Linux System Administrator's Guide 0.2

Linux System Administrator's Guide 0.2

Lars Wirzenius

The Linux Documentation Project

Lars Wirzenius, Hernesaarenkatu 15 A 2, Fin-00150 Helsinki, Finland
`lars.wirzenius@helsinki.fi`
 Published 1995.

This page is dedicated to a future dedication.

Contents

Chapter 1

Introduction to the ALPHA Versions

This is an ALPHA version of the Linux System Administrator's Guide. That means that I don't even pretend it contains anything useful, or that anything contained within it is factually correct. In fact, if you believe anything that I say in this version, and you are hurt because of it, I will cruelly laugh at your face if you complain.

Well, almost. I won't laugh, but I also will not consider myself responsible for anything.

The purpose of an ALPHA version is to get the stuff out so that other people can look at it and comment on it. The latter part is the important one: Unless the author gets feedback, the ALPHA version isn't doing anything good. Therefore, if you read this 'book', please, *please*, **please** let me hear your opinion about it. I don't care whether you think it is good or bad, I want you to tell me about it.

If at all possible, you should mail your comments directly to me, otherwise there is a largish chance I will miss them. If you want to discuss things in public (one of the `comp.os.linux` newsgroups or the mailing list), that is ok by me, but please send a copy via mail directly to me as well.

I do not much care about the format in which you send your comments, but it is essential that you clearly indicate what part of my text you are commenting on.

I can be contacted at the following e-mail addresses:

```
lars.wirzenius@helsinki.fi
wirzeniu@cc.helsinki.fi
wirzeniu@cs.helsinki.fi
wirzeniu@kruuna.helsinki.fi
wirzeniu@hydra.helsinki.fi
```

(they're all actually the same account, but I give all these, just in case there is some weird problem).

This text contains a lot of notes that I have inserted as notes to myself. They are identified with "**META:** ". They indicate things that need to be worked on, that are missing, that are wrong, or something like that. They are mostly for my own benefit and for your amusement, they are not things that I am hoping someone else will write for me.

Parts of the chapters in this version are still unwritten. If someone wishes to fill things in, I'm glad to have it, but I will proabably write it myself, anyway (mixing text from several people in one chapter usually makes for a less good manual). If you are serious about wanting to write a whole chapter or more, please contact me for ideas.

If you think that this version of the manual is missing a lot, you are right. I am including only those chapters that are at least half finished. New chapters will be released as they are written.

For reference: This is version 0.2, released 1995-02-04.

The LDP Rhyme[1]

A wondrous thing,
and beautiful,
'tis to write,
a book.

I'd like to sing,
of the sweat,
the blood and tear,
which it also took.

It started back in,
nineteen-ninety-two,
when users whined,
"we can nothing do!"

They wanted to know,
what their problem was,
and how to fix it
(by yesterday).

We put the answers in,
a Linux f-a-q,
hoped to get away,
from any more writin'.

"That's too long,
it's hard to search,
and we don't read it,
any-which-way!"

Then a few of us,
joined toghether
(virtually, you know),
to start the LDP.

We started to write,
or plan, at least,
several books,
one for every need.

The start was fun,
a lot of talk,
an outline,
then a slew.

Then silence came,
the work began,
some wrote less,
others more.

A blank screen,
oh its horrible,
it sits there,
laughs in the face.

We still await,
the final day,
when everything,
will be done.

Until then,
all we have,
is a draft,
for you to comment on.

[1] The author wishes to remain anonymous. It was posted to the mailing list by Matt Welsh.

Chapter 2

Introduction

This manual, the Linux System Administrator's Guide, describes the system administration aspects of using Linux. It is intended for people who know next to nothing about system administration (as in "what is it?"), but who already master at least the basics of normal usage, which means roughly the material covered by the (as yet unpublished) Linux User's Guide. This manual also doesn't tell you how to install Linux, that is described in the Getting Started document. There is some overlap between all manuals, however, but they all look at things from slightly different angles. See below for more information about Linux manuals.

What, then, is system administration? It is all the things that one has to do to keep a computer system in a useable shape. Things like backing up files (and restoring them if necessary), installing new programs, creating accounts for users (and deleting them when no longer needed), making certain that the filesystem is not corrupted, and so on. If a computer were a house, say, system administration would be called maintenance, and would include cleaning, fixing broken windows, and other such things. System administration is not called maintenance, because that would be too simple. [1]

The structure of this manual is such that many of the chapters should be usable independently, so that if you need information about, say, backups, you can read just that chapter. This hopefully makes the book easier to use as a reference manual, and makes it possible to read just a small part when needed, instead of having to read everything. We have tried to create a good index as well, to make it easier to find things. However, this manual is first and foremost a tutorial, and a reference manual only as a lucky coincidence.

This manual is not intended to be used completely by itself. Plenty of the rest of the

[1] There are some people who *do* call it that, but that's just because they have never read this manual, poor things.

Linux documentation is also important for system administrators. After all, a system administrator is just a user with special privileges and duties. A very important resource is the man pages, which should always be consulted when a command is not familiar. however.)

While this manual is targeted at Linux, a general principle has been that it should be useful with other UNIX based operating systems as well. Unfortunately, since there is so much variance between different versions of UNIX in general, and in system administration in particular, there is little hope for us to cover all variants. Even covering all possibilities for Linux is difficult, due to the nature of its development. There is no one official Linux distribution, so different people have different setups, many people have a setup they have built up themselves. When possible, we have tried to point out differences, and explain several alternatives. In order to cater to the hackers and DIY types that form the driving force behind Linux development, we have tried to describe how things work, rather than just listing "five easy steps" for each task. This means that there is much information here that is not necessary for everyone, but those parts are marked as such and can be skipped if you use a preconfigured system. Reading everything will, naturally, increase your understanding of the system and should make using and administering it more pleasant.

Like all other Linux related development, the work was done on a volunteer basis: we did it because we thought it might be fun and/or because we felt it should be done. However, like all volunteer work, there is a limit to how much effort we have been able to spend on this work, and also on how much knowledge and experience we have. This means that the manual is not necessarily as good as it would be if a wizard had been paid handsomely to write it and had spent a few years to perfect it. We think, of course, that it is pretty nice, but be warned. Also, on the general principle that *no* single source of information is enough, we have compiled a short bibliography of books, magazines, and papers related to UNIX system administration.

One particular point where we have cut corners and reduced our workload is that we have not covered very thoroughly many things that are already well documented in other freely available manuals. This applies especially to program specific documentation, such as all the details of using `mkfs(8)`; we only describe the purpose of the program, and as much of its usage as is necessary for the purposes of this manual. For further information, we refer the gentle reader to these other manuals. Usually, all of the referred to documentation is part of the full Linux documentation set.

While we have tried to make this manual as good as possible, we would really like to hear from you if you have any ideas on how to make it better. Bad language, factual errors, ideas for new areas to cover, rewritten sections, information about how various UNIX versions do things, we are interested in all of it. The maintainer of the manual is Lars Wirzenius. You can contact him via electronic mail with the Internet domain address `lars.wirzenius@helsinki.fi`, or by traditional paper mail using the address

Lars Wirzenius / Linux docs
Hernesaarentie 15 A 2
00150 Helsinki
Finland

META: acks

2.1 The Linux Documentation Project

The Linux Documentation Project, or LDP, is a loose team of writers, proofreaders, and editors who are working together to provide complete documentation for the Linux operating system. The overall coordinator of the project is Matt Welsh, who is heavily aided by Lars Wirzenius and Michael K. Johnson.

This manual is one in a set of several being distributed by the LDP, including a Linux Users' Guide, System Administrators' Guide, Network Administrators' Guide, and Kernel Hackers' Guide. These manuals are all available in LaTeX source format, .dvi format, and postscript output by anonymous FTP from `nic.funet.fi`, in the directory `/pub/OS/Linux/doc/doc-project`, and from `tsx-11.mit.edu`, in the directory `/pub/linux/docs/guides`.

We encourage anyone with a penchant for writing or editing to join us in improving Linux documentation. If you have Internet e-mail access, you can join the `DOC` channel of the `Linux-Activists` mailing list by sending mail to

```
linux-activists-request@niksula.hut.fi
```

with the line

```
X-Mn-Admin: join DOC
```

in the header or as the first line of the message body.

2.2 Copyright, trademarks, other legalese

UNIX is a trademark of Unix System Labratories
Linux is not a trademark, and has no connection to UNIX™ or Unix System Labratories.

Chapter 3

Directory Tree Overview

META: This chapter is waiting a major rewrite

META: Give sample contents of important files, e.g. /etc/passwd, /etc/group (not here, in the chapter that discusses the relevant subject)

META: Give a sample listing of sensible ownerships and permissions for all the important files, and *explain* why they are sensible. Explain what must be done in a specific way, and what can be altered to make the system more open or more secure.

META: timezone files, /etc/skel.

META: mention where files / programs come from, if not typically in the distribution

This chapter contains a quick overview of the most important files and directories on a UNIX system. It does not go into detail about the contents of files, only summarizes their purpose, possibly mentions connections to other files and programs, and points to the relevant document that describes things in more detail.

The set of directories and the division of files between directories is based on an assumption that some things are on a **root filesystem**, the first filesystem that is mounted when when Linux boots, while others are on other filesystems, and that some files need to be accessable before those other filesystems are mounted. The 'other' filesystem is usually called `/usr`, and everything else in the root directory is assumed to be on the root filesystem. While this configuration is not true for all systems, especially Linux systems, it was true when the directory tree was originally designed in the early history of UNIX. It is good to understand this, because many things do not otherwise make much sense (e.g. why both `/bin` and `/usr/bin`?).

3.1 The /etc and /usr/etc Directories

The `/etc` directory contains files that have to do with system administration, that are required during bootup or shutdown, that contain system-wide configuration data or are system-wide startup scripts.

The `/usr/etc` directory, on systems where it exists, is similar to `/etc`, but it typically contains only configuration files for programs in `/usr/bin`, not system-wide things. On some systems, however, `/usr/etc` is just a symlink to `/etc`, in which case both are the same thing.

META: the items below should be sorted

`/etc/rc` This is a `/bin/sh` shell script that is automatically run when the system is booted. It should start the background processes that provide useful services to user programs (e.g. `update`, `crond`, `inetd`), mount filesystems, turn on swapping areas, and do other similar tasks. In some installations `/etc/rc.local` or `/etc/rc.[0-9]` are invoked by `/etc/rc`; the intention is that all changes that need to be made for a given site are done in those files so that `/etc/rc` can be automatically updated when a new version of the operating system is installed.

META: Describe /etc/rc.[0-9] better: which is what, and so on; sysvinit: rc.d

`/etc/passwd` This is a text file that contains information about users. See the `passwd(5)` man page for more information.

`/etc/psdatabase` This is used by some versions of the Linux `ps(1)` command. It contains information about where in the kernel memory certain kernel data structures reside. `ps(1)` reads that data directly from `/dev/kmem`, and at least `/etc/psdatabase` needs to be updated when a new version of the kernel is used; sometimes `ps` itself needs recompilation and even changes. Other versions of `ps` use the `/proc` filesystem, and hence do not need any special attention when upgrading the kernel.

META: needed for procps too! wchan

`/etc/disktab` Contains the disk geometry for SCSI disks for those drivers that do not provide the information (for example the generic SCSI driver).

`/etc/fdprm` Floppy disk parameter table. This file describes what different floppy disk formats look like. The program `setfdprm(1)` looks in this file to see. See the man page `fdprm(5)` for more information.

`/etc/fstab` This file lists the filesystems and swap areas that are mounted by the `mount -a` command in `/etc/rc`. See the `mount(8)` man page for more information, and subsection 5.6.5.

META: root fs must be declared correctly, or else there will be problems with libc

`/etc/getty` This is the program that waits for somebody to log in via a terminal (or virtual console). It is started automatically by `init`, once per terminal line (or virtual console) via which it should be possible to log in. Some "terminal" lines (really, serial lines) might not be intended for logins, e.g. when a mouse is connected to that line, so `getty` is not invoked on all lines. `getty` waits until somebody enters a username and then runs `login(1)`.

`/etc/uugetty` Another version of `getty`. See the section on logins for more information.

`/etc/gettydefs` **META:** On my system this is just a symlink to `/etc/gettytab`. sysv uses this name.

`/etc/gettytab` Describes how `getty` should use the terminal lines (speeds, parity settings, and so on).

META: I think this is something like it, gotta find a man page first. bsd uses this name.

META: mail from jwl (1993-07-22)

`/etc/group` This is a file similar to `/etc/passwd`, but it describes groups instead of users. See the `group(5)` man page for more information.

`/etc/init` This is the program that is started as process 1 by the kernel at boot time. After `init` is started, the kernel booting is done. `init` then runs `/etc/rc`, starts the `gettys`, and so on.

`/etc/inittab` This file lists the processes that `init` starts.

META: for some inits, much more info.

`/etc/issue` This file contains the text that `getty` outputs before the login prompt.

`/etc/lilo` This directory contains files for LILO, a program that allows Linux to boot from a harddisk. See the LILO documentation for more information.

`/etc/magic` This is the configuration file for `file(1)`. It contains the descriptions of various file formats based on which `file` guesses the type of the file.

`/etc/motd` This contains the **message of the day** that is automatically copied to the users terminal after his password is validated. It typically contains important messages from the sysadmin, such as warnings about downtime in the future. It is also common to have a short description of the type of the computer and the operating system.

META: .hushlogin?

`/etc/mtab` This file contains information about the currently mounted filesystems. It is removed by `/etc/rc`, set up by `mount -a`, and maintained automatically by the `mount(8)` and `umount(8)` commands, and used when a list of mounted filesystems is needed, for example by `df(1)`.

`/etc/mtools` This is a configuration file for `mtools`, a package for using MS-DOS format disks with UNIX. `mtools` is typically unnecessary with Linux, since Linux itself understands the MS-DOS filesystem (although that driver can be configured out when compiling the kernel, in which case `mtools` is needed).

