ctrl-char`, where *char* stands for some character, and if, furthermore, there is a similar command that works on words instead of characters, then the command working on words is bound to the key `Meta-char`. Table 2 on page 13 shows many examples of this principle.

Invoking the command `display-time` displays the current date and time in the mode line. Similarly, the command `line-number-mode` displays the number of the line of the cursor in the mode line.

If you want to format a text paragraph, this can be done using the command `fill-paragraph`. However, this command needs to be given a prefix argument via `Ctrl-u`. It adjusts the current paragraph. The length of the line is controlled by the variable `fill-column`. Let us use this command to add a finishing touch to our short story. The result will then look as shown in Figure 25 on page 43.

Sometimes, you have to add control characters into a buffer. For example, you want to add `Ctrl-c`. This character is displayed as “`^c`”. But how do we enter this character? The answer is the command `quoted-insert`, which is bound to the key `Ctrl-q`. Pressing `Ctrl-q` will enter the next key that is pressed next literally into the buffer.

Finally, the command `just-one-space` swallows redundant spaces. It is bound to the key `Meta-␣`.

| command | key sequence | short description |
|--------------------------|------------------------|--|
| info | Ctrl-h i | start the <i>Info</i> system |
| Info-follow-nearest-node | <code>return</code> | follow a node reference near point |
| Info-scroll-next | <code>space</code> | scroll screen down visit following node if at end of page |
| Info-scroll-prev | <code>backspace</code> | scroll screen up visit preceding node if at end of page |
| Info-next | <code>n</code> | visit next node |
| Info-prev | <code>p</code> | visit previous |
| Info-up | <code>u</code> | visit the superior node |
| Info-top-node | <code>t</code> | move to top node of current manual |
| Info-directory | <code>d</code> | move to <i>Info</i> directory |
| Info-last | <code>l</code> | go back to node last visited visit the |
| Info-search | <code>s</code> | search for a string |
| Info-index | <code>i</code> | search the index |
| beginning-of-buffer | <code>b</code> | move to beginning of buffer |
| Info-next-reference | <code>tab</code> | move cursor to next cross reference |
| Info-prev-reference | <code>Meta-tab</code> | move cursor to previous cross reference |
| Info-menu | <code>m</code> | ask for name of menu entry move to given entry |
| Info-follow-reference | <code>f</code> | ask for name of cross reference move to cross reference |
| Info-bookmark | <code>x</code> | set a bookmark |
| Info-visit-file | <code>v</code> | load an <i>Info</i> document |
| Info-quit | <code>q</code> | quit the <i>Info</i> system |

Table 16: The *Info* System.

2.14 What Else is There

Plenty. Really. The manual describing Emacs has over 600 pages. And then there is the manual covering the programming language *Emacs Lisp*. This language is needed if you want to create serious extensions of *XEmacs*. This manual has about 900 pages. Of course, these manuals are written in a very condensed style, so they cover much more material per page than I have covered in the few pages that I have dedicated to *XEmacs*. Furthermore, there are hundreds of extensions of *XEmacs* that are described in separate manuals. Some of these extensions are described in a few pages, others require more than 100 pages. This should give you a rough estimate about the wealth of features available in *XEmacs*. I can not resist to showing of some of the more cool stuff that there is. Figure 26 on page 44 shows that *XEmacs* can edit Japanese text.

While in *XEmacs*, some games are available. You should definitely try the command

| command | key sequence | short description |
|------------------|--------------|---|
| transpose-chars | Ctrl-t | interchange characters around the cursor |
| transpose-words | Meta-t | interchange words around the cursor |
| display-time | | display the current time and date in the mode line |
| line-number-mode | | show the line number of the cursor in the mode line |
| fill-paragraph | | justify paragraph |
| quoted-insert | Ctrl-q | read next character and insert it verbatim |
| just-one-space | Meta-_ | remove redundant spaces |

Table 17: Various useful commands.

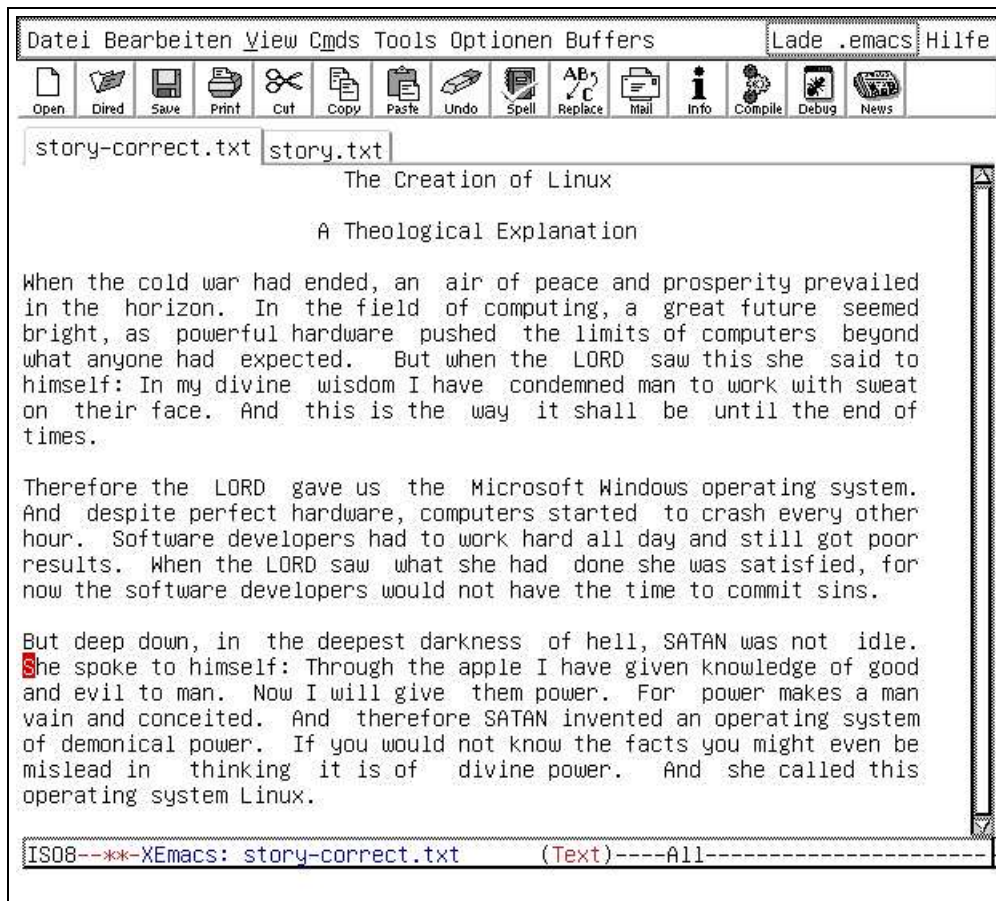


Figure 25: Short story after formatting with fill-paragraph.

hanoi showing how to solve the famous “*towers of Hanoi*” puzzle. By using Ctrl-u to provide a prefix argument you can run it for an arbitrary number of disks. Did I mention tetris? This game is shown in Figure 27 on page 45. Finally, when you think that all this is more than you can take, try the doctor command.

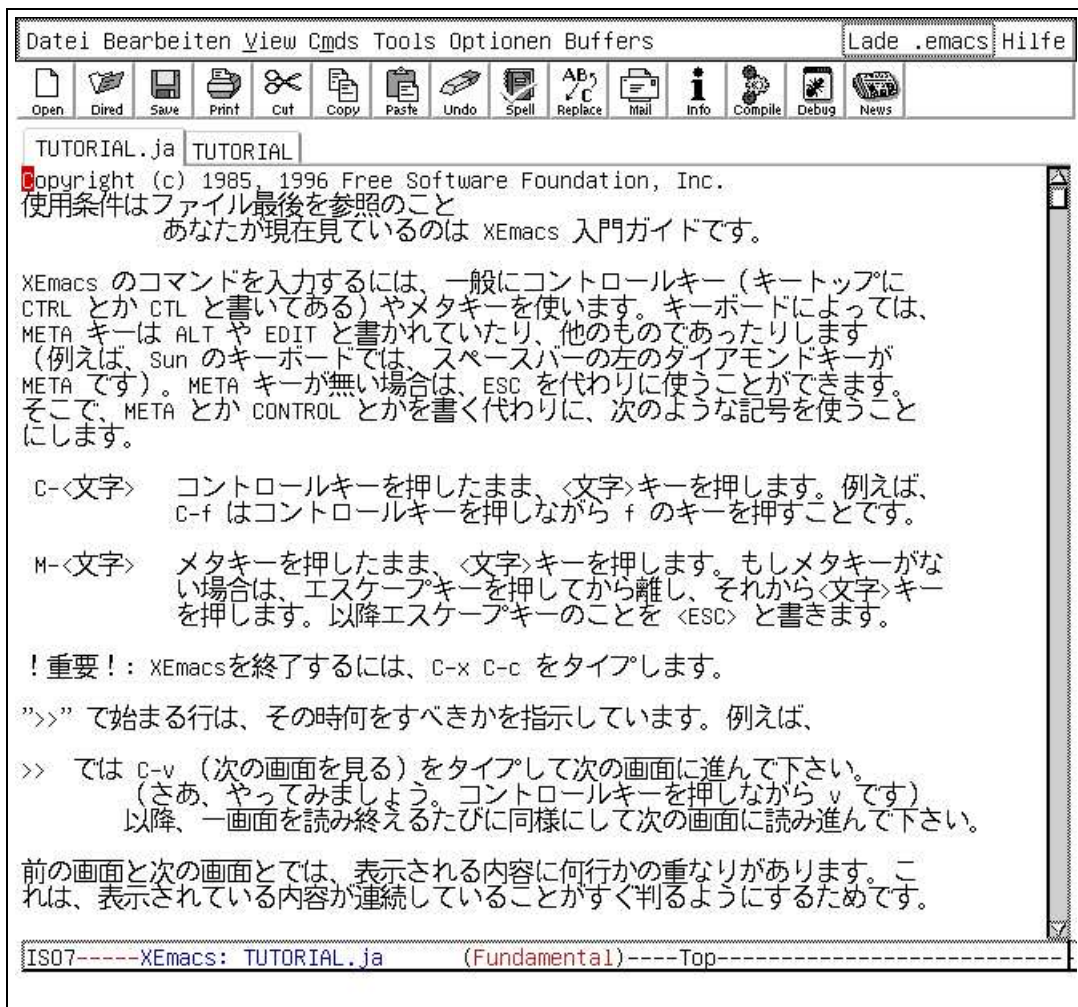


Figure 26: Editing Japanese text with XEmacs.

3 The File System

It is time to discuss the file system of *Linux*. To begin with, what precisely is a *file system*? This question is best answered via an analogy. Consider an office of a company. This office needs to store various contracts, letters, the addresses of customers, etc. The storage of these items needs to be organized for otherwise it would be impossible to find something. All contracts with a given customer may be collected in a folder. All folders with contracts may be stored in a cabinet, while another cabinet might holds the folders containing the employment contracts.

With a computer, things are somewhat similar. There are lots of data around. On a first level, this data is organized in files. In turn, files are organized in directories and these directories may themselves be part of their parent directories. The analogy with the office can be roughly sketched as follows:

| | | |
|--------------------------------|---|--------------|
| letter, contract, address list | ≈ | file |
| folder | ≈ | subdirectory |
| cabinet | ≈ | directory |

In *Linux*, directories can be nested arbitrarily. That is, you can create a directory and in this directory you can create a number of files and subdirectories. Then in each of

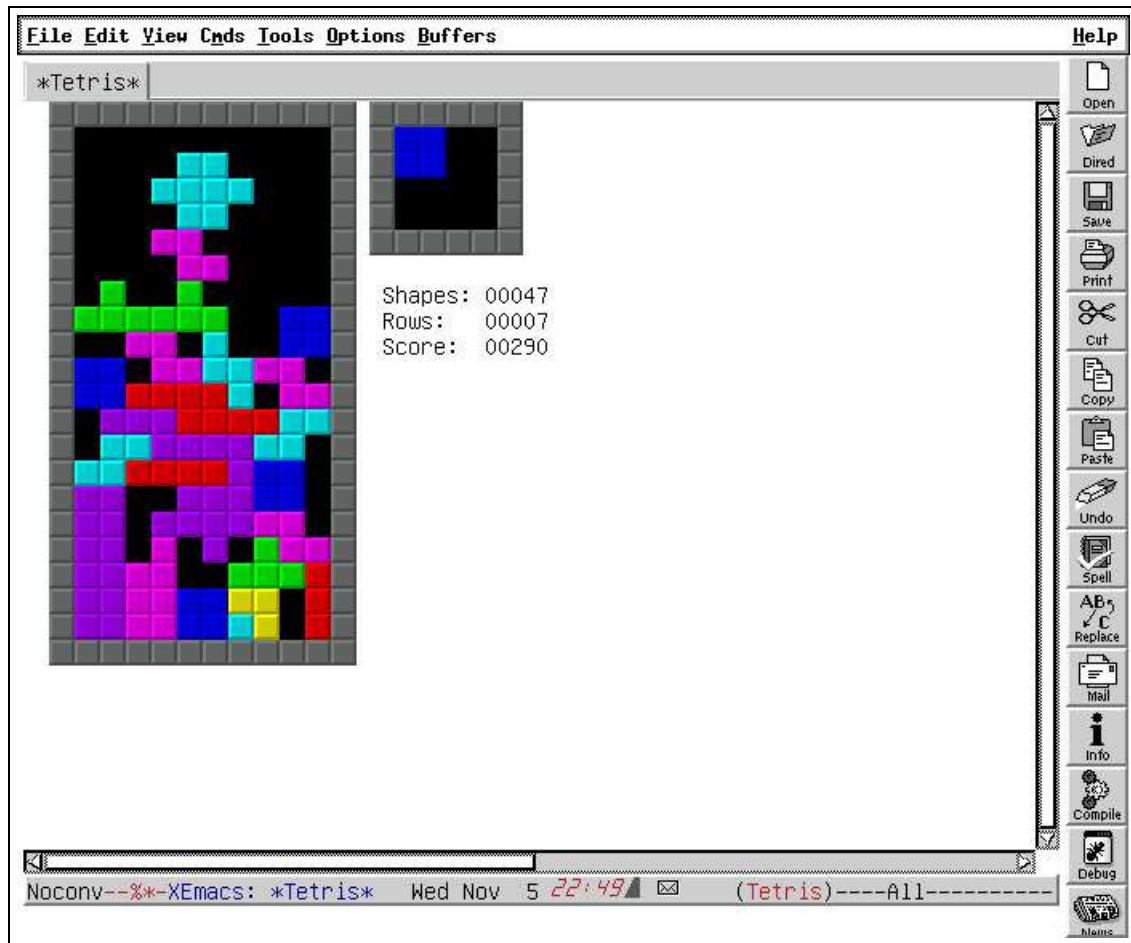


Figure 27: Tetris

the subdirectories you can again create files and subdirectories. And so on. Therefore, you can structure your data in any way that you consider convenient. Let us take a look at a typical file system as it is shown in Figure 28 on page 46. First, there is the so called *root directory*. This directory has the name “/”. This is the only directory that has no *parent directory*. Here a directory *A* is the *parent directory* of a directory *B* if *B* is a subdirectory of *A*. In our example, the root directory has the following subdirectories:

```
bin, dev, etc, home, lib, tmp, usr, var
```

The directory *home* is somewhat special. It contains the *home directories* of the users. In our example, there are two users: “anja” and “bernd”. Normally, the user “anja” will own all the files that are in the directory “anja” or in any of its subdirectories. Similar for the user “bernd”. When the user “anja” first logs in, she will start her session in her home directory. In our example, the directory “anja” has two subdirectories: *Prog* and *Texte*. The directory *Texte* contains two files: *Kap1.tex* and *Kap2.tex*.

