Slackware Linux Basics

For Slackware Linux 12.0

Daniël de Kok

Slackware Linux Basics: For Slackware Linux 12.0

by Daniël de Kok

Published Sun May 18 14:41:12 CEST 2008
Copyright © 2002-2008 Daniël de Kok

License

Table of Contents

List of Figures

List of Tables

Preface

This book aims to provide an introduction to Slackware Linux. It addresses people who have little or no GNU/Linux experience, and covers the Slackware Linux installation, basic GNU/Linux commands and the configuration of Slackware Linux. After reading this book, you should be prepared to use Slackware Linux for your daily work, and more than that. Hopefully this book is useful as a reference to more experienced Slackware Linux users as well.

Thanks to the rapid development of open source software, there are now comprehensive desktop environments and applications for GNU/Linux. Most current distributions and books focus on using GNU/Linux with such environments. I chose to ignore most of the graphical applications for this book, and tried to focus this book on helping you, as a reader, to learn using GNU/Linux in a more traditional UNIX-like way. I am convinced that this approach is often more powerful, and helps you to learn GNU/Linux well, and not just one distribution or desktop environment. The UNIX philosophy is described in the overview of UNIX philosophy

I wish everybody a good time with Slackware Linux, and I hope that you will find this book is useful for you.

Daniël de Kok

Part I. Getting started

Table of Contents

Chapter 1. About this book

1.1. Availability

This book was written in DocBook/XML, and converted to HTML and XSL:FO with **xsltproc**. The latest version of the book is always available from: http://www.slackbasics.org/.

1.2. Conventions

This section gives a short summary of the conventions in this book.

File names

File or directory names are printed as: `/path/to/file`. For example: `/etc/fstab`

Commands

Commands are printed as bold text. For example: **ls -l**

Screen output

Screen output is printed like this:

```
Hello world!
```

If commands are being entered in the screen output the commands will be printed as bold text:

```
$ command
Output
```

If a command is executed as root, the shell will be displayed as "#". If a command is executed as a normal non-privileged user, the shell will be displayed as "$".

Notes

Some sections contain extra notes. It is not necessary to know the information in notes, but notes may provide valuable information, or pointers to information. Notes are printed like this:

Note

This is a note.

Chapter 2. An introduction to Slackware Linux

2.1. What is Linux?

Linux is a UNIX-like kernel, which is written by Linus Torvalds and other developers. Linux runs on many different architectures, for example on IA32, IA64, Alpha, m68k, SPARC and PowerPC machines. The latest kernel and information about the Linux kernel can be found on the Linux kernel website: http://www.kernel.org.

The Linux kernel is often confused with the GNU/Linux operating system. Linux is only a kernel, not a complete operating system. GNU/Linux consists of the GNU operating system with the Linux kernel. The following section gives a more extensive description of GNU/Linux.

2.2. What is GNU/Linux?

In 1984 Richard Stallman started an ambitious project with the goal to write a free UNIX-like operating system. The name of this system is GNU, which is an acronym of "GNU's Not UNIX". Around 1990, all major components of the GNU operating system were written, except the kernel. Two years earlier, in 1988, it was decided that the GNU project would use the Mach 3.0 microkernel as the foundation of its kernel. However, it took until 1991 for Mach 3.0 to be released under a free software license. In the the same year Linus Torvalds started to fill the kernel gap in the GNU system by writing the Linux kernel. GNU/Linux thus refers to a GNU system running with the Linux kernel.

The GNU kernel, named "HURD" was still under development when this book was written, and is available as the GNU/HURD operating system. There are some other kernels that are ported to the GNU operating system as well. For instance, the Debian project has developed a version of the GNU operating system that works with the NetBSD kernel.

2.3. What is Slackware Linux?

Slackware Linux is a GNU/Linux distribution, which is maintained and developed by Patrick Volkerding. A distribution is a coherent collection of software that provides a usable GNU/Linux system. Volkerding started using GNU/Linux because he needed a LISP interpreter for a project. At the time the dominant GNU/Linux distribution was Softlanding System Linux (SLS Linux). Slackware Linux started out as a private collection of Volkerding's patches for SLS Linux. The first publicly available Slackware Linux release was 1.0, which was released on July 16, 1993.

In contrast to many other GNU/Linux distributions, Slackware Linux adheres to the so-called KISS (Keep It Simple Stupid) principle. This means that Slackware Linux does not have complex graphical tools to configure the system. As a result the learning curve of Slackware Linux can be high for inexperienced GNU/Linux users, but it provides more transparency and flexibility. Besides that you get a deeper understanding of GNU/Linux with no-frills distributions like Slackware Linux.

Another distinguishing aspect of Slackware Linux, that also "complies" with the KISS principle, is the Slackware Linux package manager. Slackware Linux does not have a complex package manager like RPM or dpkg. Packages are normal tgz (tar/gzip) files, often with an additional installation script and a package description file. For novice users tgz is much more powerful than RPM, and avoids dependency problems. Another widely known feature of Slackware Linux is its initialization scripts. In contrast to most other GNU/Linux distributions Slackware Linux does not have a directory for each runlevel with symbolic links to services that have to be started or killed in that runlevel. It uses a simpler approach in which you can enable or disable services by twiddling the executable bit of an initialization script.

The packages in Slackware Linux are compiled with as little modifications as possible. This means you can use most general GNU/Linux documentation.

2.4. The UNIX philosophy

Since GNU/Linux is a free reimplementation of the UNIX operating system, it is a good idea to look at the philosophy that made UNIX widely loved. Doug McIlroy summarized the UNIX philosophy in three simple rules:

* Write programs that do one thing and do it well.

* Write programs to work together.

* Write programs to handle text streams, because that is a universal interface.

Odds are that you do not intend to write programs for GNU/Linux. However, even as a user these basic UNIX rules can mean a lot to you. Once you get to know the essential commands that have been part of UNIX for many years, you will be able to combine simple programs to solve complex problems. Keep this in mind while you learn Slackware Linux; try to get a feeling for how you can divide complex tasks in simple combined operations.

2.5. Free and open source software

Most packages in Slackware Linux are published under a free software or open source license. Under these licenses software may be used, studied, changed and distributed freely. Practically, this means that the software is available and redistributable in source and binary form. Although the free software and open source software movements share many licenses and principles, there are subtle differences between both movements. The open source movement tends to focus on the economic and technical advantages of sharing source code, while the free software movement puts accent on the ethical side of providing sources and binaries freely. As the GNU website puts it: "Free software is a matter of liberty, not price. To understand the concept, you should think of free as in free speech, not as in free beer.[1]" In the spirit of free and open source software the source code of almost all packages is included in the official Slackware Linux CD set or DVD.

2.6. Slackware Linux 12.0 features

* **Linux 2.6.21.5** - Slackware Linux uses as modern high-performance Linux kernel. The kernel includes support for all modern disk controllers, LVM, Software RAID, encrypted disks, and multiple processors/cores. By default, *udev* is enabled for automatic management of device nodes.

* **HAL** - the HAL (Hardware Abstraction Layer) is now included too. This provides a uniform API for desktop applications to use hardware. It makes automatic mounting of disks and CDs considerably easier under Xfce and KDE.

* **X11 7.2.0** - This is the first version of Slackware Linux to use modular X. This means that the X11 components are separated in many small packages for easier maintenance and lighter weight upgrades.

* **GCC 4.1.2** - Slackware Linux 12.0 includes a completely revised toolchain based on the GNU Compiler Collection 4.1.2. GCC provides C, C++, Objective-C, Fortran-77/95, and Ada 95 compilers. Additionally, version 2.5 of the GNU C library is used.

* **Apache 2.2.4** - Apache was upgraded to a new major version. Apache 2.x is a substantial rewrite of the old 1.3.x series.

* **The K Desktop Environment (KDE) 3.5.7** - The full KDE environment is provided, which includes KOffice, the Konqueror web browser, multimedia programs, development tools, and many more useful applications.

---

[1] http://www.gnu.org/philosophy/free-sw.html

- **Xfce 4.4.1** - Xfce is a lightweight desktop environment based on GTK2. It embodies the UNIX spirit of modularity and reusability.

2.7. Getting Slackware Linux

Slackware Linux is freely downloadable from the official Slackware Linux mirrors. The list of Slackware mirrors is available at http://www.slackware.com/getslack/.

You can also order Slackware Linux as a CD set or DVD from the Slackware Store. Many Internet shops also provide Slackware Linux cheaply on CD-ROM or DVD, but you are only supporting Slackware Linux financially if you buy an official CD set or DVD. The Slackware Store also offers Slackware Linux subscriptions. A subscriber automatically receives new Slackware Linux releases at a reduced price.

If you would like to have more information about purchasing Slackware Linux, visit the Slackware Store website at http://store.slackware.com/.

Chapter 3. Sources of help

There is a wealth of information available about many subjects related to GNU/Linux. Most general documentation applies to Slackware Linux, because the software in the distribution has been compiled from source code that has been altered as little as possible. This chapter provides some pointers to information and documentation that can be found on an installed Slackware Linux system, and on the Internet.

3.1. On your system

Linux HOWTO's

The Linux HOWTOs are a collection of documents which cover specific subjects related to GNU/Linux. Most Linux HOWTOs are not tailored to a specific distribution, therefore they are very useful for Slackware Linux users. The *linux-howtos* package in the "f" software set contains the HOWTO collection. After installing this package the HOWTOs are available from the `/usr/doc/Linux-HOWTOs/` directory. Slackware Linux also contains a small collection of Linux-related FAQs (FAQs are documents that answer Frequently Asked Questions). The Linux FAQs are installed to the `/usr/doc/Linux-FAQs/` directory, and are available from the *linux-faqs* package, which is also part of the "f" software set.

Manual pages

Most UNIX-like commands are covered by a traditional UNIX help system called the *manual pages*. You can read the manual page of a program with the **man** command. Executing **man** with the name of a command as an argument shows the manual page for that command. For instance,

$ **man ls**

shows the manual page of the **ls** command.

If you do not know the exact name of a manual page or command, you can search through the manual pages with a keyword. The `-k` option is provided to make use of this facility:

```
$ man -k rmdir
hrmdir                  (1)  - remove an empty HFS directory
rmdir                   (1)  - remove empty directories
rmdir                   (2)  - delete a directory
```

The manual page collection is very extensive, and covers more subjects than just commands. The following sections of manual pages are available:

1 Executable programs or shell commands
2 System calls (functions provided by the kernel)
3 Library calls (functions within program libraries)
4 Special files (usually found in /dev)
5 File formats and conventions eg /etc/passwd
6 Games

7   Miscellaneous (including macro packages and conven-
    tions), e.g. man(7), groff(7)
8   System administration commands (usually only for root)
9   Linux routines [Non standard]

If there is more than one section that has a manual page with a specific name, as with for instance **rmdir**, you can choose what page you want to see by adding the section number of the manual page before the manual page name. For example:

```
man 2 rmdir
```

If you would like to print a manual page to a printer that you have set up, you can pipe the output of **man** to the **lpr** command. When the *-t* option of the **man** command is used, **man** will output the manual page in Postscript format, rather than ASCII. For example, you can use the following command to print the **cat** manual page:

```
$ man -t cat | lpr
```

3.2. On the Internet

There are many websites and forums related to GNU/Linux and Slackware Linux on the Internet. But many sites often disappear as fast as they appeared, and the information on many web sites is fragmentary. The following resources have been around for a longer time, and provide good content.

The Slackware Linux website

The Slackware Linux website may be a bit outdated at times, but it provides many useful resources:

• A news page that announces new releases and lists other important news that is relevant to Slackware Linux.

• An overview of the changes to the distribution is provided in a structured format called a *ChangeLog*. ChangeLogs are provided for the current development version, as well as the latest stable release.

• There are two mailing lists to which you can subscribe. The *slackware-announce* list is used to announce new Slackware Linux releases, and security updates are announced on the *slackware-security* list.

• A list of mirrors where you can download Slackware Linux. The mirrors are indexed per country. Additional information such as the download protocols the mirrors support, and the speed of the mirrors is also included.

• Documentation of various kinds, including a list of frequently asked questions and the *Slackware Linux Essentials* book.

The Slackware Linux website is available at: http://www.slackware.com/

LinuxQuestions

LinuxQuestions is a large GNU/Linux forum with many helpful members. Particularly interesting is the Slackware Linux subforum, where you can seek assistance to help you with problems that you may have with Slackware Linux. The LinuxQuestions forum is available at: http://www.linuxquestions.org/

alt.os.linux.slackware

alt.os.linux.slackware is a Slackware Linux newsgroup. You can read newsgroups with a newsreader like tin or slrn, through the newsgroup server of your Internet service provider. On this newsgroup it is expected that you have read all necessary documentation before posting questions. If you have not done that, the chance of getting "flamed" is large.

Chapter 4. General concepts

This chapter gives an introduction to some general UNIX and GNU/Linux concepts. It is important to read this chapter thoroughly if you do not have any UNIX or GNU/Linux experience. Many concepts covered in this chapter are used in this book and in GNU/Linux.

4.1. Multitasking

Introduction

One of UNIX's traditional strengths is multitasking. Multitasking means that multiple programs can be run at the same time. You may wonder why this is important, because most people use only one application at a time. Multitasking is a bare necessity for UNIX-like systems. Even if you have not started any applications, there are programs that run in the background. For instance, some programs provide network services, while others show a login prompt and wait until a user logs in on a (virtual) terminal. Programs that are running in the background are often called *daemon processes*[1].

Processes and threads

After a program is loaded from a storage medium, an instance of the program is started. This instance is called a *process*. A process has its own protected memory, named the *process address space*. The process address space has two important areas: the *text* area and the *data* area. The *text* area is the actual program code; it is used to tell the system what to do. The *data* area is used to store constant and runtime data. The operating system gives every process time to execute. On single processor systems processes are not really running simultaneously. In reality a smart scheduler in the kernel divides CPU time among processes, giving the illusion that processes run simultaneously. This process is called *time-sharing*. On systems with more than one CPU or CPU cores, more than one process can run simultaneously, but time-sharing is still used to divide the available CPU time among processes.

New processes are created by duplicating a running process with the **fork** system call. Figure 4.1, "Forking of a process" shows a fork() call in action schematically. The parent process issues a fork() call. The kernel will respond to this call by duplicating the process, and naming one process the *parent*, and the other process the *child*.

Figure 4.1. Forking of a process

---
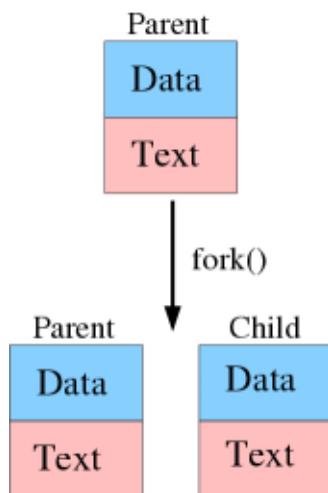
[1] The word *daemon* should not to be confused with the word *demon*, the word *daemon* refers to supernatural beings in Greek mythology.

Forking can be used by a program to create two processes that can run simultaneously on multiprocessor machines. However, this is often not ideal, because both processes will have their own process address space. The initial duplication of the process memory takes relatively much time, and it is difficult to share data between two processes. This problem is solved by a concept named *multithreading*. Multithreading means that multiple instances of the text area can run at the same time, sharing the data area. These instances, named threads, can be executed in parallel on multiple CPUs.

4.2. Filesystem hierarchy

Structure

Operating systems store data in filesystems. A filesystem is basically a tree-like structure of directories that hold files, like the operating system, user programs and user data. Most filesystems can also store various metadata about files and directories, for instance access and modification times. In GNU/Linux there is only one filesystem hierarchy, this means GNU/Linux does not have drive letters (e.g. A:, C:, D:) for different filesystems, like DOS and Windows. The filesystem looks like a tree, with a root directory (which has no parent directory), branches, and leaves (directories with no subdirectories). The root directory is alway denoted with a slash ("/"). Directories are separated by the same character.

Figure 4.2. The filesystem structure
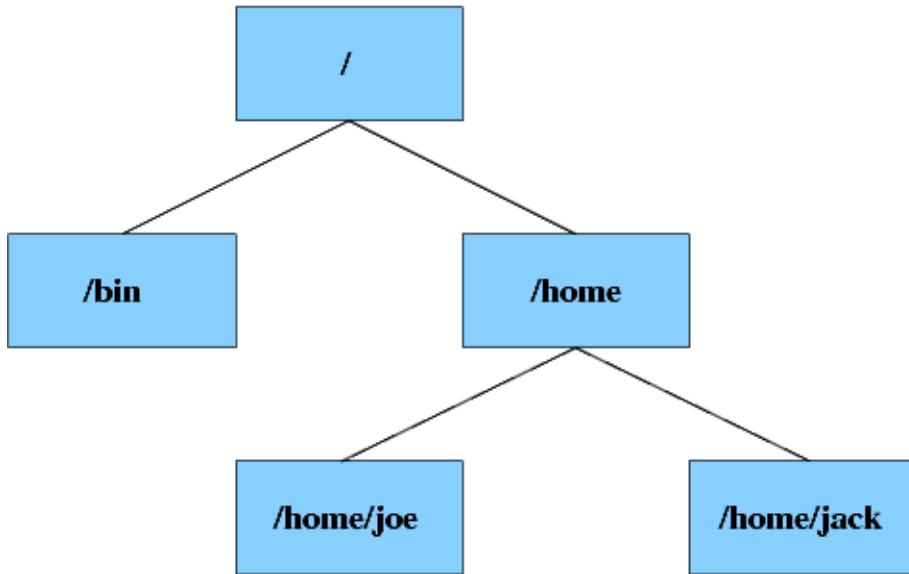
Figure 4.2, "The filesystem structure" shows the structure of a filesystem. You can see that the root directory / has two child directories: `bin` and `home`. The `home` directory has two child directories, `joe` and `jack`. The diagram shows the full pathname of each directory. The same notation is used for files. Suppose that there is a file named `memo.txt` in the `/home/jack` directory, the full path of the file is `/home/jack/memo.txt`.

Each directory has two special entries, ".", and "..". The first refers to the directory itself, the second to the parent directory. These entries can be used for making relative paths. If you are working in the `jack` directory, you can refer to the the `/home/joe` directory with `../joe`.

Mounting

You might wonder how it is possible to access other devices or partitions than the hard disk partition which holds the root filesystem. Linux uses the same approach as UNIX for accessing other filesystems. Linux allows the system

administrator to connect a device to any directory in the filesystem structure. This process is named *mounting*. For example, one could mount the CD-ROM drive to the /cdrom directory. If the mount is correct, the files on the CD-ROM can be accessed through this directory. The mounting process is described in detail in Section 8.8, "Mounting filesystems".

Common directories

The Filesystem Hierarchy Standard Group has attempted to create a standard that describes which directories should be available on a GNU/Linux system. Currently, most major distributions use the Filesystem Hierarchy Standard (FHS) as a guideline. This section describes some mandatory directories on GNU/Linux systems.

Please note that GNU/Linux does not have a separate directory for each application (like Windows). Instead, files are ordered by function and type. For example, the binaries for most common user programs are stored in /usr/bin, and their libraries in /usr/lib. This is a short overview of the important directories on a Slackware Linux system:

- **/bin**: essential user binaries that should still be available in case the /usr is not mounted.

- **/dev**: device files. These are special files used to access certain devices.

- **/etc**: the /etc directory contains all important configuration files.

- **/home**: contains home directories for individual users.

- **/lib**: essential system libraries (like glibc), and kernel modules.

- **/root**: home directory for the *root* user.

- **/sbin**: essential binaries that are used for system administration.

- **/tmp**: a world-writable directory for temporary files.

- **/usr/bin**: stores the majority of the user binaries.

- **/usr/lib**: libraries that are not essential for the system to boot.

- **/usr/sbin**: nonessential system administration binaries.

- **/var**: variable data files, like logs.

4.3. Devices

Introduction

In GNU/Linux virtually everything is represented as a file, including devices. Every GNU/Linux system has a directory with special files, named /dev. Each file in the /dev directory represents a device or pseudo-device. A device file has two special numbers associated with it, the *major* and the *minor* device number. The kernel knows which device a device file represents by these device numbers. The following example shows the device numbers for the /dev/zero device file:

```
$ file /dev/zero
/dev/zero: character special (1/5)
```

The **file** command can be used to determine the type of a file. This particular file is recognized as a device file that has *1* as the major device number, and *5* as the minor device number.

If you have installed the kernel source package, you can find a comprehensive list of all major devices with their minor and major numbers in `/usr/src/linux/Documentation/devices.txt`. An up-to-date list is also available on-line through the Linux Kernel Archives[2].

The Linux kernel handles two types of devices: *character* and *block* devices. Character devices can be read byte by byte, block devices can not. Block devices are read per block (for example 4096 bytes at a time). Whether a device is a character or block device is determined by the nature of the device. For example, most storage media are block devices, and most input devices are character devices. Block devices have one distinctive advantage, namely that they can be cached. This means that commonly read or written blocks are stored in a special area of the system memory, named the cache. Memory is much faster than most storage media, so much performance can be gained by performing common block read and write operations in memory. Of course, eventually changes have to be written to the storage media to reflect the changes that were made in the cache.

ATA and SCSI devices

There are two kinds of block devices that we are going to look into in detail, because understanding the naming of these devices is crucial for partitioning a hard disk and mounting. Almost all modern computers with an x86 architecture use ATA hard disks and CD-ROMs. Under Linux these devices are named in the following manner:

```
/dev/hda - master device on the first ATA channel
/dev/hdb - slave device on the first ATA channel
/dev/hdc - master device on the second ATA channel
/dev/hdd - slave device on the second ATA channel
```

On most computers with a single hard disk, the hard disk is the master device on the first ATA channel (`/dev/hda`), and the CD-ROM is the master device on the second ATA channel. Hard disk partitions are named after the disk that holds them plus a number. For example, `/dev/hda1` is the first partition of the disk represented by the `/dev/hda` device file.

SCSI hard disks and CD-ROM drives follow an other naming convention. SCSI is not commonly used in most low-end machines, but USB drives and Serial ATA (SATA) drives are also represented as SCSI disks. The following device notation is used for SCSI drives:

```
/dev/sda - First SCSI disk
/dev/sdb - Second SCSI disk
/dev/sdc - Third SCSI disk
/dev/scd0 - First CD-ROM
/dev/scd1 - Second CD-ROM
/dev/scd2 - Third CD-ROM
```

Partitions names are constructed in the same way as ATA disks; `/dev/sda1` is the first partition on the first SCSI disk.

If you use the software RAID implementation of the Linux kernel, RAID volumes are available as `/dev/mdn`, in which *n* is the volume number starting at 0.

---

[2] ftp://ftp.kernel.org/pub/linux/docs/device-list/

Chapter 5. Installing Slackware Linux

5.1. Booting the installation CD-ROM

The easiest method for booting the installation system is by using the installation CD-ROM. The Slackware Linux installation CD-ROM is a bootable CD, which means that the BIOS can boot the CD, just like it can boot, for example, a floppy disk. Most modern systems have a BIOS which supports CD-ROM booting.

If the CD is not booted when you have the CD inserted in the CD-ROM drive during the system boot, the boot sequence is probably not correctly configured in the BIOS. Enter the BIOS setup (usually this can be done by holding the <Del> or <Esc> key when the BIOS screen appears) and make sure the CD-ROM is on the top of the list in the boot sequence. If you are using a SCSI CD-ROM you may have to set the boot sequence in the SCSI BIOS instead of the system BIOS. Consult the SCSI card manual for more information.

When the CD-ROM is booted, a pre-boot screen will appear. Normally you can just press <ENTER> to proceed loading the default (`hugesmp.s`) Linux kernel. This kernel requires at least a Pentium Pro CPU. You can boot an alternative kernel by entering the kernel name on the prompt, and pressing <ENTER>. The following table lists the different kernels that are available from the Slackware Linux CD or DVD.

Table 5.1. Installation kernels

Linux	Description
huge.s	Previously, there were specific kernels for different sets of disk controlers. The new huge kernels include support for all common ATA, SATA and SCSI controllers. This kernel does not have SMP support, and works on i486 and newer CPUs. If you have a Pentium Pro or newer CPU, it is recommended to use the *hugesmp.s* kernel, even on uniprocessor systems.
hugesmp.s	The *hugesmp.s* kernel has support for common ATA, SATA, and SCSI controllers. Additionally, this kernel has SMP support. This is the recommended kernel on Pentium Pro and newer CPUs.
speakup.s	This kernel is comparable to the *huge.s* kernel, but also includes support for hardware speech synthesizers.

After booting the installation system, you will be asked whether you are using a special (national) keyboard layout or not. If you have a normal US/International keyboard, which is the most common, you can just press <Enter> at this question. After that the login prompt will appear. Log on as "root", no password will be requested. After login, the shell is started and you can begin installing Slackware Linux. The installation procedure will be explained briefly in this chapter.

5.2. Partitioning a hard disk

Installing Slackware Linux requires at least one Linux partition, creating a swap partition is also recommended. To be able to create a partition there has to be free unpartitioned space on the disk. There are some programs that can resize partitions. For example, FIPS can resize FAT partitions. Commercial programs like Partition Magic can also resize other partition types.

After booting the Slackware Linux CD-ROM and logging on, there are two partitioning programs at your disposal: **fdisk** and **cfdisk**. **cfdisk** is the easiest of both, because it is controlled by a menu interface. This section describes the **cfdisk** program.

To partition the first harddisk you can simply execute **cfdisk**. If you want to partition another disk or a SCSI disk you have to specify which disk you want to partition (**cfdisk /dev/device**). ATA hard disks have the following device naming: /dev/hdn, with "n" replaced by a character. E.g. the "primary master" is named /dev/hda, the "secondary slave" is named /dev/hdd. SCSI disks are named in the following way: /dev/sdn, with "n" replaced by the device character (the first SCSI disk = a, the fourth SCSI disk = d).

Figure 5.1. The cfdisk parition tool

```
                          cfdisk 2.12

                     Disk Drive: /dev/sda
                  Size: 4294967296 bytes, 4294 MB
            Heads: 255    Sectors per Track: 63    Cylinders: 522

     Name         Flags       Part Type  FS Type          [Label]         Size (MB)
    --------------------------------------------------------------------------------
     sda1                     Primary    Linux ReiserFS                    1003.49
     sda2                     Primary    Linux swap                         131.61
                             Pri/Log    Free Space                        3158.51













       [Bootable]  [ Delete ]  [  Help  ]  [Maximize]  [ Print  ]
       [  Quit  ]  [  Type  ]  [ Units  ]  [ Write  ]

              Toggle bootable flag of the current partition_
```

After starting **cfdisk**, the currently existing partitions are shown, as well as the amount of free space. The list of partitions can be navigated with the "up" and "down" arrow keys. At the bottom of the screen some commands are displayed, which can be browsed with the "left" and "right" arrow keys. A command can be executed with the <Enter> key.

You can create a Linux partition by selecting "Free Space" and executing the "New" command. **cfdisk** will ask you whether you want to create a primary or logical partition. The number of primary partitions is limited to four. Linux can be installed on both primary and logical partitions. If you want to install other operating systems besides Slackware Linux that require primary partitions, it is a good idea to install Slackware Linux onto a logical partition. The type of the new partition is automatically set to "Linux Native", so it is not necessary to set the partition type.

The creation of a swap partition involves the same steps as a normal Linux partition, but the type of the partition has to be changed to "Linux Swap" after the partition is created. The suggested size of the swap partition depends

on your own needs. The swap partition is used to store programs if the main (RAM) memory is full. If you have a harddisk of a reasonable size, it is a good idea to make a 256MB or 512MB swap partition, which should be enough for normal usage. After creating the partition, the partition type can be changed to "Linux Swap" by selecting the "Type" command. The **cfdisk** program will ask for the type number. "Linux Swap" partitions have type number 82. Normally number 82 is already selected, so you can go ahead by pressing the <Enter> key.

If you are satisfied with the partitioning you can save the changes by executing the "Write" command. This operation has to be confirmed by entering **yes**. After saving the changes you can quite **cfdisk** with the **Quit** command. It is a good idea to reboot the computer before starting the installation, to make sure that the partitioning changes are active. Press <ctrl> + <alt> + <del> to shut Linux down and restart the computer.

5.3. Installing Slackware Linux

The Slackware Linux installer is started by executing **setup** in the installation disk shell. Setup will show a menu with several choices. You can see a screenshot of the installer in Figure 5.2, "The setup tool". Every option is required for a complete Slackware Linux installation, but once you start, the **setup** program will guide you through the options.

Figure 5.2. The setup tool
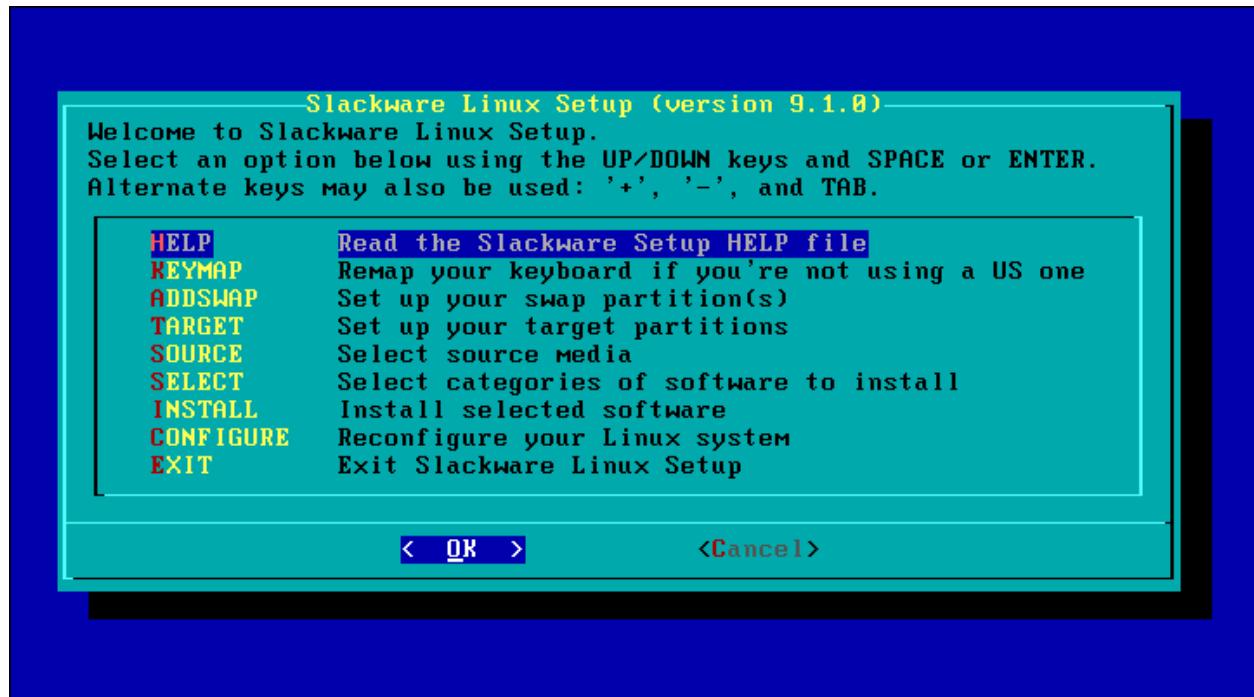
The first part of the installation is named "ADDSWAP". The **setup** tool will look for a partition with the "Linux Swap" type, and ask you if you want to format and activate the swap partition (see figure Figure 5.3, "Setting up the swap partition"). Normally you can just answer "Yes".

Figure 5.3. Setting up the swap partition
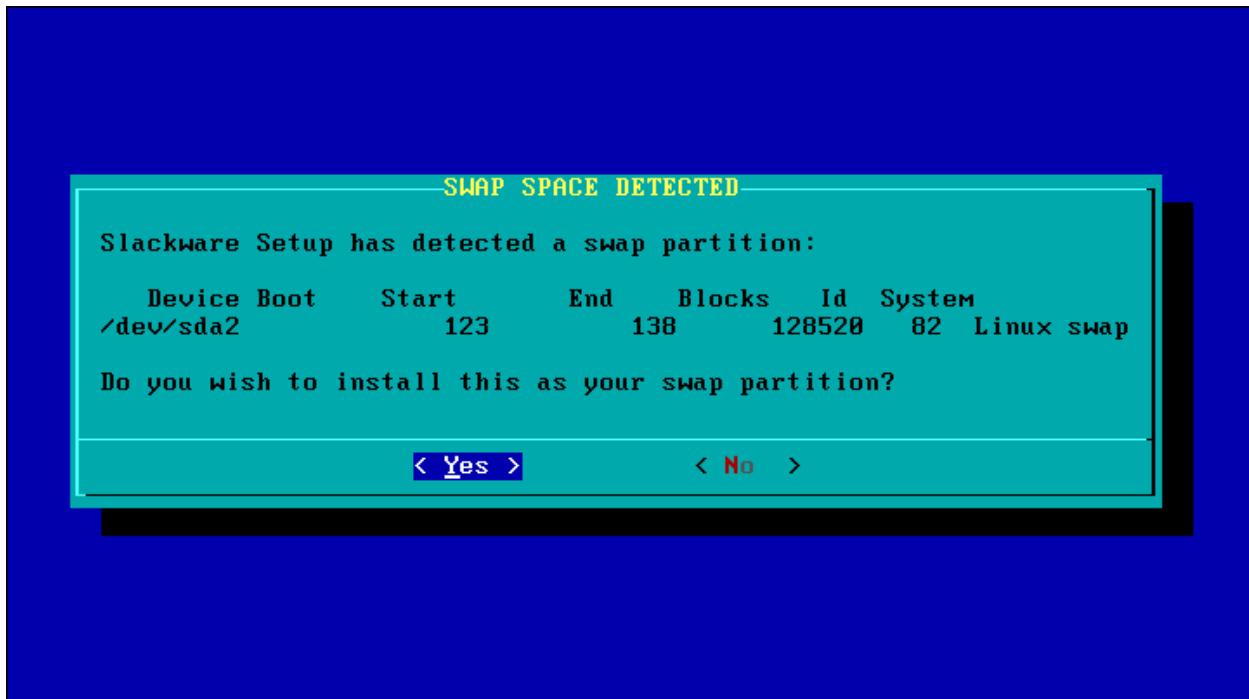
After setting up the swap space the "TARGET" menu is launched, which you can see in Figure 5.4, "Selecting a partition to initialize". It is used to initialize the Slackware Linux partitions. Setup will display all partitions with the "Linux Native" type.

Figure 5.4. Selecting a partition to initialize
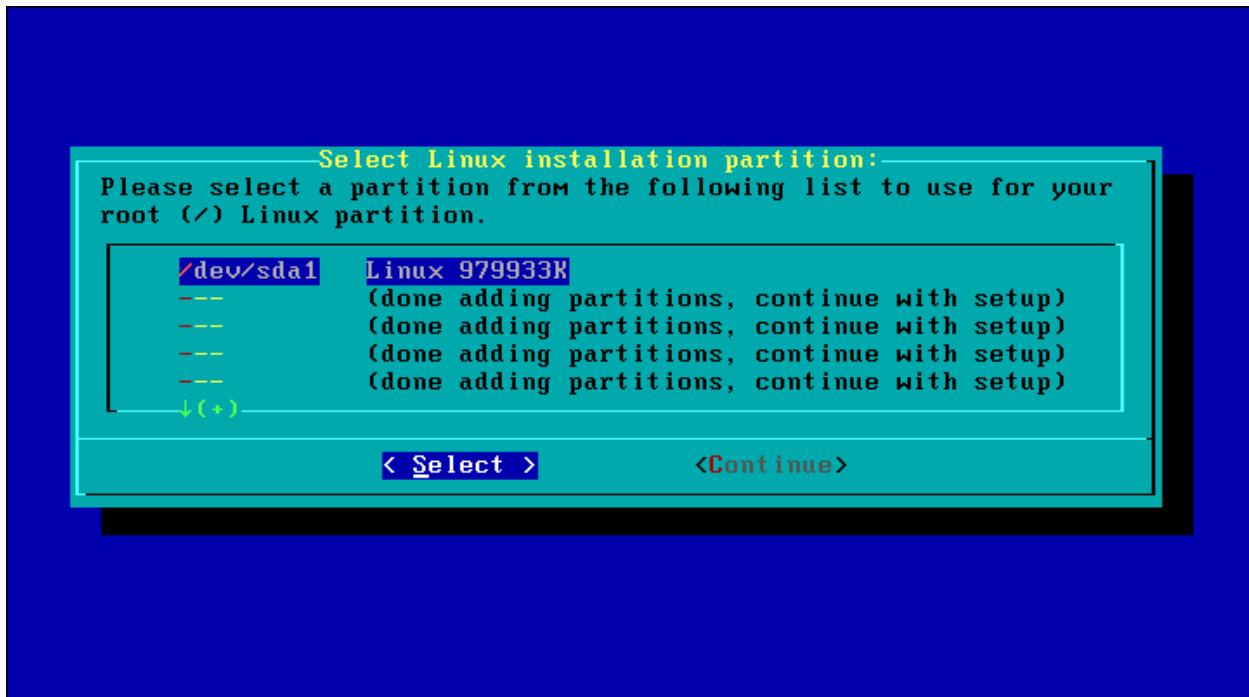
After selecting one partition, the setup tool will ask whether you want to format a partition or not, and if you want to format it, whether you want to check the disk for bad sectors or not (Figure 5.5, "Formatting the partition"). Checking the disk can take a lot of time.

Figure 5.5. Formatting the partition
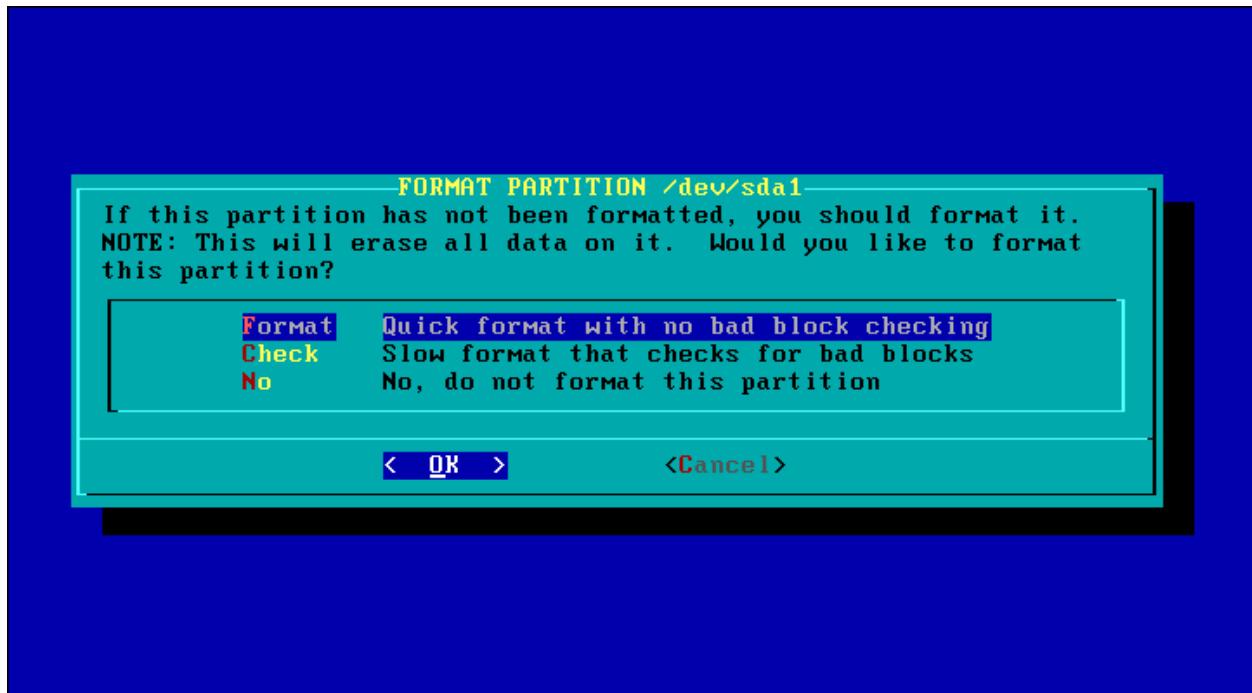
After selecting to format a partition, you can specify which filesystem should be used (Figure 5.6, "Selecting a filesystem type"). Normally you can choose the ext2, ext3 and reiserfs filesystems. Ext2 was the standard Linux filesystem for many years, but it does not support journaling. A journal is a special file or area of a partition in which all filesystem operations are logged. When the system crashes, the filesystem can be repaired rapidly, because the kernel can use the log to see what disk operations were performed. Ext3 is the same filesystem as Ext2, but adds journaling. Reiserfs is a filesystem that also provides journaling. Reiserfs uses balanced trees, which make many filesystem operations faster than with Ext2 or Ext3, especially when working with many small files. A disadvantage is that Reiserfs is newer, and can be a bit more unstable.

Figure 5.6. Selecting a filesystem type
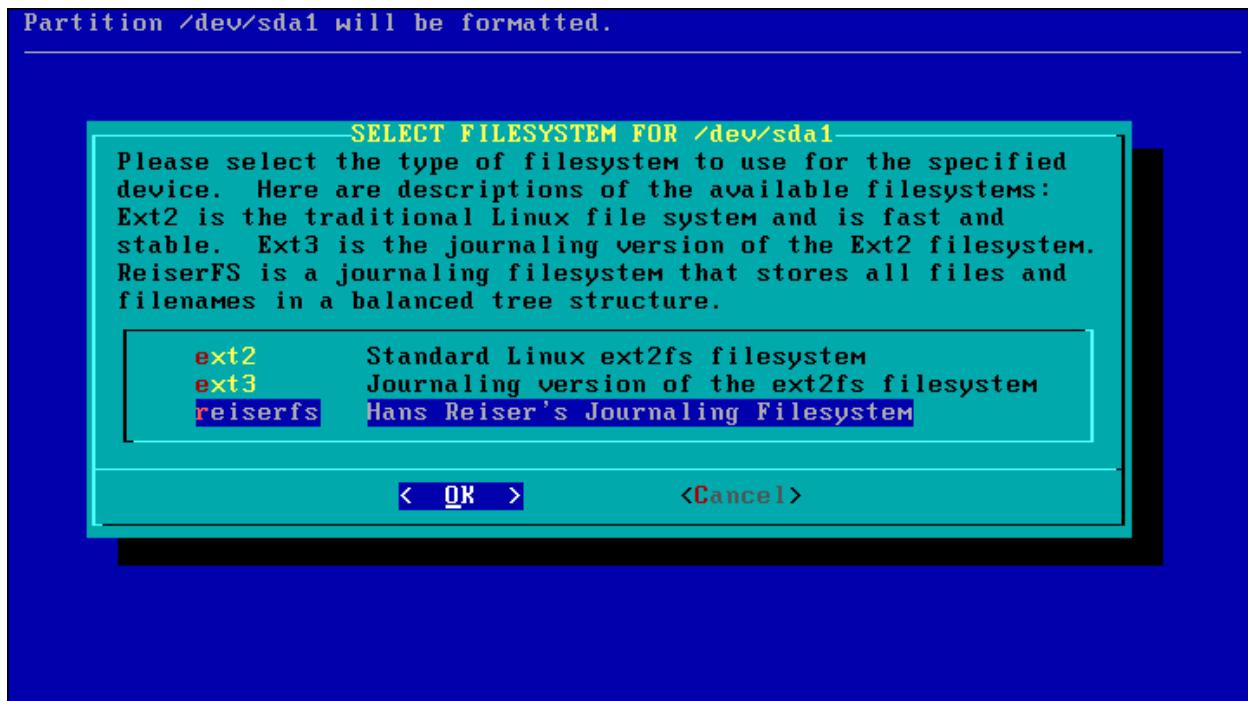
The first initialized partition is automatically mounted as the root (/) partition. For other partitions the mount point can be selected after the initialization. You could, for example, make separate partitions for /, /var, /tmp, /home and /usr. This provides extra protection against crashes. Since / is rarely changed after the installation if you create these partitions, the chance that it will be in the middle of a write operation during a crash is much smaller. Besides that, it is safer to create a separate filesystem for /home. If a program has a security vulnerability, a user could create a hard link to the binary of that program, if the /home directory is on the same filesystem as the /{s}bin /usr/ {s}bin, or /usr/local/{s}bin directories. This user will then still be able to access the old, vulnerable binary after the program is upgraded.

The next step is to select the source medium (Figure 5.7, "Selecting the source medium"). This dialog offers several choices, like installing Slackware Linux from a CD-ROM or installing Slackware Linux via NFS. Slackware Linux is usually installed from CD-ROM, so this is what we are going to look at. After selecting "CD-ROM" you will be asked whether you want to let setup look for the CD-ROM itself ("Auto") or you want to select the CD-ROM device yourself ("Manual"). If you select "Manual" the setup tool will show a list of devices. Select the device holding the Slackware Linux CD-ROM.

Figure 5.7. Selecting the source medium
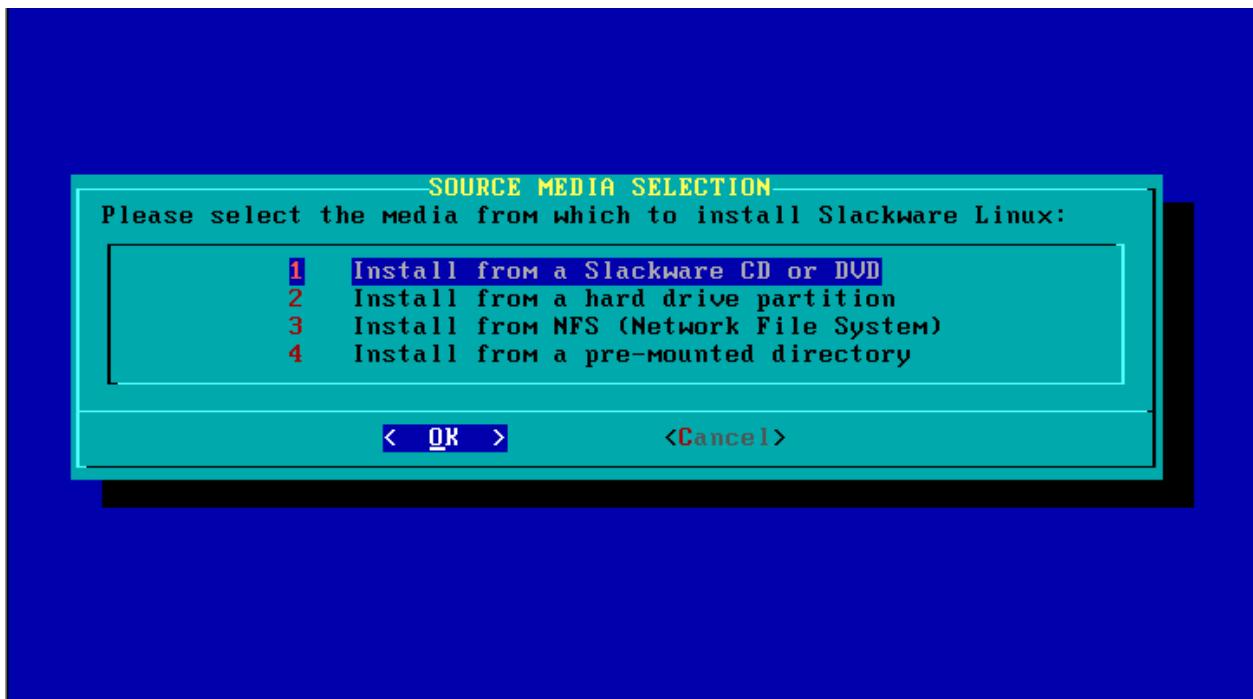
After choosing an installation source the setup tool will ask you which disk sets (series) you want to install packages from (Figure 5.8, "Selecting the disk sets"). A short description of each disk set is listed.