`/etc/shadow` This file contains the shadow passwords on systems where the shadow password software is installed. Shadow passwords mean that the password is not stored to the world-readable `/etc/passwd` file, but into `/etc/shadow` which only `root` can read. This way it is not possible to get the encrypted version of a password, and hence not possible to decipher it either.

`/etc/login.defs` This is a configuration file for the `login(1)` command.

`/etc/printcap` Like `/etc/termcap`, but intended for printers. Different syntax.

`/etc/profile` This file is sourced at login time by the Bourne shell `/bin/sh` (and `/bin/bash`; although they are the same on most Linux systems), and the Korn shell `/bin/ksh`, before the user's own `.profile` file. This allows the sysadmin to introduce global settings easily.

`/etc/securetty` Identifies secure terminals. `root` can login only from the terminals listed in this file. Typically only the virtual consoles are listed, so that it becomes impossible (or at least harder) to gain superuser privileges by breaking into a system over a modem or a network.

`/etc/shells` Lists trusted shells. The `chsh(1)` command allows users to change their login shell only to shells listed in this file. `ftpd`, the server process that provides FTP services for a machine, will check that the user's shell is listed in `/etc/shells` and will not let people log in unles the shell is listed there.

`/etc/startcons` **META:** old sls? not used anymore? forget it?

/etc/termcap The terminal capability database. Describes by what "escape sequences" various terminals can be controlled. Programs are written so that instead of directly outputting an escape sequence that only works on a particular brand of terminal, they look up the correct sequence to do whatever it is they want to do in /etc/termcap. As a result most programs work with most kinds of terminals. /etc/termcap is a text file; see termcap(5) for more information.

/etc/ttytype Lists default terminal types for terminal lines. Used by login(1).

/etc/update This is one of the background programs that is started by /etc/rc or init(8) (depending on which init is installed). It syncs (forces all unwritten data in the buffer cache to be physically written) every 30 seconds. The idea is to make certain that if there is a power failure, a kernel panic, or some other horrible thing that completely ruins everything, you won't lose more than 30 seconds' worth of writes at the most.

/etc/utmp This is a binary file that records information about who, if anybody, is currently logged in on each terminal, and some other information about the login. Each terminal has its own record in the file. When a user logs in, login(1) updates the record for the terminal in question to show that somebody is logged in, and when he logs out, init(8) records that information.

/etc/wtmp This is like /etc/utmp, except that all the records written are appended instead of overwritten on the existing records. This means that /etc/wtmp grows indefinitely, although slowly. If and when it grows big enough, it will have to be trimmed.

/etc/ftpusers, /etc/ftpaccess, /etc/rpc, /etc/rpcinit, /etc/exports These have to do with networking. See the Linux Network Administrator's Guide for more information.

3.2 Devices, /dev

The /dev directory contains the special device files for all the devices. See the man pages for descriptions for which file stands for which device and what the device is.

META: include list of at least the most important devices

3.3 The Program Directories, `/bin`, `/usr/bin`, and Others

Programs in UNIX are usually scattered in many directories. The two most important ones are `/bin` and `/usr/bin`. Traditionally, all programs intended for the user (as opposed to the sysadmin) to run are in these directories. `/bin` contains the stuff that is needed on the root partition, that is everything which is needed before `/usr` is mounted (assuming it is on a different partition), and also tools useful for recovering from disasters. `/usr/bin` contains most other things.

Usually `/bin` and `/usr/bin` contain programs that are part of the operating system, i.e. they are provided by the OS vendor, not by the user or a third party. Most systems have a place where locally written software, and freeware snarfed from Usenet or other places is installed. This is typically called `/usr/local`, with subdirectories `bin`, `etc`, `lib`, and `man` (and others as necessary). This way those programs are not in the way when a new version of the operating system is installed, especially since the upgrade procedure might wipe out all of `/bin` and `/usr/bin`. (`/usr/local` would then preferably be either a mount point or a symbolic link so that it is not erased during the upgrade.)

In addition to these, many people prefer to install large packages in their own directories so that they can be easily uninstalled, and so that they don't have to worry about overwriting existing files from other packages when installing. It can also be nice to have all files that belong to a package in a central place. An example of this is that, on Linux, X and X programs are usually installed to `/usr/X386`, which is a similar directory tree to `/usr/local`. TeX and GNU Emacs are also usually installed this way on Linux.

On some systems[1] with shared libraries, the directory `/sbin` holds statically linked versions of some of the most important programs. The intent is that if the shared library images become corrupt or are accidentally removed, it is still possible to fix things without having to boot. Typical binaries would be `ln`, `cp`, `mount`, and `sync`.

META: check mail from geyer (1993-07-22) and SunOS

See section 3.5 for an example of how the shared libraries might become fouled up.

The drawback of `/sbin` is that statically linked binaries take a lot more disk space. When a binary of `ln` might be a couple of kilobytes when linked with shared libraries, it might be a couple of hundred kilobytes if linked statically. If you are willing to have to boot if you mess up your shared libraries—and have an emergency boot disk always available—it is never necessary to have `/sbin`. If you can't afford to boot, and

[1] SunOS, and some Linux distributions at least

you can afford the disk space, then `/sbin` is a very good idea. There is not, however, a single answer that is correct for everyone on this issue.

META: hlu's sln is 320 bytes!

3.4 The `/usr/lib` Directory

The `/usr/lib` directory contains code libraries, configuration and data files for individual programs, auxiliary programs that are only invoked by other programs, and other stuff like that. The contents are pretty varied. Some programs want to have their support files directly in `/usr/lib`, some use a subdirectory below that.

`/usr/lib` is used only by programs in `/usr/bin` (programs in `/bin` shouldn't need `/usr/lib`, since `/usr` might not have been mounted). Programs in `/usr/local/bin` use `/usr/local/lib`, and programs that are installed in their own directory tree (like X in `/usr/X386`) use a place in that tree (`/usr/X386/lib`).

Of the files in `/usr/lib`, files named `libsomething.a` are usually libraries of subroutines in some programming language. The most important ones are `libc.a`, the standard C library (the one that contains `printf`, `strcpy`; everything except math stuff), `libm.a` (the math stuff), and `libg.a` (a debugging version of `libc.a`).

Note that on Linux there are several versions of the C libraries, static, and two types of shared libraries. The `libsomething.a` files are the static versions, the shared images are in `libsomething.so.`*version* (usually installed in `lib`, see below) and the stub functions (needed for linking with the shared images) are in `libsomething.sa`.

3.5 Shared Library Images, `/lib`

The `/lib` directory contains the shared images of the shared libraries, that is the actual machine code that is run when a routine in a shared library is called. They are not in `/usr/lib` because `/usr` can be on a different partition and might not be mounted when the shared images are needed (e.g. by programs in `/bin`).

A shared library usually has a name such as `/lib/libc.so.4.3.2`, which in this case would indicate version 4.3.2 of the standard C library. The `.so` bit indicates that it is a shared library image ("shared object", if you prefer).

There can be several versions of a shared library installed at the same time. A typical reason for this is that when a shared library is upgraded, the old version is usually kept around until the new version has proved to be reliable. Also, some programs are linked so that they require a specific version of a shared library. Most programs, however, are linked to use a name like `/lib/libc.so.4`, i.e. using only the major version number. That name is then actually a symlink to whatever version of the shared library that is installed and intended to be used on the system. The contents of `/lib` might then look like

```
cpp -> /usr/lib/gcc-lib/i386-linux/2.3.3/cpp*
libc.so.4 -> libc.so.4.3.2*
libc.so.4.3.2*
libm.so.4 -> libm.so.4.3.2*
libm.so.4.0*
libm.so.4.3.2*
```

Note that there seems to be two versions of the C math library, `libm.so.4.0` and `libm.so.4.3.2`. The symlink `libm.so.4` points at the latter, so that's what gets used unless a program explicitly requires the other one. The reason why both versions are on the disk might be that some program that is used on the system does indeed require that version.[2]

A shared library image can be updated by shutting down the system, booting with a special boot floppy, or by some other means that doesn't mount the normal root filesystem, and then copying the new shared library image to the `/lib` directory and fixing up the symlink. The system can then be brought up in a normal manner. This procedure avoids the problems described below if something goes wrong during the update, since the shared library images on the partition that is being updated aren't ever used.

This is a bit of work, however, so the way it is actually done in practice is to do it while the system is up and running. There is no problem with this, except for a distinct possibility of making it impossible to run any program that uses a shared library, if you make a mistake. The correct way to do it is

`ln -sf /lib/libc.so.`*new* `/lib/libc.so.`*major*

with appropriate values substituted for *new* and *major*. Note that it is imperative that the operation of replacing the old link with the new one is done atomically, in one operation, not as two separate commands. The following is an *incorrect* way of updating the link:

[2] Yup, that's the reason all right. I should know, it's my system...

```
rm -f /lib/libc.so.major      DON'T DO THIS
ln -s /lib/libc.so.new /lib/libc.so.major
```

The problem with the above two commands is that what happens after the `rm`? When `ln` starts it tries to use the shared library `/lib/libc.so.4`, but that was just deleted! Not only can't you create the new link, you can't run any other program that uses the shared library either! If you reboot and use a different root partition (and therefore a different set of shared libraries) to do the update, or if you use some carefully selected statically linked binaries (`ln` comes to mind), you can reduce the probability of this problem, if you are afraid you are going to make the mistake.

META: fixing the missing link; getting rid of old images

3.6 C/C++ Programming, /usr/include, /usr/g++include

META: C++ should be typeset prettier.

The `/usr/include` directory contains the standard headers for the C programming language, or at least most of them (see GCC documentation for details); `/usr/g++include` contains similar headers for C++. (If this doesn't make any sense to you, don't worry, you can probably ignore it in that case.)

The symbolic links `/usr/include/asm` and `/usr/include/linux` point to the similarly named directories in the kernel source tree (see `/usr/src`) for whatever kernel version you are running. They are needed because some type declarations are kernel version dependent. (You may need to update these links when you upgrade the kernel, or the header files.)

3.7 Source Codes, /usr/src

The customary place to keep source code for the kernel and standard programs is `/usr/src`. In particular, `/usr/src/linux` is usually the directory with the source code for the running version of the kernel. Kernel source code, or at least the headers[3], is needed because the headers of the C library refer to headers in the kernel source for kernel version dependent information, such as some types. Because of this, `/usr/include/asm` and `/usr/include/linux` are typically symbolic links to `/usr/src/linux/include/asm` and `linux`, respectively.

[3] If you're short of disk space, deleting everything but the headers can save a couple of megabytes.

3.8 Administrativia, `/usr/adm`

The `/usr/adm` directory contains administrative log files generated by various demons and system programs.

3.9 On-line Manuals, `/usr/man`

On-line man pages are stored below `/usr/man`, at least for programs in `/bin` and `/usr/bin`. Programs installed elsewhere (e.g. `/usr/local`) often get their man pages installed similarly elsewhere as well, but not always.

UNIX man pages are divided into eight numbered chapters (after the chapters in the original UNIX manuals). Each chapter has a subdirectory below `/usr/man`, called `man`*0* where *0* stands for the number of the chapter, and a man page is installed into the appropriate subdirectory. (Man page directories in other places, e.g. `/usr/local/man`, usually have a similar organization.)

META: linux has chapter 9 as well? some man pages need -mandoc instead of -man

In addition to the `troff(1)` source code for the man page in `/usr/man/man?`, its formatted version (formatted for a simple printer or a text screen) is stored into `/usr/man/cat`*0*, where *0* again stands for the chapter. This way, if you need the source (in order to print it on a laser printer, for example), you can get it from `/usr/man/man?`, and if you are satisfied with the pre-formatted version, you can get it quickly from `/usr/man/cat?`. The `man(1)` command uses the correct one automatically: the formatted one if it exists, else it formats the unformatted one on the fly, saving the formatted version in the appropriate place.

Man pages can easily take up a lot of space, so they are sometimes stored in compressed format. On some systems, the `man` program understands compressed files and automatically uncompresses the man page while it is read.

Another way to save space is to not have both formatted and unformatted man pages installed. (Or to not have them installed at all, of course.)

3.10 More Manuals, `/usr/info`

Besides the traditional documentation form for UNIX, man pages, Linux also uses the **Info** system develop by the GNU Project (based on the ITS info system). See Texinfo documentation for more information. The formatted info documents are put into `/usr/info` (or `/usr/emacs/info`, or `/usr/lib/emacs/info`, or `/usr/local/emacs/info`; the possibilities are almost endless, but `/usr/info` is usually either the place itself, or a symlink to the place). The file `dir` in the info directory is the toplevel, or directory, node and should be edited to contain a link to the new document when one is added to the directory.

META: info and texinfo 3 understand compressed man pages; emacs does not

3.11 /home, Sweet /home

Users' home directories are placed in different places on different systems. The oldest convention was to put them in `/usr`, but that is confusing since `/usr` contains a lot of other things as well. `/home` is one common place, and the place most Linuxers seem to prefer (although some prefer to call it `/users`, `/usr/homes`, or something else that isn't `/usr`.) The home directory of user `liw` would then be `/home/liw`.

3.12 Temporary Files, /tmp and /usr/tmp

META: /tmp on root can be a bad idea

Many programs need to create temporary files. In order not to fill the users' directories (and disks, on systems where home directories are on different disks) with such, sometimes large and plentiful, files, the directories `/tmp` and `/usr/tmp` exist. Most programs automatically place temporary files in one of these. Conventions differ from system to system, but generally it is considered better for programs to use `/usr/tmp`, because then the root filesystem (where `/tmp` resides) need not be as large. In fact, some systems even make `/tmp` a symbolic link to `/usr/tmp` (although this only works after `/usr` has been mounted) to force temporary files out of the root filesystem (this greatly reduces the size requirements for the root filesystem). Some large systems mount an especially fast disk on `/tmp` (or `/usr/tmp`); temporary files are usually used for a short time only, or as an extension to physical memory, so having `/tmp` be fast hopefully makes the system faster on the whole.

On many systems, `/tmp` is automatically cleaned (by an appropriate command in `/etc/rc` and/or `cron(8)`), so that the temporary files won't remain and take up space when they are no longer needed. `/usr/tmp` is usually not cleaned automatically (**META:** why?).