Every file can be identified by its *path name*. This name consists of a list of all directories leading to the file, separated by the “/” symbol. For example, the path name of the file *Kap2.tex* is

```
/home/anja/Texte/Kap2.tex
```

A path name that starts with a “/” is called an *absolute path name*. An absolute path name always starts from the root directory “/”. This is in contrast to a *relative path*

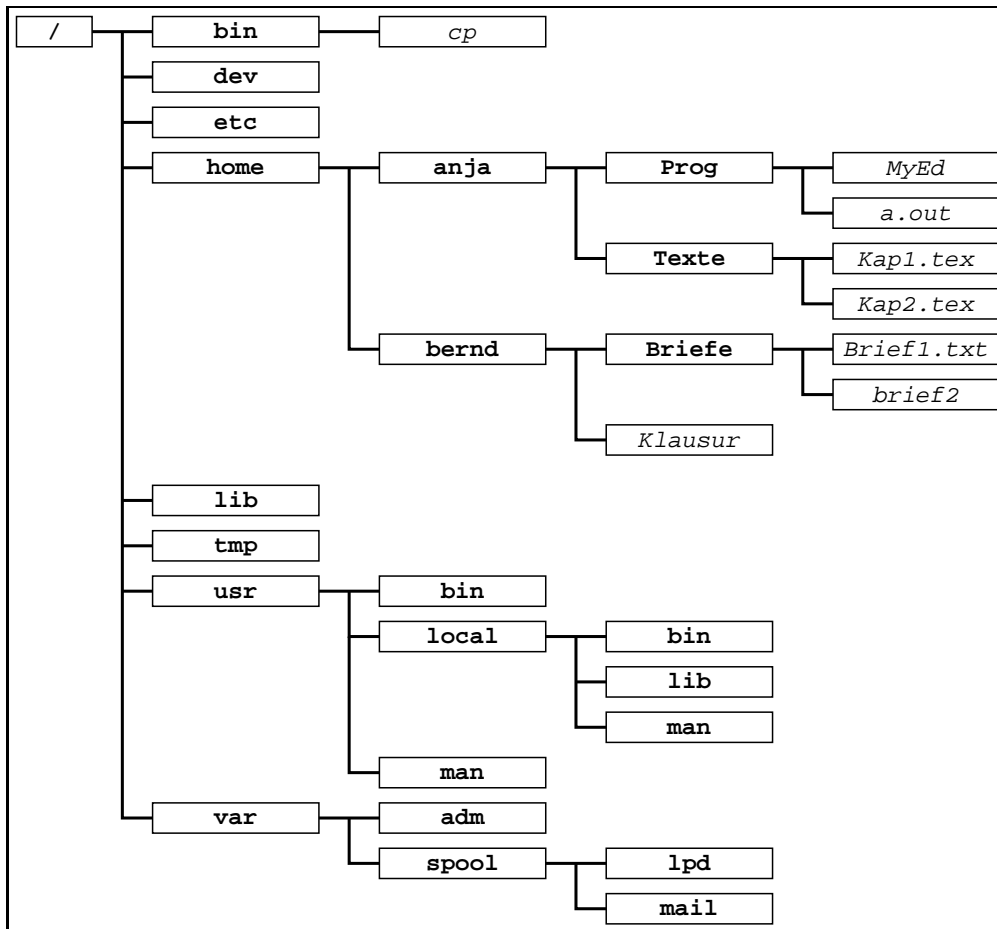


Figure 28: A typical File System.

name. To give an example, if the user “anja” is working in her home directory, then the relative path name of the file `Kap2.tex` would be

`Texte/Kap2.tex`

As the name suggests, a *relative path name* is always relative to the current *working directory*. Therefore, if the current working directory changes, the relative path name to a given file changes, too. For example, if the user “bernd” is working in his home directory, the relative path to the file `Kap2.tex` will be

`../anja/Texte/Kap2.tex`

This example introduces the notation “`..`”, which is short for the parent directory of the working directory. Therefore, if the working directory is `/home/bernd`, then “`..`” denotes the directory “`/home`”. This notation can be repeated, e.g. if the working directory is “`/home/bernd/Briefe`”, then the file “`Kap2.tex`” is denoted by the relative path name “`../../anja/Texte/Kap2.tex`”.

When writing directory names, there are three more shorthands available:

1. “`~`” is short for your home directory. In the example above, if “anja” is working in the directory “`Prog`” and she wants to refer to the file “`Kap2.tex`”, then the relative path name

`~/Texte/Kap2.tex`

would do the job.

2. “`~user`” is short for the home directory of *user*. For example, if “bernd” wants to refer to the file “`Kap2.tex`”, he could do so via the relative path name

```
~anja/Texte/Kap2.tex
```

3. “`.`” is short for the current working directory. For example, if “anja” is working in the directory `Texte`, then both “`./Kap2.tex`” and “`Kap2.tex`” would refer to the same file.

Above, we have repeatedly referred to the so called *working directory*. Basically, this is the directory where you are currently working. When you log in, the working directory is your home directory. You can change the working directory using the command “`cd`” that is described later. All the commands that you invoke will be relative to your working directory. That is, if you execute a command that reads a file and you specify just the file name but no directory information, then most commands will search for the specified file in the current working directory. Similarly, if a command creates a file and you do not specify a directory where the newly created file should be written to, then most commands will create this file in the current working directory. Sometimes, files are also created in the special directory `/tmp`. This directory stores temporary files that are only needed during a computation and can be discarded afterwards.

It is time to discuss some of the basic commands to copy, move, and remove files and directories. We give a short overview now and will discuss some of the more complex commands in greater detail later in separate subsections.

1. `mkdir dir`

creates a subdirectory with the name *dir* in the current working directory. For example, the command

```
mkdir Test
```

would create a subdirectory with the name “`Test`” in the current working directory.

2. `cp src dst`

copies the file *src* into the file *dst*. If a file with this name existed before, it is overwritten. For example, the command

```
cp Kap2.tex Kap3.tex
```

copies “`Kap2.tex`” into a new file with the name “`Kap3.tex`”

3. `ls`

lists the files and subdirectories of the current working directory.

4. `mv src dst`

renames the file “`src`” to “`dst`”. For example, the command

```
mv Kap2.tex Kap2a.tex
```

renames the file “`Kap2.tex`” to “`Kap2a.tex`”.

5. `rm file`

removes “*file*”. For example, the command

```
rm Kap2.tex
```

would remove the file “Kap2a.tex”.

Attention: In contrast to other operating systems, the `remove` command will remove the specified file immediately. It is **not** moved to some form of waste basket from where it might be retrieved.

6. `rmdir dir`

removes the directory *dir*. However, for `rmdir` to succeed, this directory must be empty! If you want to remove non-empty directories, you can do this via the “`rm`” command and its “`-r`” flag, which removes directories recursively.

7. `pwd`

reports the name of the current directory. For example, if the user “anja” issues this command right after logging in, this command would report

```
/home/anja
```

8. `cd dir`

changes the current working directory to become *dir*. For example, the command

```
cd /home/bernd
```

would change the current working directory to become “/home/bernd”

9. `touch file`

will create an empty file of the given name. For example, the command

```
touch Kap42.tex
```

creates an empty file with the name “Kap42.tex”.

10. `cat file`

dumps the content of the file *file* to the screen. For example, assume that the file “greeting” contains the text “Hello, World!”. Then issuing the command

```
cat greeting
```

would write the string “Hello, World” to the screen.

Most of the commands given above take additional arguments known as *options*. Using these options the behaviour of these commands can be fine tuned. We discuss a subset of these commands in the following subsections.

3.1 Copying Files

Basically, there are three ways to invoke the `cp` command.

1. `cp src dst`

where *src* is the name of a file and *dst* is the name of a new file. This command will copy *src* into the file *dst*. If *dst* already exists, it is overwritten without warning.

2. `cp src1 ... srcn dst`

where `src1` to `srcn` are files or directories while `dst` is the name of an existing directory. This command will copy the given files and directories into the specified directory. If this directory already contains files with this name, they are overwritten.

The command `cp` supports a number of options. We list the most important ones.

1. `cp -i src dst`

will ask before overwriting existing files.

2. `cp -r src dst`

where `src` and `dst` are directories will copy the directory `src` *recursively* into the destination directory `dst`. Here, copying *recursively* means that any subdirectories of `src` will also be copied.

3.2 Deleting Files and Directories

The command `rm` is more powerful than discussed in the introduction of this section. It can remove directories and can even do this recursively using the “-r” flag. For example, if “Test” is a directory, possibly containing some non-empty subdirectories, then the command invocation

```
rm -r Test
```

would remove “Test” and all of its subdirectories and the files contained therein.

Normally, the `rm` command will not ask for confirmation before the removal of any file. This can be changed via the “-i” flag which makes the `rm` command interactive. This flag is often used in conjunction with the “-r” flag. For example, the command

```
rm -r -i Test
```

will ask for every file and subdirectory of the directory “Test” whether it should be removed.

3.3 Listing Files and Directories

The command “`ls`” lists the contents of a directory. We view a directory as a list of *directory entries*. Each of these entries is either a file, a subdirectory, or a *link*. (The notion of a link is discussed in a separate subsection below.) The command “`ls`” has a number of useful options that are discussed in the following:

1. `ls -a`

lists the *hidden* files and directories, too. A file or directory is *hidden* if it starts with a “.”. For example, the file “.emacs” is a hidden file.

2. `ls -l`

returns a long form of the directory listing with considerably more information. For example, the following is a possible output produced by “`ls -l`”:

```
total 11
drwxr-xr-x  4 karl users  6272 2003-01-17 14:18 GRT
drwxr-xr-x  5 karl users  1512 2003-01-16 12:32 Informatik-I
drwxr-xr-x  3 karl users  3144 2003-01-17 14:23 PDV
-rw-r--r--  1 karl users   976 2002-12-10 21:15 pipeline.tex
```

The general form of each line is as follows

type mode linkno owner group size date time name

Here, the meaning of each of these items is as follows.

- (a) *type mode* specifies the *type* of each file. There are three types:
 - “-” Normal files are specified by the character “-”. In the example above, “pipeline.tex” is a normal file.
 - “d” Directories are specified by the character “d”. In the example above, “GRT” is a directory.
 - “l” Soft links are specified by the character “l”. We discuss *links* in a separate subsection.
 - “b” *Block devices* are specified by the character “b”. Usually, they refer to hardware that is read or written a block of character at a time. For example, a hard disk is associated with a block device.
 - “c” *Character devices* are specified by the character “c”. Usually, they refer to hardware that is read or written one byte at a time. For example, a keyboard is read a character at a time.
 - “p” *Named pipes* (a.k.a. *fifo*s) are used for interprocess communication. We discuss them in a separate subsection.
 - “s” *Sockets* are a facility that is needed when programs communicate over a network.
- (b) *mode* specifies the *mode* of each file. The mode of a directory entry codes certain *access rights*. This is discussed in the next subsection.
- (c) *linkno* is the number of (*hard*) links to this file. For directories, the link counter is always at least 2 since every directory has the entry “.” which is a link to itself. If a directory has subdirectories, then the link counter is increased by one for every subdirectory, since every subdirectory has the entry “..” which is a link pointing to its parent directory. Therefore, for directories *linkno* usually gives the number of subdirectories plus 2.
- (d) *owner* is the owner of the file. In the example above, this owner is always the user karl.
- (e) *group* is the group associated with the file. In the example above, all files are associated with the group users.
- (f) *size* is the size of the file in bytes.
- (g) *date* is the date of the last modification of the entry.
- (h) *time* is the time of the last modification of the entry.
- (i) *name* is the name of the entry.

3. `ls -l d1 ... dn`

lists exclusively the given files and directories. This feature is often used in conjunction with *wildcards*. These are discussed in a separate subsection.

4. `ls -R`

lists subdirectories recursively.

5. `-S`

lists entries sorted according to their file size. The default is to list files alphabetically.

6. -t

lists entries sorted according to their modification time.

There are lots of other options available that can be used to control the data that is displayed. These options are described in the *info pages* that document the “ls” command.

3.4 Maintaining Access and Ownership

Since Linux is a multiuser system it needs concepts that allow to control the access to files. With Linux, there are three levels of access.

1. The level of the *owner*.
2. The level of the *group*.
3. The level of the rest of the world.

While the concept of the owner of a file is clear, the concept of a group needs explanation. Consider a software project involving several developers. These developers will work jointly on a number of files. Therefore, these developers will make up a group and the files that they use jointly will have access rights that enable the members of the group to work jointly on these files while still preventing others from viewing or modifying these files.

Therefore, the *access mode* of a directory entry has the following general format:

op gp wp

Here, *op*, *gp*, and *wp* have the following meaning:

1. *op* is the access permission of the owner.
2. *gp* is the access permission of the group.
3. *wp* is the access permission of the rest of the world.

In Linux, there are three different ways to access a file.

1. A file can be read.
2. A file can be written.
3. A file can be executed.

Therefore, a *permission* has the following form:

rd-pr wr-pr ex-pr

Here, *rd-pr*, *wr-pr*, and *ex-pr* have the following meaning.

1. *rd-pr* controls the *read permission*. If it is set, it has the value “r”, otherwise it has the value “-”.
2. *wr-pr* controls the *write permission*. If it is set, it has the value “w”, otherwise it has the value “-”.
3. *ex-pr* controls the *execute permission*. If it is set, it has the value “x”, otherwise it has the value “-”.

For a directory, execute permission is needed to list the contents of the directory.

Using this information, we can make more sense from the directory listing produced by “ls -l”. Consider the following line:

```
-rwxr-x--x    1 karl users    976 2002-12-10 21:15 w3fetch
```

In this case, the permission of the file “w3fetch” has the value “rwxr-x--x”. To understand it, we have to split it into the three parts *op*, *gp*, and *wp* describing the permissions of owner, group, and world. We have

op = “rwx”, *gp* = “r-x”, *wp* = “--x”

We discuss these permissions in turn.

1. *op* = “rwx”.

This shows that the user has the right to read, write, and execute the file “w3fetch”.

2. *gp* = “r-x”

Since the string “r-x” does not contain the letter “w”, the members of the group *users* have the right to read and execute the file “w3fetch”, but they are not permitted to change the contents of the file.

3. *wp* = “--x”

Since the string “--x” contains only the letter “x”, the rest of the world has only the permission to execute the file “w3fetch”, but may neither read nor change the file.

3.4.1 Changing Permissions

Having understood the basic theory of permissions we next need to know how to manage these permissions. This is done with the command “chmod”. Let us start with an example:

```
chmod a+r w3fetch
```

Successful execution of this command would grant read permission (this is what the “r” stands for) to all users (this is what the “a” stands for). The general format is

```
chmod category access-or-denial permissions file-or-dir
```

Here, *category* can be one or more of the following characters:

1. “u” specifies that the permissions of the *owner* are to be changed.
2. “g” specifies that the permissions of the *group* are to be changed.
3. “o” specifies that the permissions of the *rest of the world* are to be changed. The letter “o” is short for *others*.
4. “a” specifies that the permissions of the *owner*, the *group*, and the *rest of the world* are to be changed. Here, the letter “a” is short for “*all*”.

Next, *access-or-denial* can be one of the following characters:

1. “+” grants permissions.
2. “-” revokes permissions.
3. “=” sets permissions.

This last item needs further explanation. Consider the command

```
chmod o=r w3fetch
```

Successful execution of this command would set the permission of the rest of the world to “r--” for the file “w3fetch”. That is, it does not only grant read permission to the rest of the world, it also retracts execute and write permission in the case that these had been set before.

Then, *permissions* can be any collection of the following letters:

1. “r” specifies read permission.
2. “w” specifies write permission.
3. “x” specifies execute permission.

Finally, *file-or-dir* specifies the file or directory for which the permissions should be changed. Note that in the invocation

```
chmod category access-or-denial permissions file-or-dir
```

there may be no spaces between *category*, *access-or-denial*, and *permissions*. That is, you can not write something like “chmod o - r w3fetch”. You have to write “chmod o-r w3fetch” instead. (Above, we added a little space between these words just to make things readable.) To get used to “chmod”, let us consider some examples.

1. `chmod g-r w3fetch`

This would retract the read permission for the group for the file “w3fetch”.

2. `chmod o+rx w3fetch`

This would grant both read and execute permission to the rest of the world.

3. `chmod ug+rw w3fetch`

This would add read and write permissions for both the owner and the group.

Finally, the “chmod” command is not restricted to a single file. You can specify several files. For example, if the current working directory contains the files

```
hello1.c    hello2.c    hello3.c
```

then the command

```
chmod a+r hello1.c hello2.c hello3.c
```

makes these three files readable for everyone.

3.4.2 Changing Ownership

The command “chown” is used to change the ownership of a file. It is called as

```
chown new-owner file
```

Here *new-owner* is the new owner and *file* is the file whose owner is to be changed. The command can change the ownership of several files simultaneously, the invocation

```
chown new-owner file1 ... filen
```

would change the ownership for *file₁* to *file_n*. Finally, the command “chown” can also change the ownership of a directory. It can even do this recursively. For example, if “Test” is a directory containing several subdirectories and lots of files and the ownership for all these files and subdirectories has to be changed simultaneously to *new-owner*, then this can be achieved via the following command:

```
chown -R new-owner Test
```

Only the system administrator, who is also known as *super user* or *root*, is allowed to use the command “chown”.

3.4.3 Changing the Group

The command “chgrp” is used to change the group associated with a file. It is called as

```
chgrp new-group file
```

Here *new-group* is the new group that is to be associated with *file*. The command can change the group associated with several files simultaneously, the invocation

```
chgrp new-group file1 ... filen
```

would change the group associated with *file₁* to *file_n*. Finally, the command “chgrp” can also change the group associated with a directory. It can even do this recursively. For example, if “Test” is a directory containing several subdirectories and lots of files and the group associated with these files and subdirectories has to be changed simultaneously to *new-group*, then this can be achieved via the following command:

```
chgrp -R new-group Test Only the system administrator, who is also known as super user or root, is allowed to use the command “chgrp”.
```

3.5 Wildcards

Some of the commands described above can deal with an arbitrary number of parameters. For instance, the command “mv” can deal with an arbitrary number of files. The invocation

```
mv date1.c date2.c date3 date4.c date5.c Test
```

would move the files “date1.c”, “date2.c”, “date3.c”, “date4.c”, and “date5.c” into the directory Test. However, it would be quite tedious to type all these file names on the command line. Therefore, there is an easier way to do this. The shell, which pre-processes every command, expands certain expression known as *wildcards* in a special way. A *wildcard* is an expression that denotes a set of strings. For example, the wildcard expression “[1-5]” denotes the strings “1”, “2”, “3”, “4”, and “5”. Therefore, the expression “date[1-5].c” is a shorthand for the following set of strings:

```
{ “date1.c”, “date2.c”, “date3.c”, “date4.c”, “date5.c” }.
```

As a result, the “mv” command above can be written shorter:

```
mv date[1-5].c Test
```

The above explanation assumes, that the directory where the command is issued does contain the files *date_i* for $i = 1, \dots, 5$. If, for example, there is no file with name *date4.c*, then the command would be expanded to the command

```
mv date1.c date2.c date3.c date5.c Test
```

The general rule here is that wildcards are expanded into the names of *matching* files. Next, we list all wildcard expressions that are supported by the shell. In order to clarify the semantics of these expressions, we provide examples. All these examples assume that we are working in a directory that contains the following files:

```
abc.x   abc.y   date2.c  date4.c  uvw.xy
abc.xy  date1.c  date3.c  date5.c  uvw.x   uvw.y
```

In order to test our wildcard expressions we use the “echo” command. The sole purpose of the “echo” command is to list its arguments, for example

```
echo "Hello, world."
```

would produce the familiar greeting

```
Hello, world.
```

on the screen. The “echo” command is especially well-suited to test wildcard expressions since it can not do any damage to the file system.