Figure 5.8. Selecting the disk sets
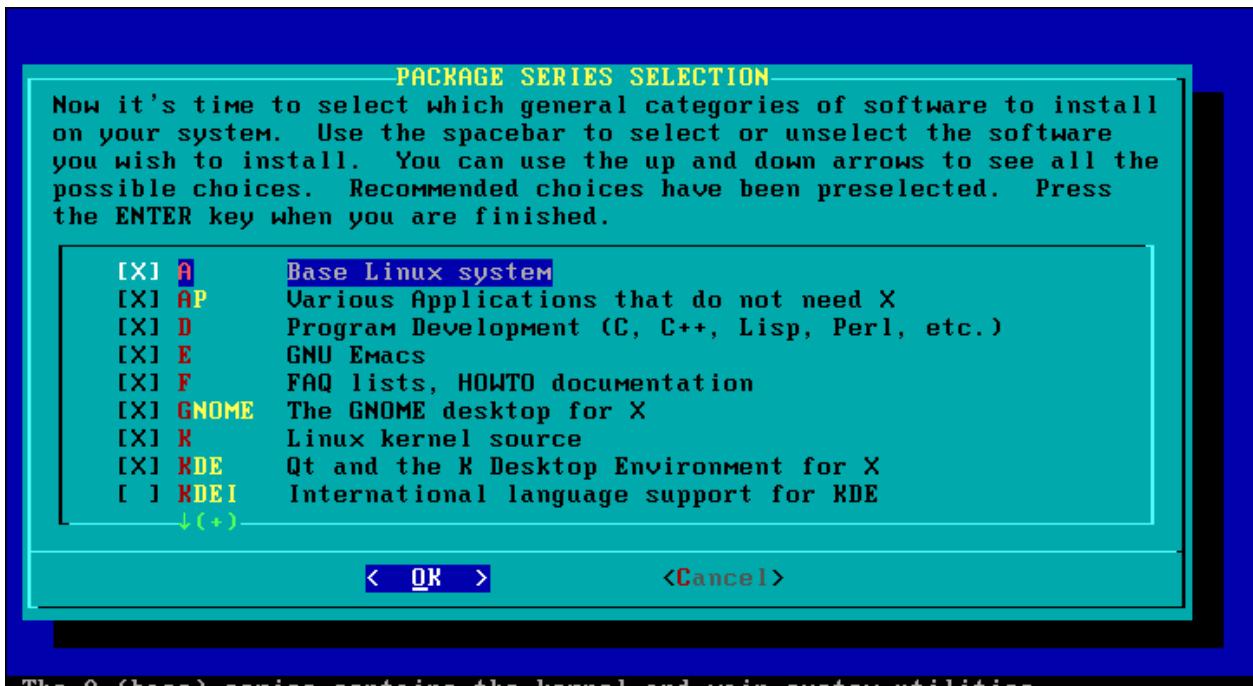
Now it is almost time to start the real installation. The next screen asks how you would like to install. The most obvious choices are "full", "menu" or "expert". Selecting "full" will install all packages in the selected disk sets. This is the

easiest way of installing Slackware Linux. The disadvantage of this choice is that it can take quite much disk space. The "menu" option will ask you for each disk set which packages you want to install. The "expert" option is almost equal to the "menu" option, but allows you to deselect some very important packages from the "a" disk set.

After the completion of the installation the setup tool will allow you to configure some parts of the system. The first dialog will ask you where you would like to install the kernel from (see Figure 5.9, "Installing the kernel"). Normally it is a good idea to install the kernel from the Slackware Linux CD-ROM, this will select the kernel you installed Slackware Linux with. You can confirm this, or select another kernel.

Figure 5.9. Installing the kernel
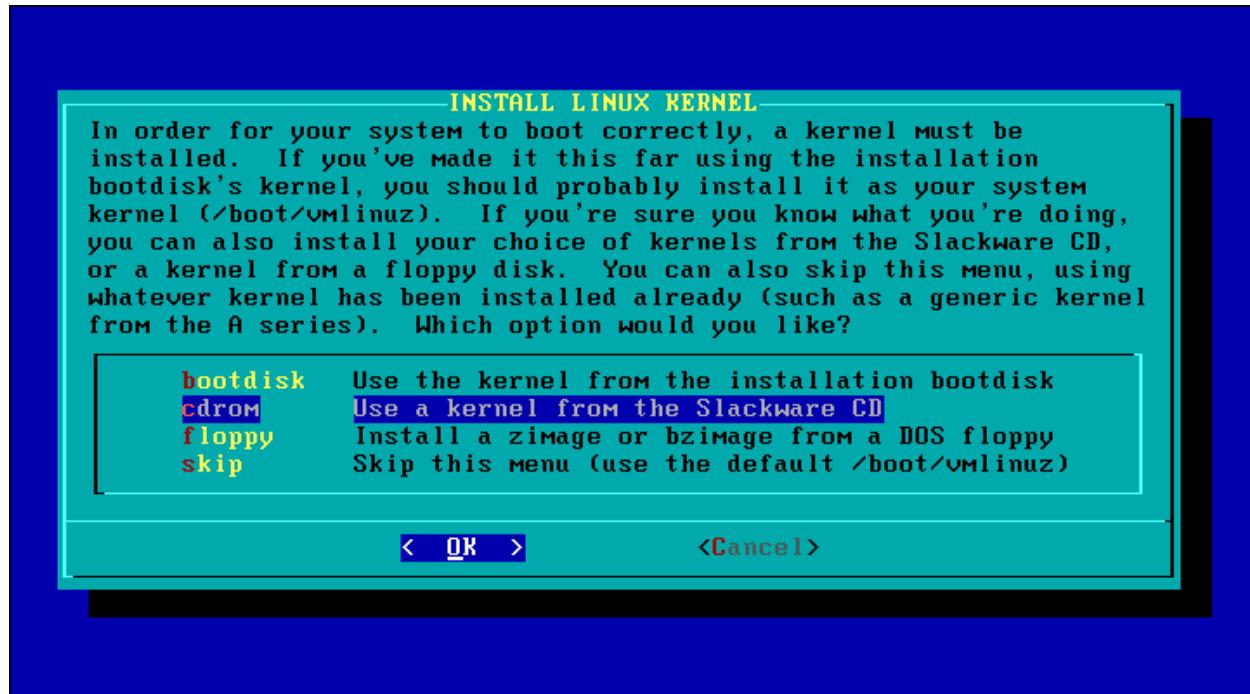
At this point you can choose to make a bootdisk (Figure 5.10, "Creating a bootdisk"). It is a good idea to make a bootdisk, you can use it to boot Slackware Linux if the LILO configuration is botched.

Figure 5.10. Creating a bootdisk
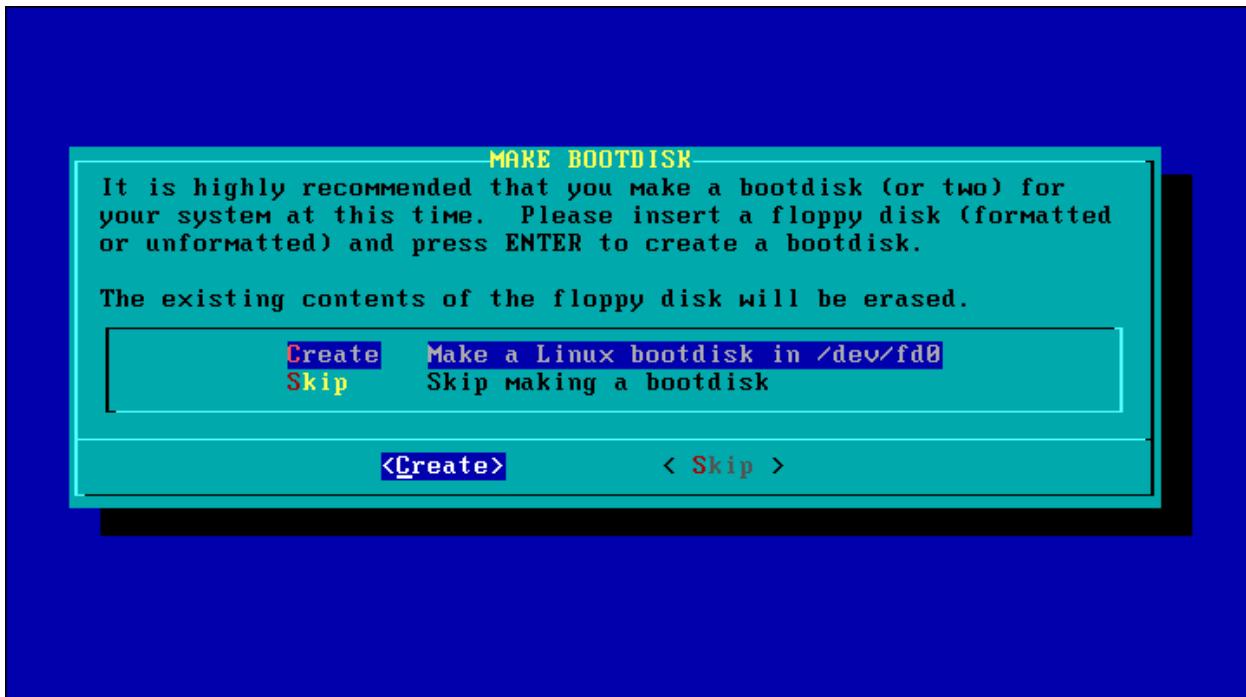
The following dialog can be used to make a link, /dev/modem, that points to your modem device. (Figure 5.11, "Selecting the default modem"). If you do not have a modem you can select *no modem*.

Figure 5.11. Selecting the default modem
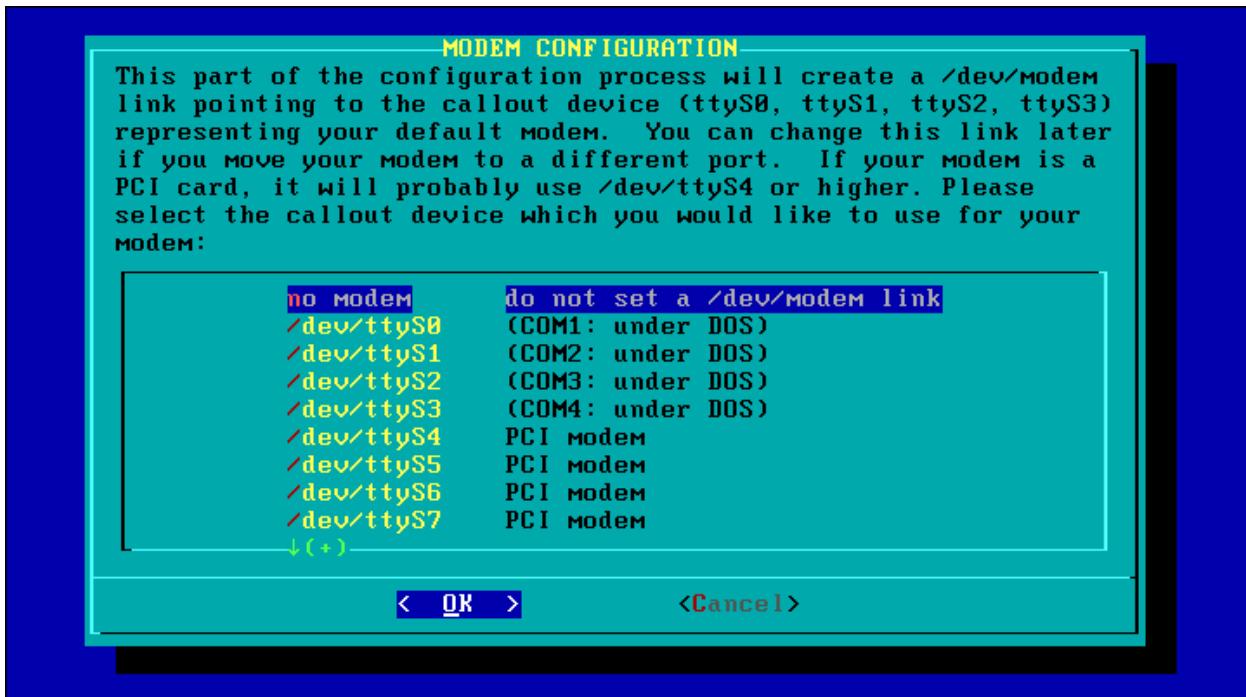
The next step is to select whether you would like to use hotplug (Figure 5.12, "Enabling hotplugging"). Hotplug is used for automatically configuring pluggable USB, PCMCIA and PCI devices. Generally speaking, it is a good idea to enable hotplugging, but some systems may have problems with the probing of the hotplug scripts.

Figure 5.12. Enabling hotplugging
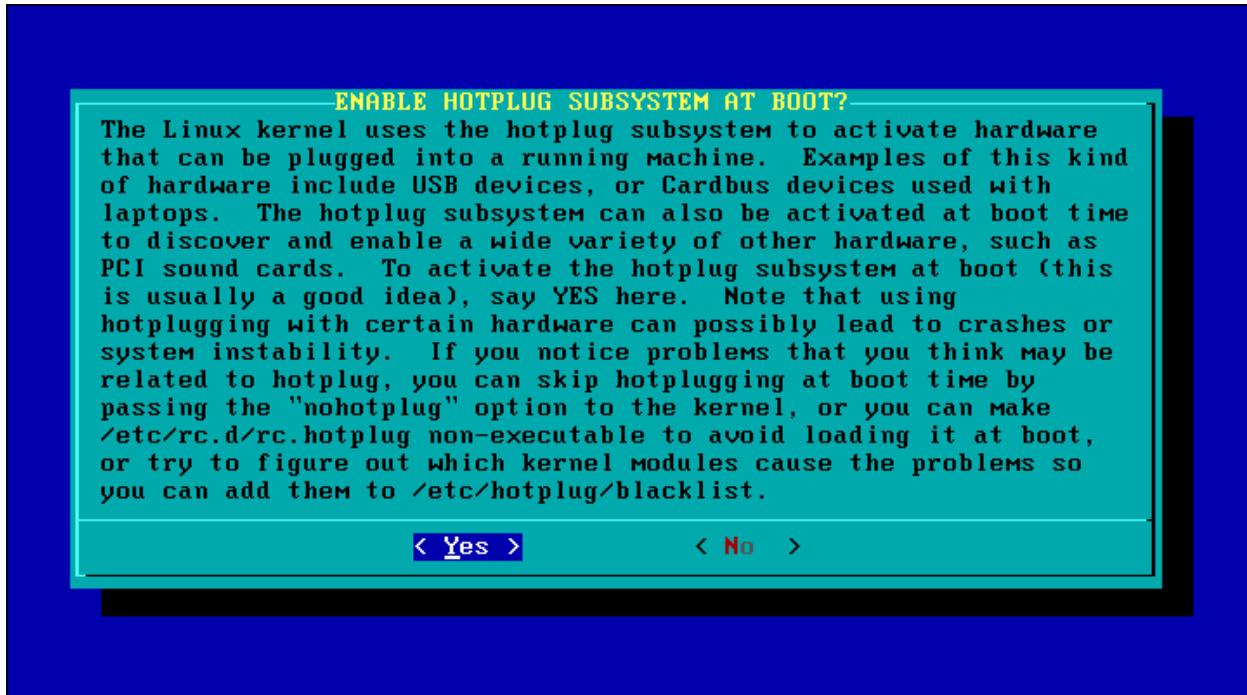
The following steps are important, the next dialogs will assist you with installing LILO, the Linux bootloader. Unless you have experience in configuring LILO it is a good idea to choose to use the *simple* option for configuration of LILO, which tries to configure LILO automatically (Figure 5.13, "Selecting the kind of LILO installation").

Figure 5.13. Selecting the kind of LILO installation
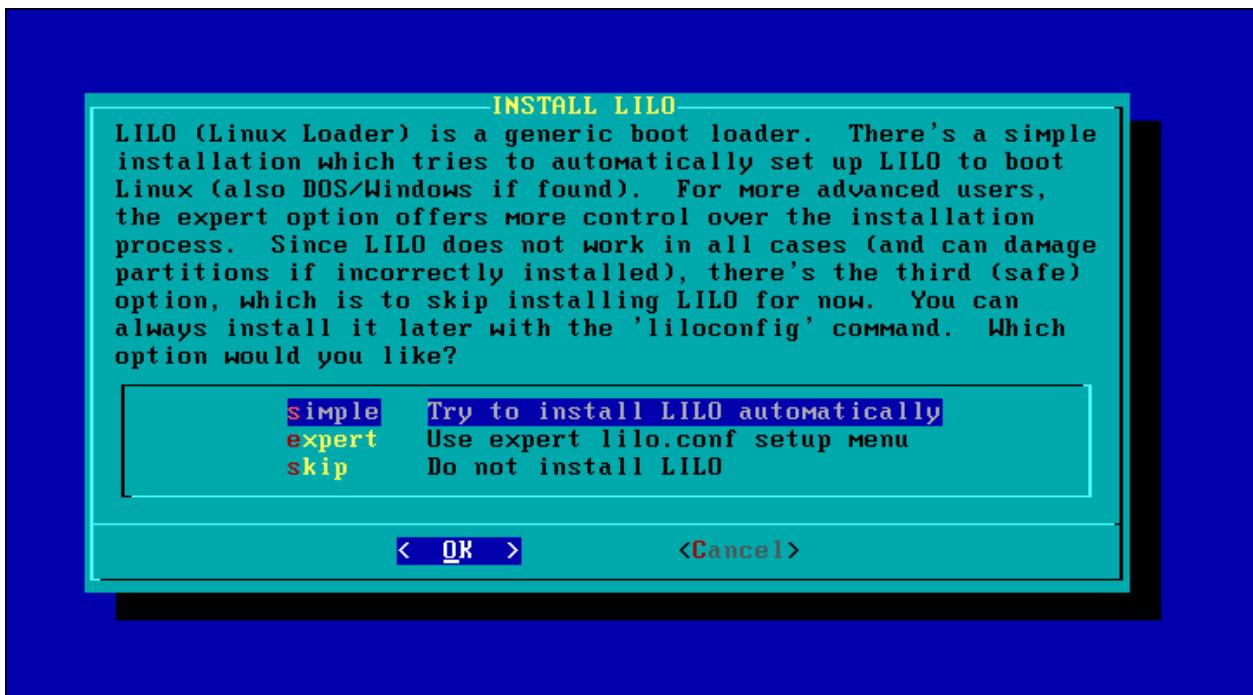
After selecting the *simple* option the LILO configuration utility will asks you whether you would like to use a framebuffer or not (Figure 5.14, "Choosing the framebuffer resolution"). Using a framebuffer will allow you to use the console in several resolutions, with other dimensions than the usual 80x25 characters. Some people who use the console extensively prefer to use a framebuffer, which allows them to keep more text on the screen. If you do not want a framebuffer console, or if you are not sure, you can choose *standard* here.

Figure 5.14. Choosing the framebuffer resolution
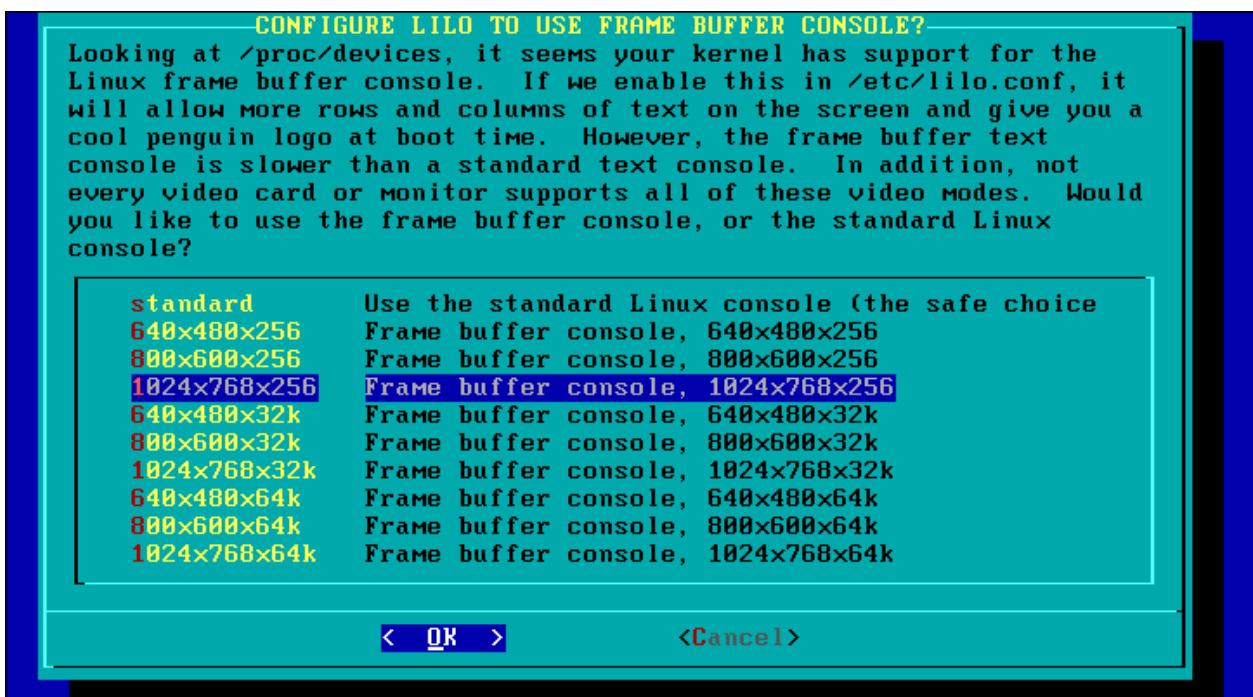
After setting the framebuffer you can pass extra parameters to the kernel (Figure 5.15, "Adding kernel parameters"). This is normally not necessary, if you do not want to pass extra parameters you can just press the <Enter> key.

Figure 5.15. Adding kernel parameters
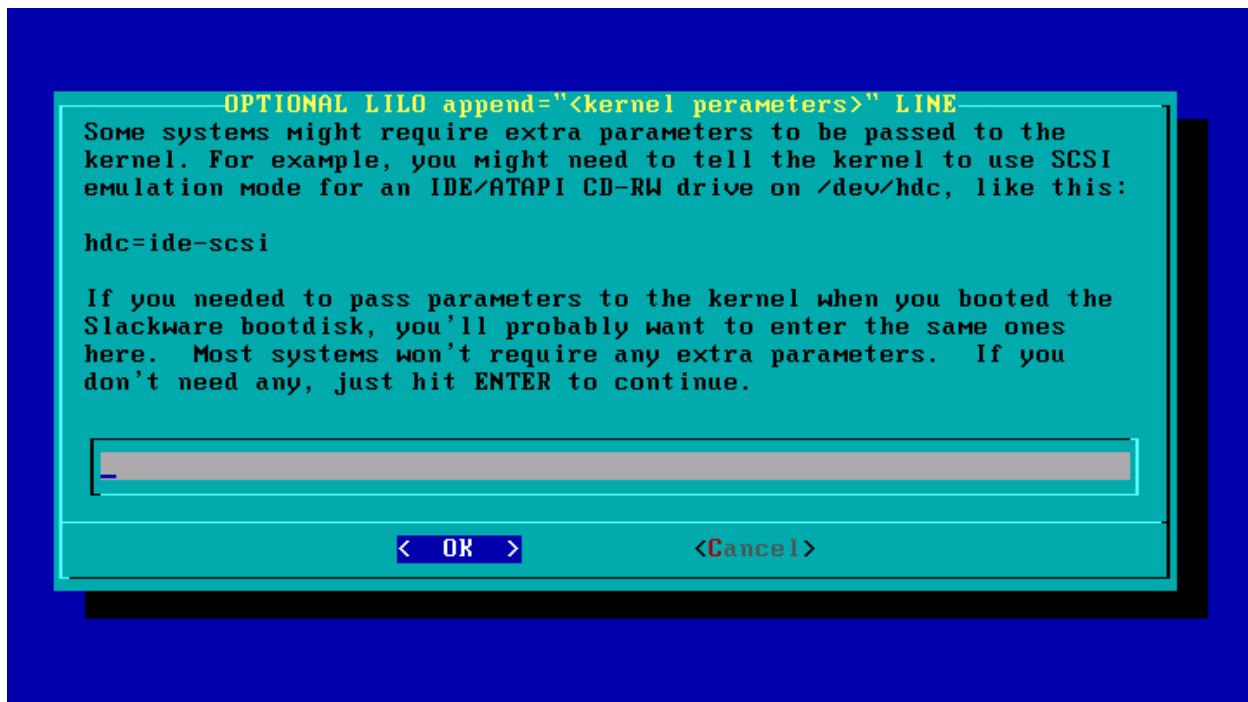
The last step of the LILO configuration is selecting where LILO should be installed (Figure 5.16, "Choosing where LILO should be installed"). *MBR* is the master boot record, the main boot record of PCs. Use this option if you want use Slackware Linux as the only OS, or if you want to use LILO to boot other operating systems. The *Root* option will install LILO in the boot record of the Slackware Linux / partition. Use this option if you use another bootloader.

Figure 5.16. Choosing where LILO should be installed
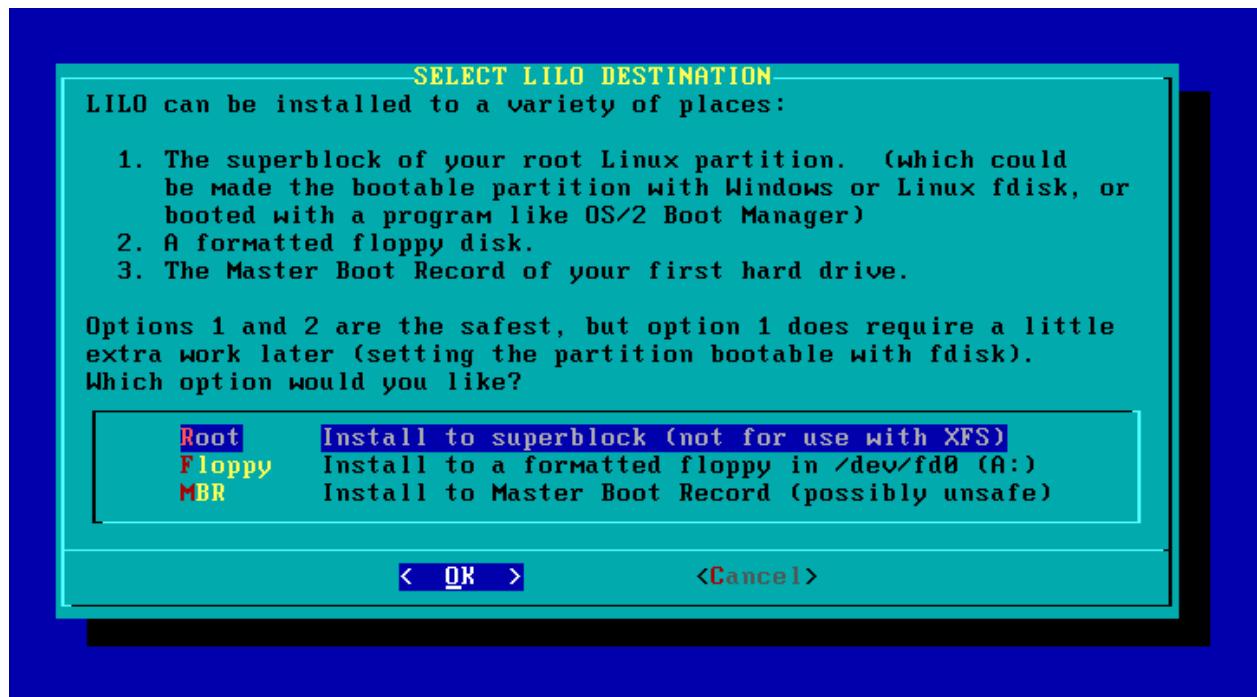
You will now be asked to configure your mouse. Select the mouse type from the dialog that appears (Figure 5.17, "Configuring a mouse").

Figure 5.17. Configuring a mouse

You will then be asked whether the **gpm** program should be loaded at boot time or not (Figure 5.18, "Choosing whether GPM should be started or not"). **gpm** is a daemon that allows you to cut and paste text on the console.

Figure 5.18. Choosing whether GPM should be started or not


```
┌──────────────GPM CONFIGURATION──────────────┐
│ The gpm program allows you to cut and paste text on    │
│ the virtual consoles using a mouse.  If you choose to  │
│ run it at boot time, this line will be added to your   │
│ /etc/rc.d/rc.gpm:                                       │
│                                                         │
│   /usr/sbin/gpm -M /dev/mouse -t ps2                    │
│                                                         │
│ Shall we load the gpm program at boot time?             │
│                                                         │
│          < Yes >              < No  >                   │
└─────────────────────────────────────────────┘
```
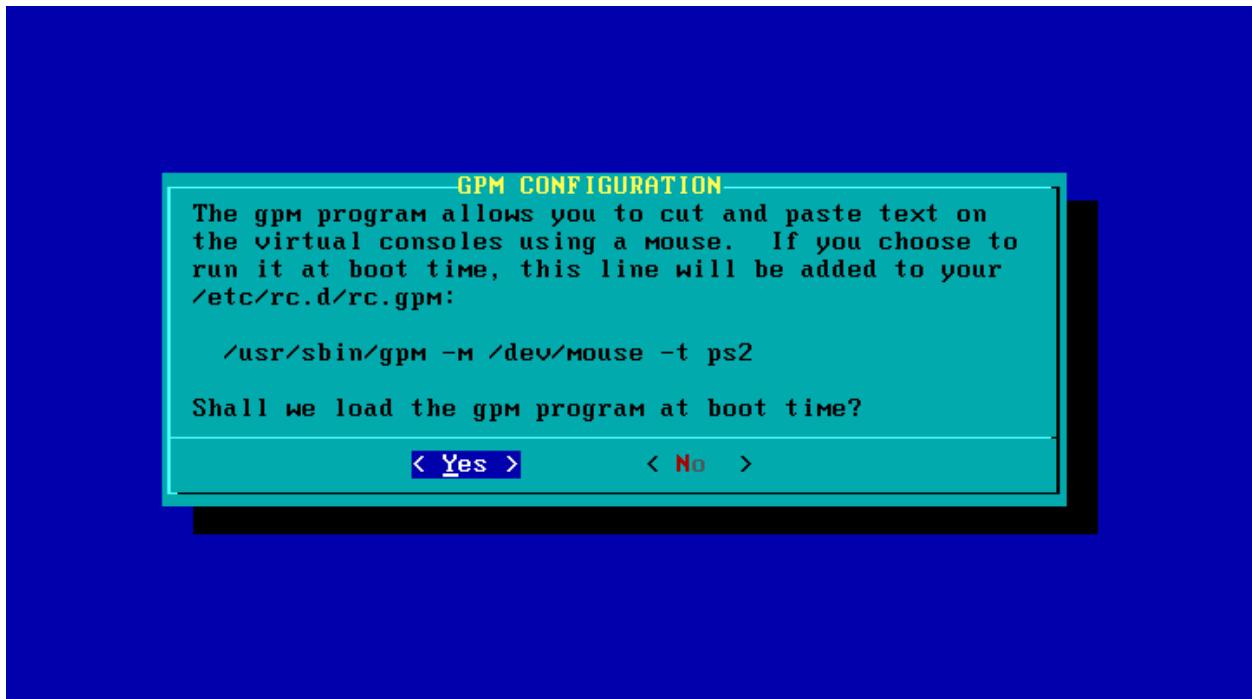
The next few steps will configure network connectivity. This is required on almost every networked system. The Slackware Linux setup will ask you if you want to set up network connectivity (Figure 5.19, "Choosing whether you would like to configure network connectivity"). If you answer "No" you can skip the next few network-related steps.

Figure 5.19. Choosing whether you would like to configure network connectivity


```
┌──────────────CONFIGURE NETWORK?──────────────┐
│ Would you like to configure your network?     │
│          < Yes >              < No  >          │
└───────────────────────────────────────────────┘
```

You will now be asked to set the hostname (Figure 5.20, "Setting the host name"). Please note that this is not the fully qualified domain name (FQDN), just the part that represents the host (normally the characters before the first dot in a FQDN).

Figure 5.20. Setting the host name
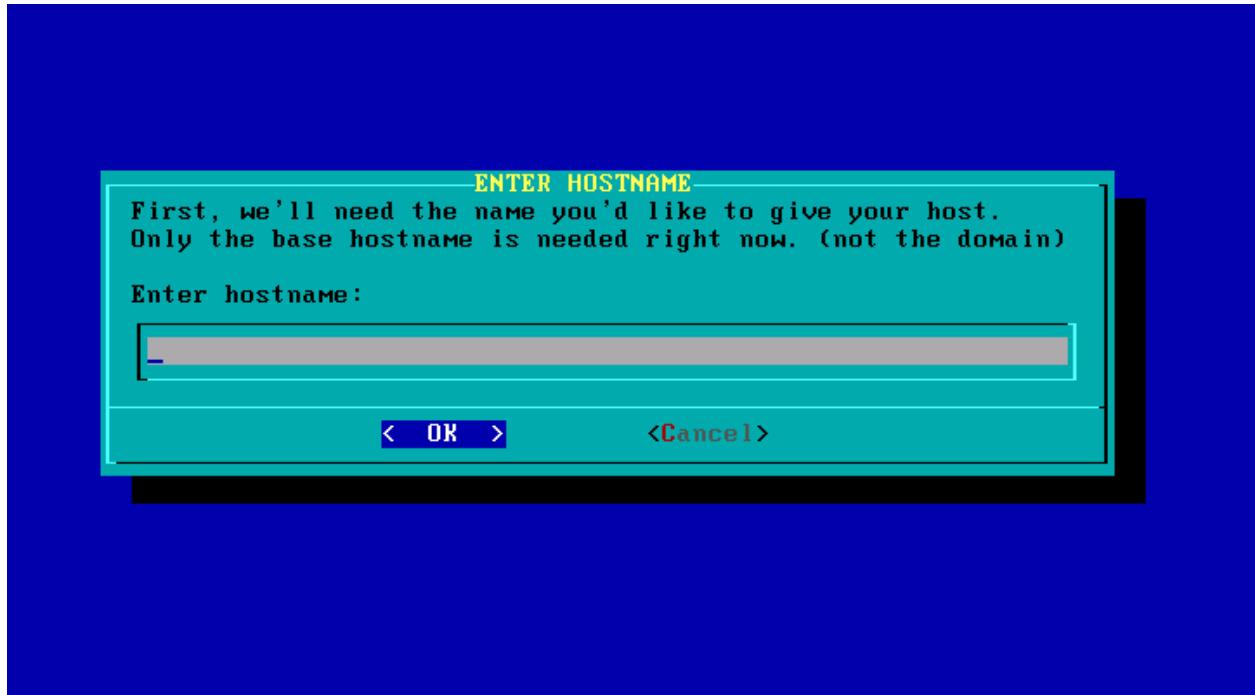
After setting the host name you can set the domain name part of the fully qualified domain name (FQDN) (Figure 5.21, "Setting the domain name").

Figure 5.21. Setting the domain name
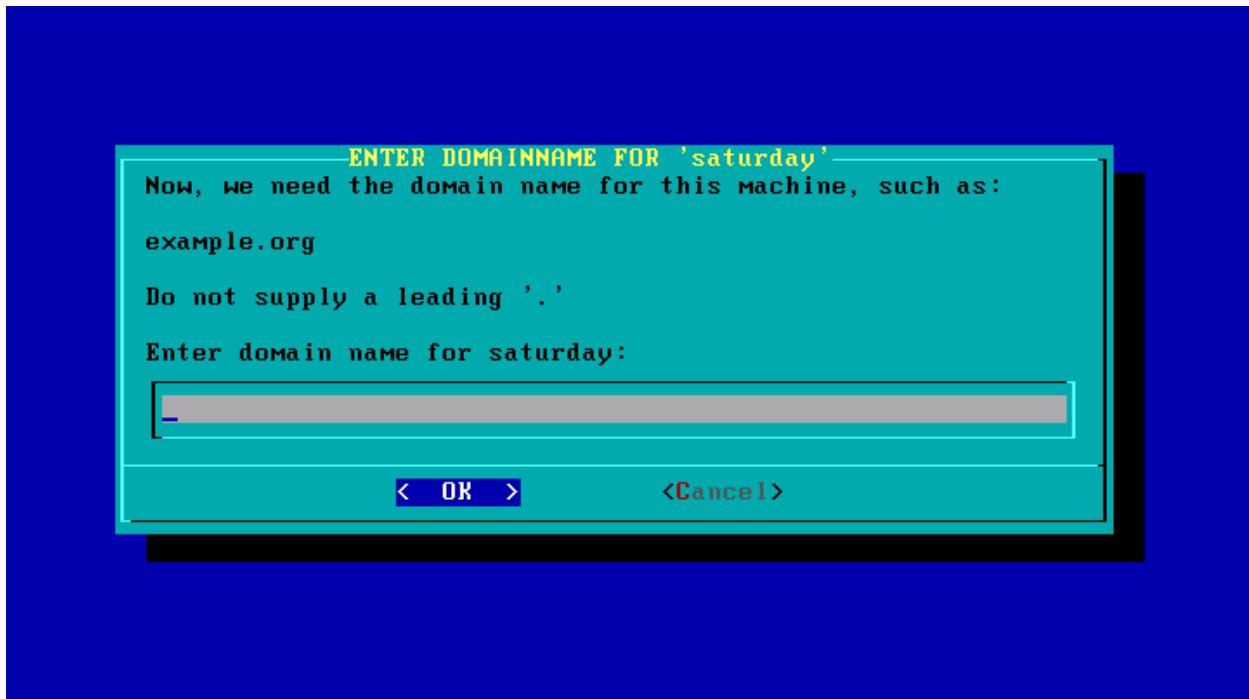
The rest of the network configuration steps depend on whether nodes in the network are configured to use a static or a dynamic IP address. Some networks have a DHCP server that automatically assigns an IP address to hosts in the network. If this is the case for the network the machine select *DHCP* during this step of the installation (Figure 5.22, "Manual or automatic IP address configuration"). When DHCP is selected you will only be asked whether a host name should be sent to the server. Normally you can leave this blank. If you use DHCP you can skip the rest of the network configuration described below.

If the network does not have a DHCP server you can choose the *static IP* option, which will allow you to set the IP address and related settings manually.

Figure 5.22. Manual or automatic IP address configuration
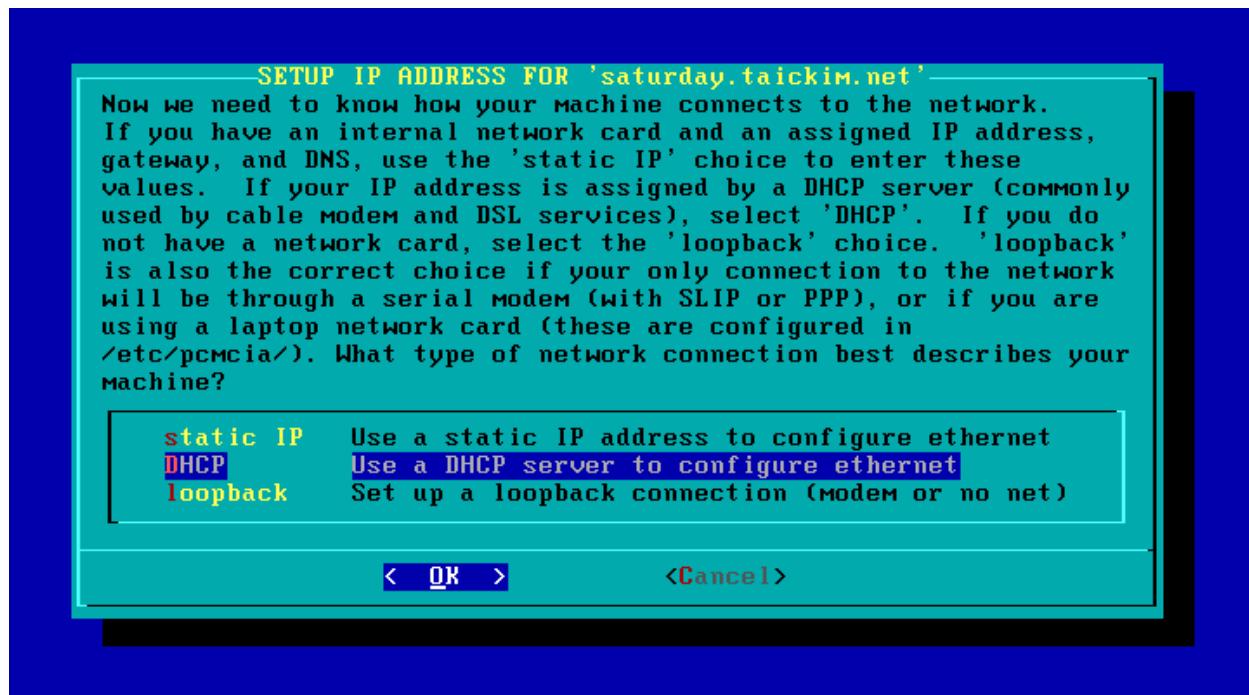
The first step of the manual configuration is to set the IP address of the first interface (eth0) of the machine (Figure 5.23, "Setting the IP addres").

Figure 5.23. Setting the IP addres
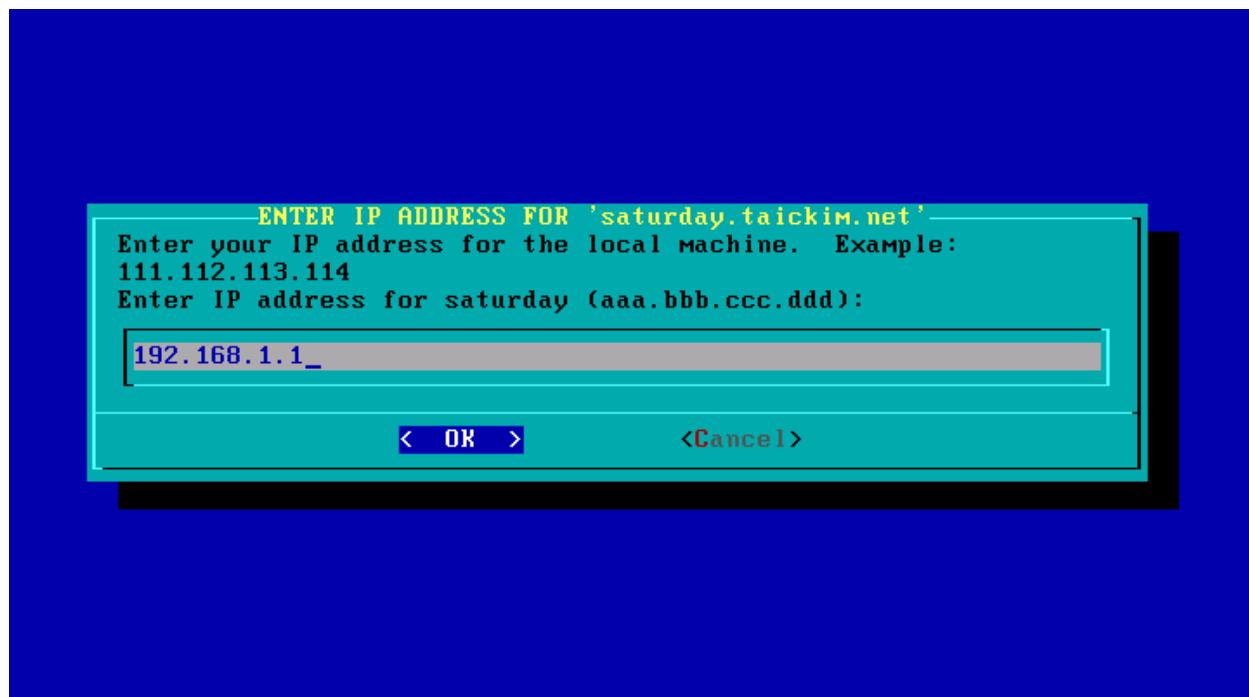
After setting the IP address you will be asked to enter the netmask. The netmask is usually dependent on the IP address class (Figure 5.24, "Setting the netmask").

Figure 5.24. Setting the netmask
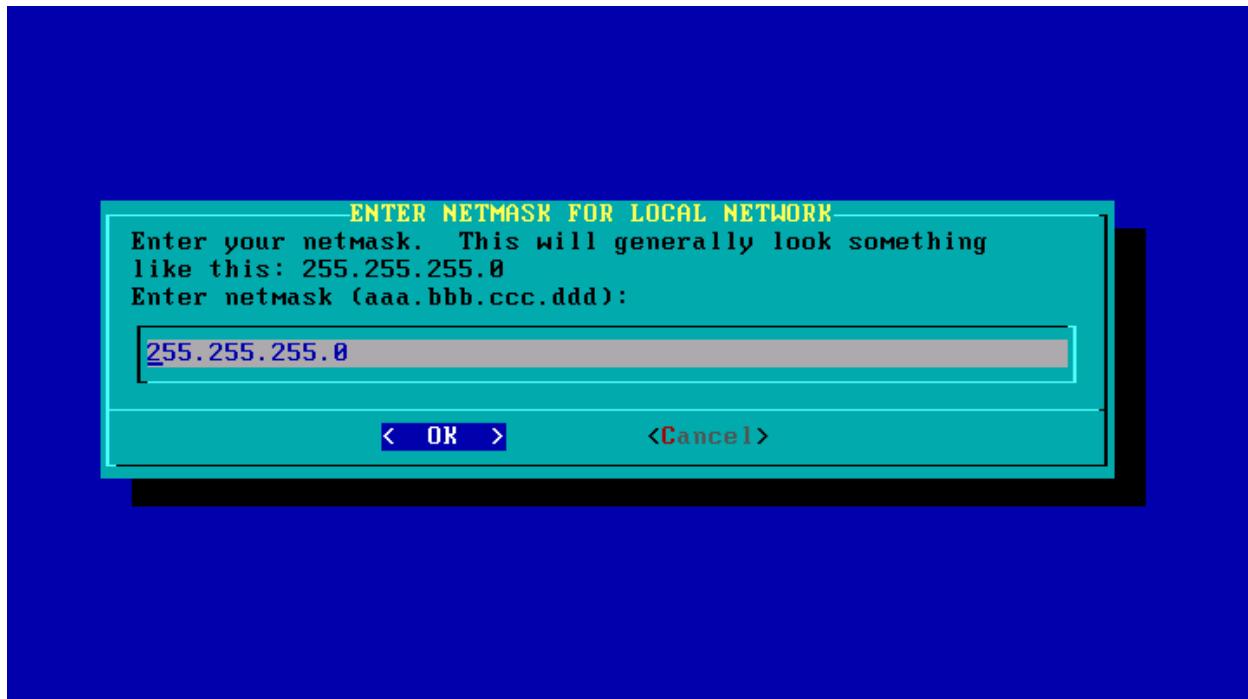
You will then be asked to set the address of the gateway (Figure 5.25, "Setting the gateway"). The gateway is the machine on the network that provides access to other networks by routing IP packets. If your network has no gateway you can just hit the <ENTER> key.