3.13 The /mnt and /swap Directories

The `/mnt` directory is the customary mounting point for temporarily mounted filesystems in Linux. It is an empty directory, and the intent is that if you have to, for instance, mount a floppy, you can use `/mnt` as its mount point and don't have to create a new directory each time. It is just for convenience, no programs contain embedded information about `/mnt`. Some people call their `/mnt` something else, such as `/floppy`. Others prefer to have subdirectories below `/mnt`, so that they can have several mount points, e.g. `/mnt/a` for the first floppy drive, or `/mnt/dos` for the MS-DOS hard disk partition.

The `/swap` directory is similarly only a commonly used mount point. As the name implies, it is intended as the mount point for the swap partition. **META:** mount understands 'none' as the mount point, /swap is not needed anymore, but it might be a good place to keep swap files, if there are many

3.14 Process Information, /proc

The `/proc` directory is where the **proc filesystem** is usually mounted. The proc filesystem provides a way in which one can get information about the processes running without having to poke in kernel memory to find it. Poking in kernel memory is both a security hole and inconvenient, since the places and methods for poking vary from kernel version to kernel version, not to mention operating system to operating system. See the Linux Kernel Hackers' Guide for more information.

3.15 The /install Directory

The `/install` directory is specific to the SLS distribution of Linux. It contains files needed for uninstallation of packages installed with the SLS installation tools.

Chapter 4

Boots And Shutdowns

This section explains what goes on when a Linux system is turned on and off, and how it should be done properly.

META: don't power cycle too often

4.1 An Overview of Boots and Shutdowns

META: explain "bootstrapping"

META: boot: bios read boot sector; boot sector reads in kernel; kernel initializes itself and runs init; init runs /etc/rc and/or other scripts and starts gettys and other programs

META: shutdown: all processes are told to quit (sigterm); wait; then kill any that didn't quit (sigkill); sync; unmount all filesystems; halt the system; just cutting the power might cause something to go wrong

4.2 The Boot Process in Closer Look

META: see Linus' notes

You can boot Linux either from a floppy or from the hard disk. The installation section in the Getting Started guide tells you how to install Linux so you can boot it the way you want to.

When the computer is booted, the BIOS will do various tests to check that everything looks all-right[1], and will then start the actual booting. It will choose a disk drive (typically the first floppy drive, if there is a floppy inserted, otherwise the first hard disk, if one is installed in the computer) and read its very first sector. This is called the **boot sector**; for a hard disk, it is also called the **master boot record** (since a hard disk can contain several partitions, each with their own boot sectors).

The boot sector contains a small program (small enough to fit into one sector) whose responsibility it is to read the actual operating system from the disk and start it. When booting Linux from a floppy disk, the boot sector contains code that just reads the first 512 kB (**META:** depends on kernel size) to a predetermined place in memory. (On a Linux boot floppy, there is no filesystem, the kernel is just stored in consecutive sectors, since this simplifies the boot process.)

When booting from the hard disk, the code in the master boot record will examine the partition table, identify the active partition (the partition that is marked to be bootable), read the boot sector from that partition, and then start the code in that boot sector. The code in the partition's boot sector does what a floppy disk's boot sector does: it will read in the kernel from the partition and start it. The details vary, however, since it is generally not useful to have a separate partition for just the kernel image, so the code in the partition's boot sector can't just read the disk in sequential order, it has to find the sectors whereever the filesystem has put them. There are several ways around this problem, but the most common way is to use LILO. (The details about how to do this are irrelevant for this discussion, however; see the LILO documentation for more information.)

When booting with LILO, it will normally go right ahead and read in and boot the default kernel. It is also possible to configure LILO to be able to boot one of several kernels, or even other operating systems than Linux, and it is possible for the user to choose which kernel or operating system is to be booted at boot time. LILO can be configured so that if one holds down the `alt`, `shift`, or `ctrl` key at boot time (i.e. when LILO is loaded), LILO will ask what is to be booted and not boot the default right away. Alternatively, LILO can be configured so that it will always ask, with an optional timeout that will cause the default kernel to be booted.

The are other boot loaders than LILO. However, since LILO has been written especially for Linux, it has some features that are useful and that only it provides, for example the ability to pass arguments to the kernel at boot time, or overriding some configuration options built into the kernel. Hence, it is usually the best choice. Among the alternatives are bootlin and bootactv[2]

[1] These is called the **power on self test**, or **POST** for short.

[2] I don't know much about any of the alternatives. If and when I learn, I will add more descriptions.

Booting from floppy and booting hard disk have both their advantages, but generally booting from the hard disk is easier, since it avoids the hassle of playing around with floppies. It is also faster. However, it can be more troublesome to install the system so it can boot from the hard disk, so many people will first boot from floppy, then, when the system is otherwise installed and working well, will install LILO and start booting from the hard disk.

After the Linux kernel has been read into the memory, by whatever means, and is started for real, roughly the following things happen:

- If the kernel was installed compressed, it will first uncompress itself. (The beginning of the compressed kernel contains a small program that does this.)
- If you have a super-VGA card that Linux thinks it recognizes and that has some special text modes (such as 100 columns by 40 rows[3]), Linux asks you which mode you want to use. (During the kernel compilation, it is possible to preset a video mode, so that this is never asked. This can also be done with LILO.)
- After this the kernel checks what other hardware there is (hard disks, floppies, network adapters...), and configures some of its device drivers appropriately; while it does this, it outputs messages about its findings. For example, when I boot, I it looks like this:

```
LILO boot:
Loading linux.
Console: colour EGA+ 80x25, 8 virtual consoles
Serial driver version 3.94 with no serial options enabled
tty00 at 0x03f8 (irq = 4) is a 16450
tty01 at 0x02f8 (irq = 3) is a 16450
lp_init: lp1 exists (0), using polling driver
Memory: 7332k/8192k available (300k kernel code, 384k reserved, 176k data)
Floppy drive(s): fd0 is 1.44M, fd1 is 1.2M
Loopback device init
Warning WD8013 board not found at i/o = 280.
Math coprocessor using irq13 error reporting.
Partition check:
  hda: hda1 hda2 hda3
VFS: Mounted root (ext filesystem).
Linux version 0.99.pl9-1 (root@haven) 05/01/93 14:12:20
```

  (The exact texts are different on different systems, depending on the hardware, the version of Linux being used, and how it has been configured.)

- After all this configration business is complete, Linux switches the processor into protected mode. The switch is not visible to the user, but is an important step

[3] Incidentally, this is the mode preferred by Linus himself.

for the kernel. A big leap for the kernel, a small step for the userkind.

- Then the kernel will try to mount the root filesystem. The place—which floppy or partition—is configurable at compilation time, with `rdev` (see man page), or with LILO (see LILO documentation). The filesystem type is detected automatically. If the mounting of the root filesystem fails, the kernel panics and halts the system (there isn't much one can do, anyway).

  META: read-only mounting of root fs, then re-mounting after fsck

- After this, the kernel starts the program `/etc/init` in the background (this will always become process number 1), which does various startup chores. The exact things it does depends on the version of `init` being used; see the section on init for more information.

  META: doshell

- The init program then starts a `getty` for virtual consoles and serial lines as configured in `/etc/gettytabs`. `getty` is the program which lets people log in via virtual consoles and serial terminals.

  META: check /etc/gettytabs filename (no s at end?)

- After this, the boot is complete, and the system is up and running normally.

META: explain doshell

4.3 More about Shutdowns

It is important to follow the correct procedures when you shut down a Linux system. If you do not do so, your filesystems may become trashed and/or the files may become scrambled. This is because Linux has a disk cache that won't write things to disk at once, but only at intervals. This greatly improves performance but also means that if you just turn off the power at a whim the cache may hold a lot of data and that what is on the disk may not be a fully working filesystem (because only some things have been written to the disk).

Another reason against just flipping the power switch is that in a multi-tasking system there can be lots of things going on in the background, and shutting the power can be quite disastrous. This is especially true for machines that several people use at the same time.

So, how does one shut down a Linux system properly? The program for doing this is called `/bin/shutdown` or `/etc/shutdown` (the place varies between systems). There are two popular ways of using it.

If you are running a system where you are the only user, the usual way of using shutdown is to quit all running programs, log out on all virtual consoles, log in as `root` on one of them (or stay logged in as `root` if you already are, but you should change to the root directory, to avoid problems with unmounting), then give the command `shutdown -q now` (substitute `now` for a number in minutes if you want a delay, though you usually don't on a single user system).

Alternatively, if your system has many users, the usual way is to use the command `shutdown` *10*, and give a short explanation of why the system is shutting down when prompted to do so. This will warn everybody that the system will shut down in ten minutes (although you can choose another time if you want, of course) and that they'd better get lost or loose data (perhaps not in these words). The warning is automatically repeated a few times before the boot, with shorter and shorter intervals as the time runs out.

META: /etc/shutdown.rc

Using either method, when the shutting down starts after any delays, all filesystems (except the root one) are unmounted, user processes (if anybody is still logged in) are killed, demons are shut down, and generally everything settles down. When that is done, `shutdown` prints out a message that you can power down the machine. Then, *and only then*, should you move your fingers towards the on/off button.

META: root fs is unmounted too?

There is another similar command, called `reboot`, which is identical to `shutdown`, but it boots the machine right away instead of asking you to power down the system. Use `reboot` instead of `shutdown` if that's what you want to do.

META: halt, fastboot, options to shutdown

Sometimes, although rarely on any good system, it is impossible to shut down properly. For instance, if the kernel panics and crashes and burns and generally misbehaves, it might be completely impossible to give any new commands, hence shutting down properly is somewhat difficult, and just about everything you can do is hope that nothing has been too severely damaged and turn off the power. If the troubles are a bit less severe (say, somebody merely hit your keyboard with an axe), and the kernel and the `update` program still run normally, it is probably a good idea to wait a couple of minutes to give `update` a chance to sync the disks, and only cut the power after that.

Some people like to shut down using the command `sync` three times, waiting for the disk I/O to stop, then turn off the power. If there are no running programs, this is about equivalent to using `shutdown`. However, it does not unmount any filesystems (this can lead to problems with the ext2fs "clean filesystem" flag). The triple-sync method is not recommended.

(In case you're wondering: the reason for *three* syncs is that in the early days of UNIX, when the commands were typed separately, that usually gave sufficient time for most disk I/O to be finished.)

Chapter 5

Using Disks and Other Storage Media

On a clear disk you can seek forever.

When you install or upgrade your system, you need to do a fair amount of work on your disks. You have to make filesystems on your disks so that files can be stored on them and reserve space for the different parts of your system.

This chapter explains all these initial activities. Usually, once you get your system set up, you won't have to go through the work again, except for using floppies. You'll need to come back to this chapter if you add a new disk or want to fine-tune your disk usage.

The basic tasks in administering disks are:

- Format your disk. This does various things to prepare it for use, such as checking for bad sectors. (Formatting is nowadays not necessary for most hard disks.)

- Partition a hard disk, if you want to use it for several activities that aren't supposed to interfere with one another. One reason for partitioning is to store different operating systems on the same disk. Another reason is to keep user files separate from system files, which simplifies back-ups and helps protect the system files from corruption.

- Make a filesystem (of a suitable type) on each disk or partition. The disk means nothing to Linux until you make a filesystem; then files can be created and accessed on it.

- Mount different filesystems in a single tree structure, either automatically, or manually as needed. (Manually mounted filesystems usually need to be unmounted manually as well.)

Chapter 6 contains information about virtual memory and disk caching, of which you also need to be aware of when using disks.

This chapter explains what you need to know for hard disks and floppies. Unfortunately, because I lack the equipment, I cannot tell you much about using other types of media, such as tapes or CD-ROM's.

5.1 Two kinds of devices

UNIX, and therefore Linux, recognizes two different kinds of devices: random-access block devices (such as disks), and serial character devices (such as tapes and serial lines). Each supported device is represented in the filesystem as a **device file**. When you read or write a device file, the data comes from or goes to the device it represents. This way no special programs (and no special application programming methodology, such as catching interrupts or polling a serial port) are necessary to access devices; for example, to send a file to the printer, one could just say

```
ttyp5 root ~ $ cat filename > /dev/lp1
ttyp5 root ~ $
```

and the contents of the file are printed (the file must, of course, be in a form that the printer understands). However, since it is not a good idea to have several people cat their files to the printer at the same time, one usually uses a special program to send the files to be printed (usually `lpr(1)`). This program makes sure that only one file is being printed at a time, and will automatically send files to the printer as soon as it finishes with the previous file. Something similar is needed for most devices. In fact, one seldom needs to worry about device files at all.

Since devices show up as files in the filesystem (in the `/dev` directory), it is easy to see just what device files exist, using `ls(1)` or another suitable command. In the output of `ls -l`, the first column contains the type of the file and its permissions. For example, inspecting a serial device gives on my system

```
ttyp5 root ~ $ ls -l /dev/cua0
crw-rw-rw-   1 root     uucp       5,  64 Nov 30  1993 /dev/cua0
ttyp5 root ~ $
```

The first character in the first column, i.e., ‘`c`’ in `crw-rw-rw-` above, tells an informed user the type of the file, in this case a character device. For ordinary files, the first character is ‘`-`’, for directories it is ‘`d`’, and for block devices ‘`b`’; see the `ls(1)` man page for further information.

Note that usually all device files exist even though the device itself might be not be installed. So just because you have a file `/dev/sda`, it doesn't mean that you really do have an SCSI hard disk. Having all the device files makes the installation programs simpler, and makes it easier to add new hardware (there is no need to find out the correct parameters for and create the device files for the new device).

5.2 Hard disks

This subsection introduces terminology related to hard disks. If you already know the terms and concepts, you can skip this subsection.

See figure 5.1 for a schematic picture of the important parts in a hard disk. A hard disk consists of one or more circular **platters**,[1] of which either or both **surfaces** are coated with a magnetic substance used for recording the data. For each surface, there is a **read-write head** that examines or alters the recorded data. The platters rotate on a common axis; a typical rotation speed is 3600 rotations per minute, although high-performance hard disks have higher speeds. The heads move along the radius of the platters; this movement combined with the rotation of the platters allows the head to access all parts of the surfaces.