1. “?” matches an arbitrary **character**. Therefore, the command

```
echo abc.?
```

produces the result

```
abc.x abc.y
```

when issued in our test directory.

2. “*” matches an arbitrary **string**. Therefore, the command

```
echo abc.*
```

produces the result

```
abc.x abc.xy abc.y
```

when issued in our test directory. The same result is produced by

```
echo ab*
```

This shows that the “*” also matches a string containing a “.” character.

I have included this example because this behaviour differs from the behaviour of wildcard expressions in the MS/DOS environment. However, there is one exception to the rule that a “*” matches any string. Assume the working directory contains a file starting with a dot. For example, assume that it contains the file “.emacs”. A file of this form is called *hidden*. Then, if we would type the command

```
echo *
```

all files would be listed with the exception of the hidden files.

3. “[$c_1 \cdots c_n$]” where the c_i are **characters**. This expression matches one of the characters in the set $\{c_1, \dots, c_n\}$. For example, the command

```
echo uvw.[xy]
```

produces the output

```
uvw.x uvw.y
```

To give another example, the command

```
echo date[12345].c
```

produces the output

```
date1.c date2.c date3.c date4.c date5.c
```

4. “[$a - b$]” where a and b are **characters**. This expression matches one of the characters in the set $\{c \mid a \leq c \leq b\}$, where the *ASCII* ordering of the characters is used. Using this form, we can write the previous example more concisely:

```
echo date[1-5].c
```

This would produce the output

```
date1.c date2.c date3.c date4.c date5.c
```

5. “[! $c_1 \cdots c_n$]” where the c_i are **characters**. This expression matches any character that is **not** in the set $\{c_1, \dots, c_n\}$. For example, the command

```
echo date[!351].c
```

produces the output

```
date2.c date4.c
```

6. “[! $a - b$]” where a and b are **characters**. This expression matches any character that is **not** in the set $\{c \mid a \leq c \leq b\}$, where the *ASCII* ordering of the characters is used. For example, the command

```
echo date[!2-4].c
```

produces the output

```
date1.c date5.c
```

3.6 Links

Sometimes it may be useful to have two different names that refer to the same program, file, or directory. For example, assume that you have written a script that assumes that the C-compiler is called “cc” and is located in the directory “/usr/bin”. Assume further that this directory does contain a C-compiler that works the same way as “cc”, but, unfortunately this compiler is called “gcc”. You can not rename “gcc” to “cc” because other users expect “gcc” to be available in the directory “/usr/bin”. One way to make your script works would be to copy the file “gcc” to “cc”. This would do the job. However, this approach would waste lots of disk space because the same file is stored twice. There is another problem associated with this approach: Assume that “gcc” contains a bug that is later fixed. If the file “gcc” is then replaced by the corrected version, the file “cc” still contains the bug and needs replacement, too. Therefore, instead of copying “gcc”, the better way is to create a *link* with the name “cc” that points to the file “gcc”. Assuming that “/usr/bin” is the current working directory, this can be done with the command

```
ln gcc cc
```

This command seemingly produces a file with the name “cc”. You can check this via the command “ls -l cc” which will produce the following output

```
-rwxrwxrwx  2 root  root          3 2002-11-22 11:16 cc
```

When the directory entry is listed this looks just like a normal file. The only hint that there is something special about this file is the fact that its *link counter* is 2 and not 1. (The link counter is the number that is displayed between the access mode and the name of the user.) The directory entry “cc” is a pointer to the program “gcc”. When the file “gcc” is changed, the file “cc” will reflect these changes automatically. Similarly, when “cc” is changed, in the file “gcc” changes, too. In effect, there is no difference between “cc” and “gcc”. Both are pointers pointing to an internal table of so called *inodes*. These *inodes* in turn point to the real location of the file on the hard disk.

The general form of the “link” command is

```
ln target name
```

This creates a new link with name *name* that point to *target*. Here, *target* can be either a file or a directory.

The links described above are so called *hard links*. These work only on the same device, that is you can not have a hard link on a hard disk drive that points to a CD-ROM drive or to another hard disk. In these cases, you need so called *soft links*. They are also known as *symbolic links*. The general command to create a *soft link* is

```
ln -s target name
```

In the invocation of the command, the only difference to the creation of a hard link is the parameter “-s”. The display of soft links by the “ls” command differs from the display of hard links: Assume that you have some directory containing a file with the name “abc”. Issuing the command

```
ln -s abc xyz
```

produces the soft link “xyz”. Using the command “ls -l xyz” this link is displayed as

```
lrwxrwxrwx  1 stroetma users          5 2003-02-24 23:34 xyz -> abc
```

Here, the first letter in the line that is displayed is the letter “l” which is short for *link* and signifies that this directory entry is a soft link.

There is another subtle difference between hard links and soft links. Assume that, initially, you have a file `abc` in your directory containing the greeting “Hello, World!” and you issue the following commands:

```
ln abc xyz
rm abc
cat xyz
```

The result of all these commands would be that the text “Hello, World!” is printed to the screen. The reason is that internally every file has a count of the number of links that point to it. This counter is known as the *link counter*. Initially, this count is 1. When the first “`ln`” command is executed, this count is incremented to 2. When the file “`abc`” is removed via the “`rm`” command, then what actually happens is the following: The link counter gets decreased by 1. Next, it is checked whether the link counter has reached 0. In our example, this is not the case. Therefore, the data will remain on disk. They would only be purged when we remove the file `xyz`.

Now with a soft link the situation is different. Assume that we start initially with the file `abc` and next issue the command sequence

```
ln -s abc xyz
rm abc
cat xyz
```

Now the result would be an error message of the form

```
File not found!
```

The reason is that setting a soft link does not change the link counter. So when we remove “`abc`”, the file and its data are both sent into the Nirvana. On the other hand, the link “`xyz`” does not notice that the file it is pointing to has been removed. It is a *symbolic link* and contains only the name of the file. This name can be seen as an *address* where to look for the data. There is no check that this address is actually valid. In our case, the address is the file name “`abc`”. Since there is no longer a file with this name, the `cat` command will produce an error message.

3.7 Adding Devices to the File System

Most of the file system is set up when *Linux* is booted. However, sometimes *devices* have to be added to the file system. For example, when a floppy is put into the floppy disk drive, the data of the floppy has to find a place in the directory hierachy. This goal is achieved by *mounting* the floppy to the file system. On my computer, I achieve this via the following command:

```
mount -t msdos /dev/fd0 /media/floppy
```

Then, the data on the floppy can be found in the directory “`/media/floppy`”. The general format of the `mount` command is

```
mount -t type device directory
```

where

1. *type* is the type of the file system to mount. For example, when mounting a floppy disk that has been formatted with MS-DOS, the mount command would be

```
mount -t msdos /dev/fd0 /media/floppy
```

If a data CD is to be mounted, the appropriate type of the file system is `iso9660`.

2. *device* is the device that is to be mounted. It is a path name of the form `/dev/xyz`. The value for `xyz` depends on the device to mount.

3. *directory* is the *mount point* under which the data will be available in the file system.

If the command `mount` is invoked with no arguments, then it lists all devices that are mounted. My system outputs the following:

```
1. /dev/hdb2 on / type reiserfs (rw)
2. proc on /proc type proc (rw)
3. devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
4. /dev/hda1 on /windows/C type ntfs \
    (ro,noexec,nosuid,nodev,gid=100,umask=0002,nls=iso8859-15)
5. tmpfs on /dev/shm type tmpfs (rw)
6. usbdevfs on /proc/bus/usb type usbdevfs (rw)
7. //blackhole/stroetma on /home/stroetma/AMADEUS type smbfs (0)
8. /dev/hdc on /media/dvd type iso9660 (ro,nosuid,nodev,unhide,user=stroetma)
```

Let us go through this output line by line.

1. The first line shows that my disk containing the *Linux* file system is device `/dev/hdb2`. The file system used is of type `reiserfs` and the file system is mounted `(rw)` meaning that I can both read and write.
2. The second line shows the so called *proc* file system. This is a virtual file system containing entries for all processes running on the system. It is discussed elsewhere.
3. The third line shows that the terminals are available under `/dev/pts`. Like *proc*, this is a virtual file system.
4. Line four shows that I have mounted a second hard disk containing my *Microsoft WindowsTM* file system as `/windows/C`. This disk uses the *ntfs* file system. I have mounted it read only `(ro)` since I do not want to crash it. After all, it crashes on its own often enough. Further, this file system is marked as `(noexec)`. This means that files residing on this file system can not be executed, even if they are executable. The reason is that a *Microsoft WindowsTM* executable can not run under *Linux*.
5. The temporary file system `tmpfs` is a file system that actually resides in memory. Therefore, access to the files on this system is very fast.
6. The next line is concerned with the *USB* devices.
7. Line seven shows that I have mounted the directory `stroetma` on the server `blackhole` via the network using the *samba* protocol `smbfs`.
8. The last line shows that the DVD is mounted read only `(ro)`.

Most of the devices shown above have been mounted automatically by the system on startup. Only the DVD has been mounted manually. The mounting of devices during startup of the system is controlled by the file `"/etc/fstab"`. This file contains a list of lines where each line specifies a device, a mount point, the type of the file system and some options used when mounting the device. On my computer, this file reads as shown in Figure 29 on page 59. Each line in this file describes one device. The format of these lines is given as shown below:

device path type options dump-frequency check

The meaning of these parameters is as follows:

1. *device* specifies the block special device or remote filesystem to be mounted.

```

/dev/hdb2      /                reiserfs  defaults          1 1
/dev/hda1      /windows/C       ntfs      \
              ro,users,gid=users,umask=0002,nls=iso8859-15 0 0
/dev/hdb1      swap            swap      pri=42            0 0
devpts         /dev/pts        devpts    mode=0620,gid=5  0 0
proc          /proc           proc      defaults          0 0
usbdevfs      /proc/bus/usb   usbdevfs  noauto            0 0
/dev/cdrecorder /media/cdrecorder auto      ro,noauto,user,exec,unhide 0 0
/dev/cdrom     /media/cdrom    auto      ro,noauto,user,exec,unhide 0 0
/dev/dvd      /media/dvd      auto      ro,noauto,user,exec,unhide 0 0
/dev/fd0      /media/floppy   auto      noauto,user,sync  0 0
/dev/sdc1     /media/camera   vfat      ro,noauto,user    0 0

//basys2002/public-it /home/stroetma/PUBLIC-IT smbfs rw,noauto,user 0 0

```

Figure 29: The file `/etc/fstab`.

2. *path* specifies the mount point for the filesystem.
3. *type* specifies the type of the file system. If “auto” is specified, then the type is detected automatically when the file system is mounted.
4. *options* is a comma separated list of options that are used when the mount command is executed. If this list contains the string “noauto”, then the associated device is not mounted automatically when the system is booted. The most common options are the following:
 - (a) *defaults* specifies that the mount should use the default options when mounting the device.
 - (b) *ro* specifies that the device is mounted read only.
 - (c) *user* specifies that any user is allowed to mount the device.
 - (d) *users* specifies that any user is allowed to mount and unmount the device. Normally, only the user who has mounted a device is allowed to unmount it.
 - (e) *uid=number* specifies the given number as user identifier for all files in the associated file system.
 - (f) *gid=number* specifies the given number as group identifier for all files in the associated file system.
 - (g) *uid=users* specifies that the *uid* for all files in the associated file system is the identifier of the user who mounted the file system. The option *gid=users* works analogously for the *gid*.
 - (h) *sync* specifies that the file system should be mounted *synchronously*, i.e. all changes to the file system take effect immediately. This makes the file system a little bit slower but is safer in case the system crashes.
 - (i) *exec* permits execution of binary files.
 - (j) *noexec* prohibits execution of binary files.

It should be noted that the options that are actually available depend on the type of the file system that is used. Furthermore, the default options are different for different type of file systems. For the file system type *reiserfs* “*exec*” is a default option, but for the file system type *ntfs* “*noexec*” is a default option, since it does not make sense to run *Microsoft Windows*TM “.exe” files on the *Linux* operating system.

5. *dump-frequency* specifies the frequency of backups of the associated file system. A frequency of 0 specifies that this file system does not need to be dumped.
6. *check* is a flag that specifies whether a file system check needs to be done on rebooting.

Of course, the same options that are given in the file “/etc/fstab” can also be used when a system is mounted manually. For example, to mount a USB memory stick manually as a read only device such that the files on it can be executed, write

```
mount -t auto /dev/sda1 /media/sda1 -O ro,exec
```

In general, on the super user *root* is allowed to mount a device. Normal users are permitted to mount the device only if the device is specified in the /etc/fstab with the option of either “user” or “users”. In this case, the mounting is done by just specifying the mount point. For example, given the /etc/fstab file shown in Figure 29 on page 59, any user can mount the DVD via the command

```
mount /media/dvd
```

Unmounting a Device The opposite of the *mount* command is the *umount* command. It is used to detach a device from the file system. This is especially important if a device is mounted *asynchronously*, i.e. if the data that is written to it is kept in memory for some time before it is actually written. For example, for performance reasons *USB* sticks are often mounted asynchronously. If you just unplug the stick without prior unmounting, data might be lost.

Hot-Plugging When examining the file /etc/fstab you might notice that sometimes this file changes apparently *on its own* without your intervention. For example, when attaching an *USB* memory stick to an *USB* port on my desktop, the following line is added to the file /etc/fstab:

```
/dev/sda1 /media/sda1 auto sync,noauto,user,exec 0 0 #HOTPLUG ...
```

This is done by the *hotplug daemon* that watches for changes in the attached hardware.

3.8 dired Mode in XEmacs

Normally, rather than using the commands sketched in the previous subsection, it is easier to manipulate files and directories via the *dired* mode in *XEmacs*. This mode is invoked via the command *dired* which is bound to the key “Ctrl-x d”. This mode offers the following features.

1. “dired-next-line” moves the cursor to the next line.
2. “dired-previous-line” moves the cursor to the previous line.
3. “dired-flag-file-deletion” flags a directory entry for deletion.
4. “dired-expunge-deletions” deletes all flagged directory entries.
5. “dired-unmark” unmarks a marked directory entry.
6. “dired-do-copy” copies a directory entry.
7. “dired-create-directory” creates a subdirectory.
8. “dired-do-rename” renames or moves a directory entry.

9. "dired-do-chmod" changes the access bits of a directory entry.
10. "dired-do-chown" changes the ownership of a directory entry.
11. "dired-do-chgrp" changes the group of a directory entry.
12. "dired-do-symlink" creates a symbolic link.
13. "dired-do-hardlink" creates a hard link.
14. "dired-quit" quits directory mode.

These commands are bound to the keys shown in Table 3.8.

| command | key sequence |
|--------------------------|---|
| dired-next-line | <code>n</code> , <code>Ctrl-n</code> , <code>Space</code> |
| dired-next-line | <code>p</code> , <code>Ctrl-p</code> , <code>Del</code> |
| dired-flag-file-deletion | <code>d</code> |
| dired-expunge-deletions | <code>x</code> |
| dired-unmark | <code>u</code> |
| dired-do-copy | <code>C</code> |
| dired-create-directory | <code>+</code> |
| dired-do-rename | <code>R</code> |
| dired-quit | <code>q</code> |
| dired-do-chmod | <code>M</code> |
| dired-do-chown | <code>O</code> |
| dired-do-chgrp | <code>G</code> |
| dired-do-symlink | <code>S</code> |
| dired-do-hardlink | <code>H</code> |

The behaviour of "dired" can be customized by setting the variable "dired-listing-switches". Often, you want to have a recursive directory listing, that is you want to list the contents of the directory together with the content of its subdirectories. To achieve this, issue the following command in *XEmacs*

```
set-variable
```

You are then prompted for the variable to set. This prompt looks as follows:

```
Set variable:
```

In response to this question, type

```
dired-listing-switches
```

and finish with a `Return`. Then, you are prompted for the new value of this variable:

```
Set dired-listing-switches to value:
```

In response, type

```
Set dired-listing-switches to value: "-lR"
```

and finish with a `Return`. Here, the letter "R" requests the directory listing to be recursive. The next time that you invoke "dired" for a directory, the listing is produced

recursively.

Attention: The procedure sketched above to create a recursive listing does not work if there is already a buffer containing a listing for the directory that is specified as the argument of “`direcd`”. In this case, this buffer needs to be killed first! Otherwise, “`direcd`” just displays this buffer without rereading the directory. Even when you explicitly request rereading of the directory via the `revert-buffer` function that is invoked by typing the character “`g`” in directory mode, the subdirectories will not be listed. To solve the problem, you need to kill the buffer containing the old directory listing. This can be done via the function `kill-buffer` which is invoked by typing `Ctrl-x k`.