Figure 5.25. Setting the gateway
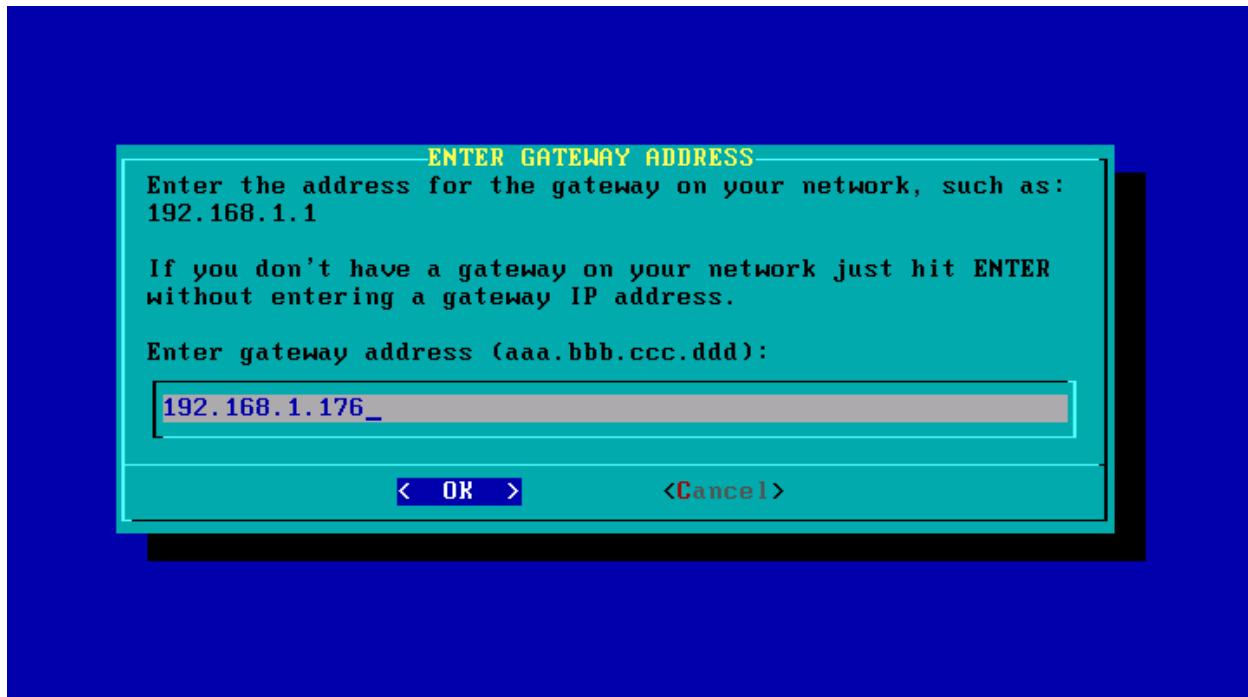
The next dialog will ask you whether you want to use a nameserver or not (Figure 5.26, "Choosing whether you want to use a nameserver or not"). A nameserver is a server that can provide the IP address for a hostname. For example, if you surf to *www.slackbasics.org*, the nameserver will "convert" the *www.slackbasics.org* name to the corresponding IP address.

Figure 5.26. Choosing whether you want to use a nameserver or not

If you chose to use a nameserver, you will be given the opportunity to set the IP address of the nameserver (Figure 5.27, "Setting the nameserver(s)").

Figure 5.27. Setting the nameserver(s)
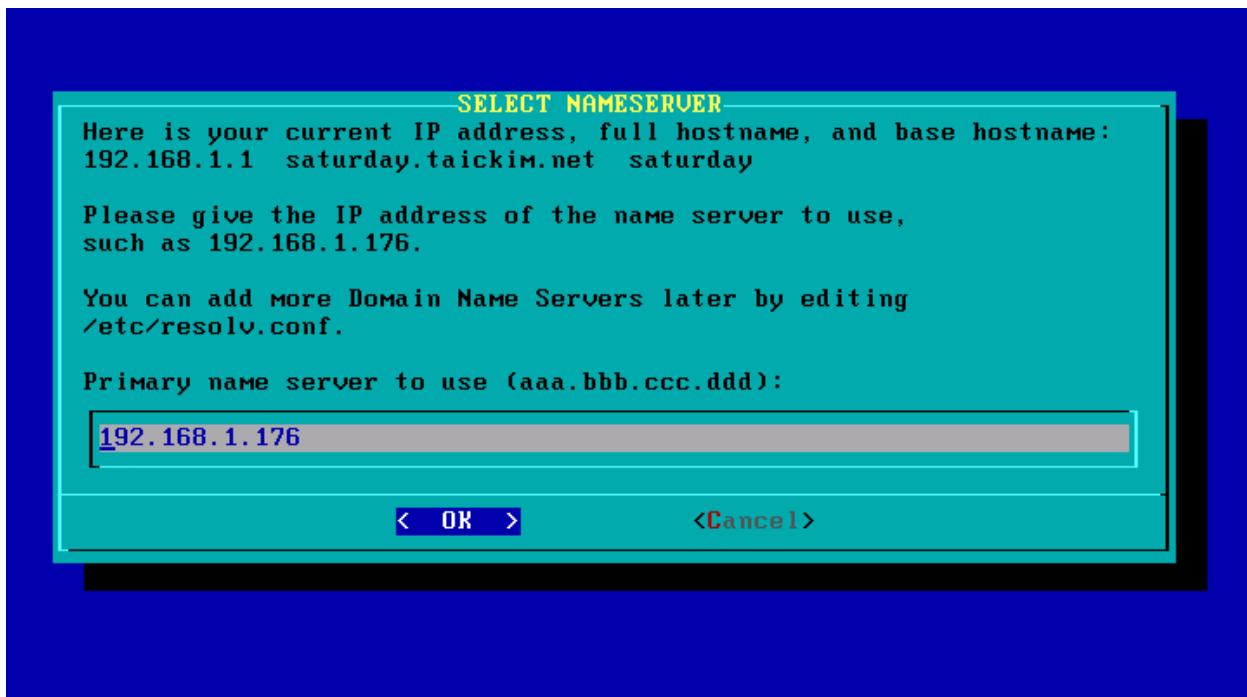
The final network settings screen provides an overview of the settings, giving you the opportunity to correct settings that have errors (Figure 5.28, "Confirming the network settings").

Figure 5.28. Confirming the network settings
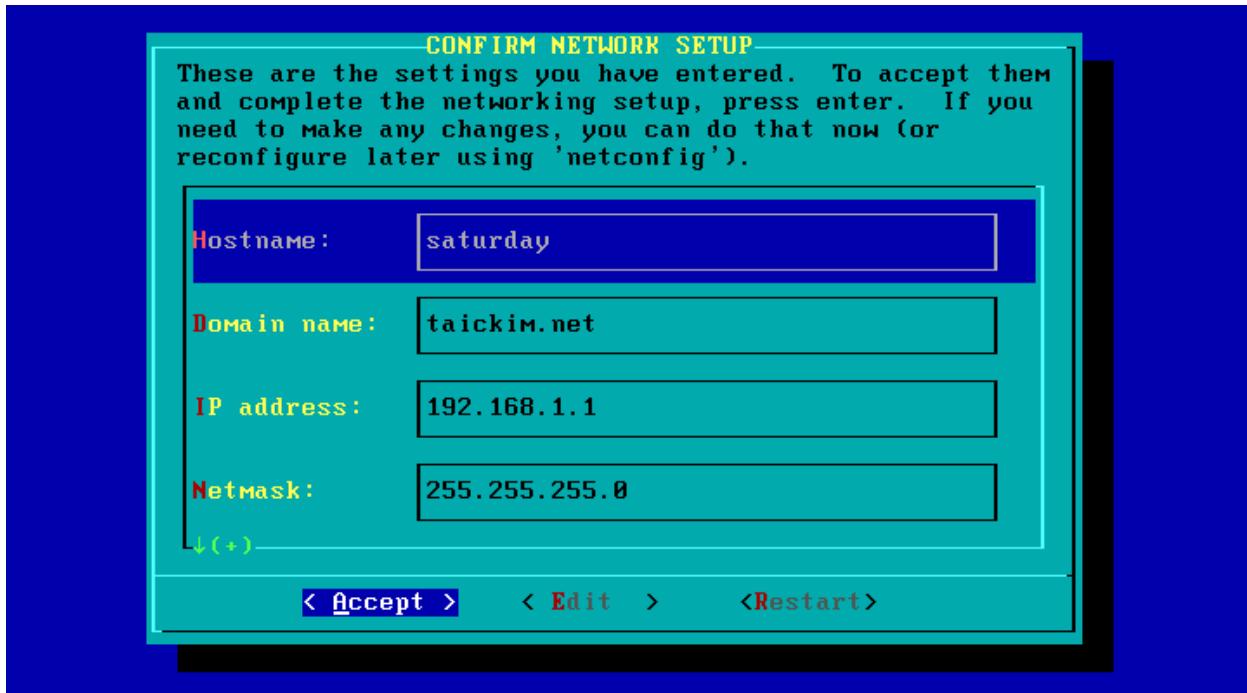
After the network configuration you can set which services should be started (Figure 5.29, "Enabling/disabling startup services"). You can mark/unmark services with the <SPACE> key.

Figure 5.29. Enabling/disabling startup services
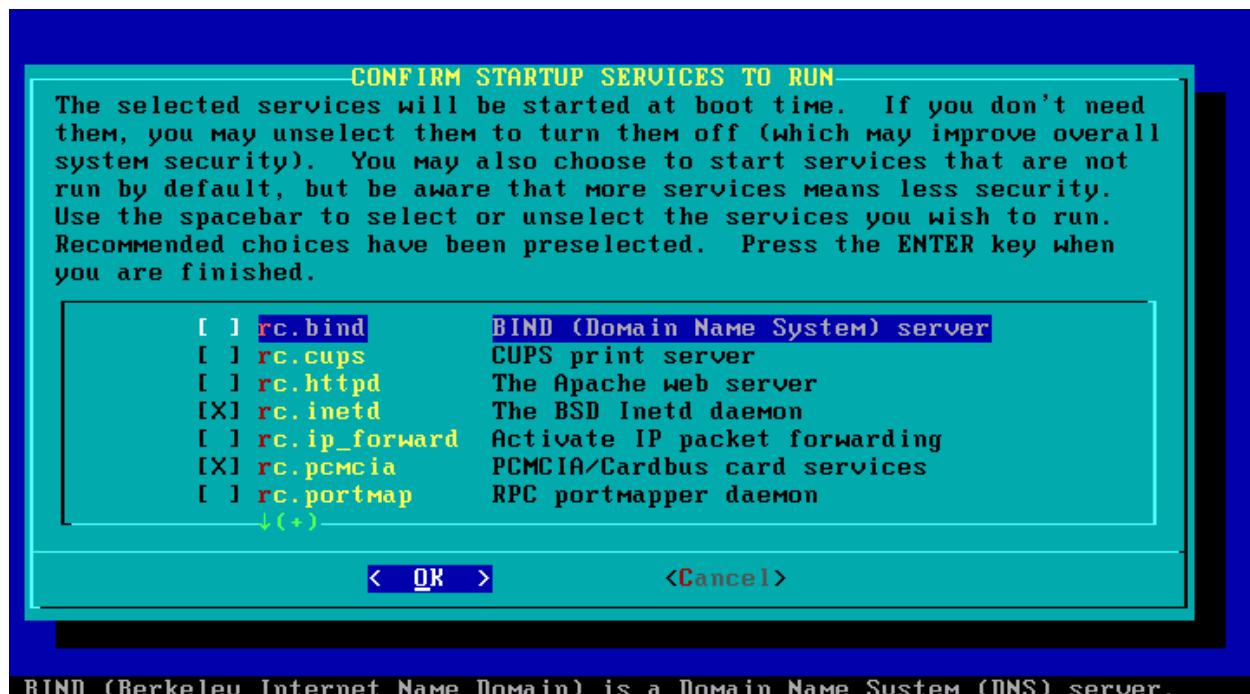
Traditionally the system clock is set to the UTC timezone on UNIX(-like) systems. If this is the case, select *Yes* during the next step (Figure 5.30, "Choosing whether the clock is set to UTC"). If you also use a non-UNIX(-like) OS on the same system, like Windows, it is usually a good idea to choose *No*, because some PC operating systems do not work with a separate system clock and software clock.

Figure 5.30. Choosing whether the clock is set to UTC
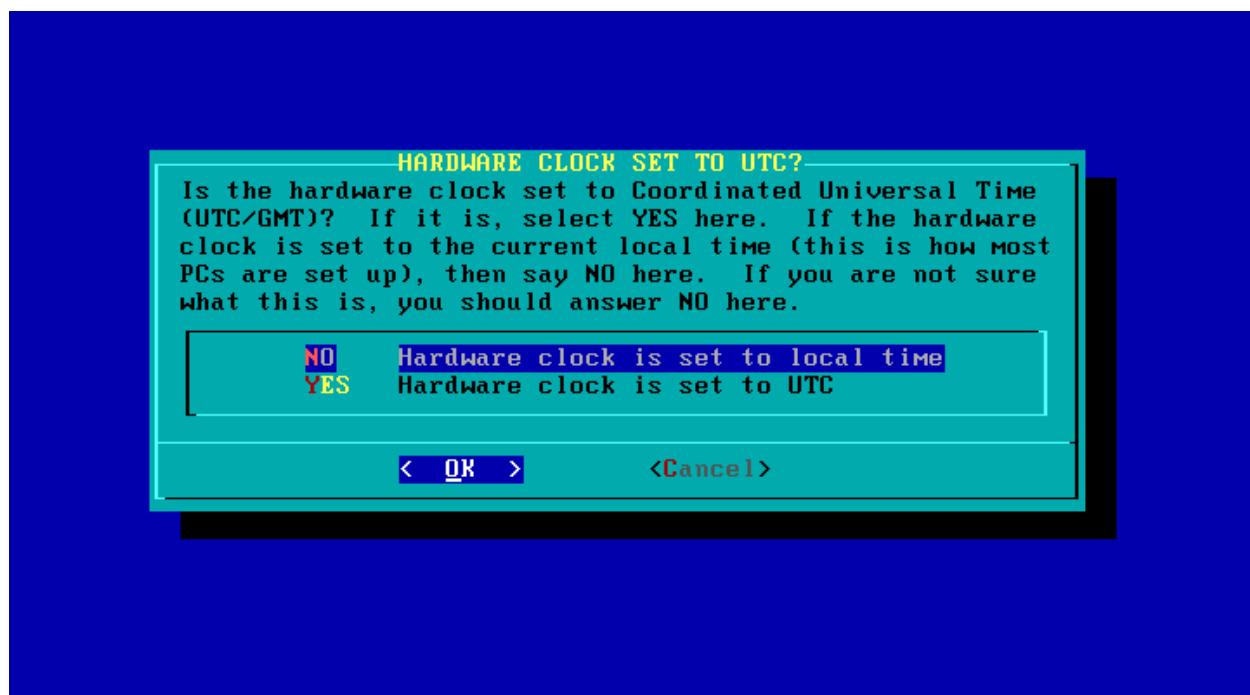
You will then be given the opportunity to select the time zone (Figure 5.31, "Setting the timezone"). This is especially important on systems that have their system clock set to UTC, without selecting the correct timezone the software clock will not match the local time.

Figure 5.31. Setting the timezone
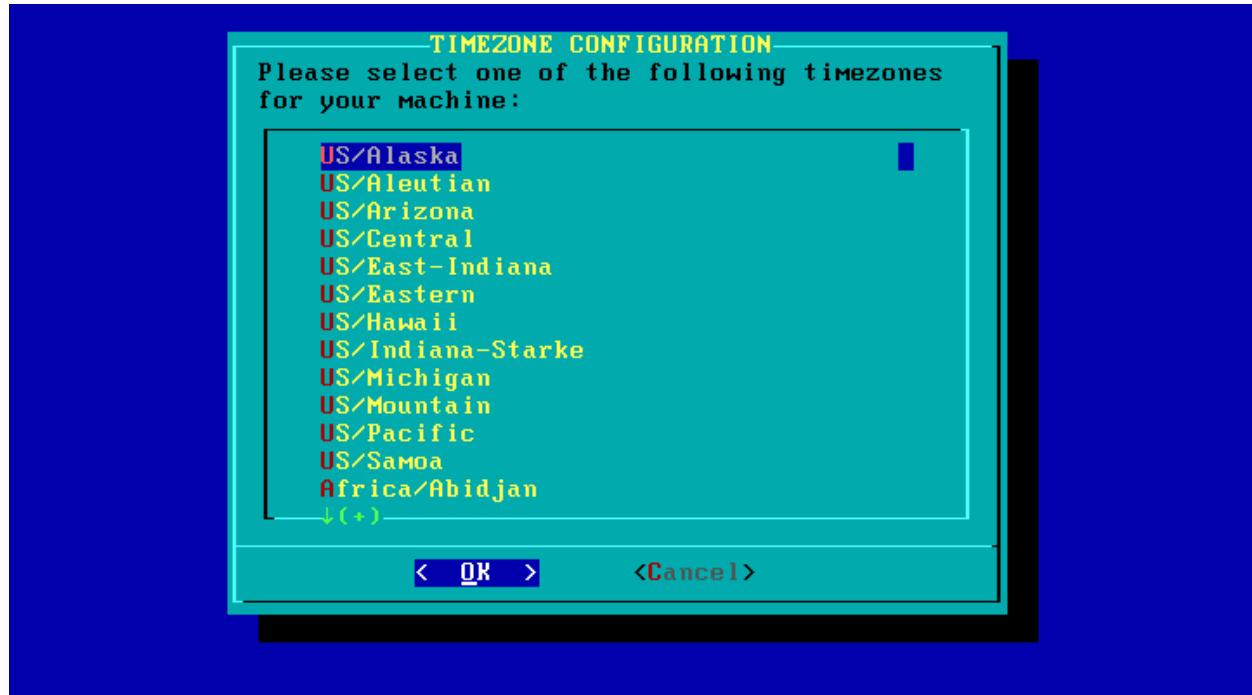
If you installed the X Window System you can now set the default window manager (Figure 5.32, "Choosing the default window manager"). The most basic functionality of a window manager is to provide basic window managing functionality like title bars. But some options, like KDE, provide a complete desktop environment.

Figure 5.32. Choosing the default window manager
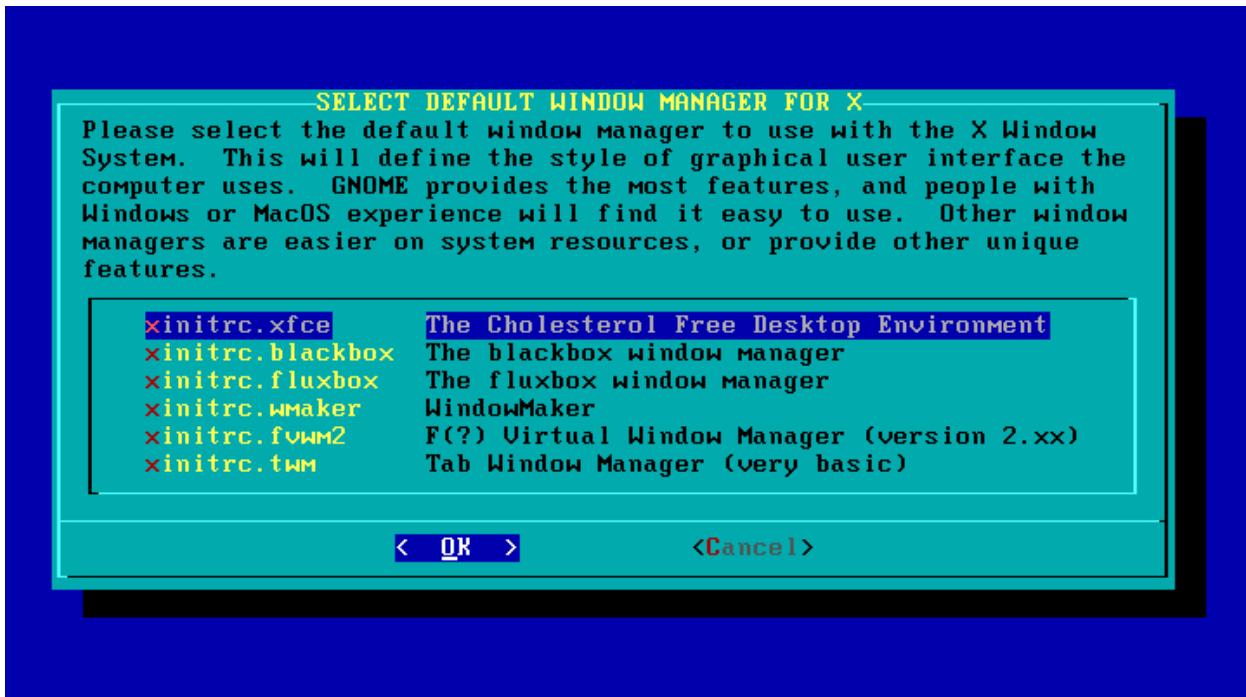
The final step is to set the root password (Figure 5.33, "Setting the root password"). The setup will ask you whether you would like to set it or not. There is no reason not to do this, and without a root password your system is dangerously insecure.

Figure 5.33. Setting the root password
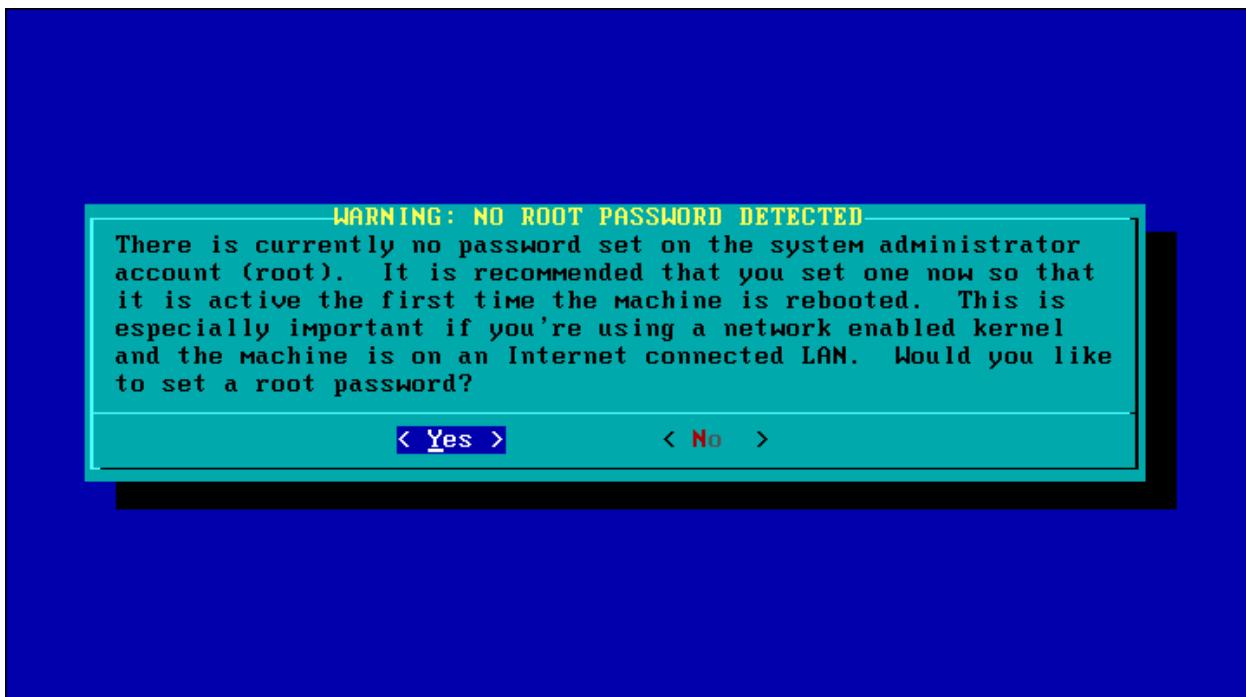
At this point you have completed the Slackware Linux installation. You can now reboot the system to start your fresh new Slackware Linux system. It wasn't that hard, was it? ;-)

Figure 5.34. Finished
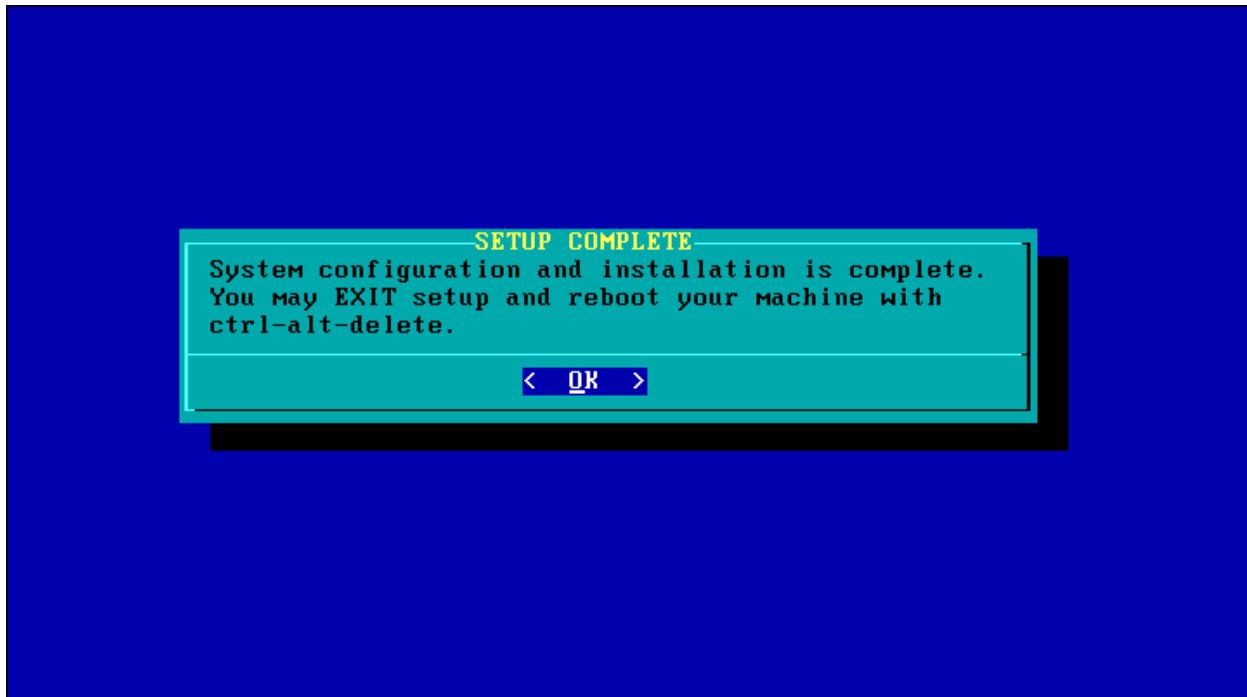
Chapter 6. Custom installation

Sometimes you may want to do a custom installation of Slackware Linux, for example to get better understanding of how GNU/Linux systems work, or to prepare an automatic installation script. This chapter outlines the steps that are required to install Slackware Linux manually. A sample installation script is also provided in Section 6.5, "Automated installation script".

6.1. Partitioning a hard disk

If you have performed a normal installation, you should not have any problems partitioning a disk. You can use the **fdisk** and **cfdisk** commands to partition disks. If you are scripting the installation of Slackware Linux it is useful to know that you can pipe **fdisk** commands to **fdisk**. For example:

```
# fdisk /dev/hda << EOF
n
p
1

+10000M
n
p
2

+512M
t
2
82
w
EOF
```

These commands create a primary Linux partition of 10000MB, and a primary Linux swap partition of 512MB. You could store the **fdisk** commands in different disk profiles, and use one of the profiles based on the specific deployment (e.g. the disk size). For example:

```
# cat /usr/share/fdisk-profiles/smalldisk | fdisk
```

6.2. Initializing and mounting filesystems

After making at least a swap and a Linux partition, you can initialize the filesystem and swap space and make use of this storage. On systems that are short on memory, you should initialize, and use swap first. We will use the partition layout used in the partitioning example listed above in the following examples. To set up and use swap, execute:

```
# mkswap /dev/hda2
# swapon /dev/hda2
```

The meaning of these commands is quite obvious. **mkswap** initializes the swap space, and **swapon** puts it to use. You will only have to execute **mkswap** once, but **swapon** has to be executed during every system boot. This can be done automatically by adding an entry for the swap partition to /etc/fstab, which we will do during a later step.

For now, it is important to initialize the target partitions. This can be done with the **mkfs** command. You can specify which filesystem should be used by adding the -t parameter. **mkfs** will automatically invoke a **mkfs.filesystem** command based on the filesystem that you have chosen. Be aware that the filesystems that can be used depends on the installation kernel that you have booted. If you have booted the *bare.i* kernel, you can use the *ext2*, *ext3* and *reiserfs* filesystems.

To initialize an *ext3* filesystem, and mount it, you should execute the following commands:

```
# mkfs -t ext3 /dev/hda1
# mount /dev/hda1 /mnt
```

If you have made separate partitions for certain directories in the root filesystem, e.g. /home, you can also initialize them and mount them at this point. For example:

```
# mkfs -t ext3 /dev/hda2
# mkdir /mnt/home
# mount /dev/hda2 /mnt/home
```

Finally, you will have to mount your source medium. If you use a CD-ROM, this is easy. Suppose that /dev/hdc is the CD-ROM device file, you can mount the CD-ROM with:

```
# mount /dev/hdc /var/log/mount
```

Using NFS as the installation medium requires some more steps. First of all, you will have to load the network disk. You can do this by running the **network** command and inserting a network disk. You can also load this disk from another medium, for example from an USB memory stick. Suppose that you have mounted a USB memory stick on /var/log/mount, you can load the network disk with: **network /var/log/mount/network.dsk**. After loading the network disk, you will have to configure the network interface. If the NFS server is on the same network, you will only have to assign an IP address to the network interface. For example, to use the address *192.168.2.201*, you could execute the following command:

```
# ifconfig eth0 192.168.2.201
```

You can then load the portmapper, which is necessary for the NFS client to function correctly:

```
# /sbin/rpc.portmap
```

If the portmapper started correctly, you can mount the NFS volume. But, first make sure that you unmount any filesystem that you have previously mounted at /var/log/mount. If no other filesystems are mounted at /var/log/mount, you can proceed with mounting the NFS volume. Suppose, that you want to mount *192.168.2.1:/home/pub/slackware-current*, you should issue the following command:

```
# mount -r -t nfs -o nolock 192.168.2.1:/home/pub/slackware-current /var/log/mount
```

If no errors where printed when the volume was mounted, it should be accessible through `/var/log/mount`

6.3. Installing packages

Everything is now set up to start installing packages from the installation medium. Since **installpkg** is available from the installation system, you can use it to install Slackware Linux packages. To install packages to the target partition(s) mounted on `/mnt`, add the `-root` option. The following command installs all packages from the source medium:

```
# installpkg -root /mnt /var/log/mount/slackware/*/*.tgz
```

If you have created tagfiles to define which packages should be installed, then you can use them now (tagfiles are described in Section 17.4, "Tagfiles"). Suppose that you have stored a tagfile for each disk set in `/usr/share/tagfiles/small-desktop`, you can install packages based on the tagfiles in the following manner:

```
# for p in a ap d e f k kde kdei l n t tcl x xap y; do
    installpkg -infobox -root /mnt \
      -tagfile /usr/share/tagfiles/small-desktop/$p/tagfile \
      /var/log/mount/slackware/$p/*.tgz
done
```

6.4. Post-install configuration

The next sections describe the bare minimum of configuration that is necessary to get a running system.

fstab

One of the necessary configuration steps is to create a **fstab** file, so that the system can look up what partitions or volumes have to be mounted. The format of the `/etc/fstab` file is described in the section called "The fstab file". As a bare minimum you will have to add entries for the `/` filesystem, the `/proc` pseudo-filesystem, the `devpts` pseudo-filesystem, and the swap partition.

With the sample partitioning used earlier in this chapter, you could create a `/etc/fstab` like this:

```
#  cat > /mnt/etc/fstab << EOF
/dev/hda2   swap                swap        defaults        0   0
/dev/hda1        /              ext3        defaults        1   1
devpts           /dev/pts       devpts      gid=5,mode=620  0   0
proc             /proc          proc        defaults        0   0
EOF
```

LILO

To make the system bootable you will have to configure and install the Linux Loader (LILO). The configuration of LILO is covered in Section 19.1, "The bootloader". For this section we will just show a sample LILO configuration, that can be used with the partition layout described in this chapter. The first step is to create the `/etc/lilo.conf` file:

```
# cat > /mnt/etc/lilo.conf << EOF
boot = /dev/hda
vga = normal
timeout = 50
image = /boot/vmlinuz
 root = /dev/hda1
 label = Slackware
 read-only
EOF
```

You can then install LILO with /mnt as the LILO root. With /mnt set as the root, LILO will use /etc/lilo.conf from the target partition:

```
# lilo -r /mnt
```

Networking

The configuration of networking in Slackware Linux is covered in Chapter 22, *Networking configuration*. This section will cover one example of a host that will use DHCP to get an IP address.

The /etc/networks file contains information about known Internet networks. Because we will get network information via DHCP, we will just use *127.0.0.1* as the local network.

```
# cat > /mnt/etc/networks << EOF
loopback        127.0.0.0
localnet        127.0.0.0
EOF
```

Although we will get a hostname via DHCP, we will still set up a temporary hostname:

```
# cat > /mnt/etc/HOSTNAME << EOF
sugaree.example.net
EOF
```

Now that the hostname is configured, the hostname and *localhost* should also be made resolvable, by creating a /etc/hosts database:

```
# cat > /mnt/etc/hosts << EOF
127.0.0.1 localhost
127.0.0.1 sugaree.example.net sugaree
EOF
```

We do not have to create a /etc/resolv.conf, since it will be created automatically by **dhcpcd**, the DHCP client. So, finally, we can set up the interface in /etc/rc.d/rc.inet1.conf:

```
# cat > /mnt/etc/rc.d/rc.inet1.conf << EOF
IPADDR[0]=""
NETMASK[0]=""
USE_DHCP[0]="yes"
DHCP_HOSTNAME[0]=""
EOF
```

You may wat to make a backup of the old `rc.inet1.conf` file first, because it contains many useful comments. Or you can use **sed** to change this specific option:

```
# sed -i 's/USE_DHCP\[0\]=""/USE_DHCP[0]="yes"/' \
    /mnt/etc/rc.d/rc.inet1.conf
```

Tuning initialization scripts

Depending on the purpose of the system that is being installed, it should be decided which initialization scripts should be started. The number of services that are available depends on what packages you have installed. You can get get a list of available scripts with **ls**:

```
# ls -l /mnt/etc/rc.d/rc.*
```

If the executable bits are set on a script, it will be started, otherwise it will not. Obviously you should keep essential scripts executable, including the runlevel-specific scripts. You can set the executable bit on a script with:

```
# chmod +x /mnt/etc/rc.d/rc.scriptname
```

Or remove it with:

```
# chmod -x /mnt/etc/rc.d/rc.scriptname
```

Configuring the dynamic linker run-time bindings

GNU/Linux uses a cache for loading dynamic libraries. Besides that many programs rely on generic version numbers of libraries (e.g. `/usr/lib/libgtk-x11-2.0.so`, rather than `/usr/lib/libgtk-x11-2.0.so.0.600.8`). The cache and library symlinks can be updated in one **ldconfig** run:

```
# chroot /mnt /sbin/ldconfig
```

You may not know the **chroot** command; it is a command that executes a command with a different root than the active root. In this example the root directory is changed to `/mnt`, and from there **chroot** runs **/sbin/ldconfig** binary. After the command has finished the system will return to the shell, which uses the original root. To use **chroot** with other commands this **ldconfig** command has to be executed once first, because without initializing the dynamic linker run-time bindings most commands will not execute, due to unresolvable library dependencies.

Setting the root password

Now that the dynamic library cache and links are set up, you can execute commands on the installed system. We will make use of this to set the *root* password. The **passwd** command can be used to set the password for an existing user (the *root* user is part of the initial /etc/passwd file). We will use the **chroot** command again to execute the command on the target partition:

```
# chroot /mnt /usr/bin/passwd root
```

Setting the timezone

On UNIX-like systems it is important to set the timezone correctly, because it is used by various parts of the system. For instance, the timezone is used by NTP to synchronize the system time correctly, or by different networking programs to compute the time difference between a client and a server. On Slackware Linux the timezone can be set by linking /etc/localtime to a timezone file. You can find the timezone for your region by browsing the /mnt/usr/share/zoneinfo directory. Most timezones can be found in the subdirectories of their respective regions. For example to use *Europe/Amsterdam* as the timezone, you can execute the following commands:

```
# cd /mnt
# rm -rf etc/localtime
# ln -sf /usr/share/zoneinfo/Europe/Amsterdam etc/localtime
```

After setting the time zone, programs still do not know whether the hardware clock is set to the local time, or to the Coordinated Universal Time (UTC) standard. If you use another operating system on the same machine that does not use UTC it is best to set the hardware clock to the local time. On UNIX-like systems it is a custom to set the system time to UTC. You can set what time the system clock uses, by creating the file /etc/hardwareclock, and putting the word *localtime* in it if your clock is set to the local time, or *UTC* when it is set to UTC. For example, to use UTC, you can create /etc/hardwareclock in the following manner:

```
# echo "UTC" > /mnt/etc/hardwareclock
```

Creating the X11 font cache

If you are going to use X11, you will have to initialize the font cache for TrueType and Type1 fonts. This can be done with the **fc-cache** command:

```
# chroot /mnt /usr/bin/fc-cache
```

6.5. Automated installation script

It is easy to combine the steps of a custom installation into one script, which performs the custom steps automatically. This is ideal for making default server installs or conducting mass roll-outs of Linux clients. This sections contains a sample script that was written by William Park. It is easy to add an installation script to the Slackware Linux medium, especially if you use an installation CD-ROM or boot the installation system from an USB flash drive.

The installation system is stored in one compressed image file, that is available on the distribution medium as `isolinux/initrd.img`. You can make a copy of this image to your hard disk, and decompress it with **gunzip**:

```
# mv initrd.img initrd.img.gz
# gunzip initrd.img.gz
```

After decompressing the image, you can mount the file as a disk, using the loopback device:

```
# mount -o loop initrd.img /mnt/hd
```

You can now add a script to the initrd file by adding it to the directory structure that is available under the mount point. After making the necessary changes, you can unmount the filesystem and compress it:

```
# umount /mnt/hd
# gzip initd.img
# mv initrd.img.gz initrd.img
```

You can then put the new `initrd.img` file on the installation medium, and test the script.

```
#! /bin/sh
# Copyright (c) 2003-2005 by William Park <opengeometry@yahoo.ca>.
# All rights reserved.
#
# Usage: slackware-install.sh

rm_ln ()  # Usage: rm_ln from to
{
 rm -rf $2; ln -sf $1 $2
}


##############################################################################
echo "Partitioning harddisk..."

(   echo -ne "n\np\n1\n\n+1000M\n" # /dev/hda1 --> 1GB swap
 echo -ne "n\np\n2\n\n+6000M\n" # /dev/hda2 --> 6GB /
 echo -ne "t\n1\n82\nw\n"
) | fdisk /dev/hda

mkswap /dev/hda1  # swap
swapon /dev/hda1

mke2fs -c /dev/hda2  # /
mount /dev/hda2 /mnt


##############################################################################
echo "Installing packages..."
```

```
mount -t iso9660 /dev/hdc /cdrom # actually, /var/log/mount
cd /cdrom/slackware
for p in [a-z]*; do  # a, ap, ..., y
 installpkg -root /mnt -priority ADD $p/*.tgz
done

cd /mnt


##############################################################################
echo "Configuring /dev/* stuffs..."

rm_ln psaux dev/mouse  # or, 'input/mice' for usb mouse
rm_ln ttyS0 dev/modem
rm_ln hdc dev/cdrom
rm_ln hdc dev/dvd


##############################################################################
echo "Configuring /etc/* stuffs..."

cat > etc/fstab << EOF
/dev/hda1    swap          swap      defaults          0  0
/dev/hda2    /             ext2      defaults          1  1
devpts       /dev/pts      devpts    gid=5,mode=620    0  0
proc         /proc         proc      defaults          0  0
#
/dev/cdrom   /mnt/cdrom    iso9660   noauto,owner,ro   0  0
/dev/fd0     /mnt/floppy   auto      noauto,owner      0  0
tmpfs        /dev/shm      tmpfs     noauto            0  0
EOF

cat > etc/networks << EOF
loopback 127.0.0.0
localnet 192.168.1.0
EOF
cat > etc/hosts << EOF
127.0.0.1 localhost
192.168.1.1 node1.example.net node1
EOF
cat > etc/resolv.conf << EOF
search example.net
nameserver 127.0.0.1
EOF
cat > etc/HOSTNAME << EOF
node1.example.net
EOF


## setup.05.fontconfig
chroot . /sbin/ldconfig  # must be run before other program
chroot . /usr/X11R6/bin/fc-cache

chroot . /usr/bin/passwd root
```

```
## setup.06.scrollkeeper
chroot . /usr/bin/scrollkeeper-update

## setup.timeconfig
rm_ln /usr/share/zoneinfo/Canada/Eastern  etc/localtime
cat > etc/hardwareclock << EOF
localtime
EOF

## setup.liloconfig
cat > etc/lilo.conf << EOF
boot=/dev/hda
delay=100
vga=normal # 80x25 char
# VESA framebuffer console:
#   pixel char 8bit 15bit 16bit 24bit
#   ----- ---- ---- ----- ----- -----
#   1600x1200  796 797 798 799
#   1280x1024 160x64 775 793 794 795
#   1024x768 128x48 773 790 791 792
#   800x600 100x37 771 787 788 789
#   640x480 80x30 769 784 785 786
image=/boot/vmlinuz  # Linux
 root=/dev/hda2
 label=bare.i
 read-only
# other=/dev/hda1  # Windows
#     label=win
#     table=/dev/hda
EOF
lilo -r .

## setup.xwmconfig
rm_ln xinitrc.fvwm95  etc/X11/xinit/xinitrc


##########################################################################
echo "Configuring /etc/rc.d/rc.* stuffs..."

cat > etc/rc.d/rc.keymap << EOF
#! /bin/sh
[ -x /usr/bin/loadkeys ] && /usr/bin/loadkeys us.map
EOF
chmod -x etc/rc.d/rc.keymap

## setup.mouse
cat > etc/rc.d/rc.gpm << 'EOF'
#! /bin/sh
case $1 in
 stop)
    echo "Stopping gpm..."
    /usr/sbin/gpm -k
    ;;
```

```
 restart)
      echo "Restarting gpm..."
      /usr/sbin/gpm -k
      sleep 1
      /usr/sbin/gpm -m /dev/mouse -t ps2
      ;;
 start)
      echo "Starting gpm..."
      /usr/sbin/gpm -m /dev/mouse -t ps2
      ;;
 *)
      echo "Usage $0 {start|stop|restart}"
      ;;
esac
EOF
chmod +x etc/rc.d/rc.gpm

## setup.netconfig
cat > etc/rc.d/rc.inet1.conf << EOF
IPADDR=(192.168.1.1)  # array variables
NETMASK=(255.255.255.0)
USE_DHCP=()   # "yes" or ""
DHCP_HOSTNAME=()
GATEWAY=""
DEBUG_ETH_UP="no"
EOF

cat > etc/rc.d/rc.netdevice << EOF
/sbin/modprobe 3c59x
EOF
chmod +x etc/rc.d/rc.netdevice

## setup.setconsolefont
mv etc/rc.d/rc.font{.sample,}
chmod -x etc/rc.d/rc.font

## setup.services
chmod +x etc/rc.d/rc.bind
chmod +x etc/rc.d/rc.hotplug
chmod +x etc/rc.d/rc.inetd
chmod +x etc/rc.d/rc.portmap
chmod +x etc/rc.d/rc.sendmail
#
chmod -x etc/rc.d/rc.atalk
chmod -x etc/rc.d/rc.cups
chmod -x etc/rc.d/rc.httpd
chmod -x etc/rc.d/rc.ip_forward
chmod -x etc/rc.d/rc.lprng
chmod -x etc/rc.d/rc.mysqld
chmod -x etc/rc.d/rc.pcmcia
chmod -x etc/rc.d/rc.samba
chmod -x etc/rc.d/rc.sshd
```

Part II. Slackware Linux Basics

Table of Contents

Chapter 7. The shell

7.1. Introduction

In this chapter we will look at the traditional working environment of UNIX systems: the shell. The shell is an interpreter that can be used interactively and non-interactively. When the shell is used non-interactively it functions as a simple, but powerful scripting language.

The procedure for starting the shell depends on whether you use a graphical or text-mode login. If you are logging on in text-mode the shell is immediately started after entering the (correct) password. If you use a graphical login manager like KDM, log on as you would normally, and look in your window manager or desktop environment menu for an entry named "XTerm", "Terminal" or "Konsole". XTerm is a terminal emulator, after the terminal emulator is started the shell comes up.

Before we go any further, we have to warn you that Slackware Linux provides more than just one shell. There are two shell flavors that have become popular over time, the Bourne shell and the C shell. In this chapter we will describe Bourne shells that conform to the IEEE 1003.1 standard. The Bash (Bourne Again Shell) and ksh (Korn Shell) shells conform well to these standards. So, it is a good idea to use one of these two shells. You can easily see what shell the system is running by executing **echo $SHELL**. This is what a Bash shell may report:

```
$ echo $SHELL
/bin/bash
```

If you are using a different shell, you can change your default shell. Before setting a different shell, you have to establish the full path of the shell. You can do this with the **which** command. For example:

```
$ which bash
/bin/bash
$ which ksh
/bin/ksh
```

On this Slackware system, the full path to the bash shell is /bin/bash, and to the ksh shell /bin/ksh. With this information, and the **chsh** command you change the default shell. The following example will set the default shell to bash:

```
$ chsh -s /bin/bash
Changing shell for daniel.
Password:
Shell changed.
```

The new shell will be activated after logging out from the current shell (with **logout** or **exit**), or by opening a new X terminal window if you are running X11.

7.2. Executing commands

An interactive shell is used to start programs by executing commands. There are two kinds of commands that a shell can start:

- *Built-in commands:* built-in commands are integrated in the shell. Commonly used built-in commands are: **cd**, **fg**, **bg**, and **jobs**.

- *External commands:* external commands are programs that are not part of the shell program, and are separately stored on the filesystem. Commonly used external commands are: **ls**, **cat**, **rm**, and **mkdir**.

All shell commands are executed with the same syntax:

```
commandname [argument1 argument2 ... argumentn]
```

The number of arguments is arbitrary, and are always passed to the command. The command can decide what it does with these arguments.

All built-in commands can always be executed, because they are part of the shell. External commands can be executed by name when the program is in the search path of the shell. Otherwise, you will have to specify the path to the program. The search path of the shell is stored in a variable named *PATH*. A variable is a named piece of memory, of which the contents can be changed. We can see the contents of the PATH variable in the following manner:

```
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/daniel/bin
```

The directory paths in the *PATH* variable are separated with the colon (*:*) character. You can use the **which** command to check whether a given command is in the current shell path. You can do this by providing the command as an argument to **which**. For example:

```
$ which pwd
/bin/pwd
$ which sysstat
/usr/bin/which: no sysstat in (/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:
```

If a program is not in the path, you can still run it by entering its absolute or relative path.

7.3. Moving around

It is often necessary to jump through various parts of a line, and to alter it, when you are editing larger commands. Both **bash** and **ksh** have keyboard shortcuts for doing common operations. There are two shell modes, in which the shortcut keys differ. These modes correspond with two popular editors for UNIX in their behavior. These editors are **vi** and **emacs**. In this book we will only cover the EMACS-like keystrokes. You can check in which mode a shell is running by printing the *SHELLOPTS* variable. In the first example the shell is used in *emacs* mode, in the second example the *vi* mode is used. You identify the mode by looking for the *emacs* or *vi* strings in the contents of the variable.

```
$ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

```
$ echo $SHELLOPTS
braceexpand:hashall:histexpand:history:interactive-comments:monitor:vi
```

If your shell is currently using the *vi* mode, you can switch to the *emacs* mode by setting the *emacs* option:

```
$ set -o emacs
```

With the *emacs* editing mode enabled, you can start using shortcuts. We will look at three kinds of shortcuts: character editing shortcuts, word editing shortcuts, and line editing shortcuts. Later in this chapter, we will also have a look at some shortcuts that are used to retrieve entries from the command history.

Character editing

The first group of shortcuts have characters as their logic unit, meaning that they allow command line editing operations on characters. Table 7.1, "Moving by character" provides an overview of the shortcuts that are used to move through a line by character.

Table 7.1. Moving by character

Keys	Description
Ctrl-b	Move a character backwards.
Ctrl-f	Move a character forward.

These shortcuts are simple, and don't do anything unexpected. Suppose that you have typed the following line:

```
find ~/music -name '*.ogg' - -print
```

The cursor will be at the end. You can now move to the start of the line by holding *Ctrl-b*:

```
find ~/music - -name '*.ogg' -print
```

Likewise, you can go back again to the end by holding *Ctrl-f*. There is an error in this line, since there is one erroneous dash. To remove this dash, you can use one of the character deletion shortcuts.

Table 7.2. Deleting characters

Keys	Description
Ctrl-h	Delete a character before the cursor. This has the same effect as using the Backspace key on most personal computers.
Ctrl-d	Delete the character the cursor is on.

You can delete the dash in two manners. The first way is to move the cursor to the dash:

```
find ~/music - -name '*.ogg' -print
```

and then press *Ctrl-d* twice. This will delete the dash character, and the space that follows the dash:

```
find ~/music ▉name '*.ogg' -print
```

Looking at the original fragment, the other approach is to position the cursor on the space after the dash:

```
find ~/music -▉ -name '*.ogg' -print
```

and then press *Ctrl-h* twice to delete the two preceding characters, namely the dash and the space before the dash. The result will be the same, except that the cursor will move:

```
find ~/music▉-name '*.ogg' -print
```

One of the nice features of most modern shells is that you can transpose (swap) characters. This is handy if you make a typing error in which two characters are swapped. Table 7.3, "Swapping characters" lists the shortcut for transposing characters.

Table 7.3. Swapping characters

Keys	Description
Ctrl-t	Swap (transpose) the characters the cursor is on, and the character before the cursor. This is handy for quickly correcting typing errors.

Suppose that you have typed the following command:

```
cat myreport.ttx
```

The extension contains a typing error if you intended to **cat** `myreport.txt`. This can be corrected with the character transpose shortcut. First move to the second character of the pair of characters that are in the wrong order:

```
cat myreport.tt▉
```

You can then press *Ctrl-t*. The characters will be swapped, and the cursor will be put behind the swapped characters:

```
cat myreport.txt▉
```

Word editing

It if often tedious to move at character level. Fortunately the Korn and Bash shells can also move through lines at a word level. Words are sequences of characters that are separated by a special character, such as a space. Table 7.4, "Moving by word" summarizes the shortcuts that can be used to navigate through a line by word.

Table 7.4. Moving by word

Keys	Description
Esc b	Move back to the start of the current or previous word.
Esc f	Move forward to the last character of the current or next word.

As you can see the letters in these shortcuts are equal to those of moving forward and backwards by character. The movement logic is a bit curious. Moving forward puts the cursor to the end of the current word, not to the first character of the next word as you may have predicted. Let's look at a quick example. In the beginning the cursor is on the first character of the line.