The processor (CPU) and the actual disk communicate through a **disk controller**. This relieves the rest of the computer from knowing how to use the drive, since the controllers for different types of disks can be made to use the same interface towards the rest of the computer. Therefore, the computer can say just “hey disk, gimme what I want”, instead of a long and complex series of electric signals to move the head to the proper location and waiting for the correct position to come under the head and doing all the other unpleasant stuff necessary. (In reality, the interface to the controller is still complex, but much less so than it would otherwise be.) The controller can also do some other stuff, such as caching, or automatic bad sector replacement.

The above is usually what one needs to understand about the hardware. There is also a bunch of other stuff, such as the motor that rotates the platters and moves the heads, and the electronics that control the operation of the mechanical parts, but that is mostly not relevant for understanding the working principle of a hard disk.

[1] The platters are made of a hard substance, e.g., aluminium, which gives the hard disk its name.

The surfaces are usually divided into concentric rings, called **tracks**, and these in turn are divided into **sectors**. This division is used to specify locations on the hard disk and to allocate disk space to files. To find a given place on the hard disk, one might say "surface 3, track 5, sector 7". Usually the number of sectors is the same for all tracks, but some hard disks put more sectors in outer tracks (all sectors are of the same physical size, so more of them fit in the longer outer tracks). Typically, a sector will hold 512 bytes of data. The disk itself can't handle smaller amounts of data than one sector.

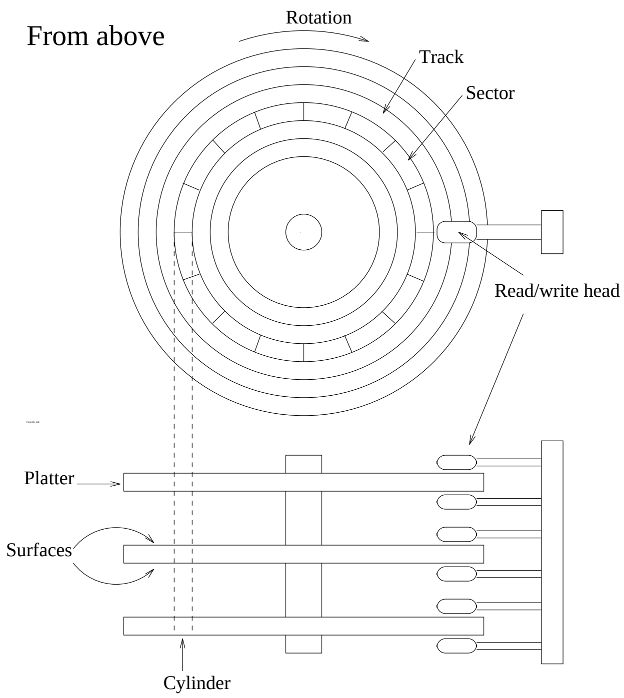

Figure 5.1: A schematic picture of a hard disk.

Each surface is divided into tracks (and sectors) in the same way. This means that when the head for one surface is on a track, the heads for the other surfaces are also on the corresponding tracks. All the corresponding tracks taken together are called a **cylinder**. It takes time to move the heads from one track (cylinder) to another, so by placing the data that is often accessed together (say, a file) so that it is within one cylinder, it is not necessary to move the heads to read all of it. This improves performance. It is not always possible to place files like this; files that are stored in several places on the disk are called **fragmented**.

The number of surfaces (or heads, which is the same thing), cylinders, and sectors vary a lot; the specification of the number of each is called the **geometry** of a hard disk. The geometry is usually stored in a special, battery-powered memory location called the **CMOS RAM**, from where the operating system can fetch it.

Unfortunately, the BIOS[2] has a design limitation, which makes it impossible to specify a track number that is larger than 1024 in the CMOS RAM, which is too little for a large hard disk. To overcome this, the hard disk controller lies about the geometry, and **translates the addresses** given by the computer into something that fits reality. For example, a hard disk might have 8 heads, 2048 tracks, and 35 sectors per track[3]. Its controller could lie to the computer and claim that it has 16 heads, 1024 tracks, and 35 sectors per track, thus not exceeding the limit on tracks, and translates the address that the computer gives it by halving the head number, and doubling the track number. The math can be more complicated in reality, because the numbers are not as nice as here (but again, the details are not relevant for understanding the principle). This translation distorts the operating system's view of how the disk is organized, thus making it impractical to use the all-data-on-one-cylinder trick to boost performance.

The translation is only a problem for IDE disks. SCSI disks use a sequential sector number (i.e., the controller translates a sequential sector number to head/cylinder/sector), and a completely different method for the CPU to talk with the controller, so they are insulated from the problem. Note, however, that the computer might not know the real geometry of an SCSI disk either.

Since Linux often will not know the real geometry of a disk, it doesn't even try to keep files within a single cylinder. Instead, it tries to assign sequentially numbered sectors to files, which almost always gives similar performance. The issue is further complicated by on-controller caches, and automatic prefetches done by the controller.

Each hard disk is represented by a separate device file. There can (usually) be only two IDE hard disks. These are known as `/dev/hda` and `/dev/hdb`, respectively. SCSI hard disks are known as `/dev/sda`, `/dev/sdb`, and so on. Similar naming conventions exist for other hard disk types Note that the device files for the hard disks give access to the entire disk, with no regard to partitions (which will be discussed below), and it's easy to mess up the partitions or the data in them if you aren't careful. The disks' device files are usually used only to get access to the master boot record (which will also be discussed below).

---

[2] The BIOS is some built-in software stored on ROM chips. It takes care, among other things, of the initial stages of booting.

[3] The numbers are completely imaginary.

5.3 Floppies

A floppy disk consists of a flexible membrane covered on one or both sides with similar magnetic substance as a hard disk. The floppy disk itself doesn't have a read-write head, that is included in the drive. A floppy corresponds to one platter in a hard disk, but is removable and one drive can be used to access different floppies, whereas the hard disk is one indivisible unit.

Like a hard disk, a floppy is divided into tracks and sectors (and the two corresponding tracks on either side of a floppy form a cylinder), but there are many fewer of them than on a hard disk.

A floppy drive can usually use several different types of disks; for example, a $3\frac{1}{2}$ inch drive can use both 720 kB and 1.44 MB disks. Since the drive has to operate a bit differently and the operating system must know how big the disk is, there are many device files for floppy drives, one per combination of drive and disk type. Therefore, `/dev/fd0H1440` is the first floppy drive (`fd0`), which must be a $3\frac{1}{2}$ inch drive, using a $3\frac{1}{2}$ inch, high density disk (`H`) of size 1440 kB (`1440`), i.e., a normal $3\frac{1}{2}$ inch HD floppy. For more information on the naming conventions for the floppy devices.

The names for floppy drives are complex, however, and Linux therefore has a special floppy device type that automatically detects the type of the disk in the drive. It works by trying to read the first sector of a newly inserted floppy using different floppy types until it finds the correct one. This naturally requires that the floppy is formatted first. The automatic devices are called `/dev/fd0`, `/dev/fd1`, and so on.

The parameters the automatic device uses to access a disk can also be set using the program `setfdprm(8)`. This can be useful if you need to use disks that do not follow any usual floppy sizes, e.g., if they have an unusual number of sectors, or if the autodetecting for some reason fails and the proper device file is missing.

Linux can handle many nonstandard floppy disk formats in addition to all the standard ones. Some of these require using special formatting programs. We'll skip these disk types for now.

5.4 Formatting

Formatting is the process of writing marks on the magnetic media that are used to mark tracks and sectors. Before a disk is formatted, its magnetic surface is a complete mess of magnetic signals. When it is formatted, some order is brought into the chaos by essentially drawing lines where the tracks go, and where they are divided into sectors. The actual details are not quite exactly like this, but that is irrelevant. What is important, is that a disk cannot be used unless it has been formatted.

The terminology is a bit confusing here: in MS-DOS, the word formatting is used to cover also the process of creating a filesystem (which will be discussed below). There, the two processes are often combined, especially for floppies. When the distinction needs to be made, the real formatting is called **low-level formatting**, while making the filesystem is called **high-level formatting**. In UNIX circles, the two are called formatting and making a filesystem, so that's what is used in this book as well.

For IDE and some SCSI disks the formatting is actually done at the factory and doesn't need to be repeated; hence most people rarely need to worry about it. In fact, formatting a hard disk can cause it to work less well, for example because a disk might need to be formatted in some very special way to allow automatic bad sector replacement to work.

Disks that need or can be formatted, often require a special program anyway, because the interface to the formatting logic inside the drive is different from drive to drive. The formatting program is often either on the controller BIOS, or is supplied as an MS-DOS program; neither of these can easily be used from within Linux.

During formatting one might encounter bad spots on the disk, called **bad blocks** or **bad sectors**. These are sometimes handled by the drive itself, but even then, if more of them develop, something needs to be done to avoid using those parts of the disk. The logic to do this is built into the filesystem; how to add the information into the filesystem is described below. Alternatively, one might create a small partition that covers just the bad part of the disk; this approach might be a good idea if the bad spot is very large, since filesystems can sometimes have trouble with very large bad areas.

Floppies are formatted with `fdformat(8)`. The floppy device file to use is given as the parameter. For example, the following command would format a high density, $3\frac{1}{2}$ inch floppy in the first floppy drive:

```
ttyp5 root ~ $ fdformat /dev/fd0H1440
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
ttyp5 root ~ $
```

Note that if you want to use an autodetecting device (e.g., `/dev/fd0`), you *must* set the parameters of the device with `setfdprm(8)` first. To achieve the same effect as above, one would have to do the following:

```
ttyp5 root ~ $ setfdprm /dev/fd0 1440/1440
ttyp5 root ~ $ fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
ttyp5 root ~ $
```

It is usually more convenient to choose the correct device file that matches the type of the floppy. Note that it is unwise to format floppies to contain more information than what they are designed for.

`fdformat` will also validate the floppy, i.e., check it for bad blocks. It will try a bad block several times (you can usually hear this, the drive noise changes dramatically). If the floppy is only marginally bad (due to dirt on the read/write head, some errors are false signals), `fdformat` won't complain, but a real error will abort the validation process. The kernel will print log messages for each I/O error it finds; these will go to the console or, if `syslog` is being used, to the file `/usr/adm/messages`. `fdformat` itself won't tell where the error is (one usually doesn't care, floppies are cheap enough that a bad one is automatically thrown away).

```
ttyp5 root ~ $ fdformat /dev/fd0H1440
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... read: Unknown error
ttyp5 root ~ $
```

The `badblocks(8)` command can be used to search any disk or partition for bad blocks (including a floppy). It does not format the disk, so it can be used to check even existing filesystems. The example below checks a $3\frac{1}{2}$ inch floppy with two bad blocks.

```
ttyp5 root ~ $ badblocks /dev/fd0H1440 1440
718
719
ttyp5 root ~ $
```

`badblocks` outputs the block numbers of the bad blocks it finds. Most filesystems can avoid such bad blocks. They maintain a list of known bad blocks, which is initialized when the filesystem is made, and can be modified later. The initial search for bad blocks can be done by the `mkfs` command (which initializes the filesystem), but later checks should be done with `badblocks` and the new blocks should be added with `fsck`. We'll describe `mkfs` and `fsck` later.

5.5 Partitions

A hard disk can be divided into several **partitions**. Each partition functions as if it were a separate hard disk. The idea is that if you have one hard disk, and want to have, say, two operating systems on it, you can divide the disk into two partitions. Each operating system uses its partition as it wishes and doesn't touch the other one's. This way the two operating systems can co-exist peacefully on the same hard disk. Without partitions one would have to buy a hard disk for each operating system.

Floppies are not partitioned. There is no technical reason against this, but since they're so small, partitions would be useful only very rarely.

5.5.1 The MBR, boot sectors and partition table

The information about how a hard disk has been partitioned is stored in its first sector (that is, the first sector of the first track on the first disk surface). The first sector is the **master boot record** (MBR) of the disk; this is the sector that the BIOS reads in and starts when the machine is first booted. The master boot record contains a small program that reads the partition table, checks which partition is active (that is, marked bootable), and reads the first sector of that partition, the partition's **boot sector** (the MBR is also a boot sector, but it has a special status and therefore a special name). This boot sector contains another small program that reads the first part of the operating system stored on that partition (assuming it is bootable), and then starts it.

The partitioning scheme is not built into the hardware, or even into the BIOS. It is only a convention that many operating systems follow. Not all operating systems do follow it, but they are the exceptions. Some operating systems support partitions, but they occupy one partition on the hard disk, and use their internal partitioning method within that partition. The latter type exists peacefully with other operating systems (including Linux), and does not require any special measures, but an operating system that doesn't support partitions cannot co-exist on the same disk with any other operating system.

As a safety precaution, it is a good idea to write down the partition table on a piece of paper, so that if it ever corrupts you don't have to lose all your files. (A bad partition table can be fixed with `fdisk`).

5.5.2 Extended and logical partitions

The original partitioning scheme for PC hard disks allowed only four partitions. This quickly turned out to be too little in real life, partly because some people want more than four operating systems (Linux, MS-DOS, OS/2, Minix, FreeBSD, NetBSD, or Windows/NT, to name a few), but primarily because sometimes it is a good idea to have several partitions for one operating system. For example, swap space is usually best put in its own partition for Linux instead of in the main Linux partition for reasons of speed (see below).

To overcome this design problem, **extended partitions** were invented. This trick allows partitioning a **primary partition** into sub-partitions. The primary partition thus subdivided is the extended partition; the subpartitions are **logical partitions**. They behave like primary[4] partitions, but are created differently.

The partition structure of a hard disk might look like that in figure 5.2. The disk is divided into three primary partitions, the second of which is divided into two logical partitions. Part of the disk is not partitioned at all. The disk as a whole and each primary partition has a boot sector.