4 Compiling and Debugging C Programs with Linux

There are many reasons for using a computer. One is to execute programs, another is to write programs. In this short section we discuss how to compile and debug C programs. We start with a simple example that shows the most basic facilities of the *Gnu* debugger *gdb*. After that, we give a short overview of those features of *gdb* that are used more frequently.

4.1 A Small Example

We discuss a small program that reverses a given string: For example, the string “Hello, World!” would be turned into “!dlroW ,olleH”. Let us assume that, as a first shot, we have written a file containing the following C code:

As it stands, this program is not correct. We will soon find out about its problems. To begin with, we need to *compile* this program. This is done with a C compiler. The C compiler can be started from *XEmacs*. In order to do this, we issue the command

```
Meta-x compile
```

After pressing `Return` the command line prompts us for the compile command as follows:

```
Compile command: make -k
```

Here, the text “Compile command: ” is the prompt string, while the string “make -k” is the default option. This default option is always offered when the `compile` command is executed the first time. Since we want to run the C compiler rather than the `make` command we edit this line until it reads as follows:

```
Compile command: cc -Wall -g -o strreverse strreverse.c
```

Here, we have called the C compiler, whose name is “`cc`”. We have added a number of options that we discuss below:

1. The option “`-Wall`” switches all warnings on. You should always use this option, since it often gives a number of useful hints about possible problems. In fact, there is only one case I can think of where using this option is not a good idea: Imagine you have to maintain some large program that has been written by somebody else. If this somebody else was **too stupid** to use the “`-Wall`” option, then there is no point for you to use this option either because if you would use it, you would first have to correct all the problems the original programmer had introduced.
2. The option “`-g`” adds debug information to the code that is produced. As long as you are still developing your program, you should use this option. Later, when your program is running, then you can remove the “`-g`” option since this will make your program a little bit faster and somewhat smaller.


```

01 #include <string.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 char* strreverse(char* str)
06 {
07     unsigned i;
08     char* reverse;
09     length = strlen(str);
10     reverse = (char*) malloc(length + 1);
11     for (i = 0; i <= length; ++i)
12     {
13         reverse[i] = str[length - i - 1];
14     }
15     reverse[length] = 0;
16     return reverse;
17 }
18
19 int main()
20 {
21     char* prompt = "String eingeben: ";
22     char* text;
23     printf("%s", prompt);
24     scanf("%[a-zA-Z0-9 ,.!?:;]", text);
25     printf("%s\n", strreverse(text));
26     return 0;
27 }

```

Figure 30: The file `strreverse.c`.

3. The option `-o strreverse` has been added to produce the executable file `strreverse`. If we had not added this option, a successful compilation would have produced a file with the name `a.out`.

When we submit the compilation command shown above by pressing Return, we see a screen that looks similar to the one shown in Figure 31 on page 64. The screen is split into two halves: The upper half shows the program, while the lower half shows the output of the C compiler. On a color display, you will see that certain keywords in the program text have been colored. This does not happen by default. In order to activate coloring of keywords the command `Meta-x font-lock-mode` has to be issued.

Let us focus our attention on the lower half of the *XEmacs* window, which contains the error messages. It informs us that the variable `length` in line number 9 is undeclared. We can jump right to this variable by issuing the command

```
next-error
```

which is bound to the key `Ctrl-x ``. (The funny symbol after `Ctrl-x` is a *backquote*. On a German keyboard you get it by pressing the shift key and hitting the key left to the backspace key.) Then we realize that we have forgotten to declare the variable `length`. Therefore, we edit line 7 reading

```
unsigned i;
```

and change it to

```
unsigned i, length;
```

```

Datei Bearbeiten View Cms Tools Optionen Buffers C Lade .emacs Hilfe
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
strreverse1.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char* strreverse(char* str)
{
    unsigned i;
    char* reverse;
    length = strlen(str);
    reverse = (char*) malloc(length + 1);
    for (i = 0; i <= length; ++i)
    {
        reverse[i] = str[length - i - 1];
    }
    reverse[length] = 0;
    return reverse;
}

IS08----XEmacs: strreverse1.c (C Font Abbrev)----Top-----
cd /home/stroetma/Kurse/PDV/
cc -Wall -g -o strreverse strreverse1.c
strreverse1.c: In function 'strreverse':
strreverse1.c:9: 'length' undeclared (first use in this function)
strreverse1.c:9: (Each undeclared identifier is reported only once
strreverse1.c:9: for each function it appears in.)

IS08--**XEmacs: *compilation* (Compilation Font:exit [exit-status 1]

```

Figure 31: Result of compiling “strreverse.c”.

After that we can again invoke the compile command. This time, the compilation succeeds and we see a screen similar to the one shown in Figure 32 on page 65.

In order to short circuit the process of compiling a program and then running it, we can write an *Emacs-Lisp* function that automatically compiles and runs a program. To do this, we add the following code to our “.emacs”file:

```

(defun my-compile()
  "compile with gcc using buffer-name"
  (interactive)
  (let ((name (buffer-name))
        (short-name (substring (buffer-name) 0 -2)))
    (compile (concat "gcc -Wall -g -o " short-name " " name
                    "; " short-name)))
  )
)
(define-key c-mode-map [ (super c) ] 'my-compile)

```

We do not have the time to discuss the mechanics of this code, but the effect is as follows: If you are editing a C program with *XEmacs* and you want to compile and test this program, you just type the key `Super c`, that is the Windows key together with the letter c. This will compile the program you are editing. The executable file produced by the compiler will have the name of the buffer you are editing without the “.c” extension and, furthermore, this executable file is then started.

```

*compilation*
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char* strreverse(char* str)
{
    unsigned i;
    char* reverse;
    length = strlen(str);
    reverse = (char*) malloc(length + 1);
    for (i = 0; i <= length; ++i)
    {
        reverse[i] = str[length - i - 1];
    }
    reverse[length] = 0;
    return reverse;
}

int main()
IS08----XEmacs: strreverse1.c (C Font Abbrev)----Top-----
cd /home/stroetma/Kurse/PDV/
cc -Wall -g -o strreverse strreverse2.c

Compilation finished at Fri Feb 7 16:05:49

IS08--**--XEmacs: *compilation* (Compilation Font:exit OK)----All-----
(No files need saving)

```

Figure 32: Result of compiling “strreverse.c” after declaring “length”.

To continue our demonstration from above, let us try to run the program `strreverse`. So we start a shell in *XEmacs* with “Meta-x shell” and, at the prompt, type the name of the program we have just generated: “`strreverse`”. After hitting `Return`, we are prompted for a string. We enter “Hello, World!” and press `Return`. To our dismay, the screen now looks as follows:

```

stroetma@stroetmannpc:~/Kurse/PDV> strreverse
String eingeben: Hello, World!
Speicherzugriffsfehler
stroetma@stroetmannpc:~/Kurse/PDV>

```

So there is a bug. In order to find this bug, we start the *Gnu* debugger via the *XEmacs* command “Meta-x gdb”. We are prompted for a file name. The file we have compiled is called “`strreverse`”, so we enter this name and press `Return`. After that, we see a screen similar to the one shown in Figure 33 on page 66.

Next, we type “run” followed by a `Return`. This time, our program prompts for a string. After entering it and pressing `Return`, *gdb* displays the following error message on the screen:

```

Program received signal SIGSEGV, Segmentation fault.
0x40073105 in _IO_vfscanf () from /lib/libc.so.6

```

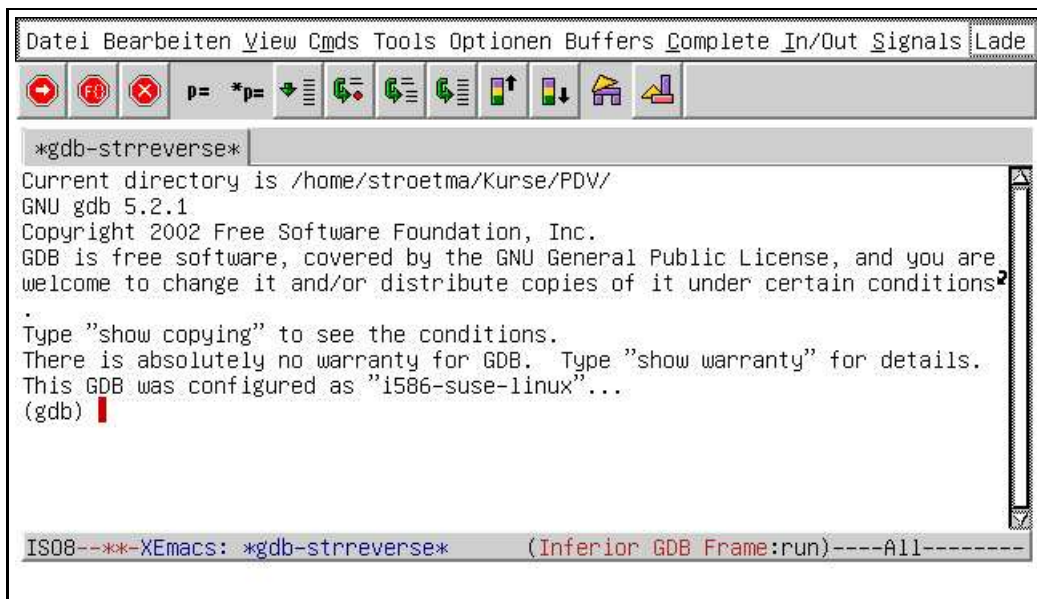


Figure 33: The Gnu debugger.

This message informs us that a *segmentation fault*³ has occurred in the function “_IO_vfscanf”, which has been loaded from the library “/lib/libc.so.6”. Since this function is a library function and the error is most probably not in this library function but in the code that called this library function, we issue the *gdb* command “up” and press Return. This time the message from *gdb* reads

```
#1 0x40076f9a in scanf () from /lib/libc.so.6
```

This message informs us that the function “_IO_vfscanf” has been called by the function “scanf()”. Since this is again a library function we repeat the “up” command to figure out how “scanf” has been called. This time the screen splits and we see a screen similar to the one shown in Figure 34 on page 67.

In the lower half of the screen we see our original program. The line that invoked the function “scanf()” is highlighted. There is just one pointer that is used in this line, the pointer “text”. This pointer has been declared two lines above the highlighted line. Unfortunately, this pointer has never been initialized, so it points to some arbitrary location. We can check this via the *gdb* print command: Entering

```
print text
```

at the *gdb* prompt and pressing Return yields

```
(gdb) p text
$1 = 0x4004e6b0 "U\211\203\b\211]"u0\001"
(gdb)
```

confirming our theory that “text” is the culprit. To correct the error, we change the line

```
char* text;
```

to become

```
char* text = (char*) malloc(81);
```

This will reserve some memory and make *text* point to this memory. After recompiling

³A *segmentation fault* occurs if memory is accessed via a pointer that points to an area of memory that is not accessible. Normally this happens when a pointer is used that has not been properly initialized.

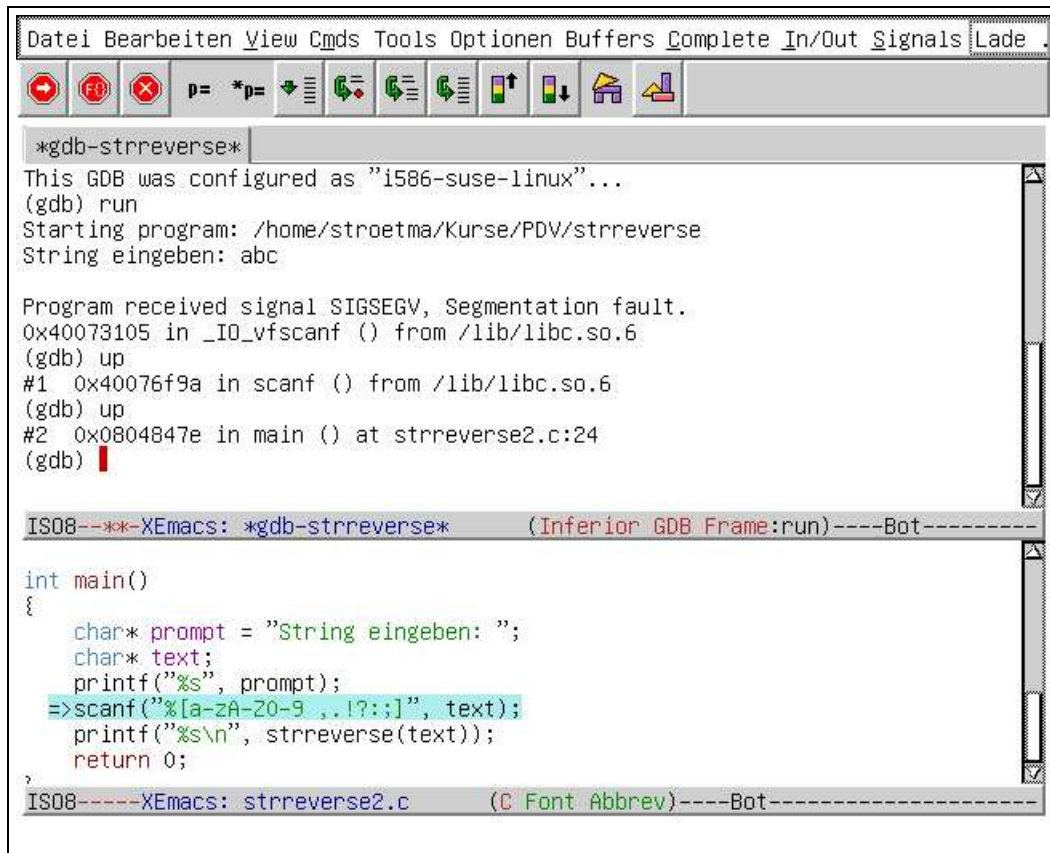


Figure 34: The Gnu debugger showing the offensive line of code.

the program, we start it again in the debugger by typing “run” at the *gdb* prompt and pressing **Return**. Then *gdb* displays the following message on the screen:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

We answer “y” and press **Return**. Again, we are prompted for a string. This time, we enter the string “abcd”, press **Return** and the program answers:

```
dcba

Program exited normally.
(gdb)
```

So finally, the program seems to work, at least for this example.

4.2 Additional Commands in *gdb*

Table 4.2 on page 70 shows a list of the most important *gdb* commands. The table has three columns.

1. The first column gives the name of the *gdb* command. You have to use this name if you enter the commands in *gdb* at the prompt. (The *gdb* prompt is the string “(gdb)”.)

2. The second column gives the name of an *XEmacs* command that invokes the associated “*gdb*” command. For example, the *XEmacs* command that invokes the *gdb* command “run” is “gdb-run”. So instead of typing “run” at the *gdb* prompt you can also type “Meta-x gdb-run”.

Note that some *gdb* commands do not have an equivalent *XEmacs* command.

3. The last column gives the name of the key sequence that can be used to invoke the *XEmacs* command.

Some of these commands are not bound to any key. Of course, if you feel that you need these commands often, then you can bind them to a suitable key yourself.

Of course, *gdb* can be called stand alone without invoking *XEmacs*. To do this, just type

```
gdb file
```

at the prompt of a shell. This will start *gdb* for the specified file. If *gdb* is invoked this way, only the commands given in the first column of Table 4.2 can be used. However, *XEmacs* provides a very convenient interface to *gdb*, which is easier to use than the raw *gdb*.

We proceed to discuss the *gdb* commands in table 4.2.

1. run: Starts the program.
2. break: Sets a *break point*. Normally, if you issue the “run” command, the program loaded by *gdb* will run until its end. To have the program stop at specified places you can set break points. There are two ways a break point can be set:

- (a) A break point can be set at a specified source line. The syntax to do this is

```
break file:lineno
```

Here *file* is the name of the file in which the break point is to be set and *lineno* specifies the line number at which the program should stop. For example, the *gdb* command

```
break strreverse.c:22
```

would set a break point at line 22 in the file *strreverse.c*.

If a break point has to be set at a specified source line, this is easier to achieve via the *XEmacs* interface. Just move the cursor to the line number where the new break point should be set and issue the *XEmacs* command “Meta-x gdb-break”, which is bound to “Ctrl-x space”.

- (b) Another way to set a break point is by specifying the name of a function. The syntax is

```
break function
```

where *function* is the name of a function. The debugger will then stop every time this function is called.

Every break point that is set has a unique number associated with it. This number is needed if the break has to be deleted later. The number is shown by *gdb* at the moment the break point is set. For example, after typing

```
break main
```

gdb might respond as follows:

```
Breakpoint 1 at 0x8048454: file strreverse.c, line 21.
```

This would inform us that this is break point number 1 and that it is set in the file “strreverse.c” at line 22.

3. `delete`: This command is used to remove a given break point. The syntax is


```
delete number
```

 where *number* is the number of the break point that has to be removed.
4. `step`: This command can be used to execute the statements of a given program one by one. This is often the best way to understand a program.
5. `next`: This is similar to the “`step`” command. The important difference comes when a function is called. The “`step`” command would step into the function, while the “`next`” steps over the function, that is the function is executed in one sweep.

This is especially useful when library function are called, since usually there is no point in debugging these functions. After all, most of the time the bugs are to be found in the code you have written yourself.
6. `finish`: Finishes execution of the current function.