```
find ~/music -name '*.ogg' -print
```

Pressing *Esc f* will move the cursor behind the last character of the first word, which is *find* in this case:

```
find ~/music -name '*.ogg' -print
```

Going forward once more will put the cursor behind *~/music*:

```
find ~/music -name '*.ogg' -print
```

Backwards movement puts the cursor on the first character of the current word, or on the first character of the previous word if the cursor is currently on the first character of a word. So, moving back one word in the previous example will put the cursor on the first letter of "music":

```
find ~/music -name '*.ogg' -print
```

Deleting words works equal to moving by word, but the characters that are encountered are deleted. Table 7.5, "Deleting words" lists the shortcuts that are used to delete words.

Table 7.5. Deleting words

Keys	Description
Alt-d	Delete the word, starting at the current cursor position.
Alt-Backspace	Delete every character from the current cursor position to the first character of a word that is encountered.

Finally, there are some shortcuts that are useful to manipulate words. These shortcuts are listed in Table 7.6, "Modifying words".

Table 7.6. Modifying words

Keys	Description
Alt-t	Swap (transpose) the current word with the previous word.
Alt-u	Make the word uppercase, starting at the current cursor position.
Alt-l	Make the word lowercase, starting at the current cursor position.
Alt-c	Capitalize the current word character or the next word character that is encountered.

Transposition swaps words. If normal words are used, it's behavior is predictable. For instance, if we have the following line with the cursor on "two"

```
one two three
```

Word transposition will swap "two" and "one":

```
two one three
```

But if there are any non-word characters, the shell will swap the word with the previous word while preserving the order of non-word characters. This is very handy for editing arguments to commands. Suppose that you made an error, and mixed up the file extension you want to look for, and the *print* parameter:

```
find ~/music -name '*.print' -ogg
```

You can fix this by putting the cursor on the second faulty word, in this case "ogg", and transposing the two words. This will give the result that we want:

```
find ~/music -name '*.ogg' -print
```

Finally, there are some shortcuts that change the capitalization of words. The Alt-u shortcut makes all characters uppercase, starting at the current cursor position till the end of the word. So, if we have the lowercase name "alice", uppercasing the name with the cursor on "i" gives "alICE". Alt-l has the same behavior, but changes letters to lowercase. So, using Alt-l on "alICE" with the cursor on "I" will change the string to "alice". Alt-c changes just the character the cursor is on, or the next word character that is encountered, to uppercase. For instance, pressing Alt-c with the cursor on "a" in "alice" will yield "Alice".

Line editing

The highest level we can edit is the line itself. Table 7.7, "Moving through lines" lists the two movement shortcuts.

Table 7.7. Moving through lines

Keys	Description
Ctrl-a	Move to the beginning of the current line.
Ctrl-e	Move to the end of the current line.

Suppose that the cursor is somewhere halfway a line:

```
find ~/music -name '*.ogg' -print
```

Pressing Ctrl-e once will move the cursor to the end of the line:

```
find ~/music -name '*.ogg' -print
```

Pressing Ctrl-a will move the cursor to the beginning of the line:

```
find ~/music -name '*.ogg' -print
```

You can also delete characters by line level. The shortcuts are listed in Table 7.8, "Deleting lines". These shortcuts work like movement, but deletes all characters that are encountered. Ctrl-k will delete the character the cursor is on, but Ctrl-x Backspace will not. Moving to the beginning of the line with Ctrl-a, followed by Ctrl-k, is a fast trick to remove a line completely.

Table 7.8. Deleting lines

Keys	Description
Ctrl-k	Delete all characters in the line, starting at the cursor position.
Ctrl-x Backspace	Delete all characters in the line up till the current cursor position.

7.4. Command history

It often happens that you have to execute commands that you executed earlier. Fortunately, you do not have to type them all over again. You can browse through the history of executed commands with the up and down arrows. Besides that it is also possible to search for a command. Press Control-r and start typing the command you want to execute. You will notice that bash will display the first match it can find. If this is not the match you were looking for you can continue typing the command (until it is unique and a match appears), or press Control-r once more to get the next match. When you have found the command you were looking for, you can execute it by pressing <Enter>.

7.5. Completion

Completion is one of the most useful functionalities of UNIX-like shells. Suppose that you have a directory with two files named `websites` and `recipe`. And suppose you want to **cat** the file `websites` (**cat** shows the contents of

a file), by specifying `websites` as a parameter to cat. Normally you would type "cat websites", and execute the command. Try typing "cat w", and hit the <Tab> key. Bash will automatically expand what you typed to "cat websites".

But what happens if you have files that start with the same letter? Suppose that you have the `recipe1.txt` and `recipe2.txt` files. Type "cat r" and hit <Tab>, Bash will complete the filename as far as it can. It would leave you with "cat recipe". Try hitting <Tab> again, and Bash will show you a list of filenames that start with "recipe", in this case both recipe files. At this point you have to help Bash by typing the next character of the file you need. Suppose you want to **cat** `recipe2`, you can push the <2> key. After that there are no problems completing the filename, and hitting <Tab> completes the command to "cat recipe2.txt".

It is worth noting that completion also works with commands. Most GNU/Linux commands are quite short, so it will not be of much use most of the time.

It is a good idea to practice a bit with completion, it can save alot of keystrokes if you can handle completion well. You can make some empty files to practice with using the **touch** command. For example, to make a file named `recipe3.txt`, execute **touch recipe3.txt**.

7.6. Wildcards

Most shells, including Bash and ksh, support wildcards. Wildcards are special characters that can be used to do pattern matching. The table listed below displays some commonly used wildcards. We are going to look at several examples to give a general idea how wildcards work.

Table 7.9. Bash wildcards

Wildcard	Matches
*	A string of characters
?	A single character
[]	A character in an array of characters

Matching a string of characters

As you can see in the table above the "*" character matches a string of characters. For example, *.html* matches everything ending with *.html*, *d*.html* matches everything starting with a *d* and ending with *.html*.

Suppose that you would like to list all files in the current directory with the *.html* extension, the following command will do the job:

```
$ ls *.html
book.html          installation.html      pkgmgmt.html   usermgmt.html
filesystem.html    internet.html          printer.html   xfree86.html
gfdl.html          introduction.html      proc.html
help.html          slackware-basics.html  shell.html
```

Likewise we could remove all files starting with an *in*:

```
$ rm in*
```

Matching single characters

The "?" wildcard works as the "*" wildcard, but matches single characters. Suppose that we have three files, `file1.txt`, `file2.txt` and `file3.txt`. The string *file?.txt* matches all three of these files, but it does not match `file10.txt` ("10" are two characters).

Matching characters from a set

The "[]" wildcard matches every character between the brackets. Suppose we have the files from the previous example, `file1.txt`, `file2.txt` and `file3.txt`. The string *file[23].txt* matches `file2.txt` and `file3.txt`, but not `file1.txt`.

7.7. Redirections and pipes

One of the main features of UNIX-like shells are redirections and pipes. Before we start to look at both techniques we have to look how most UNIX-like commands work. When a command is not getting data from a file, it will open a special pseudo-file named *stdin*, and wait for data to appear on it. The same principle can be applied for command output, when there is no explicit reason for saving output to a file, the pseudo-file *stdout* will be opened for output of data. This principle is shown schematically in Figure 7.1, "Standard input and output"

Figure 7.1. Standard input and output

You can see *stdin* and *stdout* in action with the **cat** command. If cat is started without any parameters it will just wait for input on *stdin* and output the same data on *stdout*. If no redirection is used keyboard input will be used for *stdin*, and *stdout* output will be printed to the terminal:

```
$ cat
Hello world!
Hello world!
```

As you can see cat will print data to *stdout* after inputting data to *stdin* using the keyboard.

Redirection

The shell allows you to take use of *stdin* and *stdout* using the "<" and ">". Data is redirected in which way the sharp bracket points. In the following example we will redirect the md5 summaries calculated for a set of files to a file named `md5sums`:

```
$ md5sum * > md5sums
$ cat md5sums
6be249ef5cacb10014740f61793734a8  test1
220d2cc4d5d5fed2aa52f0f48da38ebe  test2
631172a1cfca3c7cf9e8d0a16e6e8cfe  test3
```

As we can see in the **cat** output the output of the **md5sum *** output was redirected to the `md5sums` file. We can also use redirection to provide input to a command:

```
$ md5sum < test1
6be249ef5cacb10014740f61793734a8  -
```

This feeds the contents of the `test1` to **md5sum**.

Pipes

You can also connect the input and output of commands using so-called *pipes*. A pipe between commands can be made with the "|" character. Two or more combined commands are called a *pipeline*. Figure 7.2, "A pipeline" shows a schematic overview of a pipeline consisting of two commands.

Figure 7.2. A pipeline
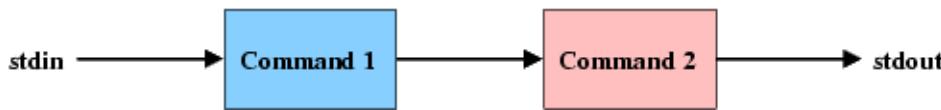
The "syntax" of a pipeline is: **command1 | command2 ... | commandn**. If you know how the most basic UNIX-like commands work you can now let these commands work together. Let's look at a quick example:

```
$ cat /usr/share/dict/american-english | grep "aba" | wc -l
123
```

The first command, **cat**, reads the dictionary file `/usr/share/dict/american-english`. The output of the **cat** command is piped to **grep**, which prints out all files containing the phrase "aba". In turn, the output of "grep" is piped to **wc -l**, which counts the number of lines it receives from *stdin*. Finally, when the stream is finished **wc** prints the number of lines it counted. So, combined three commands to count the number of words containing the phrase "aba" in this particular dictionary.

There are hundreds of small utilities that handle specific tasks. As you can imagine, together these commands provide a very powerful toolbox by making combinations using pipes.

Chapter 8. Files and directories

8.1. Some theory

Before we move on to look at practical filesystem operations, we are going to look at a more theoretical overview of how filesystems on UNIX-like systems work. Slackware Linux supports many different filesystems, but all these filesystems use virtually the same semantics. These semantics are provided through the *Virtual Filesystem* (VFS) layer, giving a generic layer for disk and network filesystems.

inodes, directories and data

The filesystem consists of two types of elements: data and metadata. The metadata describes the actual data blocks that are on the disk. Most filesystems use information nodes (inodes) to provide store metadata. Most filesystems store the following data in their inodes:

Table 8.1. Common inode fields

Field	Description
mode	The file permissions.
uid	The user ID of the owner of the file.
gid	The group ID of the group of the file.
size	Size of the file in bytes.
ctime	File creation time.
mtime	Time of the last file modification.
links_count	The number of links pointing to this inode.
i_block	Pointers to data blocks

If you are not a UNIX or Linux afficiendo, these names will probably sound bogus to you, but we will clear them up in the following sections. At any rate, you can probably deduct the relation between inodes and data from this table, and specifically the *i_block* field: every inode has pointers to the data blocks that the inode provides information for. Together, the inode and data blocks are the actual file on the filesystem.

You may wonder by now where the names of files (and directories) reside, since there is no file name field in the inode. Actually, the names of the files are separated from the inode and data blocks, which allows you to do groovy stuff, like giving the same file more than one name. The filenames are stored in so-called directory entries. These entries specify a filename and the inode of the file. Since directories are also represented by inodes, a directory structure can also be constructed in this manner.

We can simply show how this all works by illustrating what the kernel does when we execute the command **cat /home/daniel/note.txt**

1. The system reads the inode of the / directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the home directory.

2. The system reads the inode of the home directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the daniel directory.

3. The system reads the inode of the `daniel` directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the `note.txt` file.

4. The system reads the inode of the `note.txt` file, checks if the user is allowed to access this inode, and returns the data blocks to **cat** through the *read()* system call.

File permissions

As we have described earlier, Linux is a multi-user system. This means that each user has his/her own files (that are usually located in the home directory). Besides that users can be members of a group, which may give the user additional privileges.

As you have seen in the inode field table, every file has a owner and a group. Traditional UNIX access control gives read, write, or executable permissions to the file owner, file group, and other users. These permissions are stored in the *mode* field of the inode. The mode field represents the file permissions as a four digit octal number. The first digit represents some special options, the second digit stores the owner permissions, the third the group permissions, and the fourth the permissions for other users. The permissions are established by digit by using or adding one of the number in Table 8.2, "Meaning of numbers in the mode octet"

Table 8.2. Meaning of numbers in the mode octet

Number	Meaning
1	Execute (x)
2	Write (w)
4	Read (r)

Now, suppose that a file has mode *0644*, this means that the file is readable and writable by the owner (*6*), and readable by the file group (*4*) and others (*4*).

Most users do not want to deal with octal numbers, so that is why many utilities can also deal with an alphabetic representation of file permissions. The letters that are listed in Table 8.2, "Meaning of numbers in the mode octet" between parentheses are used in this notation. In the following example information about a file with *0644* permissions is printed. The numbers are replaced by three rwx triplets (the first character can list special mode options).

```
$ ls -l note.txt
-rw-r--r--  1 daniel daniel 5 Aug 28 19:39 note.txt
```

Over the years these traditional UNIX permissions have proven not to be sufficient in some cases. The POSIX 1003.1e specification aimed to extend the UNIX access control model with *Access Control Lists* (ACLs). Unfortunately this effort stalled, though some systems (like GNU/Linux) have implemented ACLs[1]. Access control lists follow the same semantics as normal file permissions, but give you the opportunity to add *rwx* triplets for additional users and groups.

The following example shows the access control list of a file. As you can see, the permissions look like normal UNIX permissions (the access rights for the user, group, and others are specified). But there is also an additional entry for the user *joe*.

```
user::rwx
```

---

[1]At the time of writing, ACLs were supported on ext2, ext3, and XFS filesystems

```
user:joe:r--
group::---
mask::r--
other::---
```

To make matters even more complex (and sophisticated), some GNU/Linux systems add more fine-grained access control through Mandatory Access Control Frameworks (MAC) like SELinux and AppArmor. But these access control frameworks are beyond the scope of this book.

Links

A directory entry that points to an inode is named a *hard link*. Most files are only linked once, but nothing holds you from linking a file twice. This will increase the *links_count* field of the inode. This is a nice way for the system to see which inodes and data blocks are free to use. If links_count is set to zero, the inode is not referred to anymore, and can be reclaimed.

Figure 8.1. The structure of a hard link
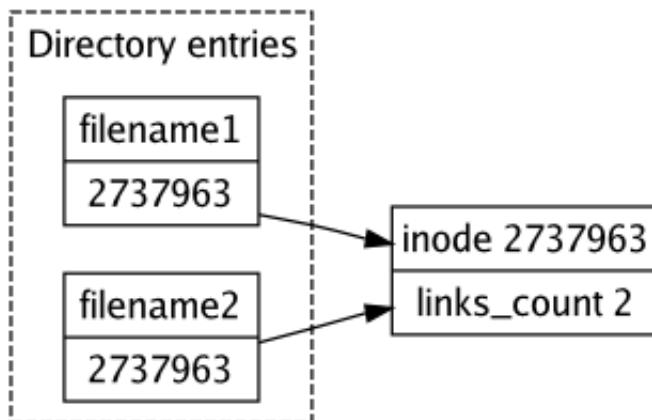
Hard links have two limitations. First of all, hard links can not interlink between filessystems, since they point to inodes. Every filesystem has its own inodes and corresponding inode numbers. Besides that, most filesystems do not allow you to create hard links to directories. Allowing creation of hard links to directories could produce directory loops, potentially leading to deadlocks and filesystem inconsistencies. In addition to that, most implementations of **rm** and **rmdir** do not know how to deal with such extra directory hard links.

Symbolic links do not have these limitations, because they point to file names, rather than inodes. When the symbolic link is used, the operating system will follow the path to that link. Symbolic links can also refer to a file that does not exist, since it just contains a name. Such links are called dangling links.

Figure 8.2. The structure of a symbolic link
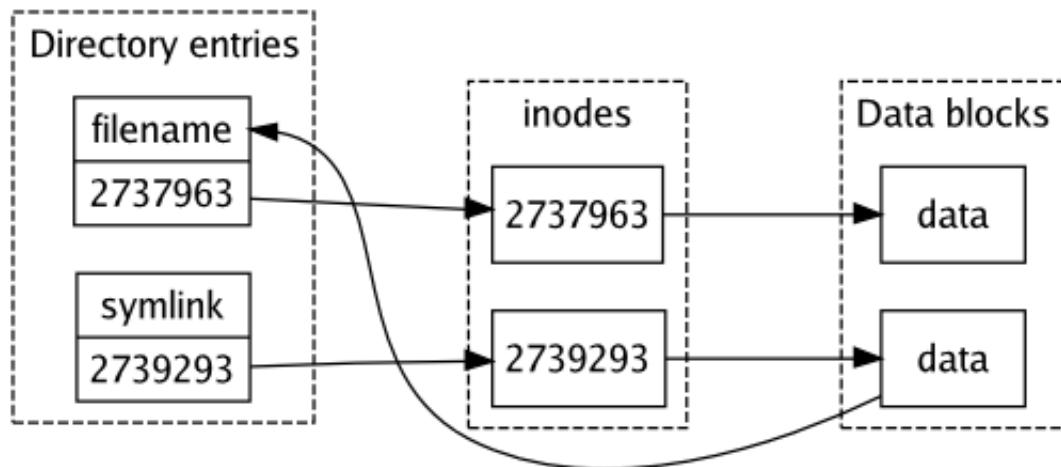
Note

If you ever get into system administration, it is good to be aware of the security implications of hard links. If the `/home` directory is on the same filesystem as any system binaries, a user will be able to create hard links to binaries. In the case that a vulnerable program is upgraded, the link in the user's home directory will keep pointing to the old program binary, effectively giving the user continuing access to a vulnerable binary.

For this reason it is a good idea to put any directories that users can write to on different filesystems. In practice, this means that it is a good idea to put at least `/home` and `/tmp` on separate filesystems.

8.2. Analyzing files

Before going to some more adventurous venues, we will start with some file and directory usage basics.

Listing files

One of the most common things that you will want to do is to list all or certain files. The **ls** command serves this purpose very well. Using **ls** without any arguments will show the contents of the actual directory:

```
$ ls
dns.txt   network-hosts.txt   papers
```

If you use a GNU/Linux distribution, you may also see some fancy coloring based on the type of file. The standard output is handy to skim through the contents of a directory, but if you want more information, you can use the `-l` parameter. This provides a so-called long listing for each file:

```
$ ls -l
total 36
-rw-rw-r--  1 daniel daniel 12235 Sep  4 15:56 dns.txt
-rw-rw-r--  1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
```

```
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

This gives a lot more information about the three directory entries that we have found with **ls**. The first column shows the file permissions. The line that shows the papers entry starts with a "d", meaning that this entry represents a directory. The second column shows the number of hard links pointing to the inode that a directory entry points to. If this is higher than 1, there is some other filename for the same file. Directory entries usually have at least two hard links, namely the link in the parent directory and the link in the directory itself (each directory has a . entry, which refers to the directory itself). The third and the fourth columns list the file owner and group respectively. The fifth column contains the file size in bytes. The sixth column the last modification time and date of the file. And finally, the last column shows the name of this entry.

Files that start with a period (.) will not be shown by most applications, including **ls**. You can list these files too, by adding the -a option to **ls**:

```
$ ls -la
total 60
drwxrwxr-x   3 daniel daniel  4096 Sep 11 10:01 .
drwx------  88 daniel daniel  4096 Sep 11 10:01 ..
-rw-rw-r--   1 daniel daniel 12235 Sep  4 15:56 dns.txt
-rw-rw-r--   1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
drwxrwxr-x   2 daniel daniel  4096 Sep  4 15:55 papers
-rw-rw-r--   1 daniel daniel     5 Sep 11 10:01 .settings
```

As you can see, three more entries have appeared. First of all, the .settings file is now shown. Besides that you can see two additional directory entries, . and .. These represent the current directory and the parent directory respectively.

Earlier in this chapter (the section called "inodes, directories and data") we talked about inodes. The inode number that a directory entry points to can be shown with the -i parameter. Suppose that I have created a hard link to the inode that points to the same inode as dns.txt, they should have the same inode number. The following **ls** output shows that this is true:

```
$ ls -i dns*
3162388 dns-newhardlink.txt
3162388 dns.txt
```

Determining the type of a file

Sometimes you will need some help to determine the type of a file. This is where the **file** utility becomes handy. Suppose that I find a file named HelloWorld.class somewhere on my disk. I suppose that this is a file that holds Java bytecode, but we can use **file** to check this:

```
$ file HelloWorld.class
HelloWorld.class: compiled Java class data, version 49.0
```

That is definitely Java bytecode. **file** is quite smart, and handles most things you throw at it. For instance, you could ask it to provide information about a device node:

```
$ file /dev/zero
/dev/zero: character special (1/5)
```

Or a symbolic link:

```
$ file /usr/X11R6/bin/X
/usr/X11R6/bin/X: symbolic link to `Xorg'
```

If you are rather interested in the file `/usr/X11R6/bin/X` links to, you can use the `-L` option of **file**:

```
$ file -L /usr/X11R6/bin/X
/usr/X11R6/bin/X: setuid writable, executable, regular file, no read permission
```

You may wonder why **file** can determine the file type relatively easy. Most files start of with a so-called *magic number*, this is a unique number that tells programs that can read the file what kind of file it is. The **file** program uses a file which describes many file types and their magic numbers. For instance, the magic file on my system contains the following lines for Java compiled class files:

```
# Java ByteCode
# From Larry Schwimmer (schwim@cs.stanford.edu)
0       belong          0xcafebabe      compiled Java class data,
>6      beshort x       version %d.
>4      beshort x       \b%d
```

This entry says that if a file starts with a long (32-bit) hexadecimal magic number *0xcafebabe*[2], it is a file that holds "compiled Java class data". The short that follows determines the class file format version.

File integrity

While we will look at more advanced file integrity checking later, we will have a short look at the **cksum** utility. **cksum** can calculate a cyclic redundancy check (CRC) for an input file. This is a mathematically sound method for calculating a unique number for a file. You can use this number to check whether a file is unchanged (for example, after downloading a file from a server). You can specify the file to calculate a CRC for as a parameter to **cksum**, and **cksum** will print the CRC, the file size in bytes, and the file name:

```
$ cksum myfile
1817811752 22638 myfile
```

Slackware Linux also provides utilities for calculating checksums based on one-way hashes (for instance MD5 or SHA-1).

Viewing files

Since most files on UNIX systems are usually text files, they are easy to view from a character-based terminal or terminal emulator. The most primitive way of looking at the contents of a file is by using **cat**. **cat** reads files that were

---

[2]Yeah, you can be creative with magic numbers too!

specified as a parameter line by line, and will write the lines to the standard output. So, you can write the contents of the file `note.txt` to the terminal with **cat note.txt**. While some systems and most terminal emulators provide support for scrolling, this is not a practical way to view large files. You can pipe the output of **cat** to the **less** paginator:

```
$ cat note.txt | less
```

or let **less** read the file directly:

```
$ less note.txt
```

The **less** paginator lets you scroll forward and backward through a file. Table 8.3, "less command keys" provides an overview of the most important keys that are used to control **less**

Table 8.3. less command keys

Key	Description
j	Scroll forward one line.
k	Scroll backwards one line.
f	Scroll forward one screen full of text.
b	Scroll backwards one screen full of text.
q	Quit less.
g	Jump to the beginning of the file.
G	Jump to the end of the file.
/*pattern*	Search for the regular expression *pattern*.
n	Search for the next match of the previously specified regular expression.
m*letter*	Mark the current position in the file with *letter*.
'*letter*	Jump to the mark *letter*

The command keys that can be quantized can be prefixed by a number. For instance *11j* scrolls forward eleven lines, and *3n* searches the third match of the previously specified regular expression.

Slackware Linux also provides an alternative to **less**, the older "more" command. We will not go into *more* here, **less** is more comfortable, and also more popular these days.

File and directory sizes

The **ls -l** output that we have seen earlier provides information about the size of a file. While this usually provides enough information about the size of files, you might want to gather information about collections of files or directories. This is where the **du** command comes in. By default, **du** prints the file size per directory. For example:

```
$ du ~/qconcord
72        /home/daniel/qconcord/src
24        /home/daniel/qconcord/ui
132       /home/daniel/qconcord
```

By default, **du** represents the size in 1024 byte units. You can explicitly specify that **du** should use 1024 byte units by adding the *-k* flag. This is useful for writing scripts, because some other systems default to using 512-byte blocks. For example:

```
$ du -k ~/qconcord
72 /home/daniel/qconcord/src
24 /home/daniel/qconcord/ui
132 /home/daniel/qconcord
```

If you would also like to see per-file disk usage, you can add the *-a* flag:

```
$ du -k -a ~/qconcord
8         /home/daniel/qconcord/ChangeLog
8         /home/daniel/qconcord/src/concordanceform.h
8         /home/daniel/qconcord/src/textfile.cpp
12        /home/daniel/qconcord/src/concordancemainwindow.cpp
12        /home/daniel/qconcord/src/concordanceform.cpp
8         /home/daniel/qconcord/src/concordancemainwindow.h
8         /home/daniel/qconcord/src/main.cpp
8         /home/daniel/qconcord/src/textfile.h
72        /home/daniel/qconcord/src
12        /home/daniel/qconcord/Makefile
16        /home/daniel/qconcord/ui/concordanceformbase.ui
24        /home/daniel/qconcord/ui
8         /home/daniel/qconcord/qconcord.pro
132       /home/daniel/qconcord
```

You can also use the name of a file or a wildcard as a parameter. But this will not print the sizes of files in subdirectories, unless *-a* is used:

```
$ du -k -a ~/qconcord/*
8         /home/daniel/qconcord/ChangeLog
12        /home/daniel/qconcord/Makefile
8         /home/daniel/qconcord/qconcord.pro
8         /home/daniel/qconcord/src/concordanceform.h
8         /home/daniel/qconcord/src/textfile.cpp
12        /home/daniel/qconcord/src/concordancemainwindow.cpp
12        /home/daniel/qconcord/src/concordanceform.cpp
8         /home/daniel/qconcord/src/concordancemainwindow.h
8         /home/daniel/qconcord/src/main.cpp
8         /home/daniel/qconcord/src/textfile.h
72        /home/daniel/qconcord/src
16        /home/daniel/qconcord/ui/concordanceformbase.ui
24        /home/daniel/qconcord/ui
```

If you want to see the total sum of the disk usage of the files and subdirectories that a directory holds, use the $-s$ flag:

```
$ du -k -s ~/qconcord
132     /home/daniel/qconcord
```

8.3. Working with directories

After having a bird's eye view of directories in the section called "inodes, directories and data", we will have a look at some directory-related commands.

Listing directories

The **ls** command that we have looked at in the section called "Listing files" can also be used to list directories in various ways. As we have seen, the default **ls** output includes directories, and directories can be identified using the first output column of a long listing:

```
$ ls -l
total 36
-rw-rw-r--  1 daniel daniel 12235 Sep  4 15:56 dns.txt
-rw-rw-r--  1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

If a directory name, or if wildcards are specified, **ls** will list the contents of the directory, or the directories that match the wildcard respectively. For example, if there is a directory `papers`, **ls paper*** will list the contents of this directory `paper`. This is often annoying if you would just like to see the matches, and not the contents of the matching directories. The $-d$ avoid that this recursion happens:

```
$ ls -ld paper*
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

You can also recursively list the contents of a directory, and its subdirectories with the $-R$ parameter:

```
$ ls -R
.:
dns.txt   network-hosts.txt   papers

./papers:
cs   phil

./papers/cs:
entr.pdf

./papers/phil:
logics.pdf
```

Creating and removing directories

UNIX provides the **mkdir** command to create directories. If a relative path is specified, the directory is created in the current active directory. The basic syntax is very simple: *mkdir <name>*, for example:

```
$ mkdir mydir
```

By default, **mkdir** only creates one directory level. So, if you use **mkdir** to create mydir/mysubdir, **mkdir** will fail if mydir does not exist already. If you would like to create both directories at once, use the *-p* parameter:

```
$ mkdir -p mydir/mysubdir
```

rmdir removes a directory. Its behavior is comparable to **mkdir**. **rmdir mydir/mysubdir** removes mydir/subdir, while **rmdir -p mydir/mysubdir** removes mydir/mysubdir and then mydir.

If a subdirectory that we want to remove contains directory entries, **rmdir** will fail. If you would like to remove a directory, including all its contents, use the **rm** command instead.

8.4. Managing files and directories

Copying

Files and directories can be copied with the **cp** command. In its most basic syntax the source and the target file are specified. The following example will make a copy of file1 named file2:

```
$ cp file1 file2
```

It is not surprising that relative and absolute paths do also work:

```
$ cp file1 somedir/file2
$ cp file1 /home/joe/design_documents/file2
```

You can also specify a directory as the second parameter. If this is the case, **cp** will make a copy of the file in that directory, giving it the same file name as the original file. If there is more than one parameter, the last parameter will be used as the target directory. For instance

```
$ cp file1 file2 somedir
```

will copy both file1 and file2 to the directory somedir. You can not copy multiple files to one file. You will have to use **cat** instead:

```
$ cat file1 file2 > combined_file
```

You can also use **cp** to copy directories, by adding the `-R`. This will recursively copy a directory and all its subdirectories. If the target directory exists, the source directory or directories will be placed under the target directory. If the target directory does not exist, it will be created if there is only one source directory.

```
$ cp -r mytree tree_copy
$ mkdir trees
$ cp -r mytree trees
```

After executing these commands, there are two copies of the directory `mytree`, `tree_copy` and `trees/mytree`. Trying to copy two directories to a nonexistent target directory will fail:

```
$ cp -R mytree mytree2 newdir
usage: cp [-R [-H | -L | -P]] [-f | -i] [-pv] src target
       cp [-R [-H | -L | -P]] [-f | -i] [-pv] src1 ... srcN directory
```

Note

Traditionally, the `-r` has been available on many UNIX systems to recursively copy directories. However, the behavior of this parameter can be implementation-dependent, and the Single UNIX Specification version 3 states that it may be removed in future versions of the standard.

When you are copying files recursively, it is a good idea to specify the behavior of what **cp** should do when a symbolic link is encountered explicitly, if you want to use **cp** in portable scripts. The Single UNIX Specification version 3 does not specify how they should be handled by default. If `-P` is used, symbolic links will not be followed, effectively copying the link itself. If `-H` is used, symbolic links specified as a parameter to **cp** may be followed, depending on the type and content of the file. If `-L` is used, symbolic links that were specified as a parameter to **cp** and symbolic links that were encountered while copying recursively may be followed, depending on the content of the file.

If you want to preserve the ownership, SGID/SUID bits, and the modification and access times of a file, you can use the `-p` flag. This will try to preserve these properties in the file or directory copy. Good implementations of **cp** provide some additional protection as well - if the target file already exists, it may not be overwritten if the relevant metadata could not be preserved.

Moving files and directories

The UNIX command for moving files, **mv**, can move or rename files or directories. What actually happens depends on the location of the files or directories. If the source and destination files or directories are on the same filesystem, **mv** usually just creates new hard links, effectively renaming the files or directories. If both are on different filesystems, the files are actually copied, and the source files or directories are unlinked.

The syntax of **mv** is comparable to **cp**. The most basic syntax renames `file1` to `file2`:

```
$ mv file1 file2
```

The same syntax can be used for two directories as well, which will rename the directory given as the first parameter to the second parameter.

When the last parameter is an existing directory, the file or directory that is specified as the first parameter, is copied to that directory. In this case you can specify multiple files or directories as well. For instance:

```
$ targetdir
$ mv file1 directory1 targetdir
```

This creates the directory `targetdir`, and moves `file1` and `directory1` to this directory.

Removing files and directories

Files and directories can be removed with the rm (1) command. This command unlinks files and directories. If there are no other links to a file, its inode and disk blocks can be reclaimed for new files. Files can be removed by providing the files that should be removed as a parameter to rm (1) . If the file is not writable, rm (1) will ask for confirmation. For instance, to remove `file1` and `file2`, you can execute:

```
$ rm file1 file2
```

If you have to remove a large number of files that require a confirmation before they can be deleted, or if you want to use rm (1) to remove files from a script that will not be run on a terminal, add the `-f` parameter to override the use of prompts. Files that are not writable, are deleted with the `-f` flag if the file ownership allows this. This parameter will also suppress printing of errors to *stderr* if a file that should be removed was not found.

Directories can be removed recursively as well with the `-r` parameter. rm (1) will traverse the directory structure, unlinking and removing directories as they are encountered. The same semantics are used as when normal files are removed, as far as the `-f` flag is concerned. To give a short example, you can recursively remove all files and directories in the `notes` directory with:

```
$ rm -r notes
```

Since rm (1) command uses the unlink (2) function, data blocks are not rewritten to an uninitialized state. The information in data blocks is only overwritten when they are reallocated and used at a later time. To remove files including their data blocks securely, some systems provide a shred (1) command that overwrites data blocks with random data. But this is not effective on many modern (journaling) filesystems, because they don't write data in place.

The unlink (1) commands provides a one on one implementation of the unlink (2) function. It is of relatively little use, because it can not remove directories.

8.5. Permissions

We touched the subject of file and directory permissions in the section called "File permissions". In this section, we will look at the chown (1) and chmod (1) commands, that are used to set the file ownership and permissions respectively. After that, we are going to look at a modern extension to permissions named Access Control Lists (ACLs).

Changing the file ownership

As we have seen earlier, every file has an owner (user) ID and a group ID stored in the inode. The chown (1) command can be used to set these fields. This can be done by the numeric IDs, or their names. For instance, to change the owner of the file `note.txt` to *john*, and its group to *staff*, the following command is used:

```
$ chown john:staff note.txt
```

You can also omit either components, to only set one of both fields. If you want to set the user name, you can also omit the colon. So, the command above can be split up in two steps:

```
$ chown john note.txt
$ chown :staff note.txt
```

If you want to change the owner of a directory, and all the files or directories it holds, you can add the -R to chown (1) :

```
$ chown -R john:staff notes
```

If user and group names were specified, rather than IDs, the names are converted by chown (1) . This conversion usually relies on the system-wide password database. If you are operating on a filesystem that uses another password database (e.g. if you mount a root filesystem from another system for recovery), it is often useful to change file ownership by the user or group ID. In this manner, you can keep the relevant user/group name to ID mappings in tact. So, changing the ownership of note to UID 1000 and GUID 1000 is done in the following (predictable) manner:

```
$ chown 1000:1000 note.txt
```

Changing file permission bits

After reading the introduction to filesystem permissions in the section called "File permissions", changing the permission bits that are stored in the inode is fairly easy with the chmod (1) command. chmod (1) accepts both numeric and symbolic representations of permissions. Representing the permissions of a file numerically is very handy, because it allows setting all relevant permissions tersely. For instance:

```
$ chmod 0644 note.txt
```

Make note.txt readable and writable for the owner of the file, and readable for the file group and others.

Symbolic permissions work with addition or subtraction of rights, and allow for relative changes of file permissions. The syntax for symbolic permissions is:

```
[ugo][-+][rwxst]
```

The first component specifies the user classes to which the permission change applies (user, group or other). Multiple characters of this component can be combined. The second component takes away rights (-), or adds rights (+). The third component is the access specifier (read, write, execute, set UID/GID on execution, sticky). Multiple components can be specified for this component too. Let's look at some examples to clear this up:

```
ug+rw        # Give read/write rights to the file user and group
chmod go-x   # Take away execute rights from the file group and others.
chmod ugo-wx # Disallow all user classes to write to the file and to
             # execute the file.
```

These commands can be used in the following manner with chmod:

```
$ chmod ug+rw note.txt
$ chmod go-x script1.sh
$ chmod ugo-x script2.sh
```

Permissions of files and directories can be changed recursively with the `-R`. The following command makes the directory `notes` world-readable, including its contents:

```
$ chmod -R ugo+r notes
```

Extra care should be taken with directories, because the *x* flag has a special meaning in a directory context. Users that have execute rights on directories can access a directory. User that don't have execute rights on directories can not. Because of this particular behavior, it is often easier to change the permissions of a directory structure and its files with help of the find (1) command .

There are a few extra permission bits that can be set that have a special meaning. The SUID and SGID are the most interesting bits of these extra bits. These bits change the active user ID or group ID to that of the owner or group of the file when the file is executed. The **su(1)** command is a good example of a file that usually has the SUID bit set:

```
$ ls -l /bin/su
-rwsr-xr-x  1 root root 60772 Aug 13 12:26 /bin/su
```

This means that the **su** command runs as the user *root* when it is executed. The SUID bit can be set with the *s* modifier. For instance, if the SUID bit was not set on /bin/su this could be done with:

```
$ chmod u+s /bin/su
```

Note

Please be aware that the SUID and SGID bits have security implications. If a program with these bits set contain a bug, it may be exploited to get privileges of the file owner or group. For this reason it is good manner to keep the number of files with the SUID and SGID bits set to an absolute minimum.

The sticky bit is also interesting when it comes to directory. It disallows users to rename of unlink files that they do not own, in directories that they do have write access to. This is usually used on world-writeable directories, like the temporary directory (/tmp) on many UNIX systems. The sticky tag can be set with the *t* modifier:

```
$ chmod g+t /tmp
```

File creation mask

The question that remains is what initial permissions are used when a file is created. This depends on two factors: the mode flag that was passed to the *open(2)* system call, that is used to create a file, and the active file creation mask. The file creation mask can be represented as an octal number. The effective permissions for creating the file are determined as *mode & ~mask*. Or, if represented in an octal fashion, you can substract the digits of the mask from the mode. For instance, if a file is created with permissions *0666* (readable and writable by the file user, file group, and others), and the effective file creation mask is *0022*, the effective file permission will be *0644*. Let's look at anothere example.

Suppose that files are still created with *0666* permissions, and you are more paranoid, and want to take away all read and write permissions for the file group and others. This means you have to set the file creation mask to *0066*, because substracting *0066* from *0666* yields *0600*

The effective file creation mask can be queried and set with the **umask** command, that is normally a built-in shell command. The effective mask can be printed by running **umask** without any parameters:

```
$ umask
0002
```

The mask can be set by giving the octal mask number as a parameter. For instance:

```
$ umask 0066
```

We can verify that this works by creating an empty file:

```
$ touch test
$ ls -l test
-rw-------  1 daniel daniel 0 Oct 24 00:10 test2
```

Access Control Lists

Access Control lists (ACLs) are an extension to traditional UNIX file permissions, that allow for more fine-grained access control. Most systems that support filesystem ACLs implement them as they were specified in the POSIX.1e and POSIX.2c draft specifications. Notable UNIX and UNIX-like systems that implement ACLs according to this draft are FreeBSD, Solaris, and Linux.

As we have seen in the section called "File permissions" access control lists allows you to use read, write and execute triplets for additional users or groups. In contrast to the traditional file permissions, additional access control lists are note stored directly in the node, but in extended attributes that are associated with files. Two thing to be aware of when you use access control lists is that not all systems support them, and not all programs support them.

Reading access control lists

On most systems that support ACLs, **ls** uses a visual indicator to show that there are ACLs associated with a file. For example:

```
$ ls -l index.html
-rw-r-----+ 1 daniel daniel 3254 2006-10-31 17:11 index.html
```

As you can see, the permissions column shows an additional plus (+) sign. The permission bits do not quite act like you expect them to be. We will get to that in a minute.

The ACLs for a file can be queried with the **getfacl** command:

```
$ getfacl index.html
# file: index.html
```

```
# owner: daniel
# group: daniel
user::rw-
group::---
group:www-data:r--
mask::r--
other::---
```

Most lines can be interpreted very easily: the file user has read/write permissions, the file group no permissions, users of the group *www-data* have read permissions, and other users have no permissions. But why does the group entry list no permissions for the file group, while **ls** does? The secret is that if there is a *mask* entry, **ls** displays the value of the mask, rather than the file group permissions.

The *mask* entry is used to restrict all list entries with the exception of that of the file user, and that for other users. It is best to memorize the following rules for interpreting ACLs:

• The *user::* entry permissions correspond with the permissions of the file owner.

• The *group::* entry permissions correspond with the permissions of the file group, unless there is a *mask::* entry. If there is a *mask::* entry, the permissions of the group correspond to the group entry with the the mask entry as the maximum of allowed permissions (meaning that the group restrictions can be more restrictive, but not more permissive).

• The permissions of other users and groups correspond to their *user:* and *group:* entries, with the value of *mask::* as their maximum permissions.

The second and third rules can clearly be observed if there us a user or group that has more rights than the mask for the file:

```
$ getfacl links.html
# file: links.html
# owner: daniel
# group: daniel
user::rw-
group::rw-                      #effective:r--
group:www-data:rw-              #effective:r--
mask::r--
other::---
```

Although read and write permissions are specified for the file and *www-data* groups, both groups will effectively only have read permission, because this is the maximal permission that the mask allows.

Another aspect to pay attention to is the handling of ACLs on directories. Access control lists can be added to directories to govern access, but directories can also have *default ACLs* which specify the initial ACLs for files and directories created under that directory.

Suppose that the directory `reports` has the following ACL:

```
$ getfacl reports
# file: reports
# owner: daniel
# group: daniel
user::rwx
```

```
group::r-x
group:www-data:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:www-data:r-x
default:mask::r-x
default:other::---
```

New files that are created in the `reports` directory get a ACL based on the entries that have *default:* as a prefix. For example:

```
$ touch reports/test
$ getfacl reports/test
# file: reports/test
# owner: daniel
# group: daniel
user::rw-
group::r-x                          #effective:r--
group:www-data:r-x                  #effective:r--
mask::r--
other::---
```

As you can see, the default ACL was copied. The execute bit is removed from the mask, because the new file was not created with execute permissions.

Creating access control lists

The ACL for a file or directory can be changed with the **setfacl** program. Unfortunately, the usage of this program highly depends on the system that is being used. To add to that confusion, at least one important flag (*-d*) has a different meanings on different systems. One can only hope that this command will get standardized.

Table 8.4. System-specific setfacl flags

Operation	Linux
Set entries, removing all old entries	*--set*
Modify entries	*-m*
Modify default ACL entries	*-d*
Delete entry	*-x*
Remove all ACL entries (except for the three required entries.	*-b*
Recalculate mask	Always recalculated, unless *-n* is used, or an mask entry expicitly specified.
Use ACL specification from a file	*-M* (modify), *-X* (delete), or *--restore*
Recursive modification of ACLs	*-R*

As we have seen in the previous section, entries can be specified for users and groups, by using the following syntax: *user/group:name:permissions*. Permissions can be specified as a triplet by using the letters *r* (read), *w* (write), or *x* (execute). A dash (-) should be used for permissions that you do not want to give to the user or group, since Solaris requires this. If you want to disallow access completely, you can use the --- triplet.

The specification for other users, and the mask follows this format: *other:r-x*. The following slightly more predictable format can also be used: *other::r-x*.

Modifying ACL entries

The simplest operation is to modify an ACL entry. This will create a new entry if the entry does not exist yet. Entries can be modified with the *-m*. For instance, suppose that we want to give the group *friend* read and write access to the file report.txt. This can be done with:

```
$ setfacl -m group:friends:rw- report.txt
```

The mask entry will be recalculated, setting it to the union of all group entries, and additional user entries:

```
$ getfacl report.txt
# file: report.txt
# owner: daniel
# group: daniel
user::rw-
group::r--
group:friends:rw-
mask::rw-
other::r--
```

You can combine multiple ACL entries by separating them with a comma character. For instance:

```
$ setfacl -m group:friends:rw-,group:foes:--- report.txt
```

Removing ACL entries

An entry can be removed with the *-x* option:

```
$ setfacl -x group:friends: report.txt
```

The trailing colon can optionally be omitted.

Making a new ACL

The *--set* option is provided create a new access control list for a file, clearing all existing entries, except for the three required entries. It is required that the file user, group and other entries are also specified. For example:

```
$ setfacl --set user::rw-,group::r--,other:---,group:friends:rwx report.txt
```

If you do not want to clean the user, group, and other permissions, but do want to clear all other ACL entries, you can use the *-b* option. The following example uses this in combination with the *-m* option to clear all ACL entries (except for user, group, and other), and to add an entry for the *friends* group:

```
$ setfacl -b -m group:friends:rw- report.txt
```

Setting a default ACL

As we have seen in the section called "Access Control Lists", directories can have default ACL entries that specify what permissions should be used for files and directories that are created below that directory. The *-d* option is used to operate on default entries:

```
$ setfacl -d -m group:friends:rwx reports
$ getfacl reports
# file: reports
# owner: daniel
# group: daniel
user::rwx
group::r-x
other::r-x
default:user::rwx
default:group::r-x
default:group:friends:rwx
default:mask::rwx
default:other::r-x
```

Using an ACL from a reference file

You can also use an ACL specification from file, rather than specifying it on the command line. An input file follows the same syntax as specifying entries as a parameter to **setfacl**, but the entries are separated by newlines, rather than by commas. This is very useful, because you can use the ACL for an existing file as a reference:

```
$ getfacl report.txt > ref
```

The *-M* option is provided to modify the ACL for a file by reading the entries from a file. So, if we have a file named report2.txt, we could modify the ACL for this file with the entries from ref with:

```
$ setfacl -M ref report2.txt
```

If you would like to start with a clean ACL, and add the entries from ref, you can add the *-b* flag that we encountered earlier:

```
$ setfacl -b -M ref report2.txt
```

Of course, it is not necessary to use this interim file. We can directly pipe the output from **getfacl** to **setfacl**, by using the symbolic name for the standard input (-), rather than the name of a file:

```
$ getfacl report.txt | setfacl -b -M - report2.txt
```

The *-X* removes the ACL entries defined in a file. This follows the same syntax as the *-x* flag, with commas replaced by newlines.

8.6. Finding files

find

The **find** command is without doubt the most comprehensive utility to find files on UNIX systems. Besides that it works in a simple and predictable way: **find** will traverse the directory tree or trees that are specified as a parameter to **find**. Besides that a user can specify an expression that will be evaluated for each file and directory. The name of a file or directory will be printed if the expression evaluates to *true*. The first argument that starts with a dash (-), exclamation mark (*!*, or an opening parenthesis ((, signifies the start of the expression. The expression can consist of various operands. To wrap it up, the syntax of **find** is: *find paths expression*.