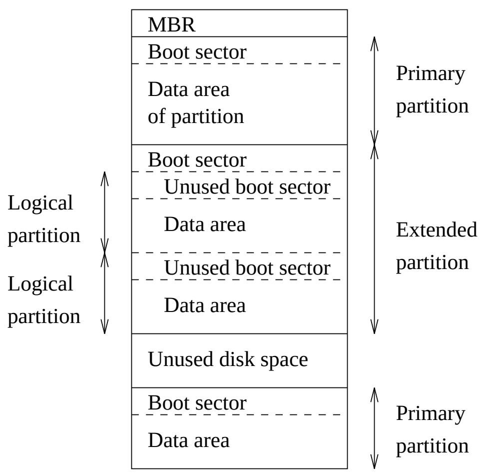

Figure 5.2: A sample hard disk partitioning.

[4] Illogical?

5.5.3 Partition types

The partition tables (the one in the MBR, and the ones for extended partitions) contain one byte per partition that identifies the type of that partition. This attempts to identify the operating system that uses the partition, or what it uses it for. The purpose is to make it possible to avoid having two operating systems accidentally using the same partition. However, in reality, operating systems do not really care about the partition type byte; e.g., Linux doesn't care at all what it is. Worse, some of them use it incorrectly; e.g., at least some versions of DR-DOS ignore the most significant bit of the byte, while others don't.

There is no standardization agency to specify what each byte value means, but some commonly accepted ones are included in the table in table 5.1. The same list is available in the Linux `fdisk(8)` program.

Table 5.1: Partition types (from the Linux `fdisk(8)` program).

0	Empty	40	Venix 80286	94	Amoeba BBT
1	DOS 12-bit FAT	51	Novell?	a5	BSD/386
2	XENIX root	52	Microport	b7	BSDI fs
3	XENIX usr	63	GNU HURD	b8	BSDI swap
4	DOS 16-bit <32M	64	Novell	c7	Syrinx
5	Extended	75	PC/IX	db	CP/M
6	DOS 16-bit ≥32M	80	Old MINIX	e1	DOS access
7	OS/2 HPFS	81	Linux/MINIX	e3	DOS R/O
8	AIX	82	Linux swap	f2	DOS secondary
9	AIX bootable	83	Linux native	ff	BBT
a	OS/2 Boot Manag	93	Amoeba		

5.5.4 Partitioning a hard disk

There are many programs for creating and removing partitions. Most operating systems have their own, and it can be a good idea to use each operating system's own, just in case it does something unusual that the others can't. Many of the programs are called `fdisk`, including the Linux one, or variations thereof. Details on using the Linux `fdisk` are given on its man page. The `cfdisk` command is similar to `fdisk`, but has a nicer (full screen) user interface.

When using IDE disks, the boot partition (the partition with the bootable kernel image files) must be completely within the first 1024 cylinders. This is because the disk is used via the BIOS during boot (before the system goes into protected mode), and BIOS can't handle more than 1024 cylinders. It is sometimes possible to use a boot partition that is only partly within the first 1024 cylinders. This works as long as all the files that are read with the BIOS are within the first 1024 cylinders. Since this is difficult to arrange, it is *a very bad idea* to do it; you never know when a kernel update or disk defragmentation will result in an unbootable system. Therefore, make sure your boot partition is completely within the first 1024 cylinders.

Some newer versions of the BIOS and IDE disks can, in fact, handle disks with more than 1024 cylinders. If you have such a system, you can forget about the problem; if you aren't quite sure of it, put it within the first 1024 cylinders.

Each partition should have an even number of sectors, since the Linux filesystems use a 1 kB block size, i.e., two sectors. An odd number of sectors will result in the last sector being unused. This won't result in any problems, but it is ugly, and some versions of `fdisk` will warn about it.

Changing a partition's size usually requires first backing up everything you want to save from that partition (preferably the whole disk, just in case), deleting the partition, creating new partition, then restoring everything to the new partition. There is a program for MS-DOS, called `fips`, which does this without requiring the backup and restore, but for other filesystems it is still necessary.

5.5.5 Device files and partitions

Each partition and extended partition has its own device file. The naming convention for these files is that a partition's number is appended after the name of the whole disk, with the convention that 1–4 are primary partitions (regardless of how many primary partitions there are) and 5–8 are logical partitions (regardless of within which primary partition they reside). For example, `/dev/hda1` is the first primary partition on the first IDE hard disk, and `/dev/sdb7` is the third extended partition on the second SCSI hard disk.

5.6 Filesystems

5.6.1 What are filesystems?

A **filesystem** is the methods and data structures that an operating uses to keep track of files on a disk or partition that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the filesystem. Thus, one might say "I have two filesystems" meaning one has two partitions on which one stores files, or that one is using the "extended filesystem", meaning the type of the filesystem.

The difference between a disk or partition and the filesystem it contains is important. A few programs—including, reasonably enough, programs that create filesystems—operate directly on the raw sectors of a disk or partition; if there is an existing file system there it can be destroyed or seriously corrupted. Most programs operate on a filesystem, and therefore won't work on a partition that doesn't contain one (or that contains one of the wrong type).

Before a partition or disk can be used as a filesystem, it needs to be initialized, and the bookkeeping data structures need to be written to the disk. This process is called **making a filesystem**.

Most UNIX filesystem types have a similar general structure, although the exact details vary quite a bit. The central concepts are **superblock**, **inode**, **data block**, **directory block**, and **indirection block**. The superblock contains information about the filesystem as a whole, such as its size (the exact information here depends on the filesystem). An inode contains all information about a file, excepts its name. The name is stored in the directory, together with the number of the inode. A directory entry consists of a filename and the number of the inode which represents the file. The inode contains the numbers of several data blocks, which are used to store the data in the file. There is space only for a few data blocks in the inode, however, and if more are needed, more space for pointers to the data blocks is allocated dynamically. These dynamically allocated blocks are indirect blocks; the name indicates that in order to find the data block, one has to find its number in the indirect block first.

UNIX filesystems usually allow one to create a **hole** in a file (this is done with `lseek(2)`; check the manual page), which means that the filesystem just pretends that at a particular place in the file there is just zero bytes, but no actual disk sectors are reserved for that place in the file (this means that the file will use a bit less disk space). This happens especially often for small binaries, Linux shared libraries, some databases, and a few other special cases. (Holes are implemented by storing a special value as the address of the data block in the indirect block or inode. This special address means that no data block is allocated for that part of the file, ergo, there is a hole in the file.)

Holes are moderately useful. On the author's system, a simple measurement showed a potential for about 4 MB of savings through holes of about 200 MB total used disk space. That system, however, contains relatively few programs and no database files. The measurement tool is described in appendix B.

5.6.2 Filesystems galore

Linux supports several types of filesystems. As of this writing the most important ones are:

minix
: The oldest, presumed to be the most reliable, but quite limited in features (some time stamps are missing, at most 30 character filenames) and restricted in capabilities (at most 64 MB per filesystem).

xia
: A modified version of the minix filesystem that lifts the limits on the filenames and filesystem sizes, but does not otherwise introduce new features. It is not very popular, but is reported to work very well.

ext2
: The most featureful of the native Linux filesystems, currently also the most popular one. It is designed to be easily upwards compatible, so that new versions of the filesystem code do not require re-making the existing filesystems.

ext
: An older version of `ext2` that wasn't upwards compatible. It is hardly ever used in new installations any more, and most people have converted to `ext2`.

In addition, support for several foreign filesystem exists, to make it easier to exchange files with other operating systems. These foreign filesystems work just like native ones, except that they may be lacking in some usual UNIX features, or have curious limitations, or other oddities.

msdos
: Compatibility with MS-DOS (and OS/2 and Windows NT) FAT filesystems.

umsdos — Extends the `msdos` filesystem driver under Linux so that Linux can see long filenames, owners, permissions, links, and device files. This allows a normal `msdos` filesystem to be used as if it were a Linux one, thus removing the need for a separate partition for Linux.

iso9660 — The standard CD-ROM filesystem; the popular Rock Ridge extensions to the CD-ROM standard that allow longer file names are supported automatically.

nfs — A networked filesystem that allows sharing a filesystem between many computers to allow easy access to the files from all of them.

hpfs — The OS/2 filesystem.

sysv — SystemV/386, Coherent, and Xenix filesystems.

The choice of filesystem to use depends on the situation. If compatibility or other reasons make one of the non-native filesystems necessary, then that one must be used. If one can choose freely, then it is probably wisest to use ext2, since it has all the features but does not suffer from lack of performance.

There is also the `proc` filesystem, usually accessible as the `/proc` directory, which is not really a filesystem at all, even though it looks like one. The `proc` filesystem makes it easy to access certain kernel data structures, such as the process list (hence the name). It makes these data structures look like a filesystem, and that filesystem can be manipulated with all the usual file tools. For example, to get a listing of all processes one might use the command

```
ttyp5 root ~ $  ls -l /proc
total 0
dr-xr-xr-x   4 root     root            0 Jan 31 20:37 1
dr-xr-xr-x   4 liw      users           0 Jan 31 20:37 63
dr-xr-xr-x   4 liw      users           0 Jan 31 20:37 94
dr-xr-xr-x   4 liw      users           0 Jan 31 20:37 95
dr-xr-xr-x   4 root     users           0 Jan 31 20:37 98
dr-xr-xr-x   4 liw      users           0 Jan 31 20:37 99
-r--r--r--   1 root     root            0 Jan 31 20:37 devices
-r--r--r--   1 root     root            0 Jan 31 20:37 dma
-r--r--r--   1 root     root            0 Jan 31 20:37 filesystems
```

```
-r--r--r--   1 root     root            0 Jan 31 20:37 interrupts
-r--------   1 root     root      8654848 Jan 31 20:37 kcore
-r--r--r--   1 root     root            0 Jan 31 11:50 kmsg
-r--r--r--   1 root     root            0 Jan 31 20:37 ksyms
-r--r--r--   1 root     root            0 Jan 31 11:51 loadavg
-r--r--r--   1 root     root            0 Jan 31 20:37 meminfo
-r--r--r--   1 root     root            0 Jan 31 20:37 modules
dr-xr-xr-x   2 root     root            0 Jan 31 20:37 net
dr-xr-xr-x   4 root     root            0 Jan 31 20:37 self
-r--r--r--   1 root     root            0 Jan 31 20:37 stat
-r--r--r--   1 root     root            0 Jan 31 20:37 uptime
-r--r--r--   1 root     root            0 Jan 31 20:37 version
ttyp5 root ~ $
```

(There will be a few extra files that don't correspond to processes, though. The above example has been shortened.)

Note that even though it is called a filesystem, no part of the `proc` filesystem touches any disk. It exists only in the kernel's imagination. Whenever anyone tries to look at any part of the `proc` filesystem, the kernel makes it look as if the part existed somewhere, even though it doesn't. So, even though there is a multi-megabyte `/proc/kmem` file, it doesn't take any disk space.

5.6.3 Which filesystem should be used?

There is usually little point in using many different filesystems. Currently, ext2fs is the most popular one, and it is probably the wisest choice. Depending on the overhead for bookkeeping structures, speed, (perceived) reliability, compatibility, and various other reasons, it may be advisable to use another file system. This needs to be decided on a case-by-case basis.

5.6.4 Creating a filesystem

Filesystems are created, i.e., initialized, with the `mkfs(8)` command. There is actually a separate program for each filesystem type. `mkfs` is just a front end that runs the appropriate program depending on the desired filesystem type. The type is selected with the `-t fstype` option.

The programs called by `mkfs` have slightly different command line interfaces. The common and most important options are summarized below; see the manual pages for more.

`-t` *fstype* Select the type of the filesystem.

`-c` Search bad bad blocks and initialize the bad block list accordingly.

`-l` *filename* Read the initial bad block list from the file *filename*.

To create an ext2 filesystem on a floppy, one would give the following commands:

```
ttyp6 root ~ $ fdformat -n /dev/fd0H1440
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
ttyp6 root ~ $ badblocks /dev/fd0H1440 1440 > bad-blocks
ttyp6 root ~ $ mkfs -t ext2 -l bad-blocks /dev/fd0H1440
mke2fs 0.5a, 5-Apr-94 for EXT2 FS 0.5, 94/03/10
360 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
360 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
ttyp6 root ~ $
```

First, the floppy was formatted (the `-n` option prevents validation, i.e., bad block checking). Then bad blocks were searched with `badblocks`, with the output redirected to a file, `bad-blocks`. Finally, the filesystem was created, with the bad block list initialized by whatever `badblocks` found.

The `-c` option could have been used with `mkfs` instead of `badblocks` and a separate file. The example below does that.

```
ttyp6 root ~ $ mkfs -t ext2 -c /dev/fd0H1440
mke2fs 0.5a, 5-Apr-94 for EXT2 FS 0.5, 94/03/10
360 inodes, 1440 blocks
```

```
72 blocks (5.00%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
360 inodes per group

Checking for bad blocks (read-only test): done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
ttyp6 root ~ $
```

The `-c` is more convenient than a separate use of `badblocks`, but `badblocks` is necessary for checking after the filesystem has been created.

The process to prepare to filesystems on hard disks or partitions is the same as for floppies, except that the formatting isn't needed.

5.6.5 Mounting and unmounting

Before one can use a filesystem, it has to be **mounted**. The operating system then does various bookkeeping things to make sure that everything works. Since all files in UNIX are in a single directory tree, the mount operation will make it look like the contents of the new filesystem are the contents of an existing subdirectory in some already mounted filesystem.

For example, figure 5.3 shows three separate filesystems, each with their own root directory. When the last two filesystems are mounted below `/home` and `/usr`, respectively, on the first filesystem, we can get a single directory tree, as in figure 5.4.

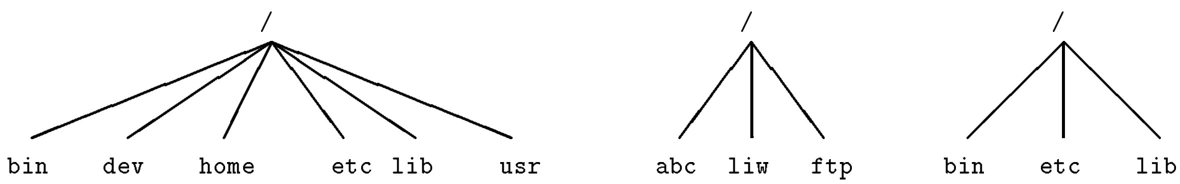

Figure 5.3: Three separate filesystems.