Suppose that during debugging you have stepped down into a function and after some time discover that this function seems to work correct. Therefore, you want to finish stepping through this function. The “`finish`” command achieves this. This will cause the execution of the function to be finished.
7. `continue`: Proceeds until the next break point is hit.
8. `up`: Move to the function that called the function you are currently inspecting. Use this command to check how the function currently being executed has been called. This is especially useful after the debugger stops in some function because a breakpoint has been hit.
9. `down`: The opposite of “`up`”.
10. `until`: Executes commands until a given line number is reached. The syntax is


```
until line
```

 where *line* is the number of the line at which the execution should stop.

Imagine you have just entered a loop that is to be executed 1000 times. Stepping through this loop one by one would be too tedious. If you want to step over this loop and the first command following the loop is in line 45, the command “`until 45`” will achieve this.
11. `print`: Print the value of a given expression. The syntax is


```
print expr
```

 where *expr* is a C expression. Most often, this will just be the name of a variable so you would type something like


```
print x
```

 If the variable “*x*” is of type pointer to type *T*, then you might also use


```
print *x
```

 in order to get the value to which “*x*” points.
12. `watch`: Set a *watch point* for a given expression. The syntax is


```
watch expr
```

 The most common case is to watch a variable. After issuing the command


```
watch c
```

 the program will stop every time the variable “*c*” is changed.

13. `help`: Prints the documentation of a given command. For example,


```
help print
```

 provides documentation about the `gdb` “`print`” command.
14. `display`: Print the value of a given expression every time the debugger stops. The syntax is


```
display expr
```

 where `expr` is a C expression. This function is useful for tracing.
15. `define`: `gdb` offers the possibility for users to define their own commands. This is a very advanced feature and is described in the *info* pages documenting `gdb`.

| <i>gdb</i> command | <i>XEmacs</i> command | key sequence |
|--------------------|-------------------------|--|
| run | <code>gdb-run</code> | |
| break | <code>gdb-break</code> | Ctrl-x space |
| delete | | |
| step | <code>gdb-step</code> | Meta-s |
| next | <code>gdb-next</code> | Ctrl-c Ctrl-n |
| finish | <code>gdb-finish</code> | Ctrl-c Ctrl-f |
| continue | <code>gdb-cont</code> | Ctrl-c Meta-c |
| up | <code>gdb-up</code> | Ctrl-c < |
| down | <code>gdb-down</code> | Ctrl-c > |
| until | | |
| watch | | |
| print | | |
| help | | |
| display | | |
| define | | |

5 Getting Help

In *XEmacs* there are four ways to get help:

1. The *manual pages* describe various commands and files. To view a manual page, use the command

```
man topic
```

where *topic* is the topic you want to get information about. For example

```
man cp
```

provides information concerning the command “`cp`”. *XEmacs* provides an interface to the “`man`” command: just type

```
Meta-x manual-entry
```

You are then prompted for a topic on which to get help. Viewing manual pages in *XEmacs* is more convenient than viewing these pages in a shell since it is easier to navigate when the manual page does not fit on a single screen. Also, you can use the search facility of *XEmacs* to quickly find some topic in a manual page.

2. The *info* system is similar to the manual pages. It is invoked by typing

```
info
```


In *XEmacs* the info system is started via the command

```
Meta-x info
```

You should definitely make yourself acquainted with the info system. If you start the info system, typing “h” provides a short tutorial on using this system.

3. The “apropos” command searches through the brief description of the manual pages for a given keyword. For example, assume you want to record a CD. In order to find out which programs are available, type

```
apropos record
```

Then the “apropos” command produces output similar to the one shown in Figure 35 on page 71. After carefully reading through this list you finally spot the command “cdrecord”. Next, you can look at the manual entry for this command to discover how it works.

4. Furthermore, there is the command `susehelp` which presents a help system with a graphical user interface.
5. If you have the name of a command and want to know what the command is about, try the `whatis` command. For example,

```
whatis xemacs
```

yields the following response from my system:

```
xemacs (1)          - Emacs:  The Next Generation
```

6. Finally, the most effective source to get information is to ask a local *wizard*.

```
stroetma@stroetmannpc:~/Kurse/PDV> apropos record
recno (3)          - record number database access method
pppdump (8)        - convert PPP record file to readable format
perlhst (1)        - the Perl history records
AddErrInfo (3)     - record information about errors
utmp (5)           - login records
wtmp (5)           - login records
RecordEval (3)    - save command on history list before evaluating
cdrecord (1)       - record audio or data Compact Discs from a master
wmrecord (1)      - General Purpose Recording Utility for Linux
```

Figure 35: Output of the command “apropos record”.

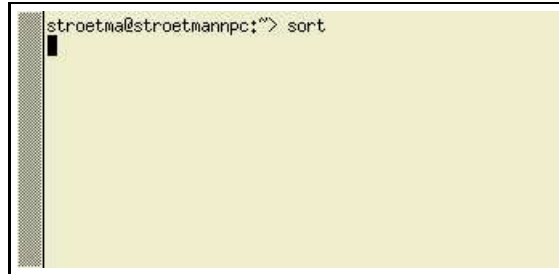
6 Input and Output Redirection and Pipes

6.1 Input and Output Redirection

A lot of programs work as follows: They read their input from the keyboard, process it and then write the results that have been computed to the screen⁴. Programs of this kind are traditionally known as *filters*. As an example, consider the “sort” program. This program considers its input line by line and outputs the lines in a sorted order. We show the working of “sort” by giving an example:

⁴Technically, this is not the whole story. Programs read their input from the so called *standard input* and write their output to the so called *standard output*. Normally, *standard input* is connected to the keyboard and *standard output* is connected to the screen. We will soon see how *standard input* and *standard output* can be reconnected to other places.

- To begin with, we type “sort” at the prompt of the shell and hit the `return` key to invoke the command.



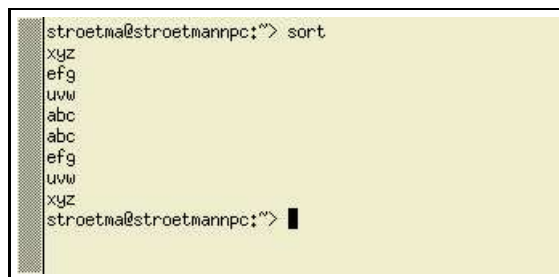
```
stroetma@stroetmannpc:~$ sort
```

Figure 36: The screen after “sort” has been started.

- Then, the screen looks as shown in Figure 36 on page 72. Nothing seems to have happened. The reason is that “sort” is waiting for input. So, let us enter some input. We enter the following lines, ending each with a `return`.

```
xyz
efg
uvw
abc
```

- After entering the last line, we press the key “Ctrl-D” to signal that the input is complete. This key is equivalent to an *end-of-file* character and signals to the shell that the input is complete. Once we have pressed this key, the sort command starts working and outputs the four lines that we have input in sorted order immediately below our input. The terminal window now looks as shown in Figure 37 on page 72.



```
stroetma@stroetmannpc:~$ sort
xyz
efg
uvw
abc
abc
efg
uvw
xyz
stroetma@stroetmannpc:~$
```

Figure 37: The screen after “sort” has finished its work.

If we really want to sort a large number of items stored in a file, we hardly want to type those items at the prompt of a shell. But we can advise “sort” to take its input from a file rather than from the keyboard by redirecting its *standard input* to a file. We show this via an example. Assume that “data.log” is a file containing data to be sorted. Then the shell command

```
sort < data.html
```

sorts these data. The output is still printed to the screen. Above, the character “<” was used to ask the shell to reconnect the standard input of “sort” to the file “data.html”. The general form of input redirection is as follows:

```
cmd-with-opts-and-args < in-file
```

Here, *cmd-with-opts-and-args* is a command together with its options and arguments. This command has to read its input from *standard input*. The input redirection character “<” orders the shell to supply the content of the file *in-file* as input to this command.

In the same way as the standard input can be taken from a file instead of the keyboard, the standard output can be written to a file instead of being written to the screen. For example, executing the command

```
echo Hello, World > echo.out
```

executes the command “echo Hello, World” and writes the output that is created to the file

“echo.out”. If this file does not exist, it is created. If it exists, its old content is discarded and replaced by the output of the “echo” command. The general form of output redirection is as follows:

```
cmd-with-opts-and-args > out-file
```

Here, *cmd-with-opts-and-args* is a command together with its options and arguments. This command has to write its output to *standard output*. The output redirection command “>” advises the shell to put the output produced by this command into the file *out-file*.

If *out-file* does already exist, then its former content will be lost. Sometimes, this is not desired. Instead, the output should be *appended* to the *out-file*. Therefore, there is another way of output redirection. If we use the redirection characters “>>” instead of the redirection character “>”, then the output is appended to the content of *out-file*. For example, assume that the file “times.log” contains a number of dates and times. Issuing the command

```
date >> times.log
```

would append a line containing the current date and time to this file.

Input and output redirection can be combined. Consider the case that “unsorted.data” is a file containing data that need to be sorted. Then issuing the command

```
sort < unsorted.data > sorted.data
```

provides the content of the file “unsorted.data” as input to the “sort” command. The output of the “sort” command is then written to the file “sorted.data”.

Often, error messages produced by a program are not written to standard output. Rather, they are written to *standard error*. Since both standard output and standard error are connected to the screen, the user does not notice that there is a difference. The difference becomes visible when standard output is reconnected to a file. Then, standard error is still connected to the screen. Let us give an example: Issuing the command

```
date --quark > data.out
```

produces the output

```
date: unknown option --quark
You get more information with 'date --help'.
```

The file “data.out” is created, but it is empty. Often, this behaviour of output redirection is just what is needed because otherwise problems would just go unnoticed. For those rare cases where also the standard error should be redirected to a file, there are two options:

1. The redirection characters “2>” redirects all error messages to the specified file. For example, the command

```
date --quark > data.out 2> error.msg
```

puts the error message into the file “error.msg”, while the output is written into the file “data.out”. So, after this command is executed, the file “data.out” will be empty and the file “error.msg” will contain an error message.

2. The redirection characters “&>” redirects both standard error and standard output to the specified file. For example, the command

```
date --quark &> data.out
```

puts both the error message and the output into the file “data.out”.

The first redirection command for standard error can also be used with two “>” characters instead of one. Then this command read “2>>”. In this case, the error messages are appended to the specified file instead of overwriting it. Finally, if both standard output and standard error have to be appended to a single file, then this can be achieved as shown below:

```
cmd-with-opts-and-args >> file 2>&1
```

The description given above is not the whole story. Further details can be found in the manual page describing the *bash*.

6.2 Pipes

The last section has shown that the standard input of a program can be taken from a file and the standard output can be redirected to a file. In this section, we will see that the standard output of one program can be connected to the standard input of another program. This way, an arbitrary number of programs can be made to work together on a single task. Although each of the programs involved may be quite simple, the combination of several simple programs can result in a very powerful command.

The syntax for connecting the standard output of one program with the standard input of another program is

```
cmd1 | cmd2
```

Here, *cmd1* and *cmd2* are commands and the output produced by *cmd1* is taken as input of *cmd2*. A construction of the form “*cmd1* | *cmd2*” is known as a *pipe*. Let us look at a simple example: The pipe

```
ls -l | sort
```

lists the directory entries of the current directory and sorts them. Admittedly, as it stands this pipe is not that useful since the directory entries will be listed according to their type and permission, which is rarely needed. We will see later a variant of this pipe that is more useful.

In order to show the power of pipes, we first need to discuss a number of small programs that are found in pipes. This is done in the following subsections.

6.2.1 The “cat” command

The basic purpose of the “cat” command is to write the contents of a file to the standard output. It is invoked as

```
cat opts file
```

This will print the contents of *file* to the standard output. The precise way this is done is controlled via the option *opts*. We list a few of the available options below:

1. “-n” numbers all lines.
2. “-T” displays `tab` characters as “^I”.
3. “-E” displays line feed characters as “\$”.
4. “-v” displays non-printing characters. Control characters are displayed using the “^” notation and meta characters are displayed using “M-” notation. However, line feeds and `tab` characters are printed literally.

The `cat` command can also be invoked with multiple file arguments. Then its invocation has the form

```
cat opts file1 ... filen
```

In this case, it concatenates (hence the command name) all these files and writes the resulting data to standard output.

If the “`cat`” command is invoked without a file argument, then it reads its input from standard input, that is it acts as a filter.

Often, the `cat` command is the source of the data of a pipe. The pipe might then have the form

```
cat file | filter1 | ... filtern
```

In this pipe, the content of *file* is transformed by *filter₁* to *filter_n*.

6.2.2 The “`cut`” command

The “`cut`” command views its input as a *table* consisting of different columns. It is used to select a specified subset of these columns. Let us illustrate this via an example. Consider the following `passwd` file containing information regarding various users:

```
stroetma:x:4678:100:Karl Stroetmann:/home/stroetma:/bin/bash
louis:x:666:100:Louis Cypher:/home/louis:/bin/cash
gates:x:5678:100:Bill Gates:/home/gates:/bin/crash
death:x:4677:100:Dr. Death:/home/death:/bin/ash
```

Now in the `passwd` file shown above, the data is organized in columns that are separated by the character “:”. We would like to list the user ids followed by the names of the users. The name can be found in the 5th column, while the numerical user id is given in the 3rd column. The “`cat`” command can select these columns. Assuming that the file whose contents are shown above has the name “`passwd`”, we can achieve our goal by issuing the command

```
cut -d":" -f3,5 passwd
```

Here, the option “`-d`” specifies that the columns are delimited via the “:” character, while the option “`-f3,5`” specifies that the 3rd and 5th column should be listed. Given the file shown above, this command produces the output shown below:

```
666:Louis Cypher
5678:Daniel Dumpfbacke
4678:Karl Stroetmann
4677:Gott Vater
```

If the “`cut`” command is invoked without a file argument, then it reads its input from standard input.

6.2.3 The “`paste`” Command

The “`paste`” command merges data of different files. It does so by interpreting the content of its argument files as different columns that should be joined into a table. In this respect, it is just opposite to the `cut` command. For example, assume that the file “`numbers`” looks as follows:

```
1
2
3
```

Assume further that the file “`data`” has the form

```
a
b
c
```

Then the command

```
paste numbers data
```

produces the following output:

```
1      a
2      b
3      c
```

By default the different columns are separated by a tab character. This can be changed via the “-d” option. If we had issued the command

```
paste -d":" numbers data
```

Then the output would have been

```
1:a
2:b
3:c
```

6.2.4 The “tr” command

The “tr” command is used to translate or delete characters. It always reads from standard input and writes to standard output. To translate characters, the command is invoked as follows

```
tr set1 set2
```

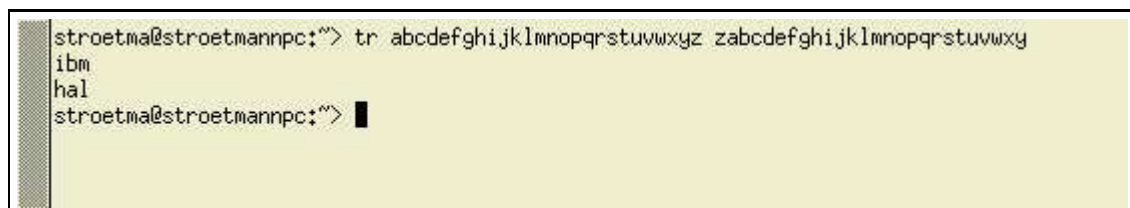
Here, *set1* and *set2* must be sets of characters. The effect of the command is that every character of *set1* is translated into the corresponding character of *set2*. For example, the invocation

```
tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

translates every lower case character into its corresponding upper case character. A variant of this command can be used to implement a trivial ciphering method: The command

```
tr abcdefghijklmnopqrstuvwxyz zabcdefghijklmnopqrstuvwxy
```

will replace every character by the character preceding it in alphabetical order. Figure 38 on page 76 shows what happens when this ciphering is applied to the input string “ibm”. There is a shorter way to specify this ciphering method using *range specification*.



```
stroetma@stroetmannpc:~$ tr abcdefghijklmnopqrstuvwxyz zabcdefghijklmnopqrstuvwxy
ibm
hal
stroetma@stroetmannpc:~$
```

Figure 38: Using the “tr” command for ciphering.

The previous command could have been written as

```
tr a-z za-y
```

Here the notation “a-z” is short for all characters with ASCII code between and including the ASCII codes of “a” and “z”.

Besides translating, the “tr” command is often used for deleting multiple characters. This is done with its *squeeze* option: The command

```
tr -s ' '
```

will squeeze multiple occurrences of blanks characters to a single occurrence. For example, different columns in the output of the “ls” command are separated by blanks. Assume that “ls -l” produces the output shown in Figure 39.