The simplest use of **find** is to use no expression. Since this matches every directory and subdirectory entry, all files and directories will be printed. For instance:

```
$ find .
.
./economic
./economic/report.txt
./economic/report2.txt
./technical
./technical/report2.txt
./technical/report.txt
```

You can also specify multiple directories:

```
$ find economic technical
economic
economic/report.txt
economic/report2.txt
technical
technical/report2.txt
technical/report.txt
```

Operands that limit by object name or type

One common scenario for finding files or directories is to look them up by name. The *-name* operand can be used to match objects that have a certain name, or match a particular wildcard. For instance, using the operand *-name 'report.txt'* will only be true for files or directories with the name `report.txt`. For example:

```
$ find economic technical -name 'report.txt'
economic/report.txt
technical/report.txt
```

The same thing holds for wildcards:

```
$ find economic technical -name '*2.txt'
economic/report2.txt
technical/report2.txt
```

Note

When using **find** you will want to pass the wildcard to **find**, rather than letting the shell expand it. So, make sure that patterns are either quoted, or that wildcards are escaped.

It is also possible to evaluate the type of the object with the *-type c* operand, where *c* specifies the type to be matched. Table 8.5, "Parameters for the '-type' operand" lists the various object types that can be used.

Table 8.5. Parameters for the '-type' operand

Parameter	Meaning
b	Block device file
c	Character device file
d	Directory
f	Regular file
l	Symbolic link
p	FIFO
s	Socket

So, for instance, if you would like to match directories, you could use the *d* parameter to *-type* operand:

```
$ find . -type d
.
./economic
./technical
```

We will look at forming a complex expression at the end of this section about **find**, but at this moment it is handy to know that you can make a boolean 'and' expression by specifying multiple operands. For instance *operand1 operand2* is true if both *operand1* and *operand2* are true for the object that is being evaluated. So, you could combine the *-name* and *-type* operands to find all directories that start with *eco*:

```
$ find . -name 'eco*' -type d
./economic
```

Operands that limit by object ownership or permissions

Besides matching objects by their name or type, you can also match them by their active permissions or the object ownership. This is often useful to find files that have incorrect permissions or ownership.

The owner (user) or group of an object can be matched with respectively the *-user username* and *-group groupname* variants. The name of a user or group will be interpreted as a user ID or group ID of the name is decimal, and could not be found on the system with getpwnam(3) or getgrnam(3). So, if you would like to match all objects of which *joe* is the owner, you can use *-user joe* as an operand:

```
$ find . -user joe
./secret/report.txt
```

Or to find all objects with the group *friend* as the file group:

```
$ find . -group friends
./secret/report.txt
```

The operand for checking file permissions *-perm* is less trivial. Like the **chmod** command this operator can work with octal and symbolic permission notations. We will start with looking at the octal notation. If an octal number is specified as a parameter to the *-perm* operand, it will match all objects that have exactly that permissions. For instance, *-perm 0600* will match all objects that are only readable and writable by the user, and have no additional flags set:

```
$ find . -perm 0600
./secret/report.txt
```

If a dash is added as a prefix to a number, it will match every object that has at least the bits set that are specified in the octal number. A useful example is to find all files which have at least writable bits set for *other* users with *-perm -0002*. This can help you to find device nodes or other objects with insecure permissions.

```
$ find /dev -perm -0002
/dev/null
/dev/zero
/dev/ctty
/dev/random
/dev/fd/0
/dev/fd/1
/dev/fd/2
/dev/psm0
/dev/bpsm0
/dev/ptyp0
```

Note

Some device nodes have to be world-writable for a UNIX system to function correctly. For instance, the /dev/null device is always writable.

The symbolic notation of *-perm* parameters uses the same notation as the **chmod** command. Symbolic permissions are built with a file mode where all bits are cleared, so it is never necessary to use a dash to take away rights. This also prevents ambiguity that could arise with the dash prefix. Like the octal syntax, prefixing the permission with a dash will match objects that have at least the specified permission bits set. The use of symbolic names is quite predictable - the following two commands repeat the previous examples with symbolic permissions:

```
$ find . -perm u+rw
./secret/report.txt
```

```
$ find /dev -perm -o+w
/dev/null
/dev/zero
/dev/ctty
/dev/random
/dev/fd/0
/dev/fd/1
/dev/fd/2
/dev/psm0
/dev/bpsm0
/dev/ptyp0
```

Operands that limit by object creation time

There are three operands that operate on time intervals. The syntax of the operand is *operand n*, where *n* is the time in days. All three operators calculate a time delta in seconds that is divided by the the number of seconds in a day (86400), discarding the remainder. So, if the delta is one day, *operand 1* will match for the object. The three operands are:

- *-atime n* - this operand evaluates to true if the initialization time of **find** minus the last access time of the object equals to *n*.

- *-ctime n* - this operand evaluates to true if the initialization time of **find** minus the time of the latest change in the file status information equals to *n*.

- *-mtime n* - this operand evaluates to true if the initialization time of **find** minus the latest file change time equals to *n*.

So, these operands match if the latest access, change, modification respectively was *n* days ago. To give an example, the following command shows all objects in /etc that have been modified one day ago:

```
$ find /etc -mtime 1
/etc
/etc/group
/etc/master.passwd
/etc/spwd.db
/etc/passwd
/etc/pwd.db
```

The plus or minus sign can be used as modifiers for the meaning of *n*. +*n* means more than *n* days, -*n* means less than *n* days. So, to find all files in /etc that were modified less than two days ago, you could execute:

```
$ find /etc -mtime -2
/etc
/etc/network/run
/etc/network/run/ifstate
```

```
/etc/resolv.conf
/etc/default
/etc/default/locale
[...]
```

Another useful time-based operand is the *-newer reffile* operand. This matches all files that were modified later that the file with filename `reffile`. The followin example shows how you could use this to list all files that have later modification times than `economic/report2.txt`:

```
$ find . -newer economic/report2.txt
.
./technical
./technical/report2.txt
./technical/report.txt
./secret
./secret/report.txt
```

Operands that affect tree traversal

Some operands affect the manner in which the **find** command traverses the tree. The first of these operands is the *-xdev* operand. *-xdev* prevents that **find** decends into directories that have a different device ID, effectively avoiding traversal of other filesystems. The directory to which the filesystem is mounted, is printed, because this operand always returns *true*. A nice example is a system where `/usr` is mounted on a different filesystem than `/`. For instance, if we search for directories with the name *bin*, this may yield the following result:

```
$ find / -name 'bin' -type d
/usr/bin
/bin
```

But if we add *-xdev* `/usr/bin` is not found, because it is on a different filesystem (and device):

```
$ find / -name 'bin' -type d -xdev
/bin
```

The *-depth* operand modifies the order in which directories are evaluated. With *-depth* the contents of a directory are evaluated first, and then the directory itself. This can be witnessed in the following example:

```
$ find . -depth
./economic/report.txt
./economic/report2.txt
./economic
./technical/report2.txt
./technical/report.txt
./technical
.
```

As you can see in the output, files in the *./economic* directory is evaluated before `.`, and `./economic/report.txt` before `./economic`. *-depth* always evaluates to *true*.

Finally, the *-prune* operand causes find not to decend into a directory that is being evaluated. *-prune* is discarded if the *-depth* operand is also used. *-depth* always evaluates to *true*.

Operands that execute external utilities

find becomes a very powerful tool when it is combined with external utilities. This can be done with the *-exec* operand. There are two syntaxes for the *-exec* operand. The first syntax is *-exec utility arguments ;*. The command *utility* will be executed with the arguments that were specified for each object that is being evaluated. If any of the arguments is *{}*, these braces will be replaced by the file being evaluated. This is very handy, especially when we consider that, if we use no additional expression syntax, operands will be evaluated from left to right. Let's look at an example:

```
$ find . -perm 0666 -exec chmod 0644 {} \;
```

The first operand returns true for files that have their permissions set to *0666*. The second operand executes *chmod 0644 filename* for each file that is being evaluated. If you were wondering why this command is not executed for every file, that is a good question. Like many other interpreters of expressions, **find** uses "short-circuiting". Because no other operator was specified, the logical *and* operator is automatically is assumed between both operands. If the first operand evaluates to *false*, it makes no sense to evaluate any further operands, because the complete expression will always evaluate to false. So, the *-exec* operand will only be evaluated if the first operand is true. Another particularity is that the semi-colon that closes the *-exec* is escaped, to prevent that the shell parses it.

A nice thing about the *-exec* operator is that it evaluates to *true* if the command terminated sucessfully. So, you could also use the *-exec* command to add additional conditions that are not represented by **find** operands. For instance, the following command prints all objects ending with *.txt* that contain the string *gross income*:

```
$ find . -name '*.txt' -exec grep -q 'gross income' {} \; -print
./economic/report2.txt
```

The **grep** command will be covered lateron. But for the moment, it is enough to know that it can be used to match text patterns. The *-print* operand prints the current object path. It is always used implicitly, except when the *-exec* or *-ok* operands are used.

The second syntax of the *-exec* operand is *-exec utility arguments {} +*. This gathers a set of all matched object for which the expression is true, and provides this set of files as an argument to the utility that was specified. The first example of the *-exec* operand can also be written as:

```
$ find . -perm 0666 -exec chmod 0644 {} +
```

This will execute the **chmod** command only once, with all files for which the expression is true as its arguments. This operand always returns *true*.

If a command executed by find returns a non-zero value (meaning that the execution of the command was not succesful), **find** should also return a non-zero value.

Operators for building complex expressions

find provides some operators that can be combined to make more complex expressions:

Operators

(expr)   Evaluates to *true* if *expr* evaluates to *true*.

expr1 [-a] expr2   Evaluates to *true* if both *expr1* and *expr2* are true. If *-a* is omitted, this operator is implicitly
                           assumed.

                           find will use short-circuiting when this operator is evaluated: *expr2* will not be evaluated
                           when *expr1* evaluates to *false*

expr1 -o expr2   Evaluates to *true* if either or both *expr1* and *expr2* are true.

                           find will use short-circuiting when this operator is evaluated: *expr2* will not be evaluated
                           when *expr1* evaluates to *true*

! expr   Negates *expr*. So, if *expr* evaluates to true, this expression will evaluate to *false* and vise
                           versa.

Since both the parentheses and exclamation mark characters are interpreted by most shells, they should usually be
escaped.

The following example shows some operators in action. This command executes **chmod** for all files that either have
their permissions set to *0666* or *0664*.

```
$ find . \( -perm 0666 -o -perm 0664 \) -exec chmod 0644 {} \;
```

which

The **which** command is not part of the Single UNIX Specification version 3, but it is provided by most sysmtems.
which locates a command that is in the user's path (as set by the PATH environment variable), printing its full path.
Providing the name of a command as its parameter will show the full path:

```
$ which ls
/bin/ls
```

You can also query the paths of multiple commands:

```
$ which ls cat
/bin/ls
/bin/cat
```

which returns a non-zero return value if the command could not be found.

whereis

This **whereis** command searches binaries, manual pages and sources of a command in some predefined places. For
instance, the following command shows the path of the **ls** and the ls(1) manual page:

```
$ whereis ls
```

```
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

locate

Slackware Linux also provides the **locate** command that searches through a file database that can be generated periodically with the **updatedb** command. Since it uses a prebuilt database of the filesystem, it is a lot faster than **command**, especially when directory entry information has not been cached yet. Though, the **locate**/**updatedb** combo has some downsides:

• New files are not part of the database until the next **updatedb** invocation.

• **locate** has no conception of permissions, so users may locate files that are normally hidden to them.

• A newer implementation, named *slocate* deals with permissions, but requires elevated privileges. This is the **locate** variation that is included with Slackware Linux.

With filesystems becoming faster, and by applying common sense when formulating **find** queries, **locate** does not really seem worth the hassle. Of course, your mileage may vary. That said, the basic usage of **locate** is *locate filename*. For example:

```
$ locate locate
/usr/bin/locate
/usr/lib/locate
/usr/lib/locate/bigram
/usr/lib/locate/code
/usr/lib/locate/frcode
[...]
```

8.7. Archives

Introduction

Sooner or later a GNU/Linux user will encounter tar archives, tar is the standard format for archiving files on GNU/Linux. It is often used in conjunction with **gzip** or **bzip2**. Both commands can compress files and archives. Table 8.6, "Archive file extensions" lists frequently used archive extensions, and what they mean.

Table 8.6. Archive file extensions

Extension	Meaning
.tar	An uncompressed tar archive
.tar.gz	A tar archive compressed with gzip
.tgz	A tar archive compressed with gzip
.tar.bz2	A tar archive compressed with bzip2
.tbz	A tar archive compressed with bzip2

The difference between **bzip2** and **gzip** is that **bzip2** can find repeating information in larger blocks, resulting in better compression. But **bzip2** is also a lot slower, because it does more data analysis.

Extracting archives

Since many software and data in the GNU/Linux world is archived with **tar** it is important to get used to extracting tar archives. The first thing you will often want to do when you receive a tar archive is to list its contents. This can be achieved by using the *t* parameter. However, if we just execute **tar** with this parameter and the name of the archive it will just sit and wait until you enter something to the standard input:

```
$ tar t test.tar
```

This happens because **tar** reads data from its standard input. If you forgot how redirection works, it is a good idea to reread Section 7.7, "Redirections and pipes". Let's see what happens if we redirect our tar archive to tar:

```
$ tar t < test.tar
test/
test/test2
test/test1
```

That looks more like the output you probably expected. This archive seems to contain a directory test, which contains the files test2 and test2. It is also possible to specify the archive file name as an parameter to **tar**, by using the *f* parameter:

```
$ tar tf test.tar
test/
test/test2
test/test1
```

This looks like an archive that contains useful files ;). We can now go ahead, and extract this archive by using the *x* parameter:

```
$ tar xf test.tar
```

We can now verify that tar really extracted the archive by listing the contents of the directory with **ls**:

```
$ ls test/
test1   test2
```

Extracting or listing files from a gzipped or bzipped archive is not much more difficult. This can be done by adding a *z* or *b* for respectively archives compressed with **gzip** or **bzip2**. For example, we can list the contents of a gzipped archive with:

```
$ tar ztf archive2.tar.gz
```

And a bzipped archive can be extracted with:

```
$ tar bxf archive3.tar.bz2
```

Creating archives

You can create archives with the *c* parameter. Suppose that we have the directory `test` shown in the previous example. We can make an archive with the `test` directory and the files in this directory with:

```
$ tar cf important-files.tar test
```

This will create the `important-files.tar` archive (which is specified with the *f* parameter). We can now verify the archive:

```
$ tar tf important-files.tar
test/
test/test2
test/test1
```

Creating a gzipped or bzipped archive goes along the same lines as extracting compressed archives: add a *z* for gzipping an archive, or *b* for bzipping an archive. Suppose that we wanted to create a **gzip** compressed version of the archive created above. We can do this with:

```
tar zcf important-files.tar.gz test
```

8.8. Mounting filesystems

Introduction

Like most Unices Linux uses a technique named "mounting" to access filesystems. Mounting means that a filesystem is connected to a directory in the root filesystem. One could for example mount a CD-ROM drive to the `/mnt/cdrom` directory. Linux supports many kinds of filesystems, like Ext2, Ext3, ReiserFS, JFS, XFS, ISO9660 (used for CD-ROMs), UDF (used on some DVDs) and DOS/Windows filesystems, like FAT, FAT32 and NTFS. These filesystems can reside on many kinds of media, for example hard drives, CD-ROMs and Flash drives. This section explains how filesystems can be mounted and unmounted.

mount

The **mount** is used to mount filesystems. The basic syntax is: "mount /dev/devname /mountpoint". The device name can be any block device, like hard disks or CD-ROM drives. The mount point can be an arbitrary point in the root filesystem. Let's look at an example:

```
# mount /dev/cdrom /mnt/cdrom
```

This mounts the /dev/cdrom on the /mnt/cdrom mountpoint. The /dev/cdrom device name is normally a link to the real CD-ROM device name (for example, /dev/hdc). As you can see, the concept is actually very simple, it just takes some time to learn the device names ;). Sometimes it is necessary to specify which kind of filesystem you are trying to mount. The filesystem type can be specified by adding the -t parameter:

```
# mount -t vfat /dev/sda1 /mnt/flash
```

This mounts the vfat filesystem on /dev/sda1 to /mnt/flash.

umount

The **umount** command is used to unmount filesystems. **umount** accepts two kinds of parameters, mount points or devices. For example:

```
# umount /mnt/cdrom
# umount /dev/sda1
```

The first command unmounts the filesystem that was mounted on /mnt/cdrom, the second commands unmounts the filesystem on /dev/sda1.

The fstab file

The GNU/Linux system has a special file, /etc/fstab, that specifies which filesystems should be mounted during the system boot. Let's look at an example:

```
/dev/hda10        swap              swap          defaults        0   0
/dev/hda5         /                 xfs           defaults        1   1
/dev/hda6         /var              xfs           defaults        1   2
/dev/hda7         /tmp              xfs           defaults        1   2
/dev/hda8         /home             xfs           defaults        1   2
/dev/hda9         /usr              xfs           defaults        1   2
/dev/cdrom        /mnt/cdrom        iso9660       noauto,owner,ro 0   0
/dev/fd0          /mnt/floppy       auto          noauto,owner    0   0
devpts            /dev/pts          devpts        gid=5,mode=620  0   0
proc              /proc             proc          defaults        0   0
```

As you can see each entry in the fstab file has five entries: fs_spec, fs_file, fs_vfstype, fs_mntops, fs_freq, and fs_passno. We are now going to look at each entry.

fs_spec

The fs_spec option specifies the block device, or remote filesystem that should be mounted. As you can see in the example several /dev/hda partitions are specified, as well as the CD-ROM drive and floppy drive. When NFS volumes are mounted an IP address and directory can be specified, for example: 192.168.1.10:/exports/data.

fs_file

fs_file specifies the mount point. This can be an arbitrary directory in the filesystem.

fs_vfstype

This option specifies what kind of filesystem the entry represents. For example this can be: ext2, ext3, reiserfs, xfs, nfs, vfat, or ntfs.

fs_mntops

The fs_mntops option specifies which parameters should be used for mounting the filesystem. The **mount** manual page has an extensive description of the available options. These are the most interesting options:

- *noauto*: filesystems that are listed in /etc/fstab are normally mounted automatically. When the "noauto" option is specified, the filesystem will not be mounted during the system boot, but only after issuing a **mount** command. When mounting such filesystem, only the mount point or device name has to be specified, for example: **mount /mnt/cdrom**

- *user*: adding the "user" option will allow normal users to mount the filesystem (normally only the superuser is allowed to mount filesystems).

- *owner*: the "owner" option will allow the owner of the specified device to mount the specified device. You can see the owner of a device using **ls**, e.g. **ls -l /dev/cdrom**.

- *noexec*: with this option enabled users can not run files from the mounted filesystem. This can be used to provide more security.

- *nosuid*: this option is comparable to the "noexec" option. With "nosuid" enabled SUID bits on files on the filesystem will not be allowed. SUID is used for certain binaries to provide a normal user to do something privileged. This is certainly a security threat, so this option should really be used for removable media, etc. A normal user mount will force the nosuid option, but a mount by the superuser will not!

- *unhide*: this option is only relevant for normal CD-ROMs with the ISO9660 filesystem. If "unhide" is specified hidden files will also be visible.

fs_freq

If the "fs_freq" is set to 1 or higher, it specifies after how many days a filesystem dump (backup) has to be made. This option is only used when dump [http://dump.sourceforge.net/] is installed, and set up correctly to handle this.

fs_passno

This field is used by **fsck** to determine the order in which filesystems are checked during the system boot.

8.9. Encrypting and signing files

Introduction

There are two security mechanisms for securing files: signing files and encrypting files. Signing a file means that a special digital signature is generated for a file. You, or other persons can use the signature to verify the integrity of the file. File encryption encodes a file in a way that only a person for which the file was intended to read can read the file.

This system relies on two keys: the private and the public key. Public keys are used to encrypt files, and files can only be decrypted with the private key. This means that one can sent his public key out to other persons. Others can use this key to send encrypted files, that only the person with the private key can decode. Of course, this means that the security of this system depends on how well the private is kept secret.

Slackware Linux provides an excellent tool for signing and encrypting files, named GnuPG. GnuPG can be installed from the "n" disk set.

Generating your private and public keys

Generating public and private keys is a bit complicated, because GnuPG uses DSA keys by default. DSA is an encryption algorithm, the problem is that the maximum key length of DSA is 1024 bits, this is considered too short for the longer term. That is why it is a good idea to use 2048 bit RSA keys. This section describers how this can be done.

Note

1024-bit keys were believed to be secure for a long time. But Bernstein's paper *Circuits for Integer Factorization: a Proposal* contests this, the bottom line is that it is quite feasible for national security agencies to produce hardware that can break keys in a relatively short amount of time. Besides that it has be shown that 512-bit RSA keys can be broken in a relatively short time using common hardware. More information about these issues can by found in this e-mail to the cypherpunks list: http://lists.saigon.com/vault/security/encryption/rsa1024.html

We can generate a key by executing:

```
$ gpg --gen-key
```

The first question is what kind of key you would like to make. We will choose *(4) RSA (sign only)*:

```
Please select what kind of key you want:
   (1) DSA and ElGamal (default)
   (2) DSA (sign only)
   (4) RSA (sign only)
Your selection? 4
```

You will then be asked what the size of the key you want to generate has to be. Type in *2048* to generate a 2048 bit key, and press enter to continue.

```
What keysize do you want? (1024) 2048
```

The next question is simple to answer, just choose what you like. Generally speaking it is not a bad idea to let the key be valid infinitely. You can always deactivate the key with a special revocation certificate.

```
Please specify how long the key should be valid.
        0 = key does not expire
     <n>  = key expires in n days
     <n>w = key expires in n weeks
     <n>m = key expires in n months
```

```
    <n>y = key expires in n years
Key is valid for? (0) 0
```

GnuPG will then ask for confirmation. After confirming your name and e-mail address will be requested. GnuPG will also ask for a comment, you can leave this blank, or you could fill in something like "Work" or "Private", to indicate what the key is used for. For example:

```
Real name: John Doe
Email address: john@doe.com
Comment: Work
You selected this USER-ID:
    "John Doe (Work) <john@doe.com>"
```

GnuPG will the ask you to confirm your user ID. After confirming it GnuPG will ask you to enter a password. Be sure to use a good password:

```
You need a Passphrase to protect your secret key.

Enter passphrase:
```

After entering the password twice GnuPG will generate the keys. But we are not done yet. GnuPG has only generated a key for signing information, not for encryption of information. To continue, have a look at the output, and look for the key ID. In the information about the key you will see *pub 2048R/*. The key ID is printed after this fragment. In this example:

```
public and secret key created and signed.
key marked as ultimately trusted.

pub  2048R/8D080768 2004-07-16 John Doe (Work) <john@doe.com>
     Key fingerprint = 625A 269A 16B9 C652 B953  8B64 389A E0C9 8D08 0768
```

the key ID is *8D080768*. If you lost the output of the key generation you can still find the key ID in the output of the **gpg --list-keys** command. Use the key ID to tell GnuPG that you want to edit your key:

```
$ gpg --edit-key <Key ID>
```

With the example key above the command would be:

```
$ gpg --edit-key 8D080768
```

GnuPG will now display a command prompt. Execute the **addkey** command on this command prompt:

```
Command> addkey
```

GnuPG will now ask the password you used for your key:

```
Key is protected.

You need a passphrase to unlock the secret key for
user: "John Doe (Work) <john@doe.com>"
2048-bit RSA key, ID 8D080768, created 2004-07-16

Enter passphrase:
```

After entering the password GnuPG will ask you what kind of key you would like to create. Choose *RSA (encrypt only)*, and fill in the information like you did earlier (be sure to use a 2048 bit key). For example:

```
Please select what kind of key you want:
   (2) DSA (sign only)
   (3) ElGamal (encrypt only)
   (4) RSA (sign only)
   (5) RSA (encrypt only)
Your selection? 5
What keysize do you want? (1024) 2048
Requested keysize is 2048 bits
Please specify how long the key should be valid.
        0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
```

And confirm that the information is correct. After the key is generated you can leave the GnuPG command prompt, and save the new key with the **save** command:

```
Command> save
```

Congratulations! You have now generated the necessary keys to encrypt and decrypt e-mails and files. You can now configure your e-mail client to use GnuPG. It is a good idea to store the contents of the .gnupg directory on some reliable medium, and store that in a safe place! If your private key is lost you can't decrypt files and messages that were encrypted with your public key. If the private key, and your password are stolen, the security of this system is completely compromised.

Exporting your public key

To make GnuPG useful, you have to give your public key to people who send you files or e-mails. They can use your public key to encrypt files, or use it to verify whether a file has a correct signature or not. The key can be exported using the `--export` parameter. It is also a good idea to specify the `--output` parameter, this will save the key in a file. The following command would save the public key of *John Doe*, used in earlier examples, to the file key.gpg:

```
$ gpg --output key.gpg --export john@doe.com
```

This saves the key in binary format. Often it is more convenient to use the so-called "ASCII armored output", which fits better for adding the key to e-mails, or websites. You export an ASCII armored version of the key by adding the `--armor` parameter:

```
$ gpg --armor --output key.gpg --export john@doe.com
```

If you look at the `key.gpg` file you will notice that the ASCII armored key is a much more comfortable format.

Signatures

With GPG you can make a signature for a file. This signature is unique, because your signature can only be made with your private key. This means that other people can check whether the file was really sent by you, and whether it was in any way altered or not. Files can be signed with the `--detach-sign` parameter. Let us look at an example. This command will make a signature for the `memo.txt` file. The signature will be stored in `memo.txt.sig`.

```
$ gpg --output memo.txt.sig --detach-sign memo.txt

You need a passphrase to unlock the secret key for
user: "John Doe (Work) <john@doe.com>"
2048-bit RSA key, ID 8D080768, created 2004-07-16

Enter passphrase:
```

As you can see, GnuPG will ask you to enter the password for your private key. After you have entered the right key the signature file (`memo.txt.sig`) will be created.

You can verify a file with its signature using the `--verify` parameter. Specify the signature file as a parameter to the `--verify` parameter. The file that needs to be verified can be specified as the final parameter:

```
$ gpg --verify memo.txt.sig memo.txt
gpg: Signature made Tue Jul 20 23:47:45 2004 CEST using RSA key ID 8D080768
gpg: Good signature from "John Doe (Work) <john@doe.com>"
```

This will confirm that the file was indeed signed by *John Doe (Work) <john@doe.com>*, with the key *8D080768*, and that the file is unchanged. Suppose the file was changed, GnuPG would have complained about it loudly:

```
$ gpg --verify memo.txt.sig memo.txt
gpg: Signature made Tue Jul 20 23:47:45 2004 CEST using RSA key ID 8D080768
gpg: BAD signature from "John Doe (Work) <john@doe.com>"
```

Encryption

One of the main features of GnuPG is encryption. Due to its use of asymmetric cryptography, the person who encrypts a file and the person who decrypts a file do not need to share a key. You can encrypt a file with the public key of another person, and that other person can decrypt it with his or her private key. You can encrypt files with the `--`

encrypt. If you do not specify a user ID for which the file should be encrypted, GnuPG will prompt for the user ID. You can specify the user ID with the *-r* parameter. In the following example, the file secret.txt will be encrypted for another person named *John Doe*:

```
$ gpg --encrypt -r "John Doe" secret.txt
```

The user ID is quoted with double quotes for making sure that the ID is interpreted as a single program argument. After the encryption is completed, the encrypted version of the file will be available as secret.txt.gpg.

The user who receives the file can decrypt it with the *--decrypt* parameter of the **gpg** command:

```
$ gpg --output secret.txt --decrypt secret.txt.gpg

You need a passphrase to unlock the secret key for
user: "John Doe (Work) <john@doe.com>"
2048-bit RSA key, ID 8D080768, created 2004-07-16 (main key ID EC3ED1AB)

Enter passphrase:

gpg: encrypted with 2048-bit RSA key, ID 8D080768, created 2004-07-16
      "John Doe (Work) <john@doe.com>"
```

In this example the *--output* parameter is used store the decrypted content in secret.txt.

Chapter 9. Text processing

Text manipulation is one of the things that UNIX excels at, because it forms the heart of the UNIX philosophy, as described in Section 2.4, "The UNIX philosophy". Most UNIX commands are simple programs that read data from the standard input, performs some operation on the data, and sends the result to the program's standard output. These programs basically act as an filters, that can be connected as a pipeline. This allows the user to put the UNIX tools to uses that the writers never envisioned. In later chapters we will see how you can build simple filters yourself.

This chapter describes some simple, but important, UNIX commands that can be used to manipulate text. After that, we will dive into regular expressions, a sublanguage that can be used to match text patterns.

9.1. Simple text manipulation

Repeating what is said

The most simple text filter is the **cat**, it does nothing else than sending the data from stdin to stdout:

```
$ echo "hello world" | cat
hello world
```

Another useful feature is that you can let it send the contents of a file to the standard output:

```
$ cat file.txt
Hello, this is the content of file.txt
```

cat really lives up to its name when multiple files are added as arguments. This will concatenate the files, in the sense that it will send the contents of all files to the standard output, in the same order as they were specified as an argument. The following screen snippet demonstrates this:

```
$ cat file.txt file1.txt file2.txt
Hello, this is the content of file.txt
Hello, this is the content of file1.txt
Hello, this is the content of file2.txt
```

Text statistics

The **wc** command provides statistics about a text file or text stream. Without any parameters, it will print the number of lines, the number of words, and the number of bytes respectively. A word is delimited by one white space character, or a sequence of whitespace characters.

The following example shows the number of lines, words, and bytes in the canonical "Hello world!" example:

```
$ echo "Hello world!" | wc
```

```
          1        2        13
```

If you would like to print just one of these components, you can use one of the *-l* (lines), *-w* (words), or *-c* (bytes) parameters. For instance, adding just the *-l* parameter will show the number of lines in a file:

```
$ wc -l /usr/share/dict/words
  235882 /usr/share/dict/words
```

Or, you can print additional fields by adding a parameter:

```
$ wc -lc /usr/share/dict/words
 235882 2493082 /usr/share/dict/words
```

Please note that, no matter the order in which the options were specified, the output order will always be the same (lines, words, bytes).

Since *-c* prints the number bytes, this parameter may not represent the number of characters that a text holds, because the character set in use maybe be wider than one byte. To this end, the *-m* parameter has been added which prints the number of characters in a text, independent of the character set. *-c* and *-m* are substitutes, and can never be used at the same time.

The statistics that **wc** provides are more useful than they may seem on the surface. For example, the *-l* parameter is often used as a counter for the output of a command. This is convenient, because many commands seperate logical units by a newline. Suppose that you would like to count the number of files in your home directory having a filename ending with .txt. You could do this by combining **find** to find the relevant files and **wc** to count the number of occurences:

```
$ find ~ -name '*.txt' -type f | wc -l
```

Manipulating characters

The **tr** command can be used to do common character operations, like swapping characters, deleting characters, and squeezing character sequences. Depending on the operation, one or two sets of characters should be specified. Besides normal characters, there are some special character sequences that can be used:

\character	This notation is used to specify characters that need escaping, most notably \n (newline), \t (horizontal tab), and \\ (backslash).
character1-character2	Implicitly insert all characters from *character1* to *character2*. This notation should be used with care, because it does not always give the expected result. For instance, the sequence *a-d* may yield *abcd* for the POSIX locale (language setting), but this may not be true for other locales.
[:class:]	Match a predefined class of characters. All possible classes are shown in Table 9.1, "tr character classes".
[character*]	Repeat *character* until the second set is as long as the first set of characters. This notation can only be used in the second set.

[character*n]   Repeat *character n* times.

Table 9.1. tr character classes

Class	Meaning
[:alnum:]	All letters and numbers.
[:alpha:]	Letters.
[:blank:]	Horizontal whitespace (e.g. spaces and tabs).
[:cntrl:]	Control characters.
[:digit:]	All digits (0-9).
[:graph:]	All printable characters, except whitespace.
[:lower:]	Lowercase letters.
[:print:]	All printable characters, including horizontal whitespace, but excluding vertical whitespace.
[:punct:]	Punctuation characters.
[:space:]	All whitespace.
[:upper:]	Uppercase letters.
[:xdigit:]	Hexadecimal digits (0-9, a-f).

Swapping characters

The default operation of **tr** is to swap (translate) characters. This means that the *n*-th character in the first set is replaced with the *n*-th character in the second set. For example, you can replace all *e*'s with *i*'s and *o*'s with *a*'s with one **tr** operation:

```
$ echo 'Hello world!' | tr 'eo' 'ia'
Hilla warld!
```

When the second set is not as large as the first set, the last character in the second set will be repeated. Though, this does not necessarily apply to other UNIX systems. So, if you want to use **tr** in a system-independent manner, explicitly define what character should be repeated. For instance

```
$ echo 'Hello world!' | tr 'eaiou' '[@*]'
H@ll@ w@rld!
```

Another particularity is the use of the repetition syntax in the middle of the set. Suppose that set 1 is *abcdef*, and set 2 *@[-*]!*. **tr** will replace *a* with *@*, *b*, *c*, *d*, and *e* with -, and *f* with *!*. Though some other UNIX systems follow replace *a* with *@*, and the rest of the set characters with -. So, a more correct notation would be the more explicit *@[-*4]!*, which gives the same results on virtually all UNIX systems:

```
$ echo 'abcdef' | tr 'abcdef' '@[-*4]!'
@----!
```

Squeezing character sequences

When the `-s` parameter is used, **tr** will squeeze all characters that are in the second set. This means that a sequence of the same characters will be reduced to one character. Let's squeeze the character "*e*":

```
$ echo "Let's squeeze this." | tr -s 'e'
Let's squeze this.
```

We can combine this with translation to show a useful example of **tr** in action. Suppose that we would like to mark al vowels with the *at* sign (@), with consecutive vowels represented by one *at* sign. This can easily be done by piping two **tr** commands:

```
$ echo "eenie meenie minie moe" | tr 'aeiou' '[@*]' | tr -s '@'
@n@ m@n@ m@n@ m@
```

Deleting characters

Finally, **tr** can be used to delete characters. If the `-d` parameter is used, all characters from the first set are removed:

```
$ echo 'Hello world!' | tr -d 'lr'
Heo wod!
```

Cutting and pasting text columns

The **cut** command is provided by UNIX systems to "cut" one or more columns from a file or stream, printing it to the standard output. It is often useful to selectively pick some information from a text. **cut** provides three approaches to cutting information from files:

1. By byte.

2. By character, which is not the same as cutting by byte on systems that use a character set that is wider than eight bits.

3. By field, that is delimited by a character.

In all three approaches, you can specify the element to choose by its number starting at *1*. You can specify a range by using a dash (-). So, *M-N* means the Mth to the Nth element. Leaving M out (*-N*) selects all elements from the first element to the Nth element. Leaving N out (*M-*) selects the Mth element to the last element. Multiple elements or ranges can be combined by separating them by commas (*,*). So, for instance, *1,3-* selects the first element and the third to the last element.

Data can be cut by field with the `-f fields` parameter. By default, the horizontal tab is used a separator. Let's have a look at **cut** in action with a tiny Dutch to English dictionary:

```
$ cat dictionary
appel     apple
```

```
banaan    banana
peer      pear
```

We can get all English words by selecting the first field:

```
$ cut -f 2 dictionary
apple
banana
pear
```

That was quite easy. Now let's do the same thing with a file that has a colon as the field separator. We can easily try this by converting the dictionary with the **tr** command that we have seen earlier, replacing all tabs with colons:

```
$ tr '\t' ':' < dictionary > dictionary-new
$ cat dictionary-new
appel:apple
banaan:banana
peer:pear
```

If we use the same command as in the previous example, we do not get the correct output:

```
$ cut -f 2 dictionary-new
appel:apple
banaan:banana
peer:pear
```

What happens here is that the delimiter could not be found. If a line does not contain the delimiter that is being used, the default behavior of **cut** is to print the complete line. You can prevent this with the `-s` parameter.

To use a different delimiter than the horizontal tab, add the `-d delimter_char` parameter to set the delimiting character. So, in this case of our `dictionary-new` file, we will ask **cut** to use the colon as a delimiter:

```
$ cut -d ':' -f 2 dictionary-new
apple
banana
pear
```

If a field that was specified does not exist in a line, that particular field is not printed.

The `-b bytes` and `-c characters` respectively select bytes and characters from the text. On older systems a character used to be a byte wide. But newer systems can provide character sets that are wider than one byte. So, if you want to be sure to grab complete characters, use the `-c` parameter. An entertaining example of seeing the `-c` parameter in action is to find the ten most common sets of the first three characters of a word. Most UNIX systems provide a list of words that are separated by a new line. We can use **cut** to get the first three characters of the words in the word list, add `uniq` to count identical three character sequences, and use **sort** to sort them reverse-numerically (**sort** is described in the section called "Sorting text"). Finally, we will use **head** to get the ten most frequent sequences:

```
$ cut -c 1-4 /usr/share/dict/words | uniq -c | sort -nr | head
    254 inte
    206 comp
    169 cons
    161 cont
    150 over
    125 tran
    111 comm
    100 disc
     99 conf
     96 reco
```

Having concluded with that nice piece of UNIX commands in action, we will move on to the **paste** command, which combines files in columns in a single text stream.

Usage of **paste** is very simple, it will combine all files given as an argument, separated by a tab. With the list of English and Dutch words, we can generate a tiny dictionary:

```
$ paste dictionary-en dictionary-nl
apple   appel
banana  banaan
pear    peer
```

You can also combine more than two files:

```
$ paste dictionary-en dictionary-nl dictionary-de
apple   appel   Apfel
banana  banaan  Banane
pear    peer    Birne
```

If one of the files is longer, the column order is maintained, and empty entries are used to fill up the entries of the shorter files.

You can use another delimiter by adding the *-d delimiter* parameter. For example, we can make a colon-separated dictionary:

```
$ paste -d ':' dictionary-en dictionary-nl
apple:appel
banana:banaan
pear:peer
```

Normally, **paste** combines files as different columns. You can also let **paste** use the lines of each file as columns, and put the columns of each file on a separate line. This is done with the *-s* parameter:

```
$ paste -s dictionary-en dictionary-nl dictionary-de
apple   banana  pear
```

```
appel    banaan   peer
Apfel    Banane   Birne
```

Sorting text

UNIX offers the **sort** command to sort text. **sort** can also check whether a file is in sorted order, and merge two sorted files. **sort** can sort in dictionary and numerical orders. The default sort order is the dictionary order. This means that text lines are compared character by character, sorted as specified in the current collating sequence (which is specified through the LC_COLLATE environment variable). This has a catch when you are sorting numbers, for instance, if you have the numbers 1 to 10 on different lines, the sequence will be 1, 10, 2, 3, etc. This is caused by the per-character interpretation of the dictionary sort. If you want to sort lines by number, use the numerical sort.

If no additional parameters are specified, **sort** sorts the input lines in dictionary order. For instance:

```
$ cat << EOF | sort
orange
apple
banana
EOF
apple
banana
orange
```

As you can see, the input is correctly ordered. Sometimes there are two identical lines. You can merge identical lines by adding the -u parameter. The two samples listed below illustrate this.

```
$ cat << EOF | sort
orange
apple
banana
banana
EOF
apple
banana
banana
orange
$ cat << EOF | sort -u
orange
apple
banana
banana
EOF
apple
banana
orange
```

There are some additional parameters that can be helpful to modify the results a bit:

• The -f parameter makes the sort case-insensitive.

- If *-d* is added, only blanks and alphanumeric characters are used to determine the order.

- The *-i* parameter makes **sort** ignore non-printable characters.

You can sort files numerically by adding the *-n* parameter. This parameter stops reading the input line when a non-numeric character was found. The leading minus sign, decimal point, thousands separator, radix character (that separates an exponential from a normal number), and blanks can be used as a part of a number. These characters are interpreted where applicable.

The following example shows numerical sort in action, by piping the output of **du** to **sort**. This works because **du** specifies the size of each file as the first field.

```
$ du -a /bin | sort -n
0        /bin/kernelversion
0        /bin/ksh
0        /bin/lsmod.modutils
0        /bin/lspci
0        /bin/mt
0        /bin/netcat
[...]
```

In this case, the output is probably not useful if you want to read the output in a paginator, because the smallest files are listed first. This is where the *-r* parameter becomes handy. This reverses the sort order.

```
$ du -a /bin | sort -nr
4692     /bin
1036     /bin/ksh93
668      /bin/bash
416      /bin/busybox
236      /bin/tar
156      /bin/ip
[...]
```

The *-r* parameter also works with dictionary sorts.

Quite often, files use a layout with multiple columns, and you may want to sort a file by a different column than the first column. For instance, consider the following score file named `score.txt`:

```
John:US:4
Herman:NL:3
Klaus:DE:5
Heinz:DE:3
```

Suppose that we would like to sort the entries in this file by the two-letter country name. **sort** allows us to sort a file by a column with the *-k col1[,col2]* parameter. Where *col1* up to *col2* are used as fields for sorting the input. If *col2* is not specified, all fields up till the end of the line are used. So, if you want to use just one column, use *-k col1,col1*. You can also specify the the starting character within a column by adding a period (.) and a character index. For instance, *-k 2.3,4.2* means that the second column starting from the third character, the third column, and the fourth column up to (and including) the second character.

There is yet another particularity when it comes to sorting by columns: by default, **sort** uses a blank as the column separator. If you use a different separator character, you will have to use the `-t char` parameter, that is used to specify the field separator.

With the `-t` and `-k` parameters combined, we can sort the scores file by country code:

```
$ sort -t ':' -k 2,2 scores.txt
Heinz:DE:3
Klaus:DE:5
Herman:NL:3
John:US:4
```

So, how can we sort the file by the score? Obviously, we have to ask sort to use the third column. But sort uses a dictionary sort by default[1]. You could use the `-n`, but **sort** also allows a more sophisticated approach. You can append the one or more of the *n*, *r>*, *f*, *d*, *i*, or *b* to the column specifier. These letters represent the **sort** parameters with the same name. If you add just the starting column, append it to that column, otherwise, add it to the ending column.

The following command sorts the file by score:

```
$ sort -t ':' -k 3n /home/daniel/scores.txt
Heinz:DE:3
Herman:NL:3
John:US:4
Klaus:DE:5
```

It is good to follow this approach, rather than using the parameter variants, because **sort** allows you to use more than one `-k` parameter. And, adding these flags to the column specification, will allow you to sort by different columns in different ways. For example using **sort** with the `-k 3,3n -k 2,2` parameters will sort all lines numerically by the third column. If some lines have identical numbers in the third column, these lines can be sorted further with a dictionary sort of the second column.

If you want to check whether a file is already sorted, you can use the `-c` parameter. If the file was in a sorted order, sort will return the value *0*, otherwise *1*. We can check this by echoing the value of the *?* variable, which holds the return value of the last executed command.

```
$ sort -c scores.txt ; echo $?
1
$ sort scores.txt | sort -c ; echo $?
0
```

The second command shows that this actually works, by piping the output of the sort of scores.txt to **sort**.

Finally, you can merge two sorted files with the `-m` parameter, keeping the correct sort order. This is faster than concatenating both files, and resorting them.

---

[1]Of course, that will not really matter in this case, because we don't use numbers higher than 9, and virtually all character sets have numbers in a numerical order).

```
# sort -m scores-sorted.txt scores-sorted2.txt
```

Differences between files

Since text streams, and text files are very important in UNIX, it is often useful to show the differences between two text files. The main utilities for working with file differences are **diff** and **patch**. **diff** shows the differences between files. The output of **diff** can be processed by **patch** to apply the changes between two files to a file. "diffs" are also form the base of version/source management systems. The following sections describe **diff** and **patch**. To have some material to work with, the following two C source files are used to demonstrate these commands. These files are named `hello.c` and `hello2.c` respectively.