The mounts could be done as in the following example:

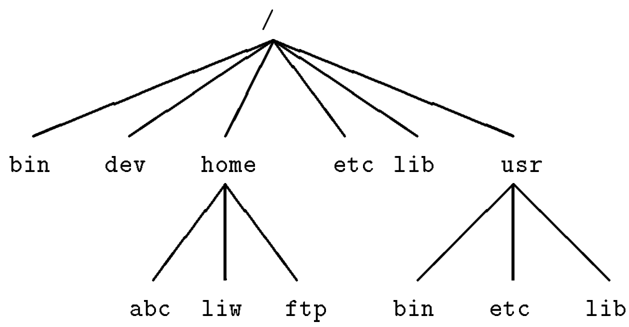

Figure 5.4: `/home` and `/usr` have been mounted.

```
ttyp6 root ~ $  mount /dev/hda2 /home
ttyp6 root ~ $  mount /dev/hda3 /usr
ttyp6 root ~ $
```

The `mount(8)` command takes two arguments. The first one is the device file corresponding to the disk or partition containing the filesystem. The second one is the directory below which it will be mounted. After these commands the contents of the two filesystems look just like the contents of the `/home` and `/usr` directories, respectively. One would then say that "`/dev/hda2` **is mounted on** `/home`", and similarly for `/usr`. To look at either filesystem, one would look at the contents of the directory on which it has been mounted, just as it were any other directory. Note the difference between the device file, `/dev/hda2`, and the mounted-on directory, `/home`. The device file gives access to the raw contents of the disk, the mounted-on directory gives access to the files on the disk. The mounted-on directory is called the **mount point**.

The mounted-on directory need not be empty, although it must exist. Any files in it, however, will be inaccessible by name while the filesystem is mounted. (Any files that have already been opened will still be accessible. Files that have hard links from other directories can be accessed using those names.) There is no harm done with this, and it can even be useful. For instance, some people like to have `/tmp` and `/usr/tmp` synonymous, and make `/tmp` be a symbolic link to `/usr/tmp`. When the system is booted, before the `/usr` filesystem is mounted, a `/usr/tmp` directory residing on the root filesystem is used instead. When `/usr` is mounted, it will make the `/usr/tmp` directory on the root filesystem inaccessible. If `/usr/tmp` didn't exist on the root filesystem, it would be impossible to use temporary files before mounting `/usr`.

If you don't intend to write anything to the filesystem, use the `-r` switch for `mount` to do a **readonly mount**. This will make the kernel stop any attempts at writing to the filesystem, and will also stop the kernel from updating file access times in the inodes. Read-only mounts are necessary for unwritable media, e.g., CD-ROM's.

The alert reader has already noticed a slight logistical problem. How is the first filesystem (called the **root filesystem**, because it contains the root directory) mounted, since it obviously can't be mounted on another filesystem? Well, the answer is that it is done by magic.[5] The root filesystem is magically mounted at boot time, and one can rely on it to always be mounted—if the root filesystem can't be mounted, the system does not boot. The name of the filesystem that is magically mounted as root is either compiled into the kernel, or set using LILO or `rdev`.

The root filesystem is usually first mounted readonly. The startup scripts will then run `fsck(8)` to verify its validity, and if there are no problems, they will **re-mount** it so that writes will also be allowed. `fsck` should not be run on a mounted filesystem, since any changes to the filesystem while `fsck` is running *will* cause trouble. Since the root filesystem is mounted readonly while it is being checked, `fsck` can fix any problems without worry.

On many systems there are other filesystems that should also be mounted automatically at boot time. These are specified in the `/etc/fstab` file; see the `fstab(5)` man page for details on the format. The details of exactly when the extra filesystems are mounted depend on many factors, and can be configured by each administrator if need be. When the chapter on booting is finished, you may read all about it there.

When a filesystem no longer needs to be mounted, it can be unmounted with `umount(8)`[6]. `umount` takes one argument: either the device file or the mount point. For example, to unmount the directories of the previous example, one could use the commands

```
ttyp6 root ~ $ umount /dev/hda2
ttyp6 root ~ $ umount /usr
ttyp6 root ~ $
```

See the man page for further instructions on how to use the command. It is imperative that you always unmount a mounted floppy. *Don't just pop the floppy out of the drive!* Because of disk caching, the data is not necessarily written to the floppy until you unmount it, so removing the floppy from the drive too early might cause the contents

[5]For more information, see the kernel source or the Kernel Hackers' Guide.

[6]It should of course be `unmount(8)`, but the `n` mysteriously disappeared in the 70's, and hasn't been seen since. Please return it to Bell Labs, NJ, if you find it.

to become garbled. If you just read from the floppy, this is not very likely, but if you write, even accidentally, the result may be catastrophic.

Mounting and unmounting requires super user priviledges, i.e., only `root` can do it. The reason for this is that if any user can mount a floppy on any directory, then it is rather easy to create a floppy with, say, a Trojan horse disguised as `/bin/sh`, or any other often used program. However, it is often necessary to allow users to use floppies, and there are several ways to do this:

- Give the users the `root` password. This is obviously bad security, but is the easiest solution. It works well if there is no need for security anyway, which is the case on many non-networked, personal systems.
- Use a program such as `sudo(8)` to allow users to use mount. This is still bad security, but doesn't directly give super user priviledges to everyone.[7]
- Make the users use `mtools`, a package for manipulating MS-DOS filesystems, without mounting them. This works well if MS-DOS floppies are the only thing that is needed, but is rather awkward otherwise.
- List the floppy devices and their allowable mount points together with the suitable options in `/etc/fstab`.

The last alternative can be implemented by adding a line like the following to `/etc/fstab`:

```
/dev/fd0 /floppy msdos user,noauto
```

The columns are: device file to mount, directory to mount on, filesystem type, and options. The `noauto` option stops this mount to be done automatically when the system is started (i.e., it stops `mount -a` from mounting it). The `user` option allows any user to mount the filesystem, and, because of security reasons, disallows execution of programs (normal or setuid) and interpretation of device files from the mounted filesystem. After this, any user can mount a floppy with an `msdos` filesystem with the following command:

```
ttyp6 root ~ $  mount /floppy
ttyp6 root ~ $
```

The floppy can (and needs to, of course) be unmounted with the corresponding `umount` command.

[7] It requires several seconds of hard thinking on the users' behalf.

5.6.6 Keeping filesystems healthy

Filesystems are complex creatures, and as such, they tend to be somewhat error-prone. A filesystem's correctness and validity can be checked using the `fsck(8)` command. It can be instructed to repair any minor problems it finds, and to alert the user if there any unrepairable problems. Fortunately, the code to implement filesystems is debugged quite effectively, so there are seldom any problems at all, and they are usually caused by power failures, failing hardware, or operator errors; for example, by not shutting down the system properly.

Most systems are setup to run `fsck` automatically at boot time, so that any errors are detected (and hopefully corrected) before the system is used. Use of a corrupted filesystem tends to make things worse: if the data structures are messed up, using the filesystem will probably mess them up even more, resulting in more data loss. However, `fsck` can take a while to run on big filesystems, and since errors almost never occur if the system has been shut down properly, a couple of tricks are used to avoid doing the checks in such cases. The first is that if the file `/etc/fastboot` exists, no checks are made. The second is that the ext2 filesystem has a special marker in its superblock that tells whether the filesystem was unmounted properly after the previous mount. This allows `e2fsck` (the version of `fsck` for the ext2 filesystem) to avoid checking the filesystem if the flag indicates that the unmount was done (the assumption being that a proper unmount indicates no problems). Whether the `/etc/fastboot` trick works on your system depends on your startup scripts, but the ext2 trick works every time you use `e2fsck`—it has to be explicitly bypassed with an option to `e2fsck` to be avoided. (See the `e2fsck(8)` man page for details on how.)

The automatic checking only works for the filesystems that are mounted automatically at boot time. Use `fsck` manually to check other filesystems, e.g., floppies.

If `fsck` finds unrepairable problems, you need either in-depth knowlege of how filesystems work in general, and the type of the corrupt filesystem in particular, or good backups. The latter is easy (although sometimes tedious) to arrange, the former can sometimes be arranged via a friend, the Linux newsgroups and mailing lists, or some other source of support, if you don't have the know-how yourself. I'd like to tell you more about it, but my lack of education and experience in this regard hinders me. The `debugfs(8)` program by Theodore T'so should be useful.

`fsck` must only be run on unmounted filesystems, never on mounted filesystems (with the exception of the read-only root during startup). This is because it accesses the raw disk, and can therefore modify the filesystem without the operating system realizing it. There *will* be trouble, if the operating system is confused.

It can be a good idea to periodically check for bad blocks. This is done with the `badblocks` command. It outputs a list of the numbers of all bad blocks it can find. This list can be fed to `fsck` to be recorded in the filesystem data structures so that the operating system won't try to use the bad blocks for storing data. The following example will show how this could be done.

```
ttyp6 root ~ $ badblocks /dev/fd0H1440 1440 > bad-blocks
ttyp6 root ~ $ fsck -t ext2 -l bad-blocks /dev/fd0H1440
Parallelizing fsck version 0.5a (5-Apr-94)
e2fsck 0.5a, 5-Apr-94 for EXT2 FS 0.5, 94/03/10
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Check reference counts.
Pass 5: Checking group summary information.

/dev/fd0H1440: ***** FILE SYSTEM WAS MODIFIED *****
/dev/fd0H1440: 11/360 files, 63/1440 blocks
ttyp6 root ~ $
```

5.7 Disks without filesystems

Not all disks or partitions are used as filesystems. A swap partition, for example, will not have a filesystem on it. Many floppies are used in a tape-drive emulating fashion, so that a `tar` or other file is written directly on the raw disk, without a filesystem. This has the advantages of making more of the disk usable (a filesystem always has some bookkeeping overhead) and more easily compatible with other systems: the `tar` file format is the same on all systems, while filesystems are different on most systems. You will quickly get used to disks without filesystems if you need them. Bootable Linux floppies also do not necessarily have a filesystem, although that is also possible.

One reason to use raw disks is to make image copies of them. For instance, if the disk contains a partially damaged filesystem, it is a good idea to make an exact copy of it before trying to fix it, since then you can start again if your fixing breaks thing even more. One way to do this is to use `dd(1)`:

```
ttyp2 root /usr/tmp $  dd if=/dev/fd0H1440 of=floppy-image
2880+0 records in
```

```
2880+0 records out
ttyp2 root /usr/tmp $  dd if=floppy-image of=/dev/fd0H1440
2880+0 records in
2880+0 records out
ttyp2 root /usr/tmp $
```

The first `dd` makes an exact image of the floppy to the file `floppy-image`, the second one writes the image to the floppy. (The user has presumably switched the floppy before the second command. Otherwise the command pair is of doubtful usefulness.)

5.8 Allocating disk space

5.8.1 Partitioning schemes

It is not easy to partition a disk in the best possible way. Worse, there is no universally correct way to do it; there are too many factors involved.

The traditional way is to have a (relatively) small root filesystem, which contains `/bin`, `/etc`, `/dev`, `/lib`, `/tmp`, and other stuff that is needed to get the system up and running. This way, the root filesystem (in its own partition or on its own disk) is all that is needed to bring up the system. The reasoning is that if the root filesystem is small and is not heavily used, it is less likely to become corrupt when the system crashes, and you will therefore find it easier to fix any problems caused by the crash. Then you create separate partitions or use separate disks for the directory tree below `/usr`, the users' home directories (often under `/home`), and the swap space. Separating the home directories (with the users' files) in their own partition makes backups easier, since it is usually not necessary to backup programs (which reside below `/usr`). In a networked environment it is also possible to share `/usr` among several machines (e.g., by using NFS), thereby reducing the total disk space required by several tens or hundreds of megabytes times the number of machines.

The problem with having many partitions is that it splits the total amount of free disk space into many small pieces. Nowadays, when disks and (hopefully) operating systems are more reliable, many people prefer to have just one partition that holds all their files. On the other hand, it can be less painful to back up (and restore) a small partition.

For a small hard disk (assuming you don't do kernel development), the best way to go is probably to have just one partition. For large hard disks, it is probably better to have a few large partitions, just in case something does go wrong. (Note that 'small' and 'large' are used in a relative sense here; your needs for disk space decide what the threshold is.)

If you have several disks, you might wish to have the root filesystem (including `/usr`) on one, and the users' home directories on another.

It is a good idea to be prepared to experiment a bit with different partitioning schemes (over time, not just while first installing the system). This is a bit of work, since it essentially requires you to install the system from scratch several times, but it is the only way to be sure you do it right.

5.8.2 Space requirements

The Linux distribution you install will give some indication of how much disk space you need for various configurations. Programs installed separately may also do the same. This will help you plan your disk space usage, but you should prepare for the future and reserve some extra space for things you will notice later that you need.

The amount you need for user files depends on what your users wish to do. Most people seem to need as much space for their files as possible, but the amount they will live happily with varies a lot. Some people do only light text processing and will survive nicely with a few megabytes, others do heavy image processing and will need gigabytes.

By the way, when comparing file sizes given in kilobytes or megabytes and disk space given in megabytes, it can be important to know that the two units can be different. Some disk manufacturers like to pretend that a kilobyte is 1000 bytes and a megabyte is 1000 kilobytes, while all the rest of the computing world uses 1024 for both factors. Therefore, my 345 MB hard disk is really a 330 MB hard disk.[8]

Swap space allocation is discusses in section 6.5.

5.8.3 Examples of hard disk allocation

I used to have a 109 MB hard disk, and am now using a 330 MB hard disk. I'll explain how and why I partitioned these disks.

The 109 MB disk I partitioned in a lot of ways, when my needs and the operating systems I used changed; I'll explain two typical scenarios. First, I used to run MS-DOS together with Linux. For that, I needed about 20 MB of hard disk, or just

[8] Sic transit discus mundi.

enough to have MS-DOS, a C compiler, an editor, a few other utilities, the program I was working on, and enough free disk space to not feel claustrophobic. For Linux, I had a 10 MB swap partition, and the rest, or 79 MB, was a single partition with all the files I had under Linux. I experimented with having separate root, `/usr`, and `/home` partitions, but there was never enough free disk space in one piece to do much interesting.