```
-rw-r--r--  1 stroetma users      2628 2003-02-07 14:56 daten.html
-rw-r--r--  1 stroetma users       224 2003-02-06 19:37 macro.el
-rw-r--r--  1 stroetma users      3028 2003-02-06 21:15 uebung2.dvi
-rw-r--r--  1 stroetma users      2948 2003-02-06 21:16 uebung2.tex
```

Figure 39: Output of the “ls” command.

Piping this output into the command “tr -s ' ’” would squeeze the blanks. The output of

```
ls -l | tr -s ' '
```

will therefore be as follows:

```
-rw-r--r-- 1 stroetma users 2628 2003-02-07 14:56 daten.html
-rw-r--r-- 1 stroetma users 224 2003-02-06 19:37 macro.el
-rw-r--r-- 1 stroetma users 3028 2003-02-06 21:15 uebung2.dvi
-rw-r--r-- 1 stroetma users 2948 2003-02-06 21:16 uebung2.tex
```

Getting rid of excessive blanks is sometimes necessary when the data is needed as input of the “cut” command discussed above: If different columns are separated by blanks, then we must always have exactly one blank as a separator.

Sometimes, the command tr is used to delete newlines. To do this, invoke tr as follows

```
tr -d "\n"
```

If tr is invoked with the option “-d”, then all characters from the set of characters given as first argument are deleted. Special characters, like the newline character, are denoted via *escape sequences*, for example, the string “\n” denotes a newline character. The complete list of *backslash escape sequences* for the tr command is shown in Table 18 on page 77. Note that it is also possible to specify an octal character in the form $\backslash o_3 o_2 o_1$, where o_3 , o_2 , and o_1 are the octal digits, that is they are digits from the set $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Therefore “ $\backslash o_3 o_2 o_1$ ” specifies the character $o_3 * 8^2 + o_2 * 8 + o_1$. The escape sequences shown in Table 18 are the same escape sequences that are used with the function “printf()” in the programming language C. Finally, note that in order to include a literal backslash character “\” you have to escape it with a second backslash, that is you have to write “\”.

The option “-s” for squeezing characters and the option “-d” for deleting characters can be combined. The general syntax for this combination is

```
tr -d -s set1 set2.
```

This invocation of tr will first remove any characters from set1 and will squeeze multiple occurrences of characters in set2. To give an example, the command

```
tr -d -s "0-9" " "
```

first removes all digits. Then, repeated occurrences of blanks are squeezed into one blank.

| escape sequence | control character | ASCII value | meaning |
|---|-------------------|----------------------------|----------------------|
| <code>\a</code> | <i>control g</i> | 7 | bell |
| <code>\b</code> | <i>control h</i> | 8 | backspace |
| <code>\f</code> | <i>control l</i> | 12 | form feed |
| <code>\n</code> | <i>control j</i> | 10 | newline |
| <code>\r</code> | <i>control m</i> | 13 | carriage return |
| <code>\t</code> | <i>control i</i> | 9 | horizontal tabulator |
| <code>\v</code> | <i>control k</i> | 11 | vertical tabulator |
| <code>\\</code> | | 92 | literal backslash |
| <code>\o₃o₂o₁</code> | | $o_3 * 64 + o_2 * 8 + o_1$ | octal character |

Table 18: Escape sequences for `tr`

6.2.5 The “sed” Command

The command name “`sed`” is short for *stream editor*. The invocation of “`sed`” that is most often seen in practice is

```
sed s/str1/str2/g
```

The effect of this command is that every occurrence of the string `str1` is replaced by `str2`. In this way, “`sed`” is a generalization of “`tr`”.

The real power of `sed` comes from the fact that we can use a *regular expression* for `str1`. Since we have not introduced regular expressions yet, we defer a discussion of this feature. The program `sed` is very powerful, entire books have been written discussing it [3]. If you have some spare time, you should read the info pages describing `sed`.

6.2.6 The “tee” Command

This command copies its standard input unchanged to standard output and also saves this input into a file specified as its argument. To see that this is useful, consider the following example: The file “`screen.xwd`” contains a screen dump in a certain format known as *X Window Dump File* format. The command “`xwdtopnm`” can be used to convert this format into a *portable bitmap graphic*. And then the command “`ppmtogif`” converts this into the *gif* format. Suppose that we need both the portable bitmap graphic and the *gif* file. The following pipe would achieve this:

```
cat screen.xwd | xwdtopnm | tee screen.ppm | ppmtogif > screen.gif
```

This command creates the files “`screen.ppm`” and “`screen.gif`” both in one sweep. The trick is that “`tee`” is used to save the intermediate file “`screen.ppm`” and at the same time passes the data over to the command “`ppmtogif`”. Using `tee` we can *tap* into a pipe and extract data from.

6.2.7 The “fgrep” Command

The “`fgrep`” command is used to search for strings in files. It is invoked as follows:

```
fgrep opts string file1 ... filen
```

The effect is, that the given files `file1` to `filen` are searched for the specified *string*. For example, the command

```
fgrep stroetma *
```

would search all files in the current working directory for the string “`stroetma`”. Among the options most often used are “`-R`” to traverse directories recursively and “`-I`” to exclude binary files from the search.

We should note that “fgrep” is a restriction of the much more powerful “grep” command. Instead of simply searching for strings, this command accepts *regular expressions* as search patterns. However, the letter “f” in fgrep is short for *fast*, so fgrep is really *fast grep*. It can be faster because it is less general.

6.2.8 The “head” Command

The command head displays the first 10 lines of the file given as argument, that is the invocation

```
head file
```

displays the first 10 lines from *file*. The option *-n* can be used to display a different number of lines:

```
head -n count file
```

displays the first *count* lines from *file*. If no file is given as argument, head reads from standard input. This command is useful for computing the maximum of a list of numbers. Assume that the file *data* contains lots of numbers, each on one line. Then the pipe

```
cat data | sort -n -r | head -n 1
```

computes the maximum of all numbers in *data*.

The command can also be used to print the first *count* bytes of a file. This is achieved with the “-c” option :

```
head -c count file
```

displays the first *count* characters from *file*.

6.2.9 The “tail” Command

The command tail displays the last 10 lines of the file given as argument, that is the invocation

```
tail file
```

displays the last 10 lines from *file*. The option *-n* can be used to display a different number of lines:

```
tail -n count file
```

displays the last *count* lines from *file*. If no file is given as argument, tail reads from standard input. This command can be used to compute the maximum of a list of numbers. Assume that the file *data* contains lots of numbers, each on one line. Then the pipe

```
cat data | sort -n | tail -n 1
```

computes the maximum of all numbers in *data*.

The command can also be used to print the last *count* bytes of a file. This is achieved with the “-c” option :

```
tail -c count file
```

displays the last *count* characters from *file*.

The tail command has another important option. This option is “-f” where the letter ‘f’ is for following a file. If tail is invoked as

```
tail -f file
```

and *file* is a file that is constantly growing, for example some log file, then every time that new lines are added to *file* these new lines are written to standard output. There are a number of additional options for tail that can be used to specify how often tail should check whether *file* has grown or the action that should be taken when *file* disappears. Consult the *info* pages for the full story.

6.2.10 Putting it All Together

Finally, we show how little programs like the ones shown above can be made to work together. Consider the directory listing shown in Figure 39 on page 76. Assume that all we are really interested in is the size of the file and its name and that we would like to have the files sorted by their name. The following commands achieve this goal:

1. `ls -l | tr -s ' ' | cut -d' ' -f5,8 | tr ' ' '\t' > inter`
2. `cat inter | cut -f1 > size`
3. `cat inter | cut -f2 > names`
4. `paste names size | sort`
5. `rm inter names size`

The output of this command is shown below:

```
daten.html      2628
macro.el        224
uebung2.dvi     3028
uebung2.tex     2948
```

Let us discuss these commands one by one.

1. `"ls -l | tr -s ' ' | cut -d' ' -f5,8 | tr ' ' '\t' > inter"`
 - (a) `"ls -l"` produces the directory listing shown in Figure 39 on page 76 and pipes it into `"tr -s ' '"`.
 - (b) `"tr -s ' '"` squeezes multiple blanks into a single blank and handles that output over to `"cut -d' ' -f5,8"`.
 - (c) `"cut -d' ' -f5,8"` selects the 5th and 8th column of this output. This are the size of the file and the name of the file, respectively.
 - (d) `"tr ' ' '\t'"` replaces blanks by tab characters, thereby adjusting the output.

Finally, the output of this first command is written into the file `"inter"`.

2. `"cat inter | cut -f1 > size"`

This takes the first column of the file `inter` and puts it into the file `size`. The first column is the column containing the file sizes.
3. `"cat inter | cut -f2 > names"`

This takes the second column of the file `inter` and puts it into the file `names`. The second column is the column containing the file names.
4. `"paste names size | sort"`

pastes the lines from the files `"names"` and `"size"` together and pipes the result to the `"sort"` command which then sorts its input and writes it to standard output.
5. `"rm inter names size;"`

has been added for cleaning up.

The example given should convince you that the combination of several small programs via pipes can produce commands that solve quite sophisticated problems.

When discussing the previous example, there is one point that should not be missed. It is tempting to think that the commands given above could be shortened to

```
ls -l | tr -s ' ' | cut -d' ' -f8,5 | tr ' ' '\t' | sort
```

Here the `cut` command has been changed from `cut -d' ' -f5,8` to `cut -d' ' -f8,5`. Unfortunately, this does not work as intended because the order in which the columns are output is always the same as in the input data. Therefore, although we have specified the columns as “-f8,5”, the 5th column is output before the 8th column. If we want them in a different order, we have to store them in different files which can then be pasted together.

7 The Shell

The *shell* is the interface⁵ between the user and the operating system. The commands that you type are first interpreted by the shell. The shell then invokes the appropriate programs. Actually, there are a number of different shells available. Nowadays, the two most prominent are the *bash* and the *tcsh*. We will only discuss the *bash* since this is the default shell but in case you want to change you shell you can do this via the command

```
chsh -l new-sh user
```

where *user* is your user name and *new-sh* specifies the new shell that you want to use.

The shell can do much more than just execute programs. For example, the input and output redirection discussed in the previous chapter are features provided by the shell. The following subsections discuss additional features of the shell.

7.1 Shell Variables

The shell provides variables that can be used to control both the shell itself and the programs that are started from the shell. Now, what exactly is a variable? A variable has two properties: It has a *name* and a *value*. The name consists of alphanumeric characters and underscores and has to start with an alphanumeric variable. The value can be accessed by prefixing the name of the variable with a “\$” character. A variable is set to a value using the assignment operator “=”. The easiest way to access the value is via the “echo” command. For example, if the following commands are executed

```
XYZ=abc
echo The value of XYZ is now $XYZ.
```

then, as a result the shell would print

```
The value of XYZ is now abc.
```

It is important to note that no white space is allowed in the assignment “XYZ=abc”. For example, writing “XYZ = abc” produces an error message stating that the command “XYZ” could not be found.

The above describes the way to set a variable interactively. But in the same way we can also set variables permanently by putting the variable assignment into a file that is read automatically when the shell is started. The file “~/.bashrc” in your home directory is read by the *bash* upon startup by every interactive *bash* shell that is not a login shell. For a login shell, the *bash* searches for the files “~/.bash_profile”, “~/.bash_login”, and “~/.profile” and loads the first one that it finds. Therefore,

⁵Actually, this is only part of the truth because between the user and the shell is a terminal program like *xterm*. However, discussing *xterm* is out of the scope of this lecture.

we can set variables permanently in these files. The user can also load a file via the command

```
source file
```

which reads *file* and executes the commands found in it.

Variables come in two flavours: *local* variables and *global* variables. The local variables are only visible to the shell, while the global ones are also visible to programs that are started by the shell. In order to convert a variable from local status to global status, it needs to be *exported*. This is done by the command “`export`”. In the example above, if we want the variable “XYZ” to become a global variable, we can achieve this via the command

```
export XYZ
```

There is a shortcut available to define and export a variable at the same time. The command

```
export XYZ=abc
```

sets the value of “XYZ” to “abc” and proceeds to export this variable.

Up to now, this is not very exciting. Why do we need variables at all? Well, a number of properties of the shell are controlled via variables. We discuss some of the most important now, for more information you should consult the *bash* manual page or the *info* page describing the *bash*.

1. “PATH” specifies the *search path* for commands. It is a colon-separated list of directories in which the shell looks for commands when it is asked to execute a command. The default path is system-dependent, and is set by the administrator. A common value is

```
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/opt/gnome/bin:.
```

You can extend this value to include some of your own directories. For example, I have a line similar to the following in my “.bashrc”

```
export PATH=/home/karl/bin:$PATH
```

This adds the directory “/home/karl/bin” to the list of directories that are searched for executable commands.

2. “MANPATH” contains the search path for manual pages. If you have manual pages of your own, then you need to add a path leading to these pages to the variable “MANPATH”.
3. “HOME” contains the path to your home directory.
4. “LOGNAME” contains your login name.
5. “HOSTNAME” contains the name of your computer.
6. “PS1” specifies the prompt string. This prompt string can contain certain backslash-escaped special characters. The most important are listed below:
 - (a) “\w” yields the current working directory.
 - (b) “\u” yields the user name.
 - (c) “\H” yields the hostname.
 - (d) “\h” yields the hostname up to the first “.” character.
 - (e) “\n” yields a newline.
 - (f) “\r” yields a carriage return.
 - (g) “\t” yields the current time in 24-hour format.
 - (h) “\T” yields the current time in 12-hour format.

(i) “\d” yields the date.

In practice, you often see excessive prompts that clutter the screen. Personally, I favor modest prompts. In my “bashrc”, the variable PS1 is therefore set as follows:

```
export PS1="\u@\h:\w>\nHonorable master, what is your command? "
```

The following variables are only needed when writing shell scripts.

7. “#” contains the number of arguments given to a shell script.
8. “*” contains a list of all arguments given to a shell script.
9. “0” contains the name of the script.
10. “1” contains the first argument given to a script. Similarly, “2” contains the second argument. The numbers from 1 to 9 can be used to refer to the corresponding arguments of a script.
11. “\$” gives the process identifier of the process running the shell script.
12. “?” gives the exit status of the last command that has been executed.

Remember that to get the *value* of a variable you have to prefix the *name* with the “\$” character. To give an example, in order to print the process identifier of the current shell you have to type

```
echo $$
```

7.2 Job Control

Sometimes, a program takes a long time to finish its task. Since Linux is a multitasking operating system, there is no need to wait for its completion. Simply append the character “&” at the end of the command name and the command is executed *asynchronously*. This concept is also known as *background execution*. For example, assume that numbercrunch is a program taking a long time. Then issuing this command as

```
numbercrunch &
```

executes this program in the background. The shell will respond with something like

```
[1] 23456
```

and print a prompt to indicate that it is ready for more input. Here, the number [1] specifies the *job number*. Every process started asynchronously from a shell has a unique number, the *job number*. This number can be used for various aspects of job control that are discussed in the following. The number 23456 is the so called *process identifier*. Every process running on a *Linux* system has a unique process identifier associated with it. This number can be used to stop the process.

After starting numbercrunch, I start my audio player. Since I want to continue working, I will execute this command in the background, too. So I issue the command

```
“xmms &”
```

Then I start netscape and “xemacs” asynchronously. In order to get an overview about all the processes that have been started asynchronously, the command “jobs” is used. It lists all those jobs that I have started asynchronously from the current shell. In our case, it would display something like

```
[1] Running          numbercrunch &
[2] Running          xmms &
[3]- Running         /usr/local/netscape/netscape &
[4]+ Running         xemacs &
```

As can be seen, all processes are numbered. In order to bring back process number n to the *foreground*, the command

```
fg %n
```

is used. In the same way, a running process can be killed. This is done via the command

```
kill %n
```

where n specifies the process number.

If you start a program and forget to put it to the background, you can send it into the background by pressing the character `Ctrl-Z`⁶. This would suspend the program. In order to get it running again, you have two options. Issuing

```
bg %
```

puts the program into the background, while the command

```
fg %
```

resumes the command in the foreground. You can put any program running in the background back into the foreground via the command

```
fg %n
```

where n is the *job number* of the process. These job numbers are the numbers listed by the `jobs` command.

Sometimes, you want to know all processes that are running on your machine, regardless of whether you started them from the current shell or not. This is achieved by the command “`ps`”, which is short for process status. For example, typing

```
ps ux
```

at the prompt of my shell yields the output shown in Figure 40 on page 84.

| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | START | TIME | COMMAND |
|----------|-------|------|------|-------|-------|-------|------|-------|------|-------------------|
| stroetma | 1931 | 0.0 | 0.2 | 5424 | 2852 | ? | S | Dec01 | 0:03 | /usr/X11R6/bin/fv |
| stroetma | 1969 | 0.0 | 0.2 | 5164 | 2536 | ? | S | Dec01 | 0:00 | /usr/X11R6/bin/fv |
| stroetma | 1980 | 0.0 | 0.1 | 2820 | 1652 | ? | S | Dec01 | 0:00 | /usr/bin/tclsh /h |
| stroetma | 1983 | 0.0 | 0.1 | 3632 | 1932 | ? | S | Dec01 | 0:14 | fetchmail -d 60 |
| stroetma | 1985 | 0.0 | 0.1 | 2820 | 1652 | ? | S | Dec01 | 0:00 | /usr/bin/tclsh /h |
| stroetma | 7600 | 0.0 | 0.1 | 4540 | 1664 | ? | S | Dec03 | 0:00 | /usr/bin/smbmount |
| stroetma | 11335 | 0.0 | 0.1 | 4856 | 1848 | pts/0 | S | Dec04 | 0:00 | ssh tux |
| stroetma | 11337 | 0.0 | 0.1 | 4856 | 1848 | pts/1 | S | Dec04 | 0:00 | ssh tux |
| stroetma | 12193 | 0.1 | 3.8 | 46300 | 39692 | ? | S | Dec04 | 5:39 | xemacs -geometry |
| stroetma | 17675 | 0.0 | 0.2 | 4116 | 2828 | ? | S | Dec05 | 0:00 | /usr/bin/ispell - |
| stroetma | 19569 | 0.0 | 0.4 | 9432 | 5120 | ttyp2 | S | 15:09 | 0:00 | xdvi -name xdvi - |
| stroetma | 19572 | 0.2 | 0.8 | 19972 | 8324 | ttyp2 | S | 15:09 | 0:29 | gs -sDEVICE=xll - |
| stroetma | 19685 | 0.0 | 0.1 | 2844 | 1128 | ? | S | 15:56 | 0:00 | /usr/bin/esd -ter |
| stroetma | 19710 | 0.0 | 0.1 | 2832 | 1464 | ttyp6 | S | 16:00 | 0:00 | /bin/bash -i |
| stroetma | 20028 | 0.0 | 0.2 | 6516 | 2832 | ? | S | 18:44 | 0:00 | xterm -e top |
| stroetma | 20030 | 0.3 | 0.0 | 1872 | 936 | pts/2 | S | 18:44 | 0:02 | top |
| stroetma | 20087 | 0.0 | 0.0 | 2668 | 716 | ttyp6 | R | 18:56 | 0:00 | ps ux |

Figure 40: Output of the Command “`ps ux`”.

This output gives us an association between the name of a command that is running

⁶Note that when you use a shell in *XEmacs* you have to precede this with a `Ctrl-Q`, for otherwise the `Ctrl-Z` is swallowed by *XEmacs* and is never seen by the shell.

and its *process identifier*. This process identifier is useful if we need to terminate a process. The command

```
kill n
```

terminates the process with the process identifier *n*. You can also kill a process by its name. This is done with the command “killall”. For example, typing

```
killall xdvi.bin
```

would kill the `xdvi` process that is running. Observe the difference between killing a process by specifying its *job number* or via its *process identifier*. In contrast to the process identifier, the job number has to be preceded by a “%” character if used in the `kill` command.

The command “`ps ux`” lists only the processes that the user has started. In order to list all processes, the command `ps` has to be invoked as

```
ps ax
```

You may have noted that the names of the commands that invoked the processes have been truncated in the output of “`ps ux`”. In order to suppress this truncation, use the command

```
ps axww
```

Here the two occurrences of the letter “w” specify that a wide listing shall be used.

7.3 Quoting and Command Expansion

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for these characters, to prevent reserved words from being recognized as such, and to prevent variable expansions.

Each of the characters listed below has special meaning to the shell and must be quoted if it is to represent itself.