```
#include <stdio.h>

void usage(char *programName);

int main(int argc, char *argv[]) {
  if (argc == 1) {
    usage(argv[0]);
    return 1;
  }

  printf("Hello %s!\n", argv[1]);

  return 0;
}

void usage(char *programName) {
  printf("Usage: %s name\n", programName);
}
```

```
#include <stdio.h>
#include <time.h>

void usage(char *programName);

int main(int argc, char *argv[]) {
  if (argc == 1) {
    usage(argv[0]);
    return 1;
  }

  printf("Hello %s!\n", argv[1]);

  time_t curTime = time(NULL);
  printf("The date is %s\n", asctime(localtime(&curTime)));

  return 0;
```

```
}

void usage(char *programName) {
  printf("Usage: %s name\n", programName);
}
```

Listing differences between files

Suppose that you received the program `hello.c` from a friend, and you modified it to give the user the current date and time. You could just send your friend the updated program. But if a file grows larger, the can become uncomfortable, because the changes are harder to spot. Besides that, your friend may have also received modified program sources from other persons. This is a typical situation where **diff** becomes handy. **diff** shows the differences between two files. It most basic syntax is **diff file file2**, which shows the differences between `file` and `file2`. Let's try this with the our source files:

```
$ diff hello.c hello2.c
1a2 ❶
> #include <time.h> ❷
12a14,17
>   time_t curTime = time(NULL);
>   printf("The date is %s\n", asctime(localtime(&curTime)));
>
```

The additions from `hello2.c` are visible in this output, but the format may look a bit strange. Actually, these are commands that can be interpreted by the **ed** line editor. We will look at a more comfortable output format after touching the surface of the default output format.

Two different elements can be distilled from this output:

❶   This is an ed command that specified that text should be appended (`a`) after line 2.

❷   This is the actual text to be appended after the second line. The ">" sign is used to mark lines that are added.

The same elements are used to add the second block of text. What about lines that are removed? We can easily see how they are represented by swapping the two parameters to **diff**, showing the differences between `hello2.c` and `hello.c`:

```
$ diff hello2.c hello.c
2d1 ❶
< #include <time.h> ❷
14,16d12
<   time_t curTime = time(NULL);
<   printf("The date is %s\n", asctime(localtime(&curTime)));
<
```

The following elements can be distinguished:

❶   This is the ed delete command (`d`), stating that line 2 should be deleted. The second delete command uses a range (line 14 to 17).

❷   The text that is going to be removed is preceded by the "<" sign.

That's enough of the ed-style output. The GNU diff program included in Slackware Linux supports so-called unified diffs. Unified diffs are very readable, and provide context by default. **diff** can provide unified output with the *-u* flag:

```
$ diff -u hello.c hello2.c
--- hello.c      2006-11-26 20:28:55.000000000 +0100 ❶
+++ hello2.c     2006-11-26 21:27:52.000000000 +0100 ❷
@@ -1,4 +1,5 @@ ❸
 #include <stdio.h> ❹
+#include <time.h> ❺

 void usage(char *programName);

@@ -10,6 +11,9 @@

   printf("Hello %s!\n", argv[1]);

+   time_t curTime = time(NULL);
+   printf("The date is %s\n", asctime(localtime(&curTime)));
+
   return 0;
 }
```

The following elements can be found in the output

❶   The name of the original file, and the timestamp of the last modification time.

❷   The name of the changed file, and the timestamp of the last modification time.

❸   This pair of numbers show the location and size of the chunk that the text below affects in the original file and the modified file. So, in this case the numbers mean that in the affected chunk in the original file starts at line 1, and is four lines long. In the modified file the affected chunk starts at line 1, and is five lines long. Different chunks in diff output are started by this header.

❹   A line that is not preceded by a minus (-) or plus (+) sign is unchanged. Unmodified lines are included because they give contextual information, and to avoid that too many chunks are made. If there are only a few unmodified lines between changes, **diff** will choose to make only one chunk, rather than two chunks.

❺   A line that is preceded by a plus sign (+) is an addition to the modified file, compared to the original file.

As with the ed-style **diff** format, we can see some removals by swapping the file names:

```
$ diff -u hello2.c hello.c

--- hello2.c     2006-11-26 21:27:52.000000000 +0100
+++ hello.c      2006-11-26 20:28:55.000000000 +0100
@@ -1,5 +1,4 @@
 #include <stdio.h>
-#include <time.h>

 void usage(char *programName);

@@ -11,9 +10,6 @@

   printf("Hello %s!\n", argv[1]);
```

```
-   time_t curTime = time(NULL);
-   printf("The date is %s\n", asctime(localtime(&curTime)));
-
    return 0;
 }
```

As you can see from this output, lines that are removed from the modified file, in contrast to the original file are preceded by the minus (-) sign.

When you are working on larger sets of files, it's often useful to compare whole directories. For instance, if you have the original version of a program source in a directory named `hello.orig`, and the modified version in a directory named `hello`, you can use the `-r` parameter to recursively compare both directories. For instance:

```
$ diff -ru hello.orig hello
diff -ru hello.orig/hello.c hello/hello.c

--- hello.orig/hello.c  2006-12-04 17:37:14.000000000 +0100
+++ hello/hello.c       2006-12-04 17:37:48.000000000 +0100
@@ -1,4 +1,5 @@
 #include <stdio.h>
+#include <time.h>

 void usage(char *programName);

@@ -10,6 +11,9 @@

    printf("Hello %s!\n", argv[1]);

+   time_t curTime = time(NULL);
+   printf("The date is %s\n", asctime(localtime(&curTime)));
+
    return 0;
 }
```

It should be noted that this will only compare files that are available in both directories. The GNU version of diff, that is used by Slackware Linux provides the *-N* parameter. This parameters treats files that exist in only one of both directories as if it were an empty file. So for instance, if we have added a file named `Makefile` to the `hello` directory, using the *-N* parameter will give the following output:

```
$ diff -ruN hello.orig hello

diff -ruN hello.orig/hello.c hello/hello.c
--- hello.orig/hello.c  2006-12-04 17:37:14.000000000 +0100
+++ hello/hello.c       2006-12-04 17:37:48.000000000 +0100
@@ -1,4 +1,5 @@
 #include <stdio.h>
```

```
+#include <time.h>

 void usage(char *programName);

@@ -10,6 +11,9 @@

   printf("Hello %s!\n", argv[1]);

+   time_t curTime = time(NULL);
+   printf("The date is %s\n", asctime(localtime(&curTime)));
+
   return 0;
 }

diff -ruN hello.orig/Makefile hello/Makefile
--- hello.orig/Makefile 1970-01-01 01:00:00.000000000 +0100
+++ hello/Makefile      2006-12-04 17:39:44.000000000 +0100
@@ -0,0 +1,2 @@
+hello: hello.c
+       gcc -Wall -o $@ $<
```

As you can see the chunk indicator says that the chunk in the original file starts at line 0, and is 0 lines long.

UNIX users often exchange the output of **diff**, usually called "diffs" or "patches". The next section will show you how you can handle diffs. But you are now able to create them yourself, by redirecting the output of diff to a file. For example:

```
$ diff -u hello.c hello2.c > hello_add_date.diff
```

If you have multiple diffs, you can easily combine them to one diff, by concatenating the diffs:

```
$ cat diff1 diff2 diff3 > combined_diff
```

But make sure that they were created from the same directory if you want to use the **patch** utility that is covered in the next section.

Modifying files with diff output

Suppose that somebody would send you the output of **diff** for a file that you have created. It would be tedious to manually incorporate all the changes that were made. Fortunately, the **patch** can do this for you. **patch** accepts diffs on the standard input, and will try to change the original file, according to the differences that are registered in the diff. So, for instance, if we have the hello.c file, and the patch that we produced previously based on the changes between hello.c and hello2.c, we can patch hello.c to become equal to its counterpart:

```
$ patch < hello_add_date.diff
patching file hello.c
```

If you have hello2.c, you can check whether the files are identical now:

```
$ diff -u hello.c hello2.c
```

There is no output, so this is the case. One of the nice features of **patch** is that it can revert the changes made through a diff, by using the *-R* parameter:

```
$ patch -R < hello_add_date.diff
```

In these examples, the original file is patched. Sometimes you may want to want to apply the patch to a file with a different name. You can do this by providing the name of a file as the last argument:

```
$ patch helloworld.c < hello_add_date.diff
patching file helloworld.c
```

You can also use **patch** with diffs that were generated with the *-r* parameter, but you have to take a bit of care. Suppose that the header of a particular file in the diff is as follows:

```
-------------------------
|diff -ruN hello.orig/hello.c hello/hello.c
|--- hello.orig/hello.c 2006-12-04 17:37:14.000000000 +0100
|+++ hello/hello.c     2006-12-04 17:37:48.000000000 +0100
-------------------------
```

If you process this diff with **patch**, it will attempt to change hello.c. So, the directory that holds this file has to be the active directory. You can use the full pathname with the *-p n*, where *n* is the number of pathname components that should be stripped. A value of *0* will use the path as it is specified in the patch, *1* will strip the first pathname component, etc. In this example, stripping the first component will result in patching of hello.c. According to the Single UNIX Specification version 3 standard, the path that is preceded by --- should be used to construct the file that should be patched. The GNU version of patch does not follow the standard here. So, it is best to strip off to the point where both directory names are equal (this is usually the top directory of the tree being changed). In most cases where relative paths are used this can be done by using *-p 1*. For instance:

```
$ cd hello.orig
$ patch -p 1 < ../hello.diff
```

Or, you can use the *-d* parameter to specify in which directory the change has to be applied:

```
$ patch -p 1 -d hello.orig < hello.diff
patching file hello.c
patching file Makefile
```

If you want to keep a backup when you are changing a file, you can use the *-b* parameter of **patch**. This will make a copy of every affected file named filename.orig, before actually changing the file:

```
$ patch -b < hello_add_date.diff
$ ls -l hello.c*
-rw-r--r-- 1 daniel daniel 382 2006-12-04 21:41 hello.c
-rw-r--r-- 1 daniel daniel 272 2006-12-04 21:12 hello.c.orig
```

Sometimes a file can not be patched. For instance, if it has already been patched, it has changed to much to apply the patch cleanly, or if the file does not exist at all. In this case, the chunks that could not be saved are stored in a file with the name `filename.rej`, where *filename* is the file that **patch** tried to modify.

9.2. Regular expressions

Introduction

In daily life, you will often want to some text that matches to a certain pattern, rather than a literal string. Many UNIX utilities implement a language for matching text patterns, *regular expressions* (regexps). Over time the regular expression language has grown, there are now basically three regular expression syntaxes:

• Traditional UNIX regular expressions.

• POSIX extended regular expressions.

• Perl-compatible regular expressions (PCRE).

POSIX regexps are mostly a superset of traditional UNIX regexps, and PCREs a superset of POSIX regexps. The syntax that an application supports differs per application, but almost all applications support at least POSIX regexps.

Each syntactical unit in a regexp expresses one of the following things:

• **A character**: this is the basis of every regular expression, a character or a set of characters to be matched. For instance, the letter *p* or the the sign *,*.

• **Quantification**: a quantifier specifies how many times the preceding character or set of characters should be matched.

• **Alternation**: alternation is used to match "a or b" in which *a* and *b* can be a character or a regexp.

• **Grouping**: this is used to group subexpressions, so that quantification or alternation can be applied to the group.

Traditional UNIX regexps

This section describes traditional UNIX regexps. Because of a lack of standardisation, the exact syntax may differ a bit per utility. Usually, the manual page of a command provides more detailed information about the supported basic or traditional regular expressions. It is a good idea to learn traditional regexps, but to use POSIX regexps for your own scripts.

Matching characters

Characters are matched by themselves. If a specific character is used as a syntactic character for regexps, you can match that character by adding a backslash. For instance, \+ matches the plus character.

A period (.) matches any character, for instance, the regexp *b.g* matches *bag*, *big*, and *blg*, but not *bit*.

The period character, often provides too much freedom. You can use square brackets (*[ ]*) to specify characters which can be matched. For instance, the regexp *b[aei]g* matches *bag*, *beg*, and *big*, but nothing else. You can also match

any character but the characters in a set by using the square brackets, and using the caret (^) as the first character. For instance, *b[^aei]g* matches any three character string that starts with *b* and ends with *g*, with the exception of *bag*, *beg*, and *big*. It is also possible to match a range of characters with a dash (-). For example, *a[0-9]* matches a followed by a single number character.

Two special characters, the caret (^) and the dollar sign (*$*), respectively match the start and end of a line. This is very handy for parsing files. For instance, you can match all lines that start with a hash (#) with the regexp *^#*.

Quantification

The simplest quantification sign that traditional regular expressions support is the (Kleene) star (*). This matches zero or arbitrary instances of the preceding character. For instance, *ba** matches *b*, *babaa*, etc. You should be aware that a single character folowed by a star without any context matches every string, because *c** also matches a string that has zero *c* characters.

More specific repetitions can be specified with backslash-escaped curly braces. *\{x,y\}* matches the preceding character at least *x* times, but not more than *y* times. So, *ba\{1,3\}* matches *ba*, *baa*, and *baaa*.

Grouping

Backslash-escaped parentheses group various characters together, so that you can apply quantification or alternation to a group of characters. For instance, *\(ab\)\{1,3\}* matches *ab*, *abab*, and *ababab*.

Alternation

A backslash-escaped pipe vertical bar (\|) allows you to match either of two expressions. This is not useful for single characters, because *a\|b* is equivalent to *[ab]*, but it is very useful in conjunction with grouping. Suppose that you would like an expression that matches *apple* and *pear*, but nothing else. This can be done easily with the vertical bar: *(apple)|(pear)*.

POSIX extended regular expressions

POSIX regular expressions build upon traditional regular expressions, adding some other useful primitives. Another comforting difference is that grouping parenthesises, quantification accolades, and the alternation sign (/) are not backslash-escaped. If they are escaped, they will match the literal characters instead, thus resulting in the opposite behavior of traditional regular expressions. Most people find POSIX extended regular expressions much more comfortable, making them more widely used.

Matching characters

Normal character matching has not changed compared to the traditional regular expressions described in the section called "Matching characters"

Quantification

Besides the Kleene star (*), that matches the preceding character or group zero or more times, POSIX extended regular expressions add two new simple quantification primitives. The plus sign (+) matches the preceding character or group one or more times. For example, *a+*, matches *a* (or any string with more consecutive *a*'s), but does not match zero *a*'s. The questions mark character (*?*) matches the preceding character zero or one time. So, *ba?d* matches *bd* and *bad*, but not *baad* or *bed*.

Curly braces are used for repetition, like traditional regular expressions. Though the backslash should be omitted. To match *ba* and *baa*, one should use *ba{1,2}* rather than *ba\{1,2\}*.

Grouping

Grouping is done in the same manner as traditional regular expressions, leaving out the escape-backslashes before the parenthesises. For example, *(ab){1,3}* matches *ab*, *abab*, and *ababab*.

Alternation

Alternation is done in the same manner as with traditional regular expressions, leaving out the escape-backslashes before the vertical bar. So, *(apple)|(pear)* matches *apple* and *pear*.

9.3. grep

Basic grep usage

We have now arrived at one of the most important utilties of the UNIX System, and the first occasion to try and use regular expressions. The **grep** command is used to search a text stream or a file for a pattern. This pattern is a regular expression, and can either be a basic regular expression or a POSIX extended regular expression (when the -*E* parameter is used). By default, **grep** will write the lines that were matched to the standard output. In the most basic syntax, you can specify a regular expression as an argument, and **grep** will search matches in the text from the standard input. This is a nice manner to practice a bit with regular expressions.

```
$ grep '^\(ab\)\{2,3\}$'
ab
abab
abab
ababab
ababab
abababab
```

The example listed above shows a basic regular expression in action, that matches a line solely consisting of two or three times the *ab* string. You can do the same thing with POSIX extended regular expressions, by adding the -*E* (for extended) parameter:

```
$ grep -E '^(ab){2,3}$'
ab
abab
abab
ababab
ababab
abababab
```

Since the default behavior of **grep** is to read from the standard input, you can add it to a pipeline to get the interesting parts of the output of the preceding commands in the pipeline. For instance, if you would like to search for the string *2006* in the third column in a file, you could combine the **cut** and **grep** command:

```
$ cut -f 3 | grep '2006'
```

grepping files

Naturally, **grep** can also directly read a file, rather than the standard input. As usual, this is done by adding the files to be read as the last arguments. The following example will print all lines from the `/etc/passwd` file that start with the string *daniel:*.

```
$ grep "^daniel" /etc/passwd
daniel:*:1001:1001:Daniel de Kok:/home/daniel:/bin/sh
```

With the `-r` option, **grep** will recursively traverse a directory structure, trying to find matches in each file that was encountered during the traversal. Though, it is better to combine **grep** with **find** and the *-exec* operand in scripts that have to be portable.

```
$ grep -r 'somepattern' somedir
```

is the non-portable functional equivalent of

```
$ find /somedir -type f -exec grep 'somepattern' {} \; -print
```

Pattern behavior

grep can also print all lines that do not match the pattern that was used. This is done by adding the `-v` parameter:

```
$ grep -Ev '^(ab){2,3}$'
ab
ab
abab
ababab
abababab
abababab
```

If you want to use the pattern in a case-insensitive manner, you can add the `-i` parameter. For example:

```
$ grep -i "a"
a
a
A
A
```

You can also match a string literally with the `-F` parameter:

```
$ grep -F 'aa*'
a
aa*
```

```
aa*
```

Using multiple patterns

As we have seen, you can use the alternation character (/) to match either of two or more subpatterns. If two patterns that you would like to match differ a lot, it is often more comfortable to make two separate patterns. **grep** allows you to use more than one pattern by separating patterns with a newline character. So, for example, if you would like to print lines that match either the *a* or *b* pattern, this can be done easily by starting a new line:

```
$ grep 'a
b'
a
a
b
b
c
```

This works, because quotes are used, and the shell passes quoted parameters literally. Though, it must be admitted that this is not quite pretty. **grep** accepts one or more *-e pattern* parameters, giving the opportunity to specify more than one parameter on one line. The **grep** invocation in the previous example could be rewritten as:

```
$ grep -e 'a' -e 'b'
```

Chapter 10. Process management

10.1. Theory

Processes

A running instance of a program is called a *process*. Each process has its own protected memory, named the *process address space*. This address space consists of two areas: the *text area* and the *data area*. The text area contains the actual program code, and tells the system what to do. The data area stores constant and runtime data of a process. Since there are many processes on a system, and only one or a few processors, the operating system kernel divides processor time between processes. This process is called *time-sharing*.

Table 10.1. The structure of a process

Field	Description
pid	The numeric process identifier
ppid	The process identifier of the parent process
euid	The effective user ID of the process.
ruid	The real user ID of the process
egid	The group ID of the process
rgid	The real group ID of the process
fd	Pointer to the list of open file descriptors
vmspace	Pointer to the process address space.

Table 10.1, "The structure of a process" lists the most important fields of information that a kernel stores about a process. Each process can be identified uniquely by its *PID* (process identifier), which is an unsigned number. As we will see later, a user can easily retrieve the PID of a process. Each process is associated with a *UID* (user ID) and *GID* (group ID) on the system. Each process has a *real UID*, which is the UID as which the process was started, and the *effective UID*, which is the UID as which the process operates. Normally, the effective UID is equal to the real UID, but some programs ask the system to change its effective UID. The effective UID determines is used for access control. This means that if a user named joe starts a command, say less, less can only open files that joe has read rights for. In parallel, a process also has an *real GID* and an *effective GID*.

Many processes open files, the handle used to operate on a file is called a *file descriptor*. The kernel manages a list of open file descriptors for each process. The *fd* field contains a pointer to the list of open files. The *vmspace* field points to the process address space of the process.

Figure 10.1. Process states
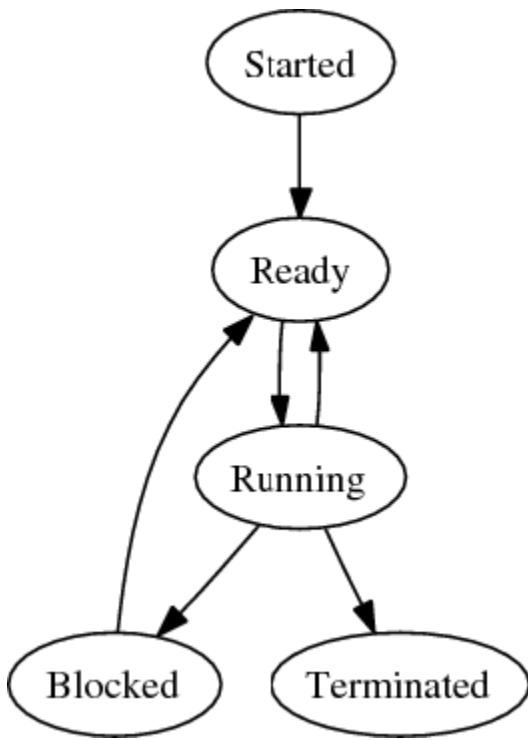
Not every process is in need of CPU time at a given moment. For instance, some processes maybe waiting for some *I/O* (Input/Output) operation to complete or may be terminated. Not taking subtleties in account, processes are normally *started*, *running*, *ready* (to run), *blocked* (waiting for I/O), or *terminated*. Figure 10.1, "Process states" shows the lifecycle of a process. A process that is terminated, but for which the process table entry is not reclaimed, is often called a *zombie process*. Zombie processes are useful to let the parent process read the exit status of the process, or reserve the process table entry temporarily.

Creating new processes

New processes are created with the *fork()* system call. This system call copies the process address space and process information of the caller, and gives the new process, named the child process, a different PID. The child process will continue execution at the same point as the parent, but will get a different return value from the *fork()* system call. Based on this return value the code of the parent and child can decide how to continue executing. The following piece of C code shows a *fork()* call in action:

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid == 0)
    printf("Hi, I am the child!\n");
  else
    printf("Hi, I am the parent, the child PID is %d!\n", pid);

  return 0;
```

}

. This little program calls *fork()*, storing the return value of *fork()* in the variable *pid*. *fork()* returns the value *0* to the child, and the PID of the child to the parent. Since this is the case, we can use a simple conditional structure to check the value of the *pid* variable, and print an appropriate message.

You may wonder how it is possible to start new programs, since the *fork()* call duplicates an existing process. That is a good questions, since with *fork()* alone, it is not possible to execute new programs. UNIX kernels also provide a set of system calls, starting with *exec*, that load a new program image in the current process. We saw at the start of this chapter that a process is a running program -- a process was constructed in memory from the program image that is stored on a storage medium. So, the *exec* family of system calls gives a running process the facilities to replace its contents with a program stored on some medium. In itself, this is not wildly useful, because every time the an *exec* call is done, the original calling code (or program) is removed from the process. This can be witnessed in the following C program:

```
#include <stdio.h>
#include <unistd.h>

int main() {
  execve("/bin/ls", NULL, NULL);

  /* This will never be printed, unless execve() fails. */
  printf("Hello world!\n");

  return 0;
}
```

This program executes **ls** with the *execve()* call. The message printed with *printf()* will never be shown, because the running program image is replaced with that of **ls**. Though, a combination of *fork()* and the *exec* functions are very powerful. A process can fork itself, and let the child "sacrifice" itself to run another program. The following program demonstrates this pattern:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid == 0)
    execve("/bin/ls", NULL, NULL);

  printf("Hello world!");

  return 0;
}
```

This program forks itself first. The program image of the child process will be replaced with **ls**, while the parent process prints the "Hello world!" message to the screen and returns.

This procedure is followed by many programs, including the shell, when a command is executed from the shell prompt. In fact all processes on a UNIX system are directly or indirectly derrived from the *init* process, which is the first program that is started on a UNIX system.

Threads

Although forks are very useful to build some parallelism[1], they can be to expensive for some purposes. Copying the whole process takes some time, and there is cost involved if processes want to share data. This is solved by offering a more lightweight alternative, namely allowing more than one thread of execution. Each thread of execution is executed separately, but the process data is shared between the threads.

Writing good *multithreaded* programs requires good knowledge of data sharing and locking. Since all data is shared, uncareful programming can lead to bugs like race conditions.

10.2. Analyzing running processes

Listing running processes

UNIX systems provide the **ps** command to show a list of running processes. Unfortunately, this command is an example of the pains of the lack of standardization. The BSD and System V variants of **ps** have their own sets of options. Fortunately, GNU/Linux implements both the System V and BSD-style parameters, as well as some (GNU-style) long options. Options preceded by a dash are interpreted as System V options and options without a dash as BSD options. We will describe the System V-style options in this section.

If **ps** is used without any parameters, it shows all processes owned by the user that invokes **ps** and that are attached to the same controlling terminal. For example:

```
$ ps
  PID TTY          TIME CMD
 8844 pts/5    00:00:00 bash
 8862 pts/5    00:00:00 ps
```

A lot of useful information can be distilled from this output. As you can see, two processes are listed: the shell that we used to call **ps** (*bash*), and the **ps** command itself. In this case there are four information fields. *PID* is the process ID of a process, *TTY* the controlling terminal, *TIME* the amount of CPU time the proces has used, and **CMD** the command or program of which a copy is running. The fields that are shown by default may vary a bit per system, but usually at least these fields are shown, with somewhat varying field labels.

Sometime you may want to have a broader view of processes that are running. Adding the −a option shows all processes that are associated with terminals. For instance:

```
$ ps -a
  PID TTY          TIME CMD
 7487 pts/1    00:00:00 less
 8556 pts/4    00:00:10 emacs-x
11324 pts/3    00:00:00 ps
```

As you can see, processes with different controlling terminals are shown. Though, in contrast to the plain **ps** output, only processes that control the terminal at the moment are shown. For instance, the shell that was used to call **ps** is not shown.

You can also print all processes that are running, including processes that are not associated with a terminal, by using the −A option:

---

[1]For instance, a web server could fork multiple child processes to handle requests

```
$ ps -A | head -n 10
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 migration/0
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:00 watchdog/0
    5 ?        00:00:00 migration/1
    6 ?        00:00:00 ksoftirqd/1
    7 ?        00:00:00 watchdog/1
    8 ?        00:00:00 events/0
    9 ?        00:00:00 events/1
```

You can print all processes with a certain user ID, with the *-U* option. This option accepts a user name as a parameter, or multiple user names that are separated by a comma. The following command shows all processes that have *xfs* or *rpc* as their user ID:

```
$ ps -U xfs,rpc
  PID TTY          TIME CMD
 2409 ?        00:00:00 portmap
 2784 ?        00:00:00 xfs
```

Likewise, you can also print processes with a particular group ID, with the *-G* option:

```
$ ps -G messagebus,haldaemon
  PID TTY          TIME CMD
 8233 ?        00:00:00 dbus-daemon
11312 ?        00:00:00 hald
11320 ?        00:00:00 hald-addon-keyb
11323 ?        00:00:00 hald-addon-acpi
```

If you would like to have a list for a physical or pseudo-terminal, you can use the `-t` option:

```
$ ps -t tty2
  PID TTY          TIME CMD
 2655 tty2     00:00:00 getty
```

10.3. Managing processes

Sending singals to a process

Signals are a crude, but effective form of inter-process communication (IPC). A signal is basically a number that is delivered to a process that has a special meaning. For all signals there are default signal handlers. Processes can install their own signal handlers, or choose to ignore signals. Some signals (normally SIGKILL and SIGSTOP) can not be ignored. All signals have convenient symbolic names.

Only a few signals are normally interesting for interactive use on UNIX(-like) systems. These are (followed by their number):

- *SIGKILL (9):* forcefully kill a process.

- *SIGTERM (15):* request a process to terminate. Since this is a request, a program could ignore it, in contrast to *SIGKILL*.

- *SIGHUP (1):* Traditionally, this has signalled a terminal hangup. But nowadays some daemons (e.g. *inetd*) reread their configuration when this signal is sent.

The **kill** command is used to send a signal to a process. By default, **kill** sends the *SIGTERM* signal. To send this signal, the process ID of the process that you would like to send this signal to should be added as a parameter. For instance:

```
$ kill 15631
```

To send another signal, you can use one of two options: *-signalnumber* or *-signalname*. So, the following commands both send the *SIGKILL* signal to the process with process ID *15631*:

```
$ kill -9 15631
```

```
$ kill -SIGKILL 15631
```

Being nice to others

In an act of altruism you can be nice to other users of computer resources. If you plan to run a CPU-time intensive process, but do not want that to interfere with work of other users on the system (or other processes), you can assign some grade of 'niceness' to a process. Practically, this means that you will be able to influence the scheduling priority of a process. Nicer processes get a lower scheduling priority. The normal niceness of a process is *0*, and can be changed by executing a program with the **nice** command. The *-n [niceness]* option can be used to specify the niceness:

```
$ nice -n 20 cputimewaster
```

The maximum number for niceness is implementation-dependent. If a program was started with **nice**, but no niceness was specified, the niceness will be set to *10*. In case you were wondering: yes, you can also be rude, but this right is restricted to the *root* user. You can boost the priority of a process by specifying a negative value as the niceness.

You can also modify the niceness of a running processes with the **renice** command. This can be done for specific process IDs (`-p PIDs`), users (`-u user/uid`), and effective groups (`-g group/gid`). The new niceness is specified as the first parameter.

The niceness of a process can only be increased. And, of course, no user except for *root* can affect the niceness of processes of other users.

Lets look at an example, to set the niceness of a process with PID *3108* to *14*, you could use the following command:

```
$ renice 14 -p 3108
```

10.4. Job control

It is often useful to group processes to allow operations on a set of processes, for instance to distribute a signal to all processes in a group rather than a single process. Not too suprisingly, these sets of processes are called *program groups* in UNIX. After a fork, a child process is automatically a member of the process group of the parent. Though, new process groups can be created by making one process a process group leader, and adding other processes to the group. The process group ID is the PID of the process group leader.

Virtually all modern UNIX shells give processes that are created through the invocation of a command their own process group. All processes in a pipeline are normally added to one process group. If the following commands that create a pipepine are executed

```
cat | tr -s ' ' | egrep 'foob.r'
```

the shell roughly performs the following steps:

1. Three child processes are forked.

2. The first process in the pipeline is put in a process group with its own PID as the process group ID, making it the process leader. The other processes in the pipeline are added to the process group.

3. The file descriptors of the processes in the pipeline are reconfigured to form a pipeline.

4. The programs in the pipeline are executed.

The shell uses process groups to implement *job control*. A shell can run multiple jobs in the background, there can be multiple stopped job, and one job can be in the foreground. A foreground job is wired to the terminal for its standard input (meaning that it is the job the gets user input).

Stopping and continuing a job

A job that is in the foreground (thus, a job that potentially accepts userinput from the terminal) can be stopped by pressing *Ctrl-z* (pressing both the 'Ctrl' and 'z' keys simultaneously). This will stop the job, and will handle control over the terminal back to the shell. Let's try this with the **sleep** command, which waits for the number of seconds provided as an argument:

```
$ sleep 3600
Ctrl-z
[1]+  Stopped                 sleep 3600
```

The process group, which we will refer to as a job has been stopped now, meaning the the **sleep** has stopped counting - it's execution is completely stopped. You can retrieve a list of jobs with the **jobs** command:

```
$ jobs
[1]+  Stopped                 sleep 3600
```

This shows the job number (*1*), its state, and the command that was used to start this job. Let's run another program, stop that too, and have another look at the job listing.

```
$ cat
Ctrl-z
[2]+  Stopped                 cat
$ jobs
[1]-  Stopped                 sleep 3600
[2]+  Stopped                 cat
```

As expected, the second job is also stopped, and was assigned job number *2*. The plus sign (+) following the first job has changed to a minus (-) sign, while the second job is now marked by a plus sign. The plus sign is used to indicate the *current job*. The **bg** and **fg** commands that we will look at shortly, will operate on the current job if no job was specified as a parameter.

Usually, when you are working with jobs, you will want to move jobs to the foreground again. This is done with the **fg** command. Executing **fg** without a parameter will put the current job in the foreground. Most shells will print the command that is moved to the foreground to give an indication of what process was moved to the foreground:

```
$ fg
cat
```

Of course, it's not always useful to put the current job in the foreground. You can put another job in the foreground by adding the job number preceded by the percentage sign (%) as an argument to **fg**:

```
$ fg %1
sleep 3600
```

Switching jobs my stopping them and putting them in the foreground is often very useful when the shell is used interactively. For example, suppose that you are editing a file with a text editor, and would like to execute some other command and then continue editing. You could stop the editor with *Ctrl-z*, execute a command, and put the editor in the foreground again with **fg**.

Background jobs

Besides running in the foreground, jobs can also run in the background. This means that they are running, but input from the terminal is not redirected to background processes. Most shells do configure background jobs to direct output to the terminal of the shell where they were started.

A process that is stopped can be continued in the background with the **bg** command:

```
$ sleep 3600

[1]+  Stopped                 sleep 3600
$ bg
[1]+ sleep 3600 &
$
```

You can see that the job is indeed running with **jobs**:

```
$ jobs
[1]+  Running                 sleep 3600 &
```

Like **fg**, you can also move another job than the current job to the background by specifying its job number:

```
$ bg %1
[1]+ sleep 3600 &
```

You can also run put a job directly in the background when it is started, by adding an trailing ampersand (&) to a command or pipeline. For instance:

```
$ sleep 3600 &
[1] 5078
```

Part III. Editing and typesetting

Table of Contents

Chapter 11. LaTeX

11.1. Introduction

LaTeX is a typesetting system that can be used to produce high-quality articles, books, letters and other publications. LaTeX is based on TeX, a lower-level typesetting language that was designed by Donald E. Knuth. LaTeX does not work like a WYSIWYG (what you see is what you get) word processor, the kind of document preparation system most people are accustomed to. With LaTeX you do not have to care about formatting the document, only about writing the document.

LaTeX files are plain-text files that contain LaTeX macros. LaTeX formats the document based on the macros that are used. In the beginning using LaTeX may be a bit awkward to a new user. But after a while you will discover that using LaTeX has some distinct advantages. To name a few:

- LaTeX-formatted documents look very professional.

- You do not have to care about the layout of your documents. You just add structure to your documents, and LaTeX takes care of the formatting.

- LaTeX files are plain text, and can easily be changed using standard UNIX tools, such as **vi**, **sed** or **awk**

- LaTeX provides very good support for typesetting things like mathematical formula, references and Postscript images.

LaTeX is very extensive, so this chapter will only touch the surface of LaTeX. But it should be enough to get you started to be able to make simple documents.

11.2. Preparing basic LaTeX documents

Minimal document structure

Each LaTeX document has some basic minimal structure that describes the document. Let's look at an example:

```
\documentclass[10pt,a4paper]{article} ❶

\title{The history of symmetric ciphers} ❷
\author{John Doe} ❸

\begin{document} ❹

This is a basic document. ❺

\end{document} ❻
```

You can already see the basic syntactical structure of LaTeX commands. A command is started with a backslash, followed by the name of the command. Each macro has a mandatory argument that is placed in accolades, and an optional argument that is placed in square brackets.

❶ The first command in every document is the *documentclass*. This command specifies what kind of document LaTeX is dealing with. The type of document is specified as a mandatory parameter. You can also specify some optional parameters, such as the font size and the paper size. In this case the font size is changed from the default 12pt to 10pt, and A4 is used as the paper size. The document classes that are available in LaTeX are shown in Table 11.1, "LaTeX document classes".

❷ After the documentclass you can add some meta-information to the document, like the title of the document. In this case the title is *The history of symmetric ciphers*.

❸ The author command specifies the author of the book.

❹ The *\begin* command marks the beginning of an environment. There are many different environments, but they all implicate certain typesetting conventions for the text that is in an environment. In this case we start an *document* environment. This is a very basic environment, LaTeX interprets everything in this environment as the body of the text.

❺ The content of the document can be placed within the *document*, in this case a friendly warning about the nature of the document.

❻ All environments eventually have to be closed. The *document* environment is the last environment that is closed, because it denotes the end of the body of the text.

Table 11.1. LaTeX document classes

Class
article
book
letter
report

Generating printable formats

Once you have a LaTeX file, you can use the **latex** command to generate a DVI (Device Independent format) file:

```
$ latex crypto.tex
This is pdfeTeX, Version 3.141592-1.21a-2.2 (Web2C 7.5.4)
entering extended mode
(./crypto.tex
LaTeX2e <2003/12/01>
Babel <v3.8d> and hyphenation patterns for american, french, german, ngerman, b
ahasa, basque, bulgarian, catalan, croatian, czech, danish, dutch, esperanto, e
stonian, finnish, greek, icelandic, irish, italian, latin, magyar, norsk, polis
h, portuges, romanian, russian, serbian, slovak, slovene, spanish, swedish, tur
kish, ukrainian, nohyphenation, loaded.
(/usr/share/texmf/tex/latex/base/article.cls
Document Class: article 2004/02/16 v1.4f Standard LaTeX document class
(/usr/share/texmf/tex/latex/base/size10.clo)) (./crypto.aux) [1] (./crypto.aux)
)
 Output written on crypto.dvi (1 page, 248 bytes).
 Transcript written on crypto.log.
```

As the LaTeX command reports a DVI file is created after running the **latex** command. You can view this file with an X viewer for DVI files, **xdvi**:

```
$ xdvi crypto.dvi
```

This file is not directly printable (although DVI files can be printed with **xdvi**). An ideal format for printable files is Postscript. You can generate a Postscript file from the DVI file with one simple command:

```
$ dvips -o crypto.ps crypto.dvi
```

The `-o` specifies the output (file) for the Postscript document. If this parameter is not specified, the output will be piped through **lpr**, which will schedule the document to be printed.

PDF (Portable Document Format) is another popular format for electronic documents. PDF can easily be generated from Postscript:

```
$ ps2pdf crypto.ps
```

The resulting output file will be the same as the input file, with the `.ps` extension replaced with **.pdf**.

Sections

Now that you know how to create a basic LaTeX document, it is a good time to add some more structure. Structure is added using sections, subsections, and subsubsections. These structural elements are made with respectively the *\section*, *\subsection* and *\subsubsection* commands. The mandatory parameter for a section is the title for the section. Normal sections, subsections and subsubsections are automatically numbered, and added to the table of contents. By adding a star after a section command, for example *\section*{title}* section numbering is suppressed, and the section is not added to the table of contents. The following example demonstrates how you can use sections:

```
\documentclass[10pt,a4paper]{article}

\title{The history of symmetric ciphers}
\author{John Doe}

\begin{document}

\section{Pre-war ciphers}

To be done.

\section{Modern ciphers}

\subsection*{Rijndael}

Rijndael is a modern block cipher that was designed by Joan Daemen and
Vincent Rijmen.

In the year 2000 the US National Institute of Standards and Technologies
selected Rijndael as the winner in the contest for becoming the Advanced
Encryption Standard, the successor of DES.

\end{document}
```

The example above is pretty straightforward, but this is a good time to look at how LaTeX treats end of line characters and empty lines. Empty lines are ignored by LaTeX, making the text a continuous flow. An empty line starts a new paragraph. All paragraphs but the first paragraph are stared with a extra space left of the first word.

Font styles

Usually you may want to work with different font styles too. LaTeX has some commands that can be used to change the appearance of the current font. The most commonly used font commands are *\emph* for emphasized text, and *\textbf*. Have a look at Table 11.2, "LaTeX font styles" for a more extensive list of font styles. Emphasis and bold text are demonstrated in this example paragraph:

```
Working with font styles is easy. \emp{This text is emphasized} and
\textbf{this text is bold}.
```

Table 11.2. LaTeX font styles

Command	Description
\emph	Add emphasis to a font.
\textbf	Print the text in bold.
\textit	Use an italic font.
\textsl	Use a font that is slanted.
\textsc	Use small capitals.
\texttt	Use a typewriter font.
\textsf	Use a sans-serif font.

Part IV. Electronic mail

Table of Contents

Chapter 12. Reading and writing e-mail with mutt

12.1. Introduction

Mutt is a mail user agent (MUA) that can be used for reading and writing e-mail. Mutt is a text-mode installation, meaning that it can be used on the console, over SSH and in an X terminal. Due to its menu interface, it is very easy to read large amounts of e-mail in a short time, and mutt can be configured to use your favorite text editor.

This chapter will discuss how you can customize mutt for your needs, how to use it, and how PGP/GnuPG support is used.

12.2. Usage

Mutt is pretty simple to use, though it may take some time to get used to the keys that are used to navigate, read and write e-mails. The next few sections describe some of the more important keys. Mutt provides a more thorough overview of available keys after pressing the <h> key.

Browsing the list of e-mails

After invoking the **mutt** command, an overview of all e-mails will show up. You can browse through the list of e-mails with the up and down arrow keys, or the <k> and <j> keys.

Reading e-mails

To read an e-mail, use the <Enter> key, after selecting an e-mail in the overview of e-mails. When reading an e-mail you can use the <Page Up> and <Page Down> to browse through an e-mail. You can still use the navigational keys used to browse the list of e-mail to browse to other e-mails.

If an e-mail has any attachments, you can see them by pressing the <v> key. You can view individual attachments by selecting them and pressing the <Enter> key. To save an attachment to a file, press the <s> key.

Sending e-mails

You can compose a new e-mail with the <c> key, or reply to a selected e-mail with the <r> key. Mutt will ask you to specify the recipient (*To:*), and a subject (*Subject:*). After entering this information an editor is launched (**vi** is used by default), which you can use to compose the e-mail. After saving the e-mail, and quitting the editor, mutt will give you the opportunity to make any changes to the e-mail. If you decide that you want to alter the e-mail, you can restart the editor with the <e> key. You can change the recipient or the subject with respectively <t> or <s>. Finally, you can send the e-mail by pressing <y>. If you would like to cancel the e-mail, press <q>. Mutt will ask you whether you want to postpone the e-mail. If you do so, you will be given the opportunity to re-do the e-mail the next time you compose a message.

12.3. Basic setup

There are a few mutt settings you often want to configure. This section describes these settings. User-specific mutt customizations can be made in the `.muttrc` in the user's home directory. You can change global mutt settings in `/etc/mutt/Muttrc`.

Customized headers

Each e-mail has headers with various information. For example, the header contains information about the path an e-mail has traversed after it has been sent. The sender (*From:*) and recipient (*To:*) e-mail addresses are also stored in the headers, as well as the subject (*Subject:*) of an e-mail.

> **Note**
>
> In reality the *To:* header is not used to determine the destination of an e-mail during the deliverance process of the e-mail. MTAs use the *envelope address* to determine the destination of the e-mail. Though, most MUAs use the *To:* address that the user fills in as the envelope address.

You can add your own headers to an e-mail with the *my_hdr* configuration option. This option has the following syntax: *my_hdr <header name>: <header contents>*. For example, you can add information about what OS you are running by adding the following line to your mutt configuration:

```
my_hdr X-Operating-System: Slackware Linux 10.2
```

You can also override some of the headers that are normally used, such as the sender address that is specified in the *From:* header:

```
my_hdr From: John Doe <john.doe@example.org>
```

The sendmail binary

By default mutt uses the sendmail MTA to deliver e-mails that were sent. You can use another command to send e-mail by altering the *sendmail* configuration variable. The sendmail replacement must handle the same parameter syntax as sendmail. For example, if you have installed MSMTP to deliver e-mails, you can configure mutt to use it by adding the following line to your mutt configuration:

```
set sendmail="/usr/bin/msmtp"
```

When you have completely replaced sendmail with another MTA, for instance Postfix, it is usually not needed to set this parameter, because most MTAs provide an alternative sendmail binary file.

12.4. Using IMAP

Normally, mutt reads e-mail from the user's local spool mailbox. However, mutt also has support for using IMAP mailboxes. IMAP (the Internet Message Access Protocol) is a protocol that is used for accessing e-mail from a remote server, and is supported by many e-mail servers. Mutt uses the following URL format for representing IMAP servers:

imap://[user@]hostname[:port]/[mailbox]

Or the following format for IMAP over SSL/TLS:

imaps://[user@]hostname[:port]/[mailbox]

You can directly use this syntax in folder-related operatings. For example, if you press "c" to change from folder, you can enter an IMAP URL. This is a bit tedious, so it is easier to store this information in your `.muttrc` file.

If you use only one IMAP account, you can set the INBOX folder of this account as the spool mailbox, and the main IMAP account as the e-mail folder. For example, adding the following lines to your `.muttrc` configurion file will set up mutt to log in to the *imap.example.org* server as the *me* user.

```
set folder=imap://me@imap.example.org/
set spoolfile=imap://me@imap.example.org/INBOX
```

12.5. Signing/encrypting e-mails

Introduction

Mutt provides excellent support for signing or encrypting e-mails with GnuPG. One might wonder why he or she should use one of these techniques. While most people do not feel the need to encrypt most of their e-mails, it generally is a good idea to sign your e-mails. There are, for example, a lot of viruses these days that use other people's e-mail addresses in the From: field of viruses. If the people who you are communicating with know that you sign your e-mails, they will not open fake e-mail from viruses. Besides that it looks much more professional if people can check your identity, especially in business transactions. For example, who would you rather trust, vampire_boy93853@hotmail.com, or someone using a professional e-mail address with digitally signed e-mails?

This section describes how you can use GnuPG with mutt, for more information about GnuPG read Section 8.9, "Encrypting and signing files".

Configuration

An example configuration for using GnuPG in mutt can be found in `/usr/share/doc/mutt/samples/gpg.rc`. In general the contents of this file to your mutt configuration will suffice. From the shell you can add the contents of this file to you `.muttrc` with the following command:

```
$ cat /usr/share/doc/mutt/samples/gpg.rc >> ~/.muttrc
```

There are some handy parameters that you can additionally set. For example, if you always want to sign e-mails, add the following line to your mutt configuration:

```
set crypt_autosign = yes
```

Another handy option is *crypt_replyencrypt*, which will automatically encrypt replies to messages that were encrypted. To enable this, add the following line to your mutt configuration:

```
set crypt_replyencrypt = yes
```

Usage

If you have set some of the automatical options, like *crypt_autosign* GnuPG usage of mutt is mostly automatic. If not, you can press the <p> key during the final step of sending an e-mail. In the bottom of the screen various GnuPG/PGP options will appear, which you can access via the letters that are enclosed in parentheses. For example, <s> signs e-mails, and <e> encrypts an e-mail. You can always clear any GnuPG option you set by pressing <p> and then <c>.

Chapter 13. Sendmail

13.1. Introduction

Sendmail is the default Mail Transfer Agent (MTA) that Slackware Linux uses. sendmail was originally written by Eric Allman, who still maintains sendmail. The primary role of the sendmail MTA is delivering messages, either locally or remotely. Delivery is usually done through the SMTP protocol. The means that sendmail can accept e-mail from remote sites through the SMTP port, and that sendmail delivers site destined for remote sites to other SMTP servers.

13.2. Installation

Sendmail is available as the *sendmail* package in the "n" disk set. If you want to generate your own sendmail configuration files, the *sendmail-cf* package is also required. For information about how to install packages on Slackware Linux, refer to Chapter 17, *Package Management*.

You can let Slackware Linux start sendmail during each boot by making the `/etc/rc.d/rc.sendmail` executable. You can do this by executing:

```
# chmod a+x /etc/rc.d/rc.sendmail
```

You can also start, stop and restart sendmail by using *start*, *stop*, and *restart* as a parameter to the sendmail initialization script. For example, you can restart sendmail in the following way:

```
# /etc/rc.d/rc.sendmail restart
```

13.3. Configuration

The most central sendmail configuration file is `/etc/mail/sendmail.cf`; this is where the behavior of sendmail is configured, and where other files are included. The syntax of `/etc/mail/sendmail.cf` is somewhat obscure, because this file is compiled from a much simpler `.mc` files that uses M4 macros that are defined for sendmail.

Some definitions can easily be changed in the `/etc/mail/sendmail.cf` file, but for other changes it is better to create your own `.mc` file. Examples in this chapter will be focused on creating a customized mc file.

Working with mc files

In this section we will look how you can start off with an initial mc file, and how to compile your own .cf file to a cf file. There are many interesting example mc files available in `/usr/share/sendmail/cf/cf`. The most interesting examples are `sendmail-slackware.mc` (which is used for generating the default Slackware Linux `sendmail.cf`), and `sendmail-slackware-tls.mc` which adds TLS support to the standard Slackware Linux sendmail configuration. If you want to create your own sendmail configuration, it is a good idea to start with a copy of the standard Slackware Linux mc file. For instance, suppose that we would like to create a configuration file for the server named *straw*, we could execute:

```
# cd /usr/share/sendmail/cf/cf
# cp sendmail-slackware.mc sendmail-straw.mc
```

and start editing `sendmail-straw.mc`. After the configuration file is modified to our tastes, M4 can be used to compile a cf file:

```
# m4 sendmail-straw.mc > sendmail-straw.cf
```

If we want to use this new configuration file as the default configuration, we can copy it to `/etc/mail/ sendmail.cf`:

```
# cp sendmail-straw.cf /etc/mail/sendmail.cf
```

Using a smarthost

If you would like to use another host to deliver e-mail to locations to which the sendmail server you are configuring can not deliver you can set up sendmail to use a so-called "smart host". Sendmail will send the undeliverable e-mail to the smart host, which is in turn supposed to handle the e-mail. You do this by defining *SMART_HOST* in your mc file. For example, if you want to use *smtp2.example.org* as the smart host, you can add the following line:

```
define(`SMART_HOST','stmp2.example.org')
```

Alternative host/domain names

By default sendmail will accept mail destined for localhost, and the current hostname of the system. You can simply add additional hosts or domains to accept e-mail for. The first step is to make sure that the following line is added to your mc configuration:

```
FEATURE(`use_cw_file')dnl
```

When this option is enabled you can add host names and domain names to accept mail for to `/etc/mail/local-host-names`. This file is a newline separated database of names. For example, the file could look like this:

```
example.org
mail.example.org
www.example.org
```

Virtual user table

Often you may want to map e-mail addresses to user names. This is needed when the user name differs from the part before the "@" part of an e-mail address. To enable this functionality, make sure the following line is added to your mc file:

```
FEATURE(`virtusertable',`hash -o /etc/mail/virtusertable.db')dnl
```

The mappings will now be read from /etc/mail/virtusertable.db. This is a binary database file that should not directly edit. You can edit /etc/mail/virtusertable instead, and generate /etc/mail/virtusertable.db from that file.

The /etc/mail/virtusertable file is a simple plain text file. That has a mapping on each line, with an e-mail address and a user name separated by a tab. For example:

```
john.doe@example.org     john
john.doe@mail.example.org          john
```

In this example both e-mail for *john.doe@example.org* and *john.doe@mail.example.org* will be delivered to the *john* account. It is also possible to deliver some e-mail destined for a domain that is hosted on the server to another e-mail address, by specifying the e-mail address to deliver the e-mail to in the second in the second column. For example:

```
john.doe@example.org     john.doe@example.com
```

After making the necessary changes to the virtusertable file you can generate the database with the following command:

```
# makemap hash /etc/mail/virtusertable < /etc/mail/virtusertable
```

Part V. System administration

Table of Contents

Chapter 14. User management

14.1. Introduction

GNU/Linux is a multi-user operating system. This means that multiple users can use the system, and they can use the system simultaneously. The GNU/Linux concepts for user management are quite simple. First of all, there are several user accounts on each system. Even on a single user system there are multiple user accounts, because GNU/Linux uses unique accounts for some tasks. Users can be members of groups. Groups are used for more fine grained permissions, for example, you could make a file readable by a certain group. There are a few reserved users and groups on each system. The most important of these is the *root* account. The *root* user is the system administrator. It is a good idea to avoid logging in as root, because this greatly enlarges security risks. You can just log in as a normal user, and perform system administration tasks using the **su** and **sudo** commands.

The available user accounts are specified in the `/etc/passwd`. You can have a look at this file to get an idea of which user account are mandatory. As you will probably notice, there are no passwords in this file. Passwords are kept in the separate `/etc/shadow` file, as an encrypted string. Information about groups is stored in `/etc/group`. It is generally speaking not a good idea to edit these files directly. There are some excellent tools that can help you with user and group administration. This chapter will describe some of these tools.

14.2. Adding and removing users

useradd

The **useradd** is used to add user accounts to the system. Running **useradd** with a user name as parameter will create the user on the system. For example:

```
# useradd bob
```

Creates the user account *bob*. Please be aware that this does not create a home directory for the user. Add the *-m* parameter to create a home directory. For example:

```
# useradd -m bob
```

This would add the user *bob* to the system, and create the `/home/bob` home directory for this user. Normally the user is made a member of the *users* group. Suppose that we would like to make *crew* the primary group for the user *bob*. This can be done using the *-g* parameter. For example:

```
# useradd -g crew -m bob
```

It is also possible to add this user to secondary groups during the creation of the account with the *-G*. Group names can be separated with a comma. The following command would create the user *bob*, which is a member of the *crew* group, and the *www-admins* and *ftp-admins* secondary groups:

```
# useradd -g crew -G www-admins,ftp-admins -m bob
```

By default the **useradd** only adds users, it does not set a password for the added user. Passwords can be set using the **passwd** command.

passwd

As you probably guessed the **passwd** command is used to set a password for a user. Running this command as a user without a parameter will change the password for this user. The password command will ask for the old password,once and twice for the new password:

```
$ passwd
Changing password for bob
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

The *root* user can set passwords for users by specifying the user name as a parameter. The **passwd** command will only ask for the new password. For example:

```
# passwd bob
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

adduser

The **adduser** command combines **useradd** and **passwd** in an interactive script. It will ask you to fill in information about the account to-be created. After that it will create an account based on the information you provided. The screen listing below shows a sample session.