When I didn't need MS-DOS anymore, I repartitioned the disk so that I had a 12 MB swap partition, and again had the rest as a single filesystem.

The 330 MB disk is partitioned into several partitions, like this:

5 MB	root filesystem
10 MB	swap partition
180 MB	`/usr` filesystem
120 MB	`/home` filesystem
15 MB	scratch partition

The scratch partition is for playing around with things that require their own partition, e.g., trying different Linux distributions, or comparing speeds of filesystems. When not needed for anything else, it is used as swap space (I like to have a *lot* of open windows).

5.8.4 Adding more disk space for Linux

Adding more disk space for Linux is easy, at least after the hardware has been properly installed (the hardware installation is outside the scope of this book). You format it if necessary, then create the partitions and filesystem as described above, and add the proper lines to `/etc/fstab` so that it is mounted automatically.

5.8.5 Tips for saving disk space

The best tip for saving disk space is to avoid installing unnecessary programs. Most Linux distributions have an option to install only part of the packages they contain, and by analyzing your needs you might notice that you don't need most of them. This will help save a lot of disk space, since many programs are quite large. Even if you do need a particular package or program, you might not need all of it. For example, some on-line documentation might be unnecessary, as might some of the Elisp files for GNU Emacs, some of the fonts for X11, or some of the libraries for programming.

If you cannot uninstall packages, you might look into compression. Compression programs such as `gzip(1)` or `zip(1)` will compress (and uncompress) individual files or groups of files. The `gzexe` system will compress and uncompress programs invisibly to the user (unused programs are compressed, then uncompressed as they are used). The experimental `DouBle` system will compress all files in a filesystem, invisibly to the programs that use them. (If you are familiar with products such as Stacker for MS-DOS, the principle is the same.)

Chapter 6

Memory Management

Minnet, jag har tappat mitt minne,
är jag svensk eller finne
kommer inte ihåg...

Inne, är jag ute eller inne
jag har luckor i minnet,
sådär små ALKO-HÅL
Men besinne,
man tätar med det brännvin man får,
fastän minnet och helan går.

(Bosse Österberg)

This section describes the Linux memory management features, i.e., virtual memory and the disk buffer cache. The purpose and workings and the things the system administrator needs to take into consideration are described.

6.1 What is virtual memory?

Linux supports **virtual memory**, that is, using a disk as an extension of RAM so that the effective size of usable memory grows correspondingly. The kernel will write the contents of a currently unused block of memory to the hard disk so that the memory can be used for another purpose. When the original contents are needed again, they are read back into memory. This is all made completely transparent to the user;

programs running under Linux only see the larger amount of memory available and don't notice that parts of them reside on the disk from time to time. Of course, reading and writing the hard disk is slower (on the order of a thousand times slower) than using real memory, so the programs don't run as fast. The part of the hard disk that is used as virtual memory is called the **swap space**.

Linux can use either a normal file in the filesystem or a separate partition for swap space. A swap partition is faster, but it is easier to change the size of a swap file (there's no need to repartition the whole hard disk, and possibly install everything from scratch). When you know how much swap space you need, you should go for a swap partition, but if you are uncertain, you can use a swap file first, use the system for a while so that you can get a feel for how much swap you need, and then make a swap partition when you're confident about its size.

You should also know that Linux allows one to use several swap partitions and/or swap files at the same time. This means that if you only occasionally need an unusual amount of swap space, you can set up an extra swap file at such times, instead of keeping the whole amount allocated all the time.

6.2 Creating a swap area

A swap file is an ordinary file; it is in no way special to the kernel. The only thing that matters to the kernel is that it has no holes, and that it is prepared for use with `mkswap(8)`.

The bit about holes is important. The swap file reserves the disk space so that the kernel can quickly swap out a page without having to go through all the things that are necessary when allocating a disk sector to a file. The kernel merely uses any sectors that have already been allocated to the file. Because a hole in a file means that there are no disk sectors allocated (for that place in the file), it is not good for the kernel to try to use them.

One good way to create the swap file without holes is through the following command:

```
ttyp5 root ~ $ dd if=/dev/zero of=/extra-swap bs=1024 count=1024
1024+0 records in
1024+0 records out
ttyp5 root ~ $
```

where `/extra-swap` is the name of the swap file and the size of is given after the `count=`. It is best for the size to be a multiple of 4, because the kernel writes out **memory pages**, which are 4 kilobytes in size. If the size is not a multiple of 4, the last couple of kilobytes may be unused.

A swap partition is also not special in any way. You create it just like any other partition; the only difference is that it is used as a raw partition, that is, it will not contain any filesystem at all. It is a good idea to mark swap partitions as type 82 (Linux swap); this will the make partition listings clearer, even though it is not strictly necessary to the kernel.

After you have created a swap file or a swap partition, you need to write a signature to its beginning; this contains some administrative information and is used by the kernel. The command to do this is `mkswap(8)`, used like this:

```
ttyp5 root ~ $ mkswap /extra-swap 1024
Setting up swapspace, size = 1044480 bytes
ttyp5 root ~ $
```

Note that the swap space is still not in use yet: it exists, but the kernel does not use it to provide virtual memory.

The Linux memory manager limits the size of each swap area to 127.5 MB. A larger swap space can be created, but only the first 127.5 MB are actually used. You can, however, use up to 16 swap spaces simultaneously, for a total of almost 2 GB.[1]

6.3 Using a swap area

An initialized swap area is taken into use with `swapon(8)`. This command tells the kernel that the swap area can be used. The path to the swap area is given as the argument, so to start swapping on a temporary swap file one might use the following command.

```
swapon /usr/tmp/temporary-swap-file ttyp5 root ~ $ swapon /extra-swap
ttyp5 root ~ $
```

Swap areas can be used automatically by listing them in the `/etc/fstab` file.

```
/dev/hda8 swap swap defaults
```

[1] A gigabyte here, a gigabyte there, pretty soon we start talking about real memory.

The startup scripts will run the command `swapon -a`, which will start swapping on all the swap areas listed in `/etc/fstab`. Therefore, the `swapon` command is usually used only when extra swap is needed.

You can monitor the use of swap areas with `free(1)`. It will tell the total amount of swap space used. The same information is available via `top(1)`, or using the `proc` filesystem in file `/proc/meminfo`. It is currently difficult to get information on the use of a specific swap area.

A swap area can be removed from use with `swapoff(8)`. It is usually not necessary to do it, except for temporary swap areas. Any pages in use in the swap area are swapped in first; if there is not sufficient physical memory to hold them, they will then be swapped out (to some other swap area). If there is not enough virtual memory to hold all of the pages Linux will start to trash; after a long while it should recover, but meanwhile the system is unusable. You should check (e.g., with `free`) that there is enough free memory before removing a swap space from use.

All the swap areas that are used automatically with `swapon -a` can be removed from use with `swapoff -a`; it looks at the file `/etc/fstab` to find what to remove. Any manually used swap areas will remain in use.

Sometimes a lot of swap space can be in use even though there is a lot of free physical memory. This can happen for instance if at one point there is need to swap, but later a big process that occupied much of the physical memory terminates and frees the memory. The swapped-out data is not automatically swapped in until it is needed, so the physical memory may remain free for a long time. There is no need to worry about this, but it can be comforting to know what is happening.

6.4 Sharing swap areas with other operating systems

Virtual memory is built into many operating systems. Since they each need it only when they are running, i.e., never at the same time, the swap areas of all but the currently running one are being wasted. It would be more efficient for them to share a single swap area. This is possible, but can require a bit of hacking. The Tips-HOWTO contains some advice on how to implement this.

6.5 Allocating swap space

Some people will tell you that you should allocate twice as much swap space as you have physical memory, but this is a bogus rule. Here's how to do it properly:

1. Estimate your total memory needs. This is the largest amount of memory you'll probably need at a time, that is the sum of the memory requirements of all the programs you want to run at the same time. This can be done by running at the same time all the programs you are likely to ever be running at the same time.

   For instance, if you want to run X, you should allocate about 8 MB for it, gcc wants several megabytes (some files need an unusually large amount, up to several tens of megabytes, but usually about four should do), and so on. The kernel will use about a megabyte by itself, and the usual shells and other small utilities perhaps a few hundred kilobytes (say a megabyte together). There is no need to try to be exact, rough estimates are fine, but you might want to be on the pessimistic side. Remember that if there are going to be several people using the system at the same time, they are all going to consume memory. (However, if two people run the same program at the same time, the total memory consumption is usually not double, since code pages and shared libraries exist only once.) The `free(8)` and `ps(1)` commands are useful for estimating the memory needs.

2. Add some security to the estimate in step 1. This is because estimates of program sizes will probably be wrong, because you'll probably forget some programs you want to run, and to make certain that you have some extra space just in case. A couple of megabytes should be fine. (It is better to allocate too much than too little swap space, but there's no need to over-do it and allocate the whole disk, since unused swap space is wasted space; see later about adding more swap.) Also, since it is nicer to deal with even numbers, you can round the value up to the next full megabyte.

3. Based on the computations in steps 1 and 2, you know how much memory you'll be needing in total. So, in order to allocate swap space, you just need to subtract the size of your physical memory from the total memory needed, and you know how much swap space you need. (On some versions of UNIX, you need to allocate space for an image of the physical memory as well, so the amount computed in step 2 is what you need and you shouldn't do the subtraction.)

6.6 The buffer cache

Reading from a disk[2] is very slow compared to accessing (real) memory. In addition, it is common to read the same part of a disk several times during relatively short periods of time. For example, one might first read an e-mail message, then read the letter into an editor when replying to it, then make the mail program read it again when copying it to a folder. Or, consider how often the command `ls` might be run on a system with many users. By reading the information from disk only once and then keeping it in memory until no longer needed, one can speed up all but the first read. This is called **disk buffering**, and the memory used for the purpose is called the **buffer cache**.

Since memory is, unfortunately, a finite, nay, scarce resource, the buffer cache usually cannot be big enough (it can't hold all the data one ever wants to use). When the cache fills up, the data that has been unused for the longest time is discarded and the memory thus freed is used for the new data.

Disk buffering works for writes as well. On the one hand, data that is written is often soon read again (e.g., a source code file is saved to a file, then read by the compiler), so putting data that is written in the cache is a good idea. On the other hand, by only putting the data into the cache, not writing it to disk at once, the program that writes runs quicker. The writes can then be done in the background, without slowing down the other programs.

Most operating systems have buffer caches (although they might be called something else), but not all of them work according to the above principles. Some are **write-through**: the data is written to disk at once (it is kept in the cache as well, of course). The cache is called **write-back** if the writes are done at a later time. Write-back is more efficient than write-through, but also a bit more prone to errors: if the machine crashes, or the power is cut at a bad moment, or the floppy is removed from the disk drive before the data in the cache waiting to be written gets written, the changes in the cache are usually lost. This might even mean that the filesystem (if there is one) is not in full working order, perhaps because the unwritten data held important changes to the bookkeeping information. Because of this, you should never turn off the power without using a proper shutdown procedure (see an as yet unwritten chapter), or remove a floppy from the disk drive until it has been unmounted (if it was mounted) or after whatever program is using it has signaled that it is finished and the floppy drive light doesn't shine anymore. The `sync(8)` command **flushes** the buffer, i.e., forces all unwritten data to be written to disk, and can be used when one wants to be sure

[2] Except a RAM disk, for obvious reasons.

that everything is safely written. In traditional UNIX systems, there is a program running in the background which does a sync every 30 seconds, so it is usually not necessary to use `sync`. Linux has an additional daemon, `bdflush(8)`, that does a more imperfect sync more frequently to avoid the sudden freeze due to heavy disk I/O that `sync` sometimes causes.

The cache does not actually buffer files, but blocks, which are the smallest units of disk I/O (under Linux, they are usually 1 kB). This way, also directories, super blocks, other filesystem bookkeeping data, and non-filesystem disks are cached.

The effectiveness of a cache is primarily decided by its size. A small cache is next to useless: it will hold so little data that all all cached data is flushed from the cache before it is reused. The critical size depends on how much data is read and written, and how often the same data is accessed. The only way to know is to experiment.

If the cache is of a fixed size, it is not very good to have it too big, either, because that might make the free memory too small and cause swapping (which is also slow). To make the most efficient use of real memory, Linux automatically uses all free RAM for buffer cache, but also automatically makes the cache smaller when programs need more memory.

Under Linux, you do not need to do anything to make use of the cache, it happens completely automatically. Except for following the proper procedures for shutdown and removing floppies, you do not need to worry about it.

Chapter 7

Backups

Hardware is indeterministically reliable.
Software is deterministically unreliable.
People are indeterministically unreliable.
Nature is deterministically reliable.

This chapter explains about why, how, and when to make backups, and how to restore things from backups.

META: backup strategies, horror stories, what media should be used, always verify, compressed vs uncompressed, discuss everything vs only the "important" stuff, automatic backups, off-site backups

7.1 Why backup?

Your data is valuable. It will cost you time and effort, and therefore money or at least personal grief and tears, to re-create it; sometimes it can't even be re-created, e.g., if it is the results of some measurements. Since it is an investment, you should protect it and take steps to avoid losing it.

There are basically four reasons why you might lose data: hardware failures, software bugs, human action, or natural disasters. Although modern hardware tends to be quite reliable, it can still break seemingly spontaneously. The most critical piece of hardware for storing data is the hard disk, which relies on tiny magnetic fields remaining intact in a world filled with electromagnetic noise. Modern software doesn't even tend to be reliable; a rock solid program is an exception, not a rule. Humans are quite deterministically unreliable, they will either make a mistake, or they will be malicious and destroy data on purpose. Nature might not be evil, but it can wreak havoc even when being good. All in all, it is a small miracle that anything works at all.

Backups are a way to protect the investment in data. By having several copies of the data, it does not matter as much if one is destroyed (the cost is only that of the restoration of the lost data from the backup).