```
| & ; ( ) < > ' ` !
```

There are three quoting mechanisms:

1. The backslash “\” escape character, which preserves the literal value of the next character that follows.
2. Enclosing strings in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when this single quote is preceded by a backslash.
3. Enclosing strings in double quotes preserves the literal value of all enclosed characters within the quotes, with the exception of the characters “\$”, “`” (this is called *backquote*), and the backslash character “\”. The characters “\$” and backquote retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters:

```
$ ` " \ <newline>
```

Next, you might ask what the backquote character is needed for. It is used for *command substitution*. Consider the following command

```
echo "The date is `date`."
```

Here, the output of the command `date` is substituted for the backquoted command `date`, so the output is something like

```
The date is Don Feb 6 21:48:25 CET 2003
```

Let us consider an example that is less contrived. Consider the command “`fgrep`” that

searches its standard input for a given string. It can also look for a whole set of strings when called as

```
fgrep -F string-list
```

This would search its standard input for all strings appearing in the newline separated list of strings specified as argument *string-list*. Assume now that this list of strings is the output of a command. We cannot use a pipe here since the standard input of `fgrep` is already used as the stream that is to be searched. But if *string-list* is the output of *cmd*, then we can write

```
fgrep -F "`cmd`"
```

Note that the list of strings produced by *cmd* has to be enclosed in double quotes. Otherwise only the first string would be interpreted as argument for the “-F” option. The command given above could then be a part of some more complex pipe. Let us make this example more concrete. Assume a file `names.data` of the form below to be given:

```
Bill Gates, Microsoft Foundation
Louis Cypher, Heating Incorporated
George W. Bush, Society for Modern Colonialism
...
```

This file contains names of persons separated by a comma from the company these persons work for. Additionally, assume a `passwd` file as shown when discussing the `cut` command to be given. Suppose we would like to get the user names for all persons listed in the file `names.data` that have an account in our `passwd` file. One way to do this is to `grep` for every person mentioned in `names.data` in `passwd`. The following pipe accomplishes this:

```
cat /etc/passwd | fgrep -F "`cut -d ',' -f1 names.data`" \
| cut -d ':' -f1
```

Note the line continuation character “\” at the end of the first line. The pipe is really one line, but since it does not fit in one line, I had to use the line continuation character. Let us explain this pipe command by command:

1. “`cat /etc/passwd`” starts the pipe by feeding the file “`/etc/passwd`” into it.
2. “`cut -d ',' -f1 names.data`” selects the first column, that is the column containing the name, from the file `names.data`
3. “`fgrep -F "`cut -d ',' -f1 names.data`”`” searches for these names in its input. It produces those lines from `/etc/passwd` that contain these names.
4. “`cut -d ':' -f1`” selects the first column from those `/etc/passwd` lines that are fed into it.

Besides the backquotes, the `bash` has another mechanism that can be used for *command substitution*. The last example can also be written as

```
fgrep -F "$(cut -d ',' -f1 names.data)"
```

The `$(cmd)` syntax has the advantage that it can be nested: *cmd* can itself contain expressions of the form `$(sub-cmd)`. These subcommands are then evaluated before *cmd* is evaluated. This feature is most often used in *obscure programming contests*. The goal in an *obscure programming contest* is to write a program that somehow works but nobody knows why.

7.4 Shell Scripts

A group of commands that is used frequently can be collected in a *shell script*. For example, I frequently view postscript files with “`ghostview`”. If these files are compressed,

I have to uncompress them first. This is done by the program “gunzip”. Since it would be boring to type these two commands to the shell, I have written a small script that does both. The script is shown in Figure 41.

```
#!/bin/bash
# uncompress a file and display it using ghostview
gunzip --stdout $1 | ghostview -
```

Figure 41: An almost trivial script.

The first line of this script says that this script shall be interpreted by the bash. In a shell script, the character “#” starts a comment. A comment extends until the end of the line, but there is one exception to this rule: If this is the first line and if, furthermore, the character immediately following the “#” is the character “!”, then the remainder of the line has to specify an *interpreter* for the script. Therefore, in the example above, we specify the executable “/bin/bash” as the interpreter for this script.

The second line of this script is just a comment describing the purpose of this script. It is good practice to let every script start with a short comment describing its purpose. This is necessary since even small scripts are hard to read and it is not uncommon that after a month the author of a script himself can no longer figure out what the script was supposed to do.

Let us move to the third line of the script. This line implements a simple pipe. The one thing that is peculiar about this pipe is the string “\$1”. In the last subsection we have already seen that any string starting with the character “\$” refers to a variable. The name of the variable consists of all alphanumeric characters and underscores that follow the “\$” character. Therefore, “\$1” refers to the variable with the name “1”. Now this variable is special: Its value is the first argument given to the script. Similarly, “2”, “3”, ... refer to the second, third, and following arguments. Up to nine arguments can be referenced this way. The number “0” refers to the *name* of the script itself. Therefore, if the script in Figure 41 is saved in a file with the name “gvcps” and is called as

```
gvcps someFile.ps.gz
```

then the value of the variable “1” is “someFile.ps.gz”. As a result, when the second line of the script in Figure 41 on page 86 is executed, this is equivalent to the command

```
gunzip --stdout someFile.ps.gz | ghostview -
```

The first program in this pipe is “gunzip”. This program can be used to uncompress files. In Linux, it is a convention that if a file ends with “.gz” then it contains data that has been compressed with the program “gzip”. In order to uncompress these files, the program “gunzip” can be used. The parameter “--stdout” tells “gunzip” to write its output to the standard output. This way it can be used as the source of a pipe.

Next, the program `ghostview` reads its input, which must be a postscript file and displays it on the screen. The parameter “-” tells “ghostview” to read its input from the standard input instead of reading it from a file. Therefore, the input of “ghostview” is the output produced by “gunzip”.

Quite a lot of commands provide the options “-” to read from standard input and “--stdout” to write to standard output. These options facilitate using these commands as part of a pipe.

The shell script in Figure 41 on page 86 is almost trivial. Let us extend it so that it takes any number of parameters. The revised script is shown in Figure 42. This script uses a *for loop* to step through a list of files that are to be decompressed and then displayed. The variable “\$*” contains a list of all arguments given to the script. In each iteration of the for loop, the variable “i” is instantiated to the next argument.

If this script is saved in a file “gvcpsmul” and the command

```
gvcpsmul file1.ps.gz file2.ps.gz file3.ps.gz
```

is issued, then the files `file1.ps.gz`, `file2.ps.gz`, and `file3.ps.gz` are decompressed and displayed simultaneously. There is a variant of the *for loop* that goes

```
#!/bin/bash
# uncompress and display lots of files
for i in $*
do
    gunzip --stdout $i | ghostview - &
done
```

Figure 42: A simple script.

without the “in \$*” part. It is excised in the shell script shown in Figure 43 on page 87. Semantically, these script are identical since the notation

```
for i; do
```

is short for

```
for i in $*; do.
```

```
#!/bin/bash
# uncompress and display lots of files
for i
do
    gunzip --stdout $i | ghostview - &
done
```

Figure 43: Another simple script with the same semantics.

Let us discuss a last variation of the script in Figure 43 on page 87. This variation is shown in Figure 44 on page 87. The line containing the for statement has been changed to

```
for i in *.ps.gz
```

Here, “*.ps.gz” is a *wildcard expression* that specifies all those files that end with the string “.ps.gz”. Their mechanics are discussed later.

```
#!/bin/bash
# uncompress and display all compressed postscript files
for i in *.ps.gz
do
    gunzip --stdout $i | ghostview - &
done
```

Figure 44: A script to show all compressed postscript files.

The previous examples show that the shell is similar to a programming language. It has its own structures for looping and transferring control. The structure for transferring control is shown in Figure 45 on page 87.

```
if cond
then
    commands1
else
    commands2
fi
```

Figure 45: The *if-then-else-fi* control construct.

Here, *cond* is some sort of test, *commands₁* is a group of commands that is executed

in case that the test succeeds and *commands₂* is a group of commands that is executed otherwise. Commands to write a test can be written using the command “test” that is build into the *bash*. This command can be used to compare strings or integers. Strings are compared using the binary infix operators “=” and “!=”, while integers are compared via the binary infix operators “-eq”, “-ne”, “-ge”, “-gt”, “-le”, “-lt”. These operators are short for *equal*, *not equal*, *greater or equal*, *greater than*, *less or equal*, and *less than*, respectively. The command “test” can also be used to test a number of properties of files, but this feature is beyond the scope of this manual. Reading the manual page for “test” gives complete information.

Let us see how all this works together. Consider the example script in Figure 46 on page 88. This is a simple script that checks whether it has been called with exactly 2 arguments. If the number of arguments is different from 2, it prints an error message and exits. Otherwise, it thanks the user for having been called correctly.

```
#!/bin/bash
# check whether the script is invoked with 2 arguments
if test $# -ne 2; then
    echo "The script has been called with $# arguments, "
    echo "but it requires to be called with exactly 2 arguments."
    echo "Exiting now ..."
    exit 42
else
    echo "Thanks for calling this script with 2 arguments!"
fi
```

Figure 46: A script checking the number if its arguments.

The final control construct we discuss are *while-loops*. The syntax for *while-loops* is as shown in Figure 47 on page 88. Here *cond* is a test and *commands* is a sequence of commands that is executed as long as the test yields true.

```
while cond
do
    commands
done
```

Figure 47: Syntax of the *while-loop*.

We illustrate the *while* construct via an example: Consider the script in Figure 48. Assume the script is called *gauss*. The script is invoked with one argument. An invocation of the form

gauss *n*
 computes the sum

$$\sum_{i=0}^n i.$$

This script also introduces the concept of *arithmetic expression evaluation*. If the shell is given a construct of the form

$\$(arith-expr)$

then the shell interprets *arith-expr* as an arithmetic expression and substitutes the result of evaluating this expression for the $\$(arith-expr)$ construct. Note that it is **not necessary** to prefix the value of a variable occurring in *arith-expr* with a “\$” character.

This feature is also useful interactively since it provides a simple calculator facility

on the command line. For example, typing

```
echo $((36 * 37 / 2))
```

yields the expected result.

```
#!/bin/bash
# compute the sum 1 + 2 + 3 + ... + n, where n is the given argument
sum=0
i=0
while test $i -le $1
do
    sum=$((sum+i))
    i=$((i+1))
done
echo "The sum is $sum"
```

Figure 48: A script computing $\sum_{i=0}^n i$ for given n .

We discuss one variation of the shell script for summing the values from 1 to n . The script shown in Figure 49 on page 89 has the same semantics as the previous script shown in 48 on page 89. The only difference is that it is written in a more compact style by writing several commands on a single line. In order for this to work, the different commands have to be separated by “;” characters.

```
#!/bin/bash
# compute the sum 1 + 2 + 3 + ... + n, where n is the given argument
sum=0; i=0
while test $i -le $1; do
    sum=$((sum+i)); i=$((i+1))
done
echo "The sum is $sum"
```

Figure 49: A more compact script computing $\sum_{i=0}^n i$.

Concerning the syntax of variable assignments, there is one fine point that often troubles beginners: In a variable assignment of the form

```
var=value
```

no space is permitted between either *var* and the “=” character nor between the “=” character and *value*. For example, writing

```
sum = 0
```

will produce the error message

```
sum: =0: No such file or directory
```

The reason is that the line above is interpreted as the invocation of a command `sum` with the argument `=0`. Since there is no such command, an error is raised.

Unfortunately, describing the shell as a programming language requires a book of its own. For lack of time we stop our treatment here. For further study I recommend the info pages describing the `bash` and Chapters 20 and 21 of [5].

8 Regular Expressions

You recognize a Linux professional by his or her ability to use *regular expressions*. The concept of regular expressions is so powerful that entire scripting languages have been based on regular expressions. The most prominent examples are *Tcl* [12, 7], *Perl* [8], and *Python* [6]. Of these, *Tcl* is the most powerful and will be discussed in one of the following lectures. The reason we did not discuss shell programming in more detail is due to the fact that it is usually much easier to write a script in any of these scripting languages than to write a shell script.⁷

Basically, regular expressions are a mathematical notation to specify *sets* of strings. One program that makes use of regular expressions is `grep`. The command

```
grep regexp file1 ... filen
```

searches the specified files for all occurrences of strings that *match* the regular expression *regexp*.

How do we specify a regular expression? In a regular expressions a character is either a *special character* or a *normal character*. A normal character just denotes itself. For example, the letters and digits are normal characters. Therefore the regular expression “abc” matches exactly the string “abc”. Not very exciting, but we haven’t discussed the special characters yet. These are the following:

. * + ? [] ^ \$ \

We discuss their purpose next:

1. “.” is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like “a.b”, which matches any three-character string that begins with “a” and ends with “b”.
2. “*” is not a construct by itself; it is a *quantifying suffix operator* that means to repeat the preceding regular expression as many times as possible. In “fo*”, the “*” applies to the “o”, so “fo*” matches one “f” character followed by any number of “o” characters. The case of zero “o” characters is allowed: “fo*” does match a single “f” character.
3. “+” is a quantifying suffix operator similar to “*” except that the preceding expression must match at least once. So, “fo+” would not match a single “f” character, but otherwise would match the same strings as “fo*”.
4. “?” is also a quantifying suffix operator similar to “*”, except that the preceding expression can match either once or not at all. For example, “fo?” matches “f” and “fo” but nothing else.

The *quantifiers* “*”, “+”, and “?” always apply to the smallest possible preceding expression. Thus, “fo+” has a repeating “o”, not a repeating “fo”. If the latter is needed, subexpressions can be put in escaped parentheses as in “\ (fo\)+”. This would match the strings “fo”, “fofo”, “fofofo”, and so on. Surrounding subexpressions by backslash escaped parenthesis is known as *grouping* subexpressions.

5. “\|” specifies an alternative. Two regular expressions A and B with “\|” in between form an expression that matches anything that either A or B matches. Thus, “fo\|bar” matches either “foo” or “bar” but no other string. “\|” applies to the largest possible surrounding expressions. Only a surrounding “\ (” ... “\)” grouping can limit the grouping power of “\|”.

⁷Still, some acquaintance with shell programming is indispensable because there are a lot of shell scripts around which have to be maintained by somebody.

6. “^” is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, “^foo” only matches a “foo” if it occurs at the beginning of a line.
7. “\$” is a special character that matches the empty string, but only at the end of a line in the text being matched. Otherwise it fails to match anything. Thus, “foo\$” only matches a “foo” if it occurs at the very end of a line.

The special characters “^” and “\$” are known as *anchors*.

8. “[” starts a *character set*, which is terminated by a “]”. In the simplest case, the characters between the two brackets form the set. Thus, “[ad]” matches either one “a” or one “d”, and “[ad]+” matches any nonempty string that is solely composed of the characters “a” and “d”.

The usual regular expression *special characters* are not special inside a character set. A completely different set of special characters exists inside character sets: The three characters “]”, “-”, and “^”.

- (a) “-” is used to specify a *range* of characters. To write a range, write two characters with a “-” between them. For example, “[a-z]” matches any lower case letter. Ranges may be intermixed freely with individual characters, as in “[a-z*.]”, which matches any lower case letter, the character “*”, or the character “.”.
- (b) “]” is used to terminate a character set. To include a literal “]”, make it the first character.
- (c) “^” is used to negate a character set. This, “[^0-9]” matches all characters that are not digits.

Often, regular expressions are used to replace substrings with different substrings. The command “sed” can be used in this fashion. If it is called as