```
# adduser

Login name for new user []: john

User ID ('UID') [ defaults to next available ]: <Enter>

Initial group [ users ]: <Enter>

Additional groups (comma separated) []: staff

Home directory [ /home/john ] <Enter>

Shell [ /bin/bash ] <Enter>

Expiry date (YYYY-MM-DD) []: <Enter>
```

```
New account will be created as follows:

---------------------------------------
Login name.......:  john
UID..............:  [ Next available ]
Initial group....:  users
Additional groups: [ None ]
Home directory...:  /home/john
Shell............:  /bin/bash
Expiry date......:  [ Never ]

This is it... if you want to bail out, hit Control-C.  Otherwise, press
ENTER to go ahead and make the account.
```
<Enter>

```
Creating new account...


Changing the user information for john
Enter the new value, or press ENTER for the default
        Full Name []:
```
John Doe
```
        Room Number []:
```
<Enter>
```
        Work Phone []:
```
<Enter>
```
        Home Phone []:
```
<Enter>
```
        Other []:
```
<Enter>
```
Changing password for john
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
```
password
```
Re-enter new password:
```
password

```
Account setup complete.
```

You can use the default values, or leave some fields empty, by tapping the <Enter> key.

userdel

Sometimes it is necessary to remove a user account from the system. GNU/Linux offers the **userdel** tool to do this. Just specify the username as a parameter to remove that user from the system. For example, the following command will remove the user account *bob* from the system:

```
# userdel bob
```

This will only remove the user account, not the user's home directory and mail spool. Just add the `-r` parameter to delete the user's home directory and mail spool too. For example:

```
# userdel -r bob
```

14.3. Avoiding root usage with su

It is a good idea to avoid logging in as *root*. There are many reasons for not doing this. Accidentally typing a wrong command could cause bad things to happen, and malicious programs can make a lot of damage when you are logged in as *root*. Still, there are many situations in which you need to have root access. For example, to do system administration, or to install new software. Fortunately the **su** can give you temporal root privileges.

Using **su** is very simple. Just executing **su** will ask you for the root password, and will start a shell with root privileges after the password is correctly entered:

```
$ whoami
bob
$ su
Password:
# whoami
root
# exit
exit
$ whoami
bob
```

In this example the user *bob* is logged on, the **whoami** output reflects this. The user executes su and enters the *root* password. **su** launches a shell with root privileges, this is confirmed by the **whoami** output. After exiting the *root* shell, control is returned to the original running shell running with the privileges of the user *bob*.

It is also possible to execute just one command as the *root* user with the *-c* parameter. The following example will run **lilo**:

```
$ su -c lilo
```

If you want to give parameters to the command you would like to run, use quotes (e.g. **su -c "ls -l /"**). Without quotes **su** cannot determine whether the parameters should be used by the specified command, or by **su** itself.

Restricting su access

You can refine access to **su** with *suauth*. It is a good security practice to only allow members of a special group to **su** to *root*. For instance, you can restrict root **su**-ing in a BSD fashion to members of the *wheel* group by adding the following line to /etc/suauth:

```
root:ALL EXCEPT GROUP wheel:DENY
```

14.4. Disk quota

Introduction

Disk quota is a mechanism that allows the system administrator to restrict the number of disk blocks and inodes that a particular user and group can use. Not all filesystems supported by Linux support quota, widely used filesystems that support quota are ext2, ext3 and XFS. Quota are turned on and managed on a per filesystem basis.

Enabling quota

Quota can be enabled per filesystem in /etc/fstab, by using the *usrquota* and *grpquota* filesystem options. For example, suppose that we have the following entry for the /home partition in /etc/fstab:

```
/dev/hda8              /home              xfs              defaults              1    2
```

We can now enable user quota by adding the *usrquota* filesystem option:

```
/dev/hda8              /home              xfs              defaults,usrquota 1    2
```

At this point the machine can be rebooted, to let the Slackware Linux initialization scripts enable quota. You can also enable quota without rebooting the machine, by remounting the partition, and running the **quotaon** command:

```
# mount -o remount /home
# quotaon -avug
```

Editing quota

User and group quotas can be edited with the "edquota" utility. This program allows you to edit quotas interactively with the **vi** editor. The most basic syntax of this command is **edquota username**. For example:

```
# edquota joe
```

This will launch the **vi** editor with the quota information for the user *joe*. It will look like this:

```
Disk quotas for user joe (uid 1143):
  Filesystem                 blocks      soft      hard      inodes      soft      hard
  /dev/hda5                    2136         0         0          64         0         0
```

In this example quotas are only turned on for one file system, namely the filesystem on /dev/hda5. As you can see there are multiple columns. The *blocks* column shows how many block the user uses on the file system, and the *inodes* column the number of inodes a user occupies. Besides that there are *soft* and *hard* columns after both *blocks* and *inodes*. These columns specify the soft and hard limits on blocks and inodes. A user can exceed the soft limit for a grace period, but the user can never exceed the hard limit. If the value of a limit is *0*, there is no limit.

Note

The term "blocks" might be a bit confusing in this context. In the quota settings a block is 1KB, not the block size of the file system.

Let's look at a simple example. Suppose that we would like to set the soft limit for the user *joe* to *250000*, and the hard limit to *300000*. We could change the quotas listed above to:

```
Disk quotas for user joe (uid 1143):
  Filesystem                    blocks       soft       hard     inodes     soft     hard
  /dev/hda5                       2136     250000     300000         64        0        0
```

The new quota settings for this user will be active after saving the file, and quitting **vi**.

Getting information about quota

It is often useful to get statistics about the current quota usage. The **repquota** command can be used to get information about what quotas are set for every user, and how much of each quota is used. You can see the quota settings for a specific partition by giving the name of the partition as a parameter. The `-a` parameter will show quota information for all partitions with quota enabled. Suppose that you would like to see quota information for `/dev/hda5`, you can use the following command:

```
# repquota /dev/hda5
*** Report for user quotas on device /dev/hda5
Block grace time: 7days; Inode grace time: 7days
                        Block limits              File limits
User            used    soft    hard   grace   used  soft  hard  grace
----------------------------------------------------------------------
root      --       0       0       0              3     0     0
[..]
joe       --    2136  250000  300000             64     0     0
[..]
```

Chapter 15. Printer configuration

15.1. Introduction

GNU/Linux supports a large share of the available USB, parallel and network printers. Slackware Linux provides two printing systems, CUPS (Common UNIX Printing System) and LPRNG (LPR Next Generation). This chapter covers the CUPS system.

Independent of which printing system you are going to use, it is a good idea to install some printer filter collections. These can be found in the "ap" disk set. If you want to have support for most printers, make sure the following packages are installed:

```
a2ps
enscript
espgs
gimp-print
gnu-gs-fonts
hpijs
ifhp
```

Both printing systems have their own advantages and disadvantages. If you do not have much experience with configuring printers under GNU/Linux, it is a good idea to use CUPS, because CUPS provides a comfortable web interface which can be accessed through a web browser.

15.2. Preparations

To be able to use CUPS the "cups" package from the "a" disk set has to be installed. After the installation CUPS can be started automatically during each system boot by making `/etc/rc.d/rc.cups` executable. This can be done with the following command:

```
# chmod a+x /etc/rc.d/rc.cups
```

After restarting the system CUPS will also be restarted automatically. You can start CUPS on a running system by executing the following command:

```
# /etc/rc.d/rc.cups start
```

15.3. Configuration

CUPS can be configured via a web interface. The configuration interface can be accessed with a web browser at the following URL: http://localhost:631/. Some parts of the web interface require that you authenticate yourself. If an authentication window pops up you can enter "root" as the user name, and fill in the root account password.

A printer can be added to the CUPS configuration by clicking on "Administrate", and clicking on the "Add Printer" button after that. The web interface will ask for three options:

- *Name* - the name of the printer. Use a simple name, for example "epson".

- *Location* - the physical location of the printer. This setting is not crucial, but handy for larger organizations.

- *Description* - a description of the printer, for example "Epson Stylus Color C42UX".

You can proceed by clicking the "Continue" button. On the next page you can configure how the printer is connected. If you have an USB printer which is turned on, the web interface will show the name of the printer next to the USB port that is used. After configuring the printer port you can select the printer brand and model. After that the printer configuration is finished, and the printer will be added to the CUPS configuration.

An overview of the configured printers can be found on the "Printers" page. On this page you can also do some printer operations. For example, "Print Test Page" can be used to check the printer configuration by printing a test page.

15.4. Access control

The CUPS printing system provides a web configuration interface, and remote printer access through the Internet Printing Protocol (IPP). The CUPS configuration files allow you to configure fine-grained access control to printers. By default access to printers is limited to *localhost* (*127.0.0.1*).

You can refine access control in the central CUPS daemon configuration file, `/etc/cups/cupsd.conf`, which has a syntax that is comparable to the Apache configuration file. Access control is configured through *Location* sections. For example, the default global (IPP root directory) section looks like this:

```
<Location />
Order Deny,Allow
Deny From All
Allow From 127.0.0.1
</Location>
```

As you can see deny statements are handled first, and then allow statements. In the default configuration access is denied from all hosts, except for *127.0.0.1*, *localhost*. Suppose that you would like to allow hosts from the local network, which uses the *192.168.1.0/24* address space, to use the printers on the system you are configuring CUPS on. In this case you could add the line that is bold:

```
<Location />
Order Deny,Allow
Deny From All
Allow From 127.0.0.1
Allow From 192.168.1.0/24
</Location>
```

You can refine other locations in the address space by adding additional location sections. Settings for sub-directories override global settings. For example, you could restrict access to the *epson* printer to the hosts with IP addresses *127.0.0.1* and *192.168.1.203* by adding the following section:

```
<Location /printers/epson>
Order Deny,Allow
Deny From All
Allow From 127.0.0.1
```

```
Allow From 192.168.1.203
</Location>
```

15.5. Ghostscript paper size

Ghostscript is a PostStript and Portable Document Format (PDF) interpreter. Both PostScript and PDF are languages that describe data that can be printed. Ghostscript is used to convert PostScript and PDF to raster formats that can be displayed on the screen or printed. Most UNIX programs output PostScript, the CUPS spooler uses GhostScript to convert this PostScript to rasterized format that a particular printer understands.

There are some Ghostscript configuration settings that may be useful to change in some situations. This section describes how you can change the default paper size that Ghostscript uses.

Note

Some higher-end printers can directly interpret PostScript. Rasterization is not needed for these printers.

By default Ghostscript uses US letter paper as the default paper size. The paper size is configured in `/usr/share/ghostscript/x.xx/lib/gs_init.ps`, in which *x.xx* should be replaced by the Ghostscript version number. Not far after the beginning of the file there are two lines that are commented out with a percent (%) sign, that look like this:

```
% Optionally choose a default paper size other than U.S. letter.
% (a4) /PAPERSIZE where { pop pop } { /PAPERSIZE exch def } ifelse
```

You can change the Ghostscript configuration to use A4 as the default paper size by removing the percent sign and space that are at the start of the second line, changing it to:

```
% Optionally choose a default paper size other than U.S. letter.
(a4) /PAPERSIZE where { pop pop } { /PAPERSIZE exch def } ifelse
```

It is also possible to use another paper size than Letter or A4 by replacing *a4* in the example above with the paper size you want to use. For example, you could set the default paper size to US Legal with:

```
% Optionally choose a default paper size other than U.S. letter.
(legal) /PAPERSIZE where { pop pop } { /PAPERSIZE exch def } ifelse
```

It is also possible to set the paper size per invocation of Ghostscript by using the *-sPAPERSIZE=size* parameter of the **gs** command. For example, you could use add *-sPAPERSIZE=a4* parameter when you start **gs** to use A4 as the paper size for an invocation of Ghostscript.

An overview of supported paper sizes can be found in the `gs_statd.ps`, that can be found in the same directory as `gs_init.ps`.

Chapter 16. X11

16.1. X Configuration

The X11 configuration is stored in `/etc/X11/xorg.conf`. Many distributions provide special configuration tools for X, but Slackware Linux only provides the standard X11 tools (which are actually quite easy to use). In most cases X can be configured automatically, but sometimes it is necessary to edit `/etc/X11/xorg.conf` manually.

Automatic configuration

The X11 server provides an option to automatically generate a configuration file. X11 will load all available driver modules, and will try to detect the hardware, and generate a configuration file. Execute the following command to generate a `xorg.conf` configuration file:

```
$ X -configure
```

If X does not output any errors, the generated configuration can be copied to the `/etc/X11` directory. And X can be started to test the configuration:

```
$ cp /root/xorg.conf /etc/X11/
$ startx
```

Interactive configuration

X11 provides two tools for configuring X interactively, **xorgcfg** and **xorgconfig**. **xorgcfg** tries to detect the video card automatically, and starts a tool which can be used to tune the configuration. Sometimes **xorgcfg** switches to a video mode which is not supported by the monitor. In that case **xorgcfg** can also be used in text-mode, by starting it with **xorgcfg -textmode**.

xorgconfig differs from the tools described above, it does not detect hardware and will ask detailed questions about your hardware. If you only have little experience configuring X11 it is a good idea to avoid **xorgconfig**.

16.2. Window manager

The "look and feel" of X11 is managed by a so-called window manager. Slackware Linux provides the following widely-used window managers:

- WindowMaker: A relatively light window manager, which is part of the GNUStep project.

- BlackBox: Light window manager, BlackBox has no dependencies except the X11 libraries.

- KDE: A complete desktop environment, including browser, e-mail program and an office suite (KOffice).

- Xfce: A lightweight desktop environment. This is an ideal environment if you would like to have a userfriendly desktop that runs on less capable machines.

If you are used to a desktop environment, using KDE or Xfce is a logical choice. But it is a good idea to try some of the lighter window managers. They are faster, and consume less memory, besides that most KDE and Xfce applications are perfectly usable under other window managers.

On Slackware Linux the **xwmconfig** command can be used to set the default window manager. This program shows the installed window managers, from which you can choose one. You can set the window manager globally by executing **xwmconfig** as root.

Chapter 17. Package Management

17.1. Pkgtools

Introduction

Slackware Linux does not use a complex package system, unlike many other Linux distributions. Package have the `.tgz` extension, and are usually ordinary tarballs which contain two extra files: an installation script and a package description file. Due to the simplicity of the packages the Slackware Linux package tools do not have the means to handle dependencies. But many Slackware Linux users prefer this approach, because dependencies often cause more problems than they solve.

Slackware Linux has a few tools to handle packages. The most important tools will be covered in this chapter. To learn to understand the tools we need to have a look at package naming. Let's have a look at an example, imagine that we have a package with the file name `bash-2.05b-i386-2.tgz`. In this case the name of the package is bash-2.05b-i386-2. In the package name information about the package is separated by the '-' character. A package name has the following meaning: *programname-version-architecture-packagerevision*

pkgtool

The **pkgtool** command provides a menu interface for some package operations. De most important menu items are *Remove* and *Setup*. The *Remove* option presents a list of installed packages. You can select which packages you want to remove with the space bar and confirm your choices with the return key. You can also deselect a package for removal with the space bar.

The *Setup* option provides access to a few tools which can help you with configuring your system, for example: **netconfig**, **pppconfig** and **xwmconfig**.

installpkg

The **installpkg** command is used to install packages. **installpkg** needs a package file as a parameter. For example, if you want to install the package `bash-2.05b-i386-2.tgz` execute:

```
# installpkg bash-2.05b-i386-2.tgz
```

upgradepkg

upgradepkg can be used to upgrade packages. In contrast to **installpkg** it only installs packages when there is an older version available on the system. The command syntax is comparable to **installpkg**. For example, if you want to upgrade packages using package in a directory execute:

```
# upgradepkg *.tgz
```

As said only those packages will be installed of which an other version is already installed on the system.

removepkg

The **removepkg** can be used to remove installed packages. For example, if you want to remove the "bash" package (it is not recommended to do that!), you can execute:

```
# removepkg bash
```

As you can see only the name of the package is specified in this example. You can also remove a package by specifying its full name:

```
# removepkg bash-2.05b-i386-2
```

17.2. Slackpkg

Introduction

Slackpkg is a package tool written by Roberto F. Batista and Evaldo Gardenali. It helps users to install and upgrade Slackware Linux packages using one of the Slackware Linux mirrors. Slackpkg is included in the `extra/` directory on the second CD of the Slackware Linux CD set.

Configuration

Slackpkg is configured through some files in `/etc/slackpkg`. The first thing you should do is configuring which mirror slackpkg should use. This can be done by editing the `/etc/slackpkg/mirrors`. This file already contains a list of mirrors, you can just uncomment a mirror close to you. For example:

```
ftp://ftp.nluug.nl/pub/os/Linux/distr/slackware/slackware-12.0/
```

This will use the Slackware Linux 12.0 tree on the ftp.nluug.nl mirror. Be sure to use a tree that matches your Slackware Linux version. If you would like to track slackware-current you would uncomment the following line instead (when you would like to use the NLUUG mirror):

```
ftp://ftp.nluug.nl/pub/os/Linux/distr/slackware/slackware-current/
```

Slackpkg will only accept one mirror. Commenting out more mirrors will not work.

Importing the Slackware Linux GPG key

By default slackpkg checks packages using the package signatures and the public Slackware Linux GPG key. Since this is a good idea from a security point of view, you probably do not want to change this behaviour. To be able to verify packages you have to import the *security@slackware.com* GPG key. If you have not used GPG before you have to create the GPG directory in the home directory of the *root* user:

```
# mkdir ~/.gnupg
```

The next step is to search for the public key of *security@slackware.com*. We will do this by querying the *pgp.mit.edu* server:

```
# gpg --keyserver pgp.mit.edu --search security@slackware.com
gpg: keyring `/root/.gnupg/secring.gpg' created
gpg: keyring `/root/.gnupg/pubring.gpg' created
gpg: searching for "security@slackware.com" from HKP server pgp.mit.edu
Keys 1-2 of 2 for "security@slackware.com"
(1)     Slackware Linux Project <security@slackware.com>
          1024 bit DSA key 40102233, created 2003-02-25
(2)     Slackware Linux Project <security@slackware.com>
          1024 bit DSA key 40102233, created 2003-02-25
Enter number(s), N)ext, or Q)uit >
```

As you can see we have got two (identical) hits. Select the first one by entering "1". GnuPG will import this key in the keyring of the *root* user:

```
Enter number(s), N)ext, or Q)uit > 1
gpg: key 40102233: duplicated user ID detected - merged
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 40102233: public key "Slackware Linux Project <security@slackware.com>" import
gpg: Total number processed: 1
gpg:               imported: 1
```

Be sure to double check the key you received. The key ID and fingerprint of this particular key can be found on the Internet on many trustworthy sites. The key ID is, as mentioned above *40102233*. You can get the key fingerprint with the *--fingerprint* parameter:

```
# gpg --fingerprint security@slackware.com
pub  1024D/40102233 2003-02-26 Slackware Linux Project <security@slackware.com>
     Key fingerprint = EC56 49DA 401E 22AB FA67  36EF 6A44 63C0 4010 2233
sub  1024g/4E523569 2003-02-26 [expires: 2012-12-21]
```

Once you have imported and checked this key you can start to use slackpkg, and install packages securely.

Updating the package lists

Before upgrading and installing packages you have to let slackpkg download the package lists from the mirror you are using. It is a good idea to do this regularly to keep these lists up to date. The latest package lists can be fetched with:

```
$ slackpkg update
```

Upgrading packages

The *upgrade* parameter is used to upgrade installed packages. You have to add an extra parameter to actually tell **slackpkg** what you want to upgrade, this differs for a stable Slackware Linux version and slackware-current. Upgrades for stable Slackware Linux releases are in the patches directory of FTP mirrors. You can update a slackware-stable installation (e.g. Slackware Linux 12.0) with:

```
# slackpkg upgrade patches
```

In this case **slackpkg** will use the packages from the `patches` directory. In slackware-current updated packages are put in the normal `slackware` package sub-directories. So, we can pass that as an parameter to **slackpkg upgrade**:

```
# slackpkg upgrade slackware
```

You can also upgrade individual packages by specifying the name of the package to be upgraded, for example:

```
# slackpkg upgrade pine
```

Installing packages

The `install` is used to install packages:

```
# slackpkg install rexima
```

Be aware that neither slackpkg, nor the Slackware Linux package tools do dependency checking. If some program does not work due to missing libraries, you have to add them yourself with slackpkg.

17.3. Getting updates through rsync

Another popular method of keeping Slackware Linux up to date is by keeping a local mirror. The ideal way of doing this is via rsync. rsync is a program that can synchronize two trees of files. The advantage is that rsync only transfers the differences in files, making it very fast. After syncing with a mirror you can upgrade Slackware Linux with **upgradepkg**, or make a new installation CD. The following example synchronizes a local current tree with an up-to-date tree from on a mirror:

```
# rsync -av --delete \
--exclude=slackware/kde \
--exclude=slackware/kdei \
--exclude=slackware/gnome \
--exclude=bootdisks \
--exclude=extra \
--exclude=testing \
--exclude=pasture \
--exclude=rootdisks \
--exclude=source \
--exclude=zipslack \
rsync://fill-in-mirror/pub/slackware/slackware-current/ \
/usr/share/mirrors/slackware-current
```

The `-a` parameter implies a few other options that try to make a copy that is as exact as possible (in terms of preserving symlinks, permissions and owners). The `--delete` deletes files that are not available on the mirror anymore. It is good idea to use this parameter, because otherwise your tree may get bloated very quickly with older package versions. With the `--exclude` parameter you can specify which files or directories should be ignored.

After syncing the tree you can use **upgradepkg** to update your Slackware Linux installation. For example:

```
# upgradepkg /usr/share/mirrors/slackware-current/slackware/*/*.tgz
```

17.4. Tagfiles

Introduction

Tagfiles are a relatively unknown feature of Slackware Linux. A tagfile is a file that can be used to instruct **installpkg** what packages should be installed from a collection of packages. For instance, the Slackware Linux installer generates a tagfile during the *Expert* and *Menu* installation methods to store which packages should be installed during the installation process.

The nice aspect of tagfiles is that you can easily create tagfiles yourself. By writing your own tagfiles you can automate the package installation, which is ideal for larger client or server roll-outs (or smaller set-ups if it gives you more comfort than installing packages manually). The easiest way to create your own tagfiles is by starting out with the tagfiles that are part of the official Slackware Linux distribution. In the following sections we are going to look at how this is done.

Creating tagfiles

Tagfiles are simple plain-text files. Each line consists of a package name and a flag, these two elements are separated by a colon and a space. The flag specifies what should be done with a package. The fields are described in Table 17.1, "Tagfile fields". Let's look at a few lines from the tagfile in the "a" disk set:

```
aaa_base: ADD
aaa_elflibs: ADD
acpid: REC
apmd: REC
bash: ADD
bin: ADD
```

It should be noted that you can also add comments to tagfiles with the usual comment (#) character. As you can see in the snippet above there are different flags. The table listed below describes the four different flags.

Table 17.1. Tagfile fields

Flag	Meaning
ADD	A package marked by this flag will automatically be installed
SKP	A package marked by this flag will automatically be skipped
REC	Ask the user what to do, recommend installation of the package.
OPT	Ask the user what to do, the package will be described as optional.

As you can read from the table **installpkg** will only act automatically when either *ADD* or *SKP* is used.

It would be a bit tedious to write a tagfile for each Slackware Linux disk set. The official Slackware Linux distribution contains a tagfile in the directory for each disk set. You can use these tagfiles as a start. The short script listed below can be used to copy the tagfiles to the current directory, preserving the disk set structure.

```
#!/bin/sh

if [ ! $# -eq 1 ] ; then
  echo "Syntax: $0 [directory]"
  exit
fi

for tagfile in $1/*/tagfile; do
  setdir=`echo ${tagfile} | egrep -o '\w+/tagfile$' | xargs dirname`
  mkdir ${setdir}
  cp ${tagfile} ${setdir}/tagfile.org
  cp ${tagfile} ${setdir}
done
```

After writing the script to a file you can execute it, and specify a `slackware/` directory that provides the disk sets. For example:

```
$ sh copy-tagfiles.sh /mnt/flux/slackware-current/slackware
```

After doing this the current directory will contain a directory structure like this, in which you can edit the individual tag files:

```
a/tagfile
a/tagfile.org
ap/tagfile
ap/tagfile.org
d/tagfile
d/tagfile.org
e/tagfile
e/tagfile.org
[...]
```

The files that end with *.org* are backups, that you can use as a reference while editing tagfiles. Besides that they are also used in the script that is described in the next section.

Automatically generating tagfiles

With a simple script, it is also possible to build tagfiles based on the packages that are installed on the current system. I owe thanks to Peter Kaagman for coming up with this nifty idea!

First build a tagfile directory from the Slackware Linux installation media, as described in the previous section. When you have done that, you can create the following script:

```
#!/bin/sh

if [ ! $# -eq 1 ] ; then
  echo "Syntax: $0 [directory]"
  exit
fi

for tforg in $1/*/tagfile.org ; do
  tf=${tforg%.org}
  rm -f ${tf}
  for package in $(grep -v '^#' ${tforg} | cut -d ':' -f 1) ; do
    if ls /var/log/packages/${package}-[0-9]* &> /dev/null ; then
      echo "${package}: ADD" >> ${tf}
    else
      echo "${package}: SKP" >> ${tf}
    fi
  done
done
```

Suppose that you have saved it as `build-tagfiles.sh`, you can use it by specifying directory that holds the tagfiles as the first argument:

```
$ sh build-tagfiles.sh .
```

The script will mark packages that are installed as *ADD*, and packages that are not installed as *SKP*.

Using tagfiles

On an installed system you can let **installpkg** use a tagfile with the `-tagfile` parameter. For example:

```
# installpkg -infobox -root /mnt-small -tagfile a/tagfile /mnt/flux/slackware-current/s
```

Note

You have to use the `-infobox` option, otherwise the tagfiles will not be used.

Of course, tagfiles would be useless if they cannot be used during the installation of Slackware Linux. This is certainly possible: after selecting which disk sets you want to install you can choose in what way you want to select packages (the dialog is named *SELECT PROMPTING MODE*. Select *tagpath* from this menu. You will then be asked to enter the path to the directory structure with the tagfiles. The usual way to provide tagfiles to the Slackware Linux installation is to put them on a floppy or another medium, and mounting this before or during the installation. E.g. you can switch to the second console with by pressing the <ALT> and <F2> keys, and create a mount point and mount the disk with the tagfiles:

```
# mkdir /mnt-tagfiles
# mount /dev/fd0 /mnt/mnt-tagfiles
```

After mounting the disk you can return to the virtual console on which you run **setup**, by pressing <ALT> and <F1>.

Chapter 18. Building a kernel

18.1. Introduction

The Linux kernel is shortly discussed in Section 2.1, "What is Linux?". One of the advantages of Linux is that the full sources are available (as most of the Slackware Linux system). This means that you can recompile the kernel. There are many situations in which recompiling the kernel is useful. For example:

- **Making the kernel leaner**: One can disable certain functionality of the kernel, to decrease its size. This is especially useful in environments where memory is scarce.

- **Optimizing the kernel**: it is possible to optimize the kernel. For instance, by compiling it for a specific processor type.

- **Hardware support**: Support for some hardware is not enabled by default in the Linux kernel provided by Slackware Linux. A common example is support for SMP systems.

- **Using custom patches**: There are many unofficial patches for the Linux kernel. Generally speaking it is a good idea to avoid unofficial patches. But some third party software, like Win4Lin [http://www.netraverse.com], require that you install an additional kernel patch.

- Making the proper headers and build infrastucture available to build third-party modules.

This chapter focuses on the default kernel series used in Slackware Linux 12.0, Linux 2.6. Compiling a kernel is not really difficult, just keep around a backup kernel that you can use when something goes wrong. Linux compilation involves these steps:

- Configuring the kernel.

- Building the kernel.

- Building modules.

- Installing the kernel and modules.

- Updating the LILO configuration.

In this chapter, we suppose that the kernel sources are available in `/usr/src/linux`. If you have installed the kernel sources from the "k" disk set, the kernel sources are available in `/usr/src/linux-kernelversion`, and `/usr/src/linux` is a symbolic link to the real kernel source directory. So, if you use the standards Slackware Linux kernel package you are set to go.

In contrast to older kernel versions, it is not necessary to use a `/usr/src/linux` symbolink anymore. If you have extracted newer kernel sources in `/usr/src`, you can build the kernel in `/usr/src/linux-<version>`, and use that directory in the examples in this chapter.

18.2. Configuration

As laid out above, the first step is to configure the kernel source. To ease the configuration of the kernel, it is a good idea to copy the default Slackware Linux kernel configuration to the kernel sources. The Slackware Linux kernel configuration files are stored on the distribution medium as `kernels/<kernelname>/config`. Suppose that you want to use the `hugesmp.s` kernel configuration as a starting point (which is the default kernel), and that you have a Slackware Linux CD-ROM mounted at `/mnt/cdrom`, you can copy the Slackware Linux kernel configuration with:

`cp /mnt/cdrom/kernels/hugesmp.s/config /usr/src/linux/.config`

The kernel configuration of a running kernel can also be retrieved as `/proc/config.gz` if the kernel was compiled with the *CONFIG_IKCONFIG* and *CONFIG_IKCONFIG_PROC* options. The default Slackware Linux kernels have these options enabled. So, if you would like to use the configuration of the running kernel, you could run:

`zcat /proc/config.gz > /usr/src/linux/.config`

If you are using a configuration file that is for another kernel version than you are currently compiling, it is likely that both kernel versions do not have the same set of options. New options are often added (e.g., because newer drivers are added), and sometimes kernel components are removed. You can configure new options (and remove unused options) with the **make oldconfig** command:

`cd /usr/src/linux ; make oldconfig`

This will ask you for options whether you would like to compile in support (*Y*), compile support as a module (*M*), or not include support (*N*). For example:

```
IBM ThinkPad Laptop Extras (ACPI_IBM) [N/m/y/?] (NEW)
```

As you can see, the possible options are shown, with the default choice as a capital. If you just press <Enter>, the capitalized option will be used. If you want more information about an option, you can enter the question mark (*?*):

```
    IBM ThinkPad Laptop Extras (ACPI_IBM) [N/m/y/?] (NEW) ?

This is a Linux ACPI driver for the IBM ThinkPad laptops. It adds
support for Fn-Fx key combinations, Bluetooth control, video
output switching, ThinkLight control, UltraBay eject and more.
For more information about this driver see <file:Documentation/ibm-acpi.txt>
and <http://ibm-acpi.sf.net/> .

If you have an IBM ThinkPad laptop, say Y or M here.

    IBM ThinkPad Laptop Extras (ACPI_IBM) [N/m/y/?] (NEW)
```

The output of this command can be a bit verbose, because the options that were used in the configuration file, and are available in the running kernel are also shown, but their configuration will be filled in automatically based on the configuration file.

At this point you can start to actually configure the kernel in detail. There are three configuration front-ends to the kernel configuration. The first one is *config*, which just asks you what you want to do for each kernel option. This takes a lot of time. So, normally this is not a good way to configure the kernel. A more user friendly approach is the *menuconfig* front-end, which uses a menuing system that you can use to configure the kernel. There is an X front-end as well, named *xconfig*. You can start a configuration front-end by changing to the kernel source directory, and executing **make <front-end>**. For example, to configure the kernel with the menu front-end you can use the following commands:

```
# cd /usr/src/linux
# make menuconfig
```

Of course, if you prefer, you can also edit the .config file directly with your favorite text editor.

As we have seen briefly before, in the kernel configuration there are basically three options for each choice: "n" disables functionality, "y" enables functionality, and "m" compiles the functionality as a module. The default Slackware Linux kernel configuration is a very good configuration, it includes the support for most common disk controllers and filesystems, the rest is compiled as a module. Whatever choices you make, you need to make sure that both the driver for the disk controller and support for the filesystem type holding your root filesystem is included. When they are not, the kernel will not able to mount the root filesystem, and the kernel will panic because it can not hand over initialization to the **init** program.

Note

It is always a good idea to keep your old kernel in modules around, in the case that you have made a configuration error. If the to be compiled kernel has the same version number as the running kernel, you should seriously consider modifying the *CONFIG_LOCALVERSION* option. The string specified in this option is appended to the version name. For example, if the kernel has version 2.6.21.6, and *CONFIG_LOCALVERSION* is set to *"-smp-ddk"*, the kernel version will be *2.6.21.6-smp-ddk*.

If you don't modify the version in this manner, the installation of modules from the new kernel will overwrite the modules from the running kernel. This is highly uncomfortable if you need to fall back to the old kernel.

18.3. Compilation

The kernel compilation used to consist of multiple steps, but 2.6 Linux kernels can be compiled be executing **make** in the kernel source directory. This will calculate dependencies, build the kernel, and will build and link kernel modules.

```
# cd /usr/src/linux
# make
```

After the compilation is finished, the tree contains the compiled modules, and a compressed kernel image named bzImage in /usr/src/linux/arch/i386/boot. You can now continue with the installation of the kernel and its modules.

18.4. Installation

Installing the kernel

The next step is to install the kernel and the kernel modules. We will start with installing the kernel modules, because this can be done with one command within the kernel source tree:

```
# make modules_install
```

This will install the modules in /lib/modules/<kernelversion>. If you are replacing a kernel with the exactly the same version number, it is a good idea to remove the old modules before installing the new ones. E.g.:

```
# rm -rf /lib/modules/2.6.21.5-smp
```

You can "install" the kernel by copying it to the /boot directory. You can give it any name you want, but it is a good idea to use some naming convention. E.g. vmlinuz-version. For instance, if you name it vmlinuz-2.6.21.5-smp-ddk, you can copy it from within the kernel source tree with:

```
# cp arch/i386/boot/bzImage /boot/vmlinuz-2.6.21.5-smp-ddk
```

At this point you are almost finished. The last step is to add the new kernel to the Linux boot loader (LILO) configuration.

Configuring LILO

LILO (Linux Loader) is the default boot loader that Slackware Linux uses. The configuration of LILO works in two steps; the first step is to alter the LILO configuration in /etc/lilo.conf. The second step is to run the **lilo**, which will write the LILO configuration to the boot loader. The LILO configuration already has an entry for the current kernel you are running. It is a good idea to keep this entry, as a fall-back option if your new kernel does not work. If you scroll down to the bottom of /etc/lilo.conf you will see this entry, it looks comparable to this:

```
# Linux bootable partition config begins
image = /boot/vmlinuz
  root = /dev/hda5
  label = Slack
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
# Linux bootable partition config ends
```

The easiest way to add the new kernel is to duplicate the existing entry, and then editing the first entry, changing the *image*, and *label* options. After changing the example above it would look like this:

```
# Linux bootable partition config begins
image = /boot/vmlinuz-2.6.21.5-smp-ddk
  root = /dev/hda5
  label = Slack
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking

image = /boot/vmlinuz
  root = /dev/hda5
  label = SlackOld
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
# Linux bootable partition config ends
```

As you can see the first *image* entry points to the new kernel, and the label of the second entry was changed to "SlackOld". LILO will automatically boot the first image. You can now install this new LILO configuration with the **lilo** command:

```
# lilo
Added Slack *
```

```
Added SlackOld
```

The next time you boot both entries will be available, and the "Slack" entry will be booted by default.

Note

If you want LILO to show a menu with the entries configured via `lilo.conf` on each boot, make sure that you add a line that says

```
prompt
```

to `lilo.conf`. Otherwise LILO will boot the default entry that is set with *default=<name>*, or the first entry when no default kernel is set. You can access the menu with entries at any time by holding the <Shift> key when LILO is started.

Chapter 19. System initialization

This chapter describes the initialization of Slackware Linux. Along the way various configuration files that are used to manipulate the initialization process are described.

19.1. The bootloader

Arguably the most important piece of an operating system is the kernel. The kernel manages hardware resources and software processes. The kernel is started by some tiny glue between the system BIOS (Basic Input/Output System) and the kernel, called the bootloader. The bootloader handles the complications that come with loading a specific (or less specific) kernel.

Most bootloader actually work in two stages. The first stage loader loads the second stage loader, that does the real work. The boot loader is divided in two stages on x86 machines, because the BIOS only loads one sector (the so-called boot sector) that is 512 bytes in size.

Slackware Linux uses the LILO (LInux LOader) boot loader. This bootloader has been in development since 1992, and is specifically written to load the Linux kernel. Lately LILO has been replaced by the GRUB (GRand Unified Bootloader) in most GNU/Linux distributions. GRUB is available as an extra package on the Slackware Linux distribution media.

LILO configuration

LILO is configured through the `/etc/lilo.conf` configuration file. Slackware Linux provides an easy tool to configure LILO. This configuration tool can be started with the **liloconfig** command, and is described in the installation chapter (Section 5.3, "Installing Slackware Linux").

Manual configuration of LILO is pretty simple. The LILO configuration file usually starts off with some global settings:

```
# Start LILO global section
boot = /dev/sda ❶
#compact          # faster, but won't work on all systems.
prompt ❷
timeout = 50 ❸
# Normal VGA console
vga = normal ❹
```

❶   The *boot* option specifies where the LILO bootloader should be installed. If you want to use LILO as the main bootloader for starting Linux and/or other operating systems, it is a good idea to install LILO in the MBR (Master Boot Record) of the hard disk that you use to boot the system. LILO is installed to the MBR by omitting the partition number, for instance `/dev/hda` or `/dev/sda`. If you want to install LILO to a specific partition, add a partition number, like `/dev/sda1`. Make sure that you have another bootloader in the MBR, or that the partition is made active using **fdisk**. Otherwise you may end up with an unbootable system.

         Be cautious if you use partitions with a XFS filesystem! Writing LILO to an XFS partition will overwrite a part of the filesystem. If you use an XFS root (`/`) filesystem, create a non-XFS `/boot` filesystem to which you install LILO, or install LILO to the MBR.

❷   The *prompt* option will set LILO to show a boot menu. From this menu you can select which kernel or operating system should be booted. If you do not have this option enabled, you can still access the bootloader menu by holding the <Shift> key when the bootloader is started.

❸   The *timeout* value specifies how long LILO should wait before the default kernel or OS is booted. The time is specified in tenths of a second, so in the example above LILO will wait 5 seconds before it proceeds with the boot.

❹   You can specify which video mode the kernel should use with the *vga* option. When this is set to *normal* the kernel will use the normal 80x25 text mode.

The global options are followed by sections that add Linux kernels or other operating systems. Most Linux kernel sections look like this:

```
image = /boot/vmlinuz ❶
  root = /dev/sda5 ❷
  label = Slack ❸
  read-only ❹
```

❶   The *image* option specifies the kernel image that should be loaded for this LILO item.

❷   The *root* parameter is passed to the kernel, and will be used by the kernel as the root (/) filesystem.

❸   The *label* text is used as the label for this entry in the LILO boot menu.

❹   *read-only* specifies that the root filesystem should be mounted read-only. The filesystem has to be mounted in read-only state to conduct a filesystem check.

LILO installation

LILO does not read the /etc/lilo.conf file during the second stage. So, you will have to write changes to the second stage loader when you have changed the LILO configuration. This is also necessary if you install a new kernel with the same filename, since the position of the kernel on the disk may have changed. Reinstalling LILO can simply be done with the **lilo** command:

```
# lilo
Added Slack26 *
Added Slack
```

19.2. init

After the kernel is loaded and started, the kernel will start the **init** command. **init** is the parent of all processes, and takes care of starting the system initialization scripts, and spawning login consoles through **agetty**. The behavior of **init** is configured in /etc/inittab.

The /etc/inittab file is documented fairly well. It specifies what scripts the system should run for different runlevels. A runlevel is a state the system is running in. For instance, runlevel 1 is single user mode, and runlevel 3 is multi-user mode. We will have a short look at a line from /etc/inittab to see how it works:

```
rc:2345:wait:/etc/rc.d/rc.M
```

This line specifies that **/etc/rc.d/rc.M** should be started when the system switches to runlevel 2, 3, 4 or 5. The only line you probably ever have to touch is the default runlevel:

```
id:3:initdefault:
```

In this example the default runlevel is set to *3* (multiuser mode). You can set this to another runlevel by replacing 3 with the new default runlevel. Runlevel 4 can particularly be interesting on desktop machines, since Slackware Linux will try to start the GDM, KDM or XDM display manager (in this particular order). These display managers provide a graphical login, and are respectively part of GNOME, KDE and X11.

Another interesting section are the lines that specify what command should handle a console. For instance:

```
c1:1235:respawn:/sbin/agetty 38400 tty1 linux
```

This line specifies that **agetty** should be started on *tty1* (the first virtual terminal) in runlevels 1, 2, 3 and 5. The **agetty** command opens the tty port, and prompts for a login name. **agetty** will then spawn **login** to handle the login. As you can see from the entries, Slackware Linux only starts one console in runlevel 6, namely *tty6*. One might ask what happened to *tty0*, *tty0* certainly exists, and represents the active console.

Since */etc/inittab* is the right place to spawn **agetty** instances to listen for logins, you can also let one or more agetties listen to a serial port. This is especially handy when you have one or more terminals connected to a machine. You can add something like the following line to start an **agetty** instance that listens on COM1:

```
s1:12345:respawn:/sbin/agetty -L ttyS0 9600 vt100
```

19.3. Initialization scripts

As explained in the **init** (Section 19.2, "init") section, **init** starts some scripts that handle different runlevels. These scripts perform jobs and change settings that are necessary for a particular runlevel, but they may also start other scripts. Let's look at an example from `/etc/rc.d/rc.M`, the script that **init** executes when the system switches to a multi-user runlevel:

```
# Start the sendmail daemon:
if [ -x /etc/rc.d/rc.sendmail ]; then
  . /etc/rc.d/rc.sendmail start
fi
```

These lines say "execute **/etc/rc.d/rc.sendmail start** if `/etc/rc.d/rc.sendmail` is executable". This indicates the simplicity of the Slackware Linux initialization scripts. Different functionality, for instance network services, can be turned on or off, by twiddling the executable flag on their initialization script. If the initialization script is executable, the service will be started, otherwise it will not. Setting file flags is described in the section called "Changing file permission bits", but we will have a look at a quick example how you can enable and disable sendmail.

To start sendmail when the system initializes, execute:

```
# chmod +x /etc/rc.d/rc.sendmail
```

To disable starting of sendmail when the system initializes, execute:

```
# chmod -x /etc/rc.d/rc.sendmail
```

Most service-specific initialization scripts accept three parameters to change the state of the service: *start*, *restart* and *stop*. These parameters are pretty much self descriptive. For example, if you would like to restart sendmail, you could execute:

```
# /etc/rc.d/rc.sendmail restart
```

If the script is not executable, you have to tell the shell that you would like to execute the file with **sh**. For example:

```
# sh /etc/rc.d/rc.sendmail start
```

19.4. Hotplugging and device node management

Slackware Linux has supported hotplugging since Slackware Linux 9.1. When enabled, the kernel passes notifications about device events to a userspace command. Since Slackware Linux 11.0, the **udev** set of utilities handle these notifications. **udev** manages the dynamic `/dev` directory as well.

The mode of operation of **udev** for handling hotplugging of devices is fairly simple. When a device is added to the system, the kernel notifies userspace hotplug event listeners. **udev** will receive the notification of the device being added, and looks whether there are any module mappings for the device. If there are, the appropriate device driver module for the device is automatically loaded. **udev** will remove the module when it is notified of a device removal, and no devices use the loaded module anymore.

The udev subsystem is initialized in `/etc/rc.d/rc.S` by executing **/etc/rc.d/rc.udev start**. As with most functionality, you can enable or disable udev by twiddling the executable flag of the `/etc/rc.d/rc.udev` script (see Section 19.3, "Initialization scripts").

If udev automatically loads modules that you do not want to load, you can add a *blacklist* in your **modprobe** configuration in `/etc/modprobe.d/blacklist`. For example, if you would like to prevent loading of the *8139cp* module, you can add the following line (actually, this module is already blacklisted in Slackware Linux):

```
blacklist 8139cp
```

19.5. Device firmware

Introduction

Some hardware requires the system to upload firmware. Firmware is a piece of software that is used to control the hardware. Traditionally, the firmware was stored permanently in ROM (read-only memory) or non-volatile media like flash memory. However, many new devices use volatile memory to store firmware, meaning that the firmware needs to be reloaded to the device memory when the system is restarted.

Drivers for devices that require firmware have a table of the firmware files that it needs. For each firmware file that the driver needs, it will issue a firmware addition event. If udev handles hotplugging events, it will try to handle that event. The udev rules contain an entry for firmware addition events in `/etc/udev/rules.d/50-udev.rules`:

```
# firmware loader
SUBSYSTEM=="firmware", ACTION=="add", RUN+="/lib/udev/firmware.sh"
```

This means that upon firmware addition events, the `/lib/udev/firmware.sh` script should be executed. This script searches the `/lib/firmware` and `/usr/local/lib/firmware` directories for the requested firmware. If the firmware file exists, it will be loaded by copying the contents of the file to a special sysfs node.

Adding firmware

As described in the previous section, some hardware requires firmware to be uploaded to hardware by the operating system. If this is the case, and no firmware is installed, the kernel will emit an error message when the driver for that hardware is loaded. You can see the kernel output with the **dmesg** command or in the `/var/log/messages` log file. Such error message explicitly says that the firmware could not be loaded, for example:

```
ipw2100: eth1: Firmware 'ipw2100-1.3.fw' not available or load failed.
ipw2100: eth1: ipw2100_get_firmware failed: -2
```

In this case you will have to find the firmware for your device. This can usually be found by searching the web for the chipset or driver for the device (in this case *ipw2100*) and the literal term "firmware". The firmware archive often contains a file with installation instructions. Usually you can just copy the firmware files to `/lib/firmware`.

After installing the firmware, you can reload the driver with **rmmod** and **modprobe**, or by restarting the system.

Chapter 20. Security

20.1. Introduction

With the increasing usage of the Internet and wireless networks security is getting more important every day. It is impossible to cover this subject in a single chapter of an introduction to GNU/Linux. This chapter covers some basic security techniques that provide a good start for desktop and server security.

Before we go on to specific subjects, it is a good idea to make some remarks about passwords. Computer authorization largely relies on passwords. Be sure to use good passwords in all situations. Avoid using words, names, birth dates and short passwords. These passwords can easily be cracked with dictionary attacks or brute force attacks against hosts or password hashes. Use long passwords, ideally eight characters or longer, consisting of random letters (including capitals) and numbers.

20.2. Closing services

Introduction

Many GNU/Linux run some services that are open to a local network or the Internet. Other hosts can connect to these services by connecting to specific ports. For example, port 80 is used for WWW traffic. The `/etc/services` file contains a table with all commonly used services, and the port numbers that are used for these services.

A secure system should only run the services that are necessary. So, suppose that a host is acting as a web server, it should not have ports open (thus servicing) FTP or SMTP. With more open ports security risks increase very fast, because there is a bigger chance that the software servicing a port has a vulnerability, or is badly configured. The following few sections will help you tracking down which ports are open, and closing them.

Finding open ports

Open ports can be found using a port scanner. Probably the most famous port scanner for GNU/Linux is **nmap**. **nmap** is available through the "n" disk set.

The basic **nmap** syntax is: **nmap host**. The *host* parameter can either be a hostname or IP address. Suppose that we would like to scan the host that **nmap** is installed on. In this case we could specify the *localhost* IP address, *127.0.0.1*:

```
$ nmap 127.0.0.1

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on localhost (127.0.0.1):
(The 1596 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
23/tcp     open        telnet
80/tcp     open        http
6000/tcp   open        X11

Nmap run completed -- 1 IP address (1 host up) scanned in 0 seconds
```

In this example you can see that the host has five open ports that are being serviced; ftp, ssh, telnet, http and X11.

inetd

There are two ways to offer TCP/IP services: by running server applications stand-alone as a daemon or by using the internet super server, **inetd**. inetd is a daemon which monitors a range of ports. If a client attempts to connect to a port inetd handles the connection and forwards the connection to the server software which handles that kind of connection. The advantage of this approach is that it adds an extra layer of security and it makes it easier to log incoming connections. The disadvantage is that it is somewhat slower than using a stand-alone daemon. It is thus a good idea to run a stand-alone daemon on, for example, a heavily loaded FTP server.

You can check whether **inetd** is running on a host or not with **ps**, for example:

```
$ ps ax | grep inetd
 2845 ?        S      0:00 /usr/sbin/inetd
```

In this example inetd is running with PID (process ID) 2845. **inetd** can be configured using the `/etc/inetd.conf` file. Let's have a look at an example line from inetd.conf:

```
# File Transfer Protocol (FTP) server:
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  proftpd
```

This line specifies that inetd should accept FTP connections and pass them to **tcpd**. This may seem a bit odd, because **proftpd** normally handles FTP connections. You can also specify to use proftpd directly in `inetd.conf`, but it is a good idea to give the connection to **tcpd**. This program passes the connection to **proftpd** in turn, as specified. **tcpd** is used to monitor services and to provide host based access control.

Services can be disabled by adding the comment character (#) at the beginning of the line. It is a good idea to disable all services and enable services you need one at a time. After changing `/etc/inetd.conf` **inetd** needs to be restarted to activate the changes. This can be done by sending the HUP signal to the inetd process:

```
# ps ax | grep 'inetd'
   2845 ?        S      0:00 /usr/sbin/inetd
# kill -HUP 2845
```

If you do not need **inetd** at all, it is a good idea to remove it. If you want to keep it installed, but do not want Slackware Linux to load it at the booting process, execute the following command as root:

```
# chmod a-x /etc/rc.d/rc.inetd
```

Chapter 21. Miscellaneous

21.1. Scheduling tasks with cron

Murray Stokely

Slackware Linux includes an implementation of the classic UNIX cron daemon that allows users to schedule tasks for execution at regular intervals. Each user can create, remove, or modify an individual crontab file. This crontab file specifies commands or scripts to be run at specified time intervals. Blank lines in the crontab or lines that begin with a hash ("#") are ignored.

Each entry in the crontab file must contain 6 fields separated by spaces. These fields are minute, hour, day, month, day of week, and command. Each of the first five fields may contain a time or the "*" wildcard to match all times for that field. For example, to have the **date** command run every day at 6:10 AM, the following entry could be used.