```
sed s/regexp/replacement/g file
```

then it replaces all occurrences of the regular expression *regexp* in *file* with *replacement*. In *replacement*, the character “\” is special: If it is followed immediately by the digit *n*, then it specifies the *n*-th parenthesized subexpression. To illustrate this, consider a file written in \LaTeX that contains expressions of the form “\emph{string}”. We want to convert these expressions into a *Html* format. In this format the string “\emph{string}” has to be replaced by “string”. The following command performs this transformation:

```
sed 's/\\emph{\\([\\^]\\*\\)}/<em>\\1</em>/g'
```

Admittedly, this is hard to read. Even worse is the fact that the regular expression written here does not conform to the syntax that is described above. The difference is that *sed* requires that the “+” character is escaped by a backslash. Above, I have described the syntax of regular expressions as it is in *XEmacs*. The commands “grep” and “sed” require the quantifiers “+”, “*”, and “?” to be backslash escaped if used as a quantifier.

Nevertheless, let us try to make sense of the regular expression

```
\\emph{\\([\\^]\\*\\)}
```

that is used above. First, the double backslash is needed to match a single backslash. Then, we have the string “emph{” which matches itself. This is followed by “\(\(\dots\)”. The backslash escaped opening and closing parentheses are needed to *group* what is inside so that we can later reference it in the *replacement* string. Next, inside the parentheses we have the *quantified set expression*

```
[^}]\*
```

Here, “[^}” is a set expression that matches any character that is different from the character “}”. The quantifier “*” specifies that the string inside the curly braces can have arbitrary many such characters, including none at all. In effect, this part of the regular expression matches all symbols that appear between the opening brace and the closing brace in an expression of the form “\emph{string}”.

Finally, let us look at the *replacement* string

```
<em>\1</em>
```

The character sequence “\1” is replaced by the actual match of the part of the regular expression that had been enclosed in backslash escaped parentheses. The strings “” is taken literally and in the string “” the backslash is needed to escape the slash character “/”. This character needs to be backslash escaped since it delimits the end of the replacement string in the `sed` command.

The *XEmacs* command “query-replace-regexp” works in a way similar to the above “sed” command. It prompts the user first for a regular expression and then for a replacement string, where in the replacement string the character “\” is special: If it is followed immediately by the digit *n*, then it specifies the *n*-th parenthesized subexpression.

9 The find command

This introduction to Linux would be incomplete without a short introduction to the `find` command. It is called as follows

```
find [directory] [options]
```

The purpose of `find` is to search for files that are either in *directory* or in any of its subdirectories. If *directory* is omitted, then the search starts in the current working directory. The argument *options* specifies the files that are searched for. Furthermore, we can specify certain *actions* that should be performed for these files. The most simple action is just to print the file name. This action is specified via the option “-print”. This is also the default action, i.e. the action that is performed when no other action is specified. Therefore, the command

```
find
```

prints the names of all files in the current directory and all of its subdirectories. There are a number of options to restrict the search. We list the most important ones.

1. “-name *search-pattern*” where *search-pattern* can contain wildcard symbols. This would restrict the search to all those files whose name matches the given *search-pattern*. However, if *search-pattern* contains wildcard symbols, then we must protect these wildcard symbols from being expanded by the shell. This is done by enclosing them in single quotes. The following example demonstrates this:

```
find /home/karl/PDV -name '*.tex' -print
```

This command would search the directory “/home/karl/PDV” for all files that end in “.tex”.

2. “-regexp *regexp-pattern*” where *regexp-pattern* is a regular expression. This would restrict the search to all those files whose name matches the given regular expression. Now, if *regexp-pattern* contains characters that can be interpreted by the shell as wildcards, then we must protect these characters from being expanded by the shell by putting *regexp-pattern* in single quotes.

3. “-user *username*” where *username* is the name of a user. This would search for files belonging to the specified user. For example

```
find ~ -user root
```

would search my home directory and all of its subdirectories for files belonging to root.

4. “-size *size-spec*” searches for all files whose size matches the given size specification. For example,

```
find -size +42k
```

searches for all files that are larger than 42 kilobytes (that is what the “+” stands for), while

```
find -size -42k
```

would search for all files that are less than 42 kilobytes and

```
find -size 42k
```

would search for files having a size of exactly 42 kilobytes.

Next, we list the actions that are most useful.

1. “-exec *cmd opts* {} \;” executes the command *cmd* with given options *opts* for every files that is found. Here, the path leading to the file is substituted for the string “{}”. The semicolon “;” is needed to terminate the command, while it needs to be backslash escaped in order to prevent the shell from swallowing it. A typical example would be the following invocation

```
find -name '*.tex' -exec grep regexp {} \;
```

This would look for all “.tex” files in the current directory and its subdirectories and would run the grep command

```
grep regexp file
```

on all files that it finds.

2. “-ok *cmd opts* {} \;” is similar to the above: It executes the command *cmd* with given options *opts* for every files that is found. However, before executing any command, the user is questioned whether the command shall indeed be executed. This is often used in conjunction with the rm command, a typical invocation is

```
find -name core -ok rm {} \;
```

This deletes all files with the name “core” but asks for confirmation before doing a deletion.

One nuisance with the execution of commands by find is the fact that all commands are executed in the directory where find is started and not in the directory where the file is found. Consider the following situation: An instructor has asked his students to pack the solution to their exercise into a tar file, to uuencode this and mail it to the instructor. The instructor has already stored these files in one big directory “Exercises” where there is a subdirectory for every student containing a file called “exercise”, which is the uuencoded tar file the students have mailed. Next, he wants to uuencode the “exercise” file into a file “exercise.tar”, which he then wants to unpack using “tar

xf”.

The command “tar” is a simple way to pack a directory and all its files and subdirectories into a single file, which is also known as a *tape archive*. It is called as

```
tar -c -f archive directory
```

This creates a tape archive from *directory* and all files found in this directory or any of its subdirectories. This tape archive is then saved in the file *archive*. In order to get back files that have been archived, use the command

```
tar xf archive
```

It assumes *archive* to be a tape archive and extracts all the directories and files stored in it.

Email is sent using ASCII characters. Since these characters are only 7 bits wide but the characters in normal files are 8 bits wide, files have to be preprocessed before they can be transferred by email. This is done using the command

```
uencode file name
```

Here *file* is the file that is to be transformed into a seven bit representation and *name* is needed later for the inverse operation. The encoded file is written to the standard output. The inverse operation is `udecode`, which reconstructs the original eight bit file. It is called as

```
udecode file
```

This takes *file* as the file to be transformed and stores it under the name given as the parameter *name* when uuencoding the file.

In a first attempt to solve his problem, the instructor could visit the directory `Exercises` and try the command

```
find . -name exercise -exec udecode -o exercise.tar {} \;
```

Unfortunately, this will create only one file “`exercise.tar`” and this file will be in the directory `Exercises`. The reason is, that the command “`udecode -o exercise.tar`” is always executed in the directory `Exercises` and not in the subdirectories where the files are found. However, the following command will work provided that `uudetar` is the script that is shown in Figure 50 on page 94.

```
find . -name exercise -exec uudetar {} \;
```

Let us discuss this script line by line.

```
#!/bin/bash
# switch to the appropriate directory and call udecode there
cd `dirname $1`
udecode -o exercise.tar `basename $1`
tar xf exercise.tar
```

Figure 50: A script to unpack uuencoded tape archive.

The music starts to play in the third line. The purpose of the third line is to change the working directory to the local directory where the particular file “`exercise`” resides. How do we get the directory? The answer is the command “`dirname`” that takes a full name consisting of a path and a file and returns just the path. For example

```
dirname /home/karl/PDV/script.tex
```

returns the string “`/home/karl/PDV`”. In the script, this command is backquoted, so that its result is substituted as argument to the `cd` command. The “`$1`” expands to the argument that is given to the script when it is invoked by `find`.

The fourth line invokes the “`udecode`” command. This needs the filename without the path. Here the command “`basename`” comes in handy. It is the opposite of “`dirname`”, for example

```
basename /home/karl/PDV/script.tex
```

returns the string “`script.tex`”. Therefore, the fourth line converts the file 7-bit file “`exercise`” back into the 8-bit file “`exercise.tar`”. Finally, the last line extracts ev-

erything from this tape archive.

10 Conclusion

For today, we have come to the end of our adventures in *Linux Country*, where the programmers still run free, do not have to pay **Bills**, and will never be fenced in by **Gates**. We have touched about 5 % of the commands available under Linux. For most commands we were only able to discuss about 5 % of their options. So there is much left to discover. A good starting point for further studies are the books by Ball [1] and by Kofler [5]. These are introductory books. The book by Siever et. al. [9] is more complete but also much more condensed. Finally, the places to learn all the details are the manual pages and the info pages.

References

- [1] BILL BALL: *Sams Teach Yourself SuSE Linux in 24 Hours*, Sams Publishing, 1999.
- [2] DEBRA CAMERON, BILL ROSENBLATT, ERIC S. RAYMOND: *Learning Gnu Emacs*, O'Reilly & Associates, 2nd edition, 1996.
- [3] DALE DOUGHERTY AND ARNOLD ROBBINS *sed and awk*, O'Reilly & Associates, 1997.
- [4] BOB GLICKSTEIN *Writing Gnu Emacs Extensions: Editor Customizations and Creations with Lisp*, O'Reilly & Associates, 1997.
- [5] MICHAEL KOFLER: *Linux — Installation, Konfiguration, Anwendung*, Addison-Wesley, 2002.
- [6] MARK LUTZ AND DAVID ASCHER: *Learning Python*, O'Reilly & Associates, 1999.
- [7] JOHN K. OUSTERHOUT: *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- [8] RANDAL L. SCHWARTZ AND TOM PHOENIX: *Learning Perl*, O'Reilly & Associates, 2001.
- [9] ELLEN SIEVER, STEPHEN SPAINHOUR, STEPHEN FIGGINS, JESSICA P. HEKMAN: *Linux in a Nutshell*, O'Reilly & Associates
- [10] RICHARD M. STALLMAN: *Gnu Emacs Manual*, Free Software Foundation, 2002.
- [11] LARRY GREENFIELD: *The Linux Users' Guide*, 1993–1996
An online version of this book is part of the SUSE Linux distribution.
- [12] BRENT B. WELCH AND JEFFREY HOBBS: *Practical Programming in Tcl and Tk*, Prentice Hall PTR, 2003.

Index

- 666..... 85
- asynchronous execution..... 79
- background execution 79
- character translation 73
- command seperator..... 77
- compiling programs..... 60
- compressing data 83
- Ctrl key..... 8
- cursor 6
- debugger 62
- directory 43
 - copying..... 47
 - creation 46
 - deleting 46, 48
 - home directory..... 44
 - listing 46, 48
 - parent directory..... 43
 - recursive listing..... 49
 - removing..... 46, 48
 - root directory 43
 - shorthands..... 45
 - sorted listing..... 49
 - subdirectory 43
 - working..... 44, 45
- end-of-file character 69
- file..... 43
 - access control..... 49
 - access mode 49
 - changing ownership..... 52
 - changing permissions..... 50
 - changing the group..... 52
 - copying..... 46, 47
 - creation 47
 - deleting..... 46, 48
 - displaying..... 47
 - execute permission..... 50
 - group..... 49
 - hidden 48, 53
 - link counter 49, 54, 55
 - links 54
 - listing..... 47
 - mode 49
 - modification..... 49
 - modification date 49
 - owner..... 49
 - read permission 50
 - removing..... 46, 48
 - renaming..... 46
 - size 49
 - type..... 48
 - write permission..... 50
- filter..... 69
- gdb..... 62
 - breakpoints
 - deleting..... 66
 - setting..... 65
 - commands..... 65
 - single stepping..... 66
 - up..... 63
- George W. Bush..... 3
- GNOME 3
- help..... 67
- hotplug 58
- inode..... 54
- job control..... 79
 - job number..... 79
- KDE..... 3
- Linux commands*
 - apropos..... 68
 - basename 90
 - cat..... 47, 71
 - cd..... 47
 - chgrp..... 52
 - chmod..... 50
 - chown..... 52
 - cp..... 46, 47
 - cut..... 72
 - dirname..... 90
 - echo..... 53
 - fgrep..... 75
 - find..... 88
 - ghostview..... 83
 - grep..... 86
 - gunzip..... 83
 - gzip..... 83
 - kill..... 80
 - killall..... 80
 - ln..... 54
 - ls..... 46, 48
 - man..... 68
 - mkdir..... 46
 - mount..... 55

| | | | |
|-------------------------------|--------|---------------------------------|--------|
| mv..... | 46 | segmentation fault..... | 63 |
| paste..... | 72 | shell..... | 6, 77 |
| ps..... | 80 | arithmetic expresions..... | 85 |
| pwd..... | 46 | command substitution..... | 81 |
| rm..... | 46, 48 | comparison..... | 83 |
| rmdir..... | 46 | exporting a variable..... | 78 |
| sed..... | 75, 87 | global variable..... | 78 |
| sort..... | 69 | if then else..... | 84 |
| susehelp..... | 68 | if-then-else..... | 83 |
| tar..... | 89 | local variable..... | 78 |
| tee..... | 75 | prompt..... | 79 |
| touch..... | 47 | quoting special characters..... | 81 |
| tr..... | 73 | shell variable..... | 78 |
| umount..... | 58 | test..... | 83 |
| uudecode..... | 89 | variable assignment..... | 78 |
| uuencode..... | 89 | variables..... | 78 |
| LORD..... | 9 | #..... | 79, 84 |
| magic..... | 3 | \$..... | 79 |
| manual page..... | 67 | *..... | 79, 83 |
| moron..... | 6 | ?..... | 79 |
| mount..... | 55 | HOME..... | 78 |
| path name..... | 44 | HOSTNAME..... | 79 |
| absolute..... | 44 | LOGNAME..... | 78 |
| relative..... | 44 | MANPATH..... | 78 |
| pipes..... | 71 | PATH..... | 78 |
| political correct..... | 17 | PS1..... | 79 |
| process | | while loop..... | 84 |
| bringing into foreground..... | 80 | shell scripts..... | 82 |
| killing..... | 80 | arguments..... | 82 |
| sending to background..... | 80 | comments..... | 82 |
| process identifier..... | 79 | for loop..... | 83 |
| regular expressions..... | 86 | name..... | 82 |
| anchor..... | 87 | standard error..... | 70 |
| character set..... | 87 | standard input..... | 69 |
| grouping subexpressions..... | 86 | redirection..... | 70 |
| quantifier..... | 86 | standard output..... | 69 |
| range..... | 87 | redirection..... | 70 |
| replacement..... | 87 | stream editor..... | 75 |
| special characters..... | 86 | super key..... | 26 |
| \$..... | 87 | tape archive..... | 89 |
| *..... | 86 | Teletubbies..... | 3 |
| +..... | 86 | uncompressing data..... | 83 |
| | 86 | wildcards..... | 52, 83 |
| ?..... | 86 | Windows key..... | 26 |
| [..... | 87 | working directory | |
|]..... | 87 | changing..... | 47 |
| ^..... | 87 | printing..... | 46 |
| | 86 | <i>XEmacs</i> | 5 |
| suffix operator..... | 86 | abbreviations..... | 25 |
| SATAN..... | 9 | buffer..... | 13 |

| | | | |
|-----------------------------------|--------|----------------------------|----|
| command | 11 | replacement..... | 17 |
| completion..... | 23 | searching | |
| invocation on startup..... | 27 | incremental..... | 16 |
| name..... | 11 | show line numbers | 40 |
| quit..... | 12 | splitting the screen..... | 19 |
| command line..... | 9 | starting a shell..... | 62 |
| compiling programs | 60 | transpose characters | 40 |
| completion | 21 | undo..... | 19 |
| copying text..... | 13 | universal-argument | 12 |
| creation..... | 9 | variables | 26 |
| cursor movement..... | 12 | setting..... | 26 |
| customizing..... | 22, 26 | yanking text..... | 13 |
| key bindings | 26 | | |
| directory mode..... | 58 | | |
| dirred..... | 58 | | |
| echo area | 9 | | |
| exit | 12 | | |
| file | | | |
| open | 7 | | |
| saving | 9 | | |
| formatting a paragraph | 40 | | |
| games | 42 | | |
| doctor | 42 | | |
| hanoi..... | 42 | | |
| tetris | 42 | | |
| help..... | 32 | | |
| invoking | 6 | | |
| key binding | 26 | | |
| changing..... | 26 | | |
| key bindings | | | |
| listing | 32 | | |
| key combination | 7 | | |
| keyboard macro | | | |
| editing..... | 31 | | |
| searching | 19 | | |
| keyboard macros | 27 | | |
| interactive | 28 | | |
| traps and pitfalls | 29 | | |
| kill ring..... | 13 | | |
| mark..... | 13 | | |
| reactivation..... | 30 | | |
| menu bar..... | 11 | | |
| mini buffer | 9 | | |
| minibuffer completion..... | 22 | | |
| mode line | 9 | | |
| change info..... | 9 | | |
| editing mode | 9 | | |
| encoding..... | 9 | | |
| position..... | 9 | | |
| multiple files, editing..... | 19 | | |
| point..... | 13 | | |
| query replace regular expressions | 88 | | |
| recursive editing..... | 19 | | |
| region..... | 13 | | |