```
10 6 * * * date
```

A user crontab may be viewed with the **crontab -l** command. For a deeper introduction to the syntax of a crontab file, let us examine the default *root* crontab.

```
# crontab -l
# If you don't want the output of a cron job mailed to you, you have to direct
# any output to /dev/null.  We'll do this here since these jobs should run
# properly on a newly installed system, but if they don't the average newbie
# might get quite perplexed about getting strange mail every 5 minutes. :^)
#
# Run the hourly, daily, weekly, and monthly cron jobs.
# Jobs that need different timing may be entered into the crontab as before,
# but most really don't need greater granularity than this.  If the exact
# times of the hourly, daily, weekly, and monthly cron jobs do not suit your
# needs, feel free to adjust them.
#
# Run hourly cron jobs at 47 minutes after the hour:
47❶ *❷ *❸ *❹ *❺ /usr/bin/run-parts /etc/cron.hourly 1> /dev/null❻
#
# Run daily cron jobs at 4:40 every day:
40 4 * * * /usr/bin/run-parts /etc/cron.daily 1> /dev/null
#
# Run weekly cron jobs at 4:30 on the first day of the week:
30 4 * * 0 /usr/bin/run-parts /etc/cron.weekly 1> /dev/null
#
# Run monthly cron jobs at 4:20 on the first day of the month:
20 4 1 * * /usr/bin/run-parts /etc/cron.monthly 1> /dev/null
```

❶   The first field, 47, specifies that this job should occur at 47 minutes after the hour.
❷   The second field, *, is a wildcard, so this job should occur *every* hour.
❸   The third field, *, is a wildcard, so this job should occur *every* day.
❹   The fourth field, *, is a wildcard, so this job should occur *every* month.
❺   The fifth field, *, is a wildcard, so this job should occur *every* day of the week.
❻   The sixth field, **/usr/bin/run-parts /etc/cron.hourly 1> /dev/null**, specifies the command that should be run at the time specification defined in the first five fields.

The default root crontab is setup to run scripts in /etc/cron.monthly on a monthly basis, the scripts in /etc/
cron.weekly on a weekly basis, the scripts in /etc/cron.daily on a daily basis, and the scripts in /etc/
cron.hourly on an hourly basis. For this reason it is not strictly necessary for an administrator to understand the
inner workings of cron at all. With Slackware Linux, you can simply add a new script to one of the above directories to
schedule a new periodic task. Indeed, perusing those directories will give you a good idea of the work that Slackware
Linux does behind the scenes on a regular basis to keep things like the **slocate** database updated.

21.2. Hard disk parameters

Many modern disks offer various features for increasing disk performance and improving integrity. Many of these
features can be tuned with the **hdparm** command. Be careful with changing disk settings with this utility, because
some changes can damage data on your disk.

You can get an overview of the active settings for a disk by providing the device node of a disk as a parameter to
hdparm:

```
# hdparm /dev/hda

/dev/hda:
 multcount     =  0 (off)
 IO_support    =  1 (32-bit)
 unmaskirq     =  1 (on)
 using_dma     =  1 (on)
 keepsettings =  0 (off)
 readonly      =  0 (off)
 readahead     = 256 (on)
 geometry      = 65535/16/63, sectors = 78165360, start = 0
```

A common cause for bad disk performance is that DMA was not automatically used by the kernel for a disk. DMA
will speed up I/O throughput and offload CPU usage, by making it possible for the disk to directly transfer data from
the disk to the system memory. If DMA is turned off, the overview of settings would shows this line:

```
using_dma     =  0 (off)
```

You can easily turn on DMA for this disk with the *-d* parameter of **hdparm**:

```
# hdparm -d 1 /dev/hda

/dev/hda:
 setting using_dma to 1 (on)
 using_dma     =  1 (on)
```

You can do this during every boot by adding the hdparm command to /etc/rc.d/rc.local.

The *-i* parameter of **hdparm** is also very useful, because it gives detailed information about a disk:

```
# hdparm -i /dev/hda
```

```
/dev/hda:

 Model=WDC WD400EB-00CPF0, FwRev=06.04G06, SerialNo=WD-WCAAT6022342
 Config={ HardSect NotMFM HdSw>15uSec SpinMotCtl Fixed DTR>5Mbs FmtGapReq }
 RawCHS=16383/16/63, TrkSize=57600, SectSize=600, ECCbytes=40
 BuffType=DualPortCache, BuffSize=2048kB, MaxMultSect=16, MultSect=off
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=78163247
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio1 pio2 pio3 pio4
 DMA modes:  mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 *udma5
 AdvancedPM=no WriteCache=enabled
 Drive conforms to: device does not report version:

 * signifies the current active mode
```

21.3. Monitoring memory usage

In some situations it is handy to diagnose information about how memory is used. For example, on a badly performing server you may want to look whether RAM shortage is causing the system to swap pages, or maybe you are setting up a network service, and want to find the optimum caching parameters. Slackware Linux provides some tools that help you analyse how memory is used.

vmstat

vmstat is a command that can provide statistics about various parts of the virtual memory system. Without any extra parameters **vmstat** provides a summary of some relevant statistics:

```
# vmstat
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 0  0      0 286804   7912  98632    0    0   198     9 1189   783  5  1 93  1
```

Since we are only looking at memory usage in this section, we will only have a look at the *memory* and *swap* fields.

- *swpd*: The amount of virtual memory being used.

- *free*: The amount of memory that is not used at the moment.

- *buff*: The amount of memory used as buffers.

- *cache*: The amount of memory used as cached.

- *si*: The amount of memory that is swapped in from disk per second.

- *si*: The amount of memory that is swapped to disk per second.

It is often useful to see how memory usage changes over time. You can add an interval as a parameter to **vmstat**, to run **vmstat** continuously, printing the current statistics. This interval is in seconds. So, if you want to get updated statistics every second, you can execute:

```
# vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 2  0      0 315132   8832  99324    0    0   189    10 1185   767  5  1 93  1
 1  0      0 304812   8832  99324    0    0     0     0 1222  6881 24  8 68  0
 0  0      0 299948   8836  99312    0    0     0     0 1171  1824 41  9 49  0
 [...]
```

Additionally, you can tell **vmstat** how many times it should output these statistics (rather than doing this infinitely).
For example, if you would like to print these statistics every two seconds, and five times in total, you could execute
vmstat in the following manner:

```
# vmstat 2 5
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 2  0      0 300996   9172  99360    0    0   186    10 1184   756  5  1 93  1
 0  1      0 299012   9848  99368    0    0   336     0 1293  8167 20  8 21 51
 1  0      0 294788  11976  99368    0    0  1054     0 1341 12749 14 11  0 76
 2  0      0 289996  13916  99368    0    0   960   176 1320 17636 22 14  0 64
 2  0      0 284620  16112  99368    0    0  1086   426 1351 21217 25 18  0 58
```

Part VI. Network administration

Table of Contents

Chapter 22. Networking configuration

22.1. Hardware

Network cards (NICs)

Drivers for NICs are installed as kernel modules. The module for your NIC has to be loaded during the initialization of Slackware Linux. On most systems the NIC is automatically detected and configured during the installation of Slackware Linux. You can reconfigure your NIC with the **netconfig** command. **netconfig** adds the driver (module) for the detected card to `/etc/rc.d/rc.netdevice`.

It is also possible to manually configure which modules should be loaded during the initialization of the system. This can be done by adding a **modprobe** line to `/etc/rc.d/rc.modules`. For example, if you want to load the module for 3Com 59x NICs (3c59x.o), add the following line to `/etc/rc.d/rc.modules`

```
/sbin/modprobe 3c59x
```

PCMCIA cards

Supported PCMCIA cards are detected automatically by the PCMCIA software. The pcmcia-cs packages from the "a" disk set provides PCMCIA functionality for Slackware Linux.

22.2. Configuration of interfaces

Network cards are available under Linux through so-called "interfaces". The **ifconfig** command can be used to display the available interfaces:

```
# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:20:AF:F6:D4:AD
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1301 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1529 errors:0 dropped:0 overruns:0 carrier:0
          collisions:1 txqueuelen:100
          RX bytes:472116 (461.0 Kb)  TX bytes:280355 (273.7 Kb)
          Interrupt:10 Base address:0xdc00

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:77 errors:0 dropped:0 overruns:0 frame:0
          TX packets:77 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:8482 (8.2 Kb)  TX bytes:8482 (8.2 Kb)
```

Network cards get the name ethn, in which n is a number, starting with 0. In the example above, the first network card (eth0) already has an IP address. But unconfigured interfaces have no IP address, the **ifconfig** will not show IP addresses for unconfigured interfaces. Interfaces can be configured in the `/etc/rc.d/rc.inet1.conf` file. You can simply read the comments, and fill in the required information. For example:

```
# Config information for eth0:
IPADDR[0]="192.168.1.1"
NETMASK[0]="255.255.255.0"
USE_DHCP[0]=""
DHCP_HOSTNAME[0]=""
```

In this example the IP address 192.168.1.1 with the 255.255.255.0 netmask is assigned to the first ethernet interface (eth0). If you are using a DHCP server you can change the *USE_DHCP=""* line to *USE_DHP[n]="yes"* (swap "n" with the interface number). Other variables, except *DHCP_HOSTNAME* are ignored when using DHCP. For example:

```
IPADDR[1]=""
NETMASK[1]=""
USE_DHCP[1]="yes"
DHCP_HOSTNAME[1]=""
```

The same applies to other interfaces. You can activate the settings by rebooting the system or by executing **/etc/rc.d/rc.inet1**. It is also possible to reconfigure only one interface with **/etc/rc.d/rc.inet1 ethX_restart**, in which *ethX* should be replaced by the name of the interface that you would like to reconfigure.

22.3. Configuration of interfaces (IPv6)

Introduction

IPv6 is the next generation internet protocol. One of the advantages is that it has a much larger address space. In IPv4 (the internet protocol that is commonly used today) addresses are 32-bit, this address space is almost completely used right now, and there is a lack of IPv4 addresses. IPv6 uses 128-bit addresses, which provides an unimaginable huge address space (2^{128} addresses). IPv6 uses another address notation, first of all hex numbers are used instead of decimal numbers, and the address is noted in pairs of 16-bits, separated by a colon (":"). Let's have a look at an example address:

```
fec0:ffff:a300:2312:0:0:0:1
```

A block of zeroes can be replaced by two colons ("::"). Thus, thee address above can be written as:

```
fec0:ffff:a300:2312::1
```

Each IPv6 address has a prefix. Normally this consists of two elements: 32 bits identifying the address space the provider provides you, and a 16-bit number that specifies the network. These two elements form the prefix, and in this case the prefixlength is 32 + 16 = 48 bits. Thus, if you have a /48 prefix you can make 2^{16} subnets and have 2^{80} hosts on each subnet. The image below shows the structure of an IPv6 address with a 48-bit prefix.

Figure 22.1. The anatomy of an IPv6 address
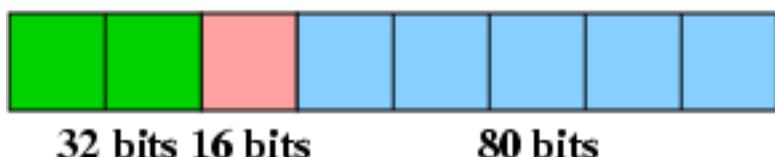
There are a some specially reserved prefixes, most notable include:

Table 22.1. Important IPv6 Prefixes

Prefix	Description
fe80::	Link local addresses, which are not routed.
fec0::	Site local addresses, which are locally routed, but not on or to the internet.
2002::	6to4 addresses, which are used for the transition from IPv4 to IPv6.

Slackware Linux IPv6 support

The Linux kernel binaries included in Slackware Linux do not support IPv6 by default, but support is included as a kernel module. This module can be loaded using **modprobe**:

```
# modprobe ipv6
```

You can verify if IPv6 support is loaded correctly by looking at the kernel output using the **dmesg**:

```
$ dmesg
[..]
IPv6 v0.8 for NET4.0
```

IPv6 support can be enabled permanently by adding the following line to /etc/rc.d/rc.modules:

```
/sbin/modprobe ipv6
```

Interfaces can be configured using **ifconfig**. But it is recommended to make IPv6 settings using the **ip** command, which is part of the "iputils" package that can be found in the extra/ directory of the Slackware Linux tree.

Adding an IPv6 address to an interface

If there are any router advertisers on a network there is a chance that the interfaces on that network already received an IPv6 address when the IPv6 kernel support was loaded. If this is not the case an IPv6 address can be added to an interface using the **ip** utility. Suppose we want to add the address "fec0:0:0:bebe::1" with a prefix length of 64 (meaning "fec0:0:0:bebe" is the prefix). This can be done with the following command syntax:

```
# ip -6 addr add <ip6addr>/<prefixlen> dev <device>
```

For example:

```
# ip -6 addr add fec0:0:0:bebe::1/64 dev eth0
```

22.4. Wireless interfaces

Wireless interfaces usually require some additional configuration, like setting the ESSID, WEP keys and the wireless mode. Interface settings that are specific to wireless interfaces can be set in the /etc/rc.d/rc.wireless.conf

file. The **/etc/rc.d/rc.wireless** script configures wireless interfaces based on descriptions from `/etc/rc.d/`
`rc.wireless.conf`. In `rc.wireless.conf` settings are made per interface MAC address. By default this file
has a section that matches any interface:

```
## NOTE : Comment out the following five lines to activate the samples below ...
## --------- START SECTION TO REMOVE -----------
## Pick up any Access Point, should work on most 802.11 cards
*)
    INFO="Any ESSID"
    ESSID="any"
    ;;
## ---------- END SECTION TO REMOVE ------------
```

It is generally a good idea to remove this section to make per-card settings. If you are lazy and only have one wireless
card, you can leave this section in and add any configuration parameters you need. Since this section matches any
wireless interface the wireless card you have will be matched and configured. You can now add a sections for your
wireless interfaces. Each section has the following format:

```
<MAC address>)
    <settings>
;;
```

You can find the MAC address of an interface by looking at the **ifconfig** output for the interface. For example, if a
wireless card has the *eth1* interface name, you can find the MAC address the following way:

```
# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:01:F4:EC:A5:32
          inet addr:192.168.2.2  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:1 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:504 (504.0 b)
          Interrupt:5 Base address:0x100
```

The hexadecimal address that is printed after *HWaddr* is the MAC address, in this case *00:01:F4:EC:A5:32*.
When you have found the MAC address of the interface you can add a section for the device to `/etc/rc.d/`
`rc.wireless.conf`. For example:

```
00:01:F4:EC:A5:32)
    INFO="Cabletron Roamabout WLAN NIC"
    ESSID="home"
    CHANNEL="8"
    MODE="Managed"
    KEY="1234-5678-AB"
    ;;
```

This will set the interface with MAC address *00:01:F4:EC:A5:32* to use the ESSID *home*, work in *Managed* mode on
channel *8*. The key used for WEP encryption is *1234-5678-AB*. There are many other parameters that can be set. For
an overview of all parameters, refer to the last example in `rc.wireless.conf`.

After configuring a wireless interface, you can activate the changes by executing the network initialization script **/etc/rc.d/rc.inet1**. You can see the current wireless settings with the **iwconfig** command:

```
eth1      IEEE 802.11-DS  ESSID:"home"  Nickname:"HERMES I"
          Mode:Managed  Frequency:2.447 GHz  Access Point: 02:20:6B:75:0C:56
          Bit Rate:2 Mb/s   Tx-Power=15 dBm   Sensitivity:1/3
          Retry limit:4   RTS thr:off   Fragment thr:off
          Encryption key:1234-5678-AB
          Power Management:off
          Link Quality=0/92  Signal level=134/153  Noise level=134/153
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:27  Invalid misc:0   Missed beacon:0
```

22.5. Resolving

Hostname

Each computer on the internet has a hostname. If you do not have a hostname that is resolvable with DNS, it is still a good idea to configure your hostname, because some software uses it. You can configure the hostname in `/etc/HOSTNAME`. A single line with the hostname of the machine will suffice. Normally a hostname has the following form: host.domain.tld, for example darkstar.slackfans.org. Be aware that the hostname has to be resolvable, meaning that GNU/Linux should be able to convert the hostname to an IP address. You can make sure the hostname is resolvable by adding it to `/etc/hosts`. Read the following section for more information about this file.

/etc/hosts

`/etc/hosts` is a table of IP addresses with associated hostnames. This file can be used to name computers in a small network. Let's look at an example of the `/etc/hosts` file:

```
127.0.0.1               localhost
192.168.1.1             tazzy.slackfans.org tazzy
192.168.1.169           flux.slackfans.org
```

The *localhost* line should always be present. It assigns the name *localhost* to a special interface, the loopback. In this example the names *tazzy.slackfans.org* and *tazzy* are assigned to the IP address 192.168.1.1, and the name *flux.slackfans.org* is assigned to the IP address 192.168.1.169. On the system with this file both computers are available via the mentioned hostnames.

It is also possible to add IPv6 addresses, which will be used if your system is configured for IPv6. This is an example of a `/etc/hosts` file with IPv4 and IPv6 entries:

```
# IPv4 entries
127.0.0.1               localhost
192.168.1.1             tazzy.slackfans.org tazzy
192.168.1.169           gideon.slackfans.org

# IPv6 entries
::1   localhost
fec0:0:0:bebe::2 flux.slackfans.org
```

Please note that "::1" is the default IPv6 loopback.

/etc/resolv.conf

The `/etc/resolv.conf` file is used to specify which nameservers the system should use. A nameserver converts hostnames to IP addresses. Your provider should have given you at least two name name server addresses (DNS servers). You can add these nameservers to `/etc/resolv.conf` by adding the line *nameserver ipaddress* for each nameserver. For example:

```
nameserver 192.168.1.1
nameserver 192.168.1.169
```

You can check wether the hostnames are tranlated correctly or not with the **host hostname** command. Swap *hostname* with an existing hostname, for example the website of your internet service provider.

22.6. IPv4 Forwarding

IPv4 forwarding connects two or more networks by sending packets which arrive on one interface to another interface. This makes it possible to let a GNU/Linux machine act as a router. For example, you can connect multiple networks, or your home network with the internet. Let's have a look at an example:

Figure 22.2. Router example
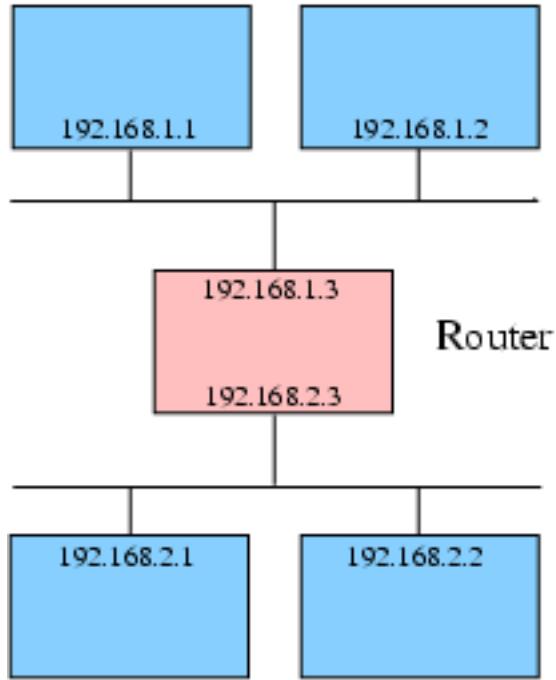
In dit example there are two networks, 192.168.1.0 and 192.168.2.0. Three hosts are connected to both network. One of these hosts is connected to both networks with interfaces. The interface on the 192.168.1.0 network has IP address 192.168.1.3, the interface on the 192.168.2.0 network has IP address 192.168.2.3. If the host acts as a router between both networks it forwards packets from the 192.168.1.0 network to the 192.168.2.0 network and vise versa. Routing of normal IPv4 TCP/IP packages can be enabled by enabling IPv4 forwarding.

IPv4 forwarding can be enabled or disabled under Slackware Linux by changing the executable bit of the `/etc/rc.d/rc.ip_forward` file. If the executable bit is set on this file, IP forwarding will be enabled during the system

boot, otherwise it will not. You can check whether the executable bit is enabled with **ls -l** (a description of the **ls** command can be found in the section called "Listing files").

It is also possible to enable IPv4 forwarding on a running system with the following command (0 disables forwarding, 1 enables forwarding):

```
# echo 0 > /proc/sys/net/ipv4/ip_forward
```

Be cautious! By default there are no active packet filters. This means that anyone can access other networks. Traffic can be filtered and logged with the iptables kernel packet filter. Iptables can be administrated through the **iptables** command. NAT (Network Address Translation) is also a subset of iptables, and can be controlled and enabled through the **iptables** command. NAT makes it possible to "hide" a network behind one IP address. This allows you to use the internet on a complete network with only one IP address.

Chapter 23. IPsec

23.1. Theory

IPsec is a standard for securing IP communication through authentication, and encryption. Besides that it can compress packets, reducing traffic. The following protocols are part of the IPsec standard:

- AH (Authentication Header) provides authenticity guarantee for transported packets. This is done by checksumming the packages using a cryptographic algorithm. If the checksum is found to be correct by the receiver, the receiver can be assured that the packet is not modified, and that the packet really originated from the reported sender (provided that the keys are only known by the sender and receiver).

- ESP (Encapsulating Security Payload) is used to encrypt packets. This makes the data of the packet confident, and only readable by the host with the right decryption key.

- IPcomp (IP payload compression) provides compression before a packet is encrypted. This is useful, because encrypted data generally compresses worse than unencrypted data.

- IKE (Internet Key Exchange) provides the means to negotiate keys in secrecy. Please note that IKE is optional, keys can be configured manually.

There are actually two modes of operation: *transport mode* is used to encrypt normal connections between two hosts, *tunnel mode* encapsulates the original package in a new header. In this chapter we are going to look at the transport mode, because the primary goal of this chapter is to show how to set up a secure connection between two hosts.

There are also two major methods of authentication. You can use manual keys, or an Internet Key Exchange (IKE) daemon, like racoon, that automatically exchanges keys securely betwoon two hosts. In both cases you need to set up a policy in the Security Policy Database (SPD). This database is used by the kernel to decide what kind of security policy is needed to communicate with another host. If you use manual keying you also have to set up Security Association Database (SAD) entries, which specifies what encryption algorithmn and key should be used for secure communication with another host. If you use an IKE daemon the security associations are automatically established.

23.2. Linux configuration

Native IPsec support is only available in Linux 2.6.x kernels. Earlier kernels have no native IPsec support. So, make sure that you have a 2.6.x kernel. The 2.6 kernel is available in Slackware Linux 10.0, 10.1, and 10.2 from the `testing` directory on CD2 of the Slackware Linux CD sets, or any of the official Slackware Linux mirrors. The 2.6 kernel is the default kernel since Slackware Linux 12.0. The default Slackware Linux 2.6 kernel has support for AH, ESP and IPcomp in for both IPv4 and IPv6. If you are compiling a custom kernel enable use at least the following options in your kernel configuration:

```
CONFIG_INET_AH=y
CONFIG_INET_ESP=y
CONFIG_INET_IPCOMP=y
```

Or you can compile support for IPsec protocols as a module:

```
CONFIG_INET_AH=m
CONFIG_INET_ESP=m
```

```
CONFIG_INET_IPCOMP=m
```

In this chapter we are only going to use AH and ESP transformations, but it is not a bad idea to enable IPComp transformation for further configuration of IPsec. Besides support for the IPsec protocols, you have to compile kernel support for the encryption and hashing algorithms that will be used by AH or ESP. Linux or module support for these algorithms can be enabled by twiddling the various *CONFIG_CRYPTO* options. It does not hurt to compile all ciphers and hash algorithms as a module.

When you choose to compile IPsec support as a module, make sure that the required modules are loaded. For example, if you are going to use ESP for IPv4 connections, load the *esp4* module.

Compile the kernel as usual and boot it.

23.3. Installing IPsec-Tools

The next step is to install the IPsec-Tools [http://ipsec-tools.sourceforge.net]. These tools are ports of the KAME [http://www.kame.net] IPsec utilities. Download the latest sources and unpack, configure and install them:

```
# tar jxf ipsec-tools-x.y.z.tar.bz2
# cd ipsec-tools-x.y.z
# CFLAGS="-O2 -march=i486 -mcpu=i686" \
  ./configure --prefix=/usr \
              --sysconfdir=/etc \
              --localstatedir=/var \
              --enable-hybrid \
              --enable-natt \
              --enable-dpd \
              --enable-frag \
              i486-slackware-linux
# make
# make install
```

Replace *x.y.z* with the version of the downloaded sources. The most notable flags that we specify during the configuration of the sources are:

- *--enable-dpd*: enables dead peer detection (DPD). DPD is a method for detecting wether any of the hosts for which security associations are set up is unreachable. When this is the case the security associations to that host can be removed.

- *--enable-natt*: enables NAT traversal (NAT-T). Since NAT alters the IP headers, this causes problems for guaranteeing authenticity of a packet. NAT-T is a method that helps overcoming this problem. Configuring NAT-T is beyond the scope of this article.

23.4. Setting up IPsec with manual keying

Introduction

We will use an example as the guideline for setting up an encrypted connection between to hosts. The hosts have the IP addresses *192.168.1.1* and *192.168.1.169*. The "transport mode" of operation will be used with AH and ESP transformations and manual keys.

Writing the configuration file

The first step is to write a configuration file we will name `/etc/setkey.conf`. On the first host (192.168.1.1) the following `/etc/setkey.conf` configuration will be used:

```
#!/usr/sbin/setkey -f

# Flush the SAD and SPD
flush;
spdflush;

add 192.168.1.1 192.168.1.169 ah 0x200 -A hmac-md5
0xa731649644c5dee92cbd9c2e7e188ee6;
add 192.168.1.169 192.168.1.1 ah 0x300 -A hmac-md5
0x27f6d123d7077b361662fc6e451f65d8;

add 192.168.1.1 192.168.1.169 esp 0x201 -E 3des-cbc
0x656c8523255ccc23a66c1917aa0cf30991fce83532a4b224;
add 192.168.1.169 192.168.1.1 esp 0x301 -E 3des-cbc
0xc966199f24d095f3990a320d749056401e82b26570320292

spdadd 192.168.1.1 192.168.1.169 any -P out ipsec
        esp/transport//require
        ah/transport//require;

spdadd 192.168.1.169 192.168.1.1 any -P in ipsec
        esp/transport//require
        ah/transport//require;
```

The first line (a line ends with a ";") adds a key for the header checksumming for packets coming from 192.168.1.1 going to 192.168.1.169. The second line does the same for packets coming from 192.168.1.169 to 192.168.1.1. The third and the fourth line define the keys for the data encryption the same way as the first two lines. Finally the "spadd" lines define two policies, namely packets going out from 192.168.1.1 to 192.168.1.169 should be (require) encoded (esp) and "signed" with the authorization header. The second policy is for incoming packets and it is the same as outgoing packages.

Please be aware that you should not use these keys, but your own secretly kept unique keys. You can generate keys using the `/dev/random` device:

```
# dd if=/dev/random count=16 bs=1 | xxd -ps
```

This command uses dd to output 16 bytes from `/dev/random`. Don't forget to add "0x" at the beginning of the line in the configuration files. You can use the 16 byte (128 bits) for the hmac-md5 algorithm that is used for AH. The 3des-cbc algorithm that is used for ESP in the example should be fed with a 24 byte (192 bits) key. These keys can be generated with:

```
# dd if=/dev/random count=24 bs=1 | xxd -ps
```

Make sure that the /etc/setkey.conf file can only be read by the root user. If normal users can read the keys IPsec provides no security at all. You can do this with:

chmod 600 /etc/setkey.conf

The same /etc/setkey.conf can be created on the 192.168.1.169 host, with inverted *-P in* and *-P out* options. So, the /etc/setkey.conf will look like this:

```
#!/usr/sbin/setkey -f

# Flush the SAD and SPD
flush;
spdflush;

add 192.168.1.1 192.168.1.169 ah 0x200 -A hmac-md5
0xa731649644c5dee92cbd9c2e7e188ee6;
add 192.168.1.169 192.168.1.1 ah 0x300 -A hmac-md5
0x27f6d123d7077b361662fc6e451f65d8;

add 192.168.1.1 192.168.1.169 esp 0x201 -E 3des-cbc
0x656c8523255ccc23a66c1917aa0cf30991fce83532a4b224;
add 192.168.1.169 192.168.1.1 esp 0x301 -E 3des-cbc
0xc966199f24d095f3990a320d749056401e82b26570320292

spdadd 192.168.1.1 192.168.1.169 any -P in ipsec
          esp/transport//require
          ah/transport//require;

spdadd 192.168.1.169 192.168.1.1 any -P out ipsec
          esp/transport//require
          ah/transport//require;
```

Activating the IPsec configuration

The IPsec configuration can be activated with the **setkey** command:

setkey -f /etc/setkey.conf

If you want to enable IPsec permanently you can add the following line to /etc/rc.d/rc.local on both hosts:

```
/usr/sbin/setkey -f /etc/setkey.conf
```

After configuring IPsec you can test the connection by running **tcpdump** and simultaneously pinging the other host. You can see if AH and ESP are actually used in the **tcpdump** output:

```
# tcpdump -i eth0
tcpdump: listening on eth0
11:29:58.869988 terrapin.taickim.net > 192.168.1.169: AH(spi=0x00000200,seq=0x40f): ESP
11:29:58.870786 192.168.1.169 > terrapin.taickim.net: AH(spi=0x00000300,seq=0x33d7): ES
```

23.5. Setting up IPsec with automatic key exchanging

Introduction

The subject of automatical key exchange is already touched shortly in the introduction of this chapter. Put simply, IPsec with IKE works in the following steps.

1. Some process on the host wants to connect to another host. The kernel checks whether there is a security policy set up for the other host. If there already is a security association corresponding with the policy the connection can be made, and will be authenticated, encrypted and/or compressed as defined in the security association. If there is no security association, the kernel will request a user-land IKE daemon to set up the necessary security association(s).

2. During the first phase of the key exchange the IKE daemon will try to verify the authenticity of the other host. This is usually done with a preshared key or certificate. If the authentication is successful a secure channel is set up between the two hosts, usually called a IKE security association, to continue the key exchange.

3. During the second phase of the key exchange the security associations for communication with the other host are set up. This involves choosing the encryption algorithm to be used, and generating keys that are used for encryption of the communication.

4. At this point the first step is repeated again, but since there are now security associations the communication can proceed.

The **racoon** IKE daemon is included with the KAME IPsec tools, the sections that follow explain how to set up racoon.

Using racoon with a preshared key

As usual the first step to set up IPsec is to define security policies. In contrast to the manual keying example you should not set up security associations, because racoon will make them for you. We will use the same host IPs as in the example above. The security policy rules look like this:

```
#!/usr/sbin/setkey -f

# Flush the SAD and SPD
flush;
spdflush;

spdadd 192.168.1.1 192.168.1.169 any -P out ipsec
 esp/transport//require;

spdadd 192.168.1.169 192.168.1.1 any -P in ipsec
 esp/transport//require;
```

Cautious souls have probably noticed that AH policies are missing in this example. In most situations this is no problem, ESP can provide authentication. But you should be aware that the authentication is more narrow; it does not protect information outside the ESP headers. But it is more efficient than encapsulating ESP packets in AH.

With the security policies set up you can configure **racoon**. Since the connection-specific information, like the authentication method is specified in the phase one configuration. We can use a general phase two configuration. It is also possible to make specific phase two settings for certain hosts. But generally speaking a general configuration will often suffice in simple setups. We will also add paths for the preshared key file, and certification directory. This is an example of /etc/racoon.conf with the paths and a general phase two policy set up:

```
path pre_shared_key "/etc/racoon/psk.txt";
path certificate "/etc/racoon/certs";

sainfo anonymous {
{
 pfs_group 2;
 lifetime time 1 hour;
 encryption_algorithm 3des, blowfish 448, rijndael;
 authentication_algorithm hmac_sha1, hmac_md5;
 compression_algorithm deflate;
}
```

The *sainfo* identifier is used to make a block that specifies the settings for security associations. Instead of setting this for a specific host, the *anonymous* parameter is used to specify that these settings should be used for all hosts that do not have a specific configuration. The *pfs_group* specifies which group of Diffie-Hellman exponentiations should be used. The different groups provide different lengths of base prime numbers that are used for the authentication process. Group 2 provides a 1024 bit length if you would like to use a greater length, for increased security, you can use another group (like 14 for a 2048 bit length). The *encryption_algorithm* specifies which encryption algorithms this host is willing to use for ESP encryption. The *authentication_algorithm* specifies the algorithm to be used for ESP Authentication or AH. Finally, the *compression_algorithm* is used to specify which compression algorithm should be used when IPcomp is specified in an association.

The next step is to add a phase one configuration for the key exchange with the other host to the racoon.conf configuration file. For example:

```
remote 192.168.1.169
{
 exchange_mode aggressive, main;
 my_identifier address;
 proposal {
  encryption_algorithm 3des;
  hash_algorithm sha1;
  authentication_method pre_shared_key;
  dh_group 2;
 }
}
```

The *remote* block specifies a phase one configuration. The *exchange_mode* is used to configure what exchange mode should be used for phase. You can specify more than one exchange mode, but the first method is used if this host is the initiator of the key exchange. The *my_identifier* option specifies what identifier should be sent to the remote host. If this option committed *address* is used, which sends the IP address as the identifier. The

proposal block specifies parameter that will be proposed to the other host during phase one authentication. The *encryption_algorithm*, and *dh_group* are explained above. The *hash_algorithm* option is mandatory, and configures the hash algorithm that should be used. This can be *md5*, or *sha1*. The *authentication_method* is crucial for this configuration, as this parameter is used to specify that a preshared key should be used, with *pre_shared_key*.

With racoon set up there is one thing left to do, the preshared key has to be added to /etc/racoon/psk.txt. The syntax is very simple, each line contains a host IP address and a key. These parameters are separated with a tab. For example:

```
192.168.1.169 somekey
```

Activating the IPsec configuration

At this point the configuration of the security policies and racoon is complete, and you can start to test the configuration. It is a good idea to start **racoon** with the *-F* parameter. This will run **racoon** in the foreground, making it easier to catch error messages. To wrap it up:

```
# setkey -f /etc/setkey.conf
# racoon -F
```

Now that you have added the security policies to the security policy database, and started **racoon**, you can test your IPsec configuration. For instance, you could ping the other host to start with. The first time you ping the other host, this will fail:

```
$ ping 192.168.1.169
connect: Resource temporarily unavailable
```

The reason for this is that the security associations still have to be set up. But the ICMP packet will trigger the key exchange. ping will trigger the key exchange. You can see whether the exchange was succesful or not by looking at the **racoon** log messages in /var/log/messages, or the output on the terminal if you started **racoon** in the foreground. A succesful key exhange looks like this:

```
 Apr  4 17:14:58 terrapin racoon: INFO: IPsec-SA request for 192.168.1.169 queued due t
 Apr  4 17:14:58 terrapin racoon: INFO: initiate new phase 1 negotiation: 192.168.1.1[5
 Apr  4 17:14:58 terrapin racoon: INFO: begin Aggressive mode.
 Apr  4 17:14:58 terrapin racoon: INFO: received Vendor ID: DPD
 Apr  4 17:14:58 terrapin racoon: NOTIFY: couldn't find the proper pskey, try to get on
 Apr  4 17:14:58 terrapin racoon: INFO: ISAKMP-SA established 192.168.1.1[500]-192.168.
 Apr  4 17:14:59 terrapin racoon: INFO: initiate new phase 2 negotiation: 192.168.1.1[0
 Apr  4 17:14:59 terrapin racoon: INFO: IPsec-SA established: ESP/Transport 192.168.1.1
 Apr  4 17:14:59 terrapin racoon: INFO: IPsec-SA established: ESP/Transport 192.168.1.1
```

After the key exchange, you can verify that IPsec is set up correctly by analyzing the packets that go in and out with **tcpdump**. **tcpdump** is available in the *n* diskset. Suppose that the outgoing connection to the other host goes through the *eth0* interface, you can analyze the packats that go though the *eth0* interface with **tcpdump -i eth0**. If the outgoing packets are encrypted with ESP, you can see this in the **tcpdump** output. For example:

```
# tcpdump -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
17:27:50.241067 IP terrapin.taickim.net > 192.168.1.169: ESP(spi=0x059950e8,seq=0x9)
17:27:50.241221 IP 192.168.1.169 > terrapin.taickim.net: ESP(spi=0x0ddff7e7,seq=0x9)
```

Chapter 24. The Internet super server

24.1. Introduction

There are two ways to offer TCP/IP services: by running server applications standalone as a daemon or by using the Internet super server, **inetd**. **inetd** is a daemon which monitors a range of ports. If a client attempts to connect to a port **inetd** handles the connection and forwards the connection to the server software which handles that kind of connection. The advantage of this approach is that it adds an extra layer of security and it makes it easier to log incoming connections. The disadvantage is that it is somewhat slower than using a standalone daemon. It is thus a good idea to run a standalone daemon on, for example, a heavily loaded FTP server.

24.2. Configuration

inetd can be configured using the `/etc/inetd.conf` file. Let's have a look at an example line from `inetd.conf`:

```
# File Transfer Protocol (FTP) server:
ftp      stream tcp     nowait  root     /usr/sbin/tcpd  proftpd
```

This line specifies that **inetd** should accept FTP connections and pass them to **tcpd**. This may seem a bit odd, because **proftpd** normally handles FTP connections. You can also specify to use **proftpd** directly in `inetd.conf`, but Slackware Linux normally passes the connection to **tcpd**. This program passes the connection to **proftpd** in turn, as specified. **tcpd** is used to monitor services and to provide host based access control.

Services can be disabled by adding the comment character (#) at the beginning of the line. It is a good idea to disable all services and enable services you need one at a time. After changing `/etc/inetd.conf` **inetd** needs to be restarted to activate the changes. This can be done by sending the HUP signal to the inetd process:

```
# ps ax | grep 'inetd'
  64 ?        S      0:00 /usr/sbin/inetd
# kill -HUP 64
```

Or you can use the `rc.inetd` initialization script to restart **inetd**:

```
# /etc/rc.d/rc.inetd restart
```

24.3. TCP wrappers

As you can see in `/etc/inetd.conf` connections for most protocols are made through **tcpd**, instead of directly passing the connection to a service program. For example:

```
# File Transfer Protocol (FTP) server:
ftp      stream tcp     nowait  root     /usr/sbin/tcpd  proftpd
```

In this example ftp connections are passed through **tcpd**. **tcpd** logs the connection through syslog and allows for additional checks. One of the most used features of **tcpd** is host-based access control. Hosts that should be denied are controlled via `/etc/hosts.deny`, hosts that should be allowed via `/etc/hosts.allow`. Both files have one rule on each line of the following form:

```
service: hosts
```

Hosts can be specified by hostname or IP address. The ALL keyword specifies all hosts or all services.

Suppose we want to block access to all services managed through **tcpd**, except for host "trusted.example.org". To do this the following `hosts.deny` and `hosts.allow` files should be created.

`/etc/hosts.deny`:

```
ALL: ALL
```

`/etc/hosts.allow`:

```
ALL: trusted.example.org
```

In the `hosts.deny` access is blocked to all (ALL) services for all (ALL) hosts. But `hosts.allow` specifies that all (ALL) services should be available to "trusted.example.org".

Chapter 25. Apache

25.1. Introduction

Apache is the most popular web server since April 1996. It was originally based on NCSA httpd, and has grown into a full-featured HTTP server. Slackware Linux currently uses the 1.3.x branch of Apache. This chapter is based on Apache 1.3.x.

25.2. Installation

Apache can be installed by adding the `apache` package from the "n" disk set. If you also want to use PHP, the `php` ("n" disk set) and `mysql` ("ap" disk set) are also required. MySQL is required, because the precompiled PHP depends on MySQL shared libraries. You do not have to run MySQL itself. After installing Apache it can be started automatically while booting the system by making the `/etc/rc.d/rc.httpd` file executable. You can do this by executing:

```
# chmod a+x /etc/rc.d/rc.httpd
```

The Apache configuration can be altered in the `/etc/apache/httpd.conf` file. Apache can be stopped/started/ restarted every moment with the **apachectl** command, and the `stop`, `start` and `restart` parameters. For example, execute the following command to restart Apache:

```
# apachectl restart
/usr/sbin/apachectl restart: httpd restarted
```

25.3. User directories

Apache provides support for so-call user directories. This means every user gets web space in the form of *http://host/ ~user/*. The contents of "~user/" is stored in a subdirectory in the home directory of the user. This directory can be specified using the "UserDir" option in `httpd.conf`, for example:

```
UserDir public_html
```

This specifies that the `public_html` directory should be used for storing the web pages. For example, the web pages at URL *http://host/~snail/* are stored in `/home/snail/public_html`.

25.4. Virtual hosts

The default documentroot for Apache under Slackware Linux is `/var/www/htdocs`. Without using virtual hosts every client connecting to the Apache server will receive the website in this directory. So, if we have two hostnames pointing to the server, "www.example.org" and "forum.example.org", both will display the same website. You can make separate sites for different hostnames by using virtual hosts.

In this example we are going to look how you can make two virtual hosts, one for "www.example.org", with the documentroot `/var/www/htdocs-www`, and "forum.example.org", with the documentroot `/var/www/htdocs-forum`. First of all we have to specify which IP addresses Apache should listen to. Somewhere in the `/etc/apache/ httpd.conf` configuration file you will find the following line:

```
#NameVirtualHost *:80
```

This line has to be commented out to use name-based virtual hosts. Remove the comment character (#) and change the parameter to "BindAddress IP:port", or "BindAddress *:port" if you want Apache to bind to all IP addresses the host has. Suppose we want to provide virtual hosts for IP address 192.168.1.201 port 80 (the default Apache port), we would change the line to:

```
NameVirtualHost 192.168.1.201:80
```

Somewhere below the NameVirtualHost line you can find a commented example of a virtualhost:

```
#<VirtualHost *:80>
#     ServerAdmin webmaster@dummy-host.example.com
#     DocumentRoot /www/docs/dummy-host.example.com
#     ServerName dummy-host.example.com
#     ErrorLog logs/dummy-host.example.com-error_log
#     CustomLog logs/dummy-host.example.com-access_log common
#</VirtualHost>
```

You can use this example as a guideline. For example, if we want to use all the default values, and we want to write the logs for both virtual hosts to the default Apache logs, we would add these lines:

```
<VirtualHost 192.168.1.201:80>
 DocumentRoot /var/www/htdocs-www
 ServerName www.example.org
</VirtualHost>

<VirtualHost 192.168.1.201:80>
 DocumentRoot /var/www/htdocs-forum
 ServerName forum.example.org
</VirtualHost>
```

Chapter 26. BIND

26.1. Introduction

The domain name system (DNS) is used to convert human-friendly host names (for example www.slackware.com) to IP addresses. BIND (Berkeley Internet Name Domain) is the most widely used DNS daemon, and will be covered in this chapter.

Delegation

One of the main features is that DNS requests can be delegated. For example, suppose that you own the "linuxcorp.com" domain. You can provide the authorized nameservers for this domain, you nameservers are authoritative for the "linuxcorp.com". Suppose that there are different branches within your company, and you want to give each branch authority over their own zone, that is no problem with DNS. You can delegate DNS for e.g. "sales.linuxcorp.com" to another nameserver within the DNS configuration for the "linuxcorp.com" zone.

The DNS system has so-called root servers, which delegate the DNS for millions of domain names and extensions (for example, country specific extensions, like ".nl" or ".uk") to authorized DNS servers. This system allows a branched tree of delegation, which is very flexible, and distributes DNS traffic.

DNS records types

The following types are common DNS record types:

Table 26.1. DNS records

Prefix	Description
A	An A record points to an IPv4 IP address.
AAAA	An AAAA record points to an IPv6 IP address.
CNAME	A CNAME record points to another DNS entry.
MX	A MX record specifies which should handle mail for the domain.

Masters and slaves

Two kinds of nameservers can be provided for a domain name: a master and slaves. The master server DNS records are authoritative. Slave servers download their DNS record from the master servers. Using slave servers besides a master server is recommended for high availability and can be used for load-balancing.

26.2. Making a caching nameserver

A caching nameserver provides DNS services for a machine or a network, but does not provide DNS for a domain. That means it can only be used to convert hostnames to IP addresses. Setting up a nameserver with Slackware Linux is fairly easy, because BIND is configured as a caching nameserver by default. Enabling the caching nameserver takes just two steps: you have to install BIND and alter the initialization scripts. BIND can be installed by adding the bind package

from the "n" disk set. After that bind can be started by executing the **named** command. If want to start BIND by default, make the `/etc/rc.d/rc.bind` file executable. This can be done by executing the following command as root:

 # chmod a+x /etc/rc.d/rc.bind

If you want to use the nameserver on the machine that runs BIND, you also have to alter `/etc/resolv.conf`.