7.2 Things to ponder

META: Considerations for backups: safety, speed, convenience, portability, rate of change, what to backup,

The purpose of backups is to enable you to rebuild the system if necessary. You must weigh all decisions about backups with this in mind. You should consider at least the following points (note that they influence each other to some degree):

- How paranoid to be?

  How much do you trust your system? Can you get the software (operating system, applications) from elsewhere quickly and reliably enough, or should you keep local copies? Can you get compatible versions? Will you be able to use the same hardware and software to restore? What disasters do expect? Do you care about unexpected disasters?

- What to backup?

  Can you get compatible copies of the software from elsewhere, whenever you need them, or should you keep local copies? Will it be easy to reinstall, or should you backup your complete system?

  Important things to remember are non-files, such as the partition table, boot sectors, hardware setups, and well-hidden files, such as many configuration files.

- How often?

  How much and how often does your data change? The more often it changes, the more often you will need to back up.

- What media to use?

  A good backup media is reliable (it won't fail) and portable (it will work on other machines as well).

- What tools to use?

  A good tool is reliable (will detect and correct errors in the backup medium) and portable. It should also be convenient and efficient to use.

- How to verify?

  How much do you trust your backup medium? It's hardware, it too can fail. The backup tools might not work correctly, or you might be making a silly mistake using them, so you'll have to verify your backups, i.e., check that they work correctly.

- How to automate?

  Backing up can be quite tedious, and that makes them unpleasant to do. If you can automate the process as much as possible, it will be much more reliable and, most importantly, it will be done.

7.3 Backup levels

META: "backup volumes", rotation, backup levels, when to backup and how much each time, don't keep all backup media in one place

It is rarely possible to make a full copy of all your files every day. There is too much to copy, and it is too much work and takes too long.

7.4 Backup media

META: backup media: floppies, tapes, worm, mo disks, network disks, removable hard disks, other hard disks

7.5 Various tools

META: an overview of the tools: tar, cpio, afio, tbackup (+newer version), dump, GNU's shell scripts, other shell scripts

7.6 A sample plan for a small system

META: a sample plan for a very simple system

7.7 A more demanding backup plan

META: how I do my backups

Appendix A

Design and Implementation of the Second Extended Filesystem

This appendix is a paper by the designers and implementors of the ext2 filesystem. It was first published in the Proceedings to the First Dutch International Symposium on Linux, ISBN 90 367 0385 9.

Introduction

Linux is a Unix-like operating system, which runs on PC-386 computers. It was implemented first as extension to the Minix operating system [9] and its first versions included support for the Minix filesystem only. The Minix filesystem contains two serious limitations: block addresses are stored in 16 bit integers, thus the maximal filesystem size is restricted to 64 mega bytes, and directories contain fixed-size entries and the maximal file name is 14 characters.

We have designed and implemented two new filesystems that are included in the standard Linux kernel. These filesystems, called "Extended File System" (Ext fs) and "Second Extended File System" (Ext2 fs) raise the limitations and add new features.

In this paper, we describe the history of Linux filesystems. We briefly introduce the fundamental concepts implemented in Unix filesystems. We present the implementation of the Virtual File System layer in Linux and we detail the Second Extended File System kernel code and user mode tools. Last, we present performance measurements made on Linux and BSD filesystems and we conclude with the current status of Ext2fs and the future directions.

A.1 History of Linux filesystems

In its very early days, Linux was cross-developed under the Minix operating system. It was easier to share disks between the two systems than to design a new filesystem, so Linus Torvalds decided to implement support for the Minix filesystem in Linux. The Minix filesystem was an efficient and relatively bug-free piece of software.

However, the restrictions in the design of the Minix filesystem were too limiting, so people started thinking and working on the implementation of new filesystems in Linux.

In order to ease the addition of new filesystems into the Linux kernel, a Virtual File System (VFS) layer was developed. The VFS layer was initially written by Chris Provenzano, and later rewritten by Linus Torvalds before it was integrated into the Linux kernel. It will be described in section A.3 of this paper.

After the integration of the VFS in the kernel, a new filesystem, called the "Extended File System" was implemented in April 1992 and added to Linux 0.96c. This new filesystem removed the two big Minix limitations: its maximal size was 2 giga bytes and the maximal file name size was 255 characters. It was an improvement over the Minix filesystem but some problems were still present in it. There was no support for the separate access, inode modification, and data modification timestamps. The filesystem used linked lists to keep track of free blocks and inodes and this produced bad performances: as the filesystem was used, the lists became unsorted and the filesystem became fragmented.

As a response to these problems, two new filesytems were released in Alpha version in January 1993: the Xia filesystem and the Second Extended File System. The Xia filesystem was heavily based on the Minix filesystem kernel code and only added a few improvements over this filesystem. Basically, it provided long file names, support for bigger partitions and support for the three timestamps. On the other hand, Ext2fs was based on the Extfs code with many reorganizations and many improvements. It had been designed with evolution in mind and contained space for future improvements. It will be described with more details in section A.4.

When the two new filesystems were first released, they provided essentially the same features. Due to its minimal design, Xia fs was more stable than Ext2fs. As the filesystems were used more widely, bugs were fixed in Ext2fs and lots of improvements and new features were integrated. Ext2fs is now very stable and has become the de-facto standard Linux filesystem.

The table A.1 contains a summary of the features provided by the different filesystems.

Table A.1: Summary of the filesystem features

	Minix FS	Ext FS	Ext2 FS	Xia FS
Max FS size	64 MB	2 GB	4 TB	2 GB
Max file size	64 MB	2 GB	2 GB	64 MB
Max file name	16/30 c	255 c	255 c	248 c
3 times support	No	No	Yes	Yes
Extensible	No	No	Yes	No
Var. block size	No	No	Yes	No
Maintained	Yes	No	Yes	?

A.2 Basic File System Concepts

Every Linux filesystem implements a basic set of common concepts derivated from the Unix operating system [2]: files are represented by inodes, directories are simply files containing a list of entries and devices can be accessed by requesting I/O on special files.

A.2.1 Inodes

Each file is represented by a structure, called an inode. Each inode contains the description of the file: file type, access rights, owners, timestamps, size, pointers to data blocks. The addresses of data blocks allocated to a file are stored in its inode. When a user requests an I/O operation on the file, the kernel code converts the current offset to a block number, uses this number as an index in the block addresses table and reads or writes the physical block. Figure A.1 represents the structure of an inode.

A.2.2 Directories

Directories are structured in a hierarchical tree. Each directory can contain files and subdirectories.

Directories are implemented as a special type of files. Actually, a directory is a file containing a list of entries. Each entry contains an inode number and a file name. When a process uses a pathname, the kernel code searchs in the directories to find the

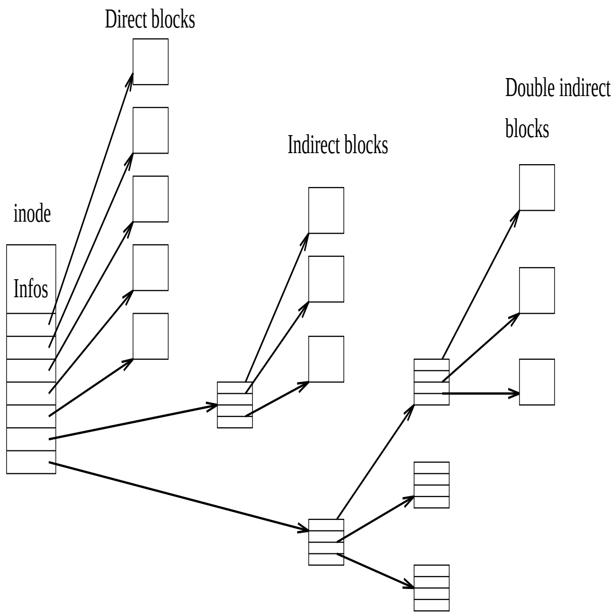

Figure A.1: Structure of an inode

corresponding inode number. After the name has been converted to an inode number, the inode is loaded into memory and is used by subsequent requests.

Figure A.2 represents a directory.

A.2.3 Links

Unix filesystems implement the concept of link. Several names can be associated with a inode. The inode contains a field containing the number associated with the file. Adding a link simply consists in creating a directory entry, where the inode number points to the inode, and in incrementing the links count in the inode. When a link is deleted, i.e. when one uses the `rm` command to remove a filename, the kernel decrements the links count and deallocates the inode if this count becomes zero.

This type of link is called a hard link and can only be used within a single filesystem: it is impossible to create cross-filesystem hard links. Moreover, hard links can only point on files: a directory hard link cannot be created to prevent the apparition of a cycle in the directory tree.

Another kind of links exists in most Unix filesystems. Symbolic links are simply files which contain a filename. When the kernel encounters a symbolic link during a

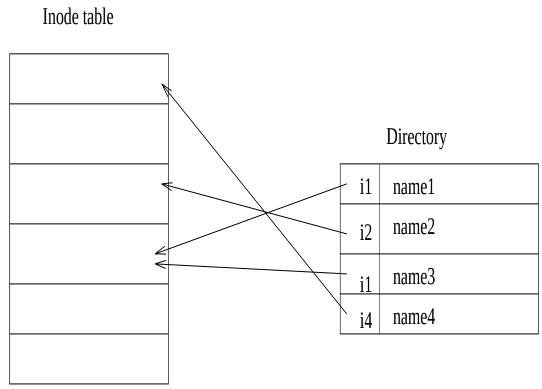

Figure A.2: Structure of a directory

pathname to inode conversion, it replaces the name of the link by its contents, i.e. the name of the target file, and restarts the pathname interpretation. Since a symbolic link does not point to an inode, it is possible to create cross-filesystems symbolic links. Symbolic links can point to any type of file, even on nonexistent files. Symbolic links are very useful because they don't have the limitations associated to hard links. However, they use some disk space, allocated for their inode and their data blocks, and cause an overhead in the pathname to inode conversion because the kernel has to restart the name interpretation when it encounters a symbolic link.

A.2.4 Device special files

In Unix-like operating systems, devices can be accessed via special files. A device special file does not use any space on the filesystem. It is only an access point to the device driver.

Two types of special files exist: character and block special files. The former allows I/O operations in character mode while the later requires data to be written in block mode via the buffer cache functions. When an I/O request is made on a special file, it is forwarded to a (pseudo) device driver. A special file is referenced by a major number, which identifies the device type, and a minor number, which identifies the unit.

A.3 The Virtual File System

A.3.1 Principle

The Linux kernel contains a Virtual File System layer which is used during system calls acting on files. The VFS is an indirection layer which handles the file oriented system calls and calls the necessary functions in the physical filesystem code to do the I/O.

This indirection mechanism is frequently used in Unix-like operating systems to ease the integration and the use of several filesystem types [5, 8].

When a process issues a file oriented system call, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical filesystem code, which is responsible for handling the structure dependent operations. Filesystem code uses the buffer cache functions to request I/O on devices. This scheme is illustrated on figure A.3.

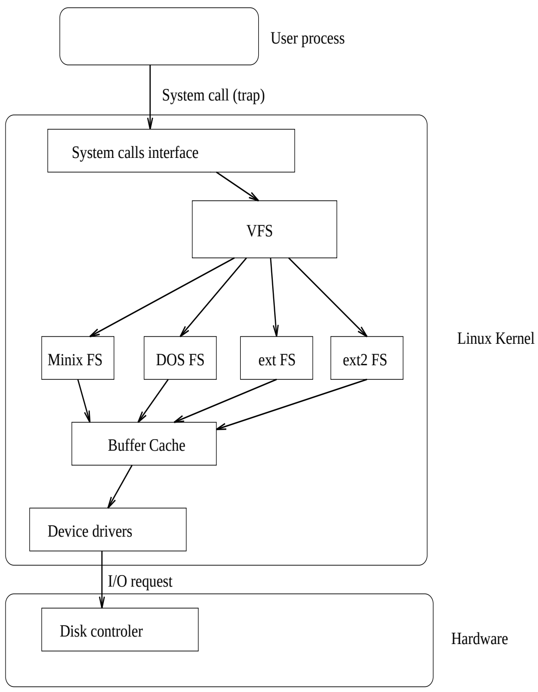

Figure A.3: The VFS Layer

A.3.2 The VFS structure

The VFS defines a set of functions that every filesystem has to implement. This interface is made up of a set of operations associated to three kinds of objects: filesystems, inodes, and open files.

The VFS knows about filesystem types supported in the kernel. It uses a table defined during the kernel configuration. Each entry in this table describes a filesystem type: it contains the name of the filesystem type and a pointer on a function called during the mount operation. When a filesystem is to be mounted, the appropriate mount function is called. This function is responsible for reading the superblock from the disk, initializing its internal variables, and returning a mounted filesystem descriptor to the VFS. After the filesystem is mounted, the VFS functions can use this descriptor to access the physical filesystem routines.

A mounted filesystem descriptor contains several kinds of data: informations that are common to every filesystem types, pointers to functions provided by the physical filesystem kernel code, and private data maintained by the physical filesystem code. The function pointers contained in the filesystem descriptors allow the VFS to access the filesystem internal routines.

Two other types of descriptors are used by the VFS: an inode descriptor and an open file descriptor. Each descriptor contains informations related to files in use and a set of operations provided by the physical filesystem code. While the inode descriptor contains pointers to functions that can be used to act on any file (e.g. `create`, `unlink`), the file descriptors contains pointer to functions which can only act on open files (e.g. `read`, `write`).

A.4 The Second Extended File System

A.4.1 Motivations

The Second Extended File System has been designed and implemented to fix some problems present in the first Extended File System. Our goal was to provide a powerful filesystem, which implements Unix file semantics and offers advanced features.

Of course, we wanted to Ext2fs to have excellent performance. We also wanted to provide a very robust filesystem in order to reduce the risk of data loss in intensive use. Last, but not least, Ext2fs had to include provision for extensions to allow users to benefit from new features without reformatting their filesystem.

A.4.2 "Standard" Ext2fs features

The Ext2fs supports standard Unix file types: regular files, directories, device special files and symbolic links.

Ext2fs is able to manage filesystems created on really big partitions. While the original kernel code restricted the maximal filesystem size to 2 GB, recent work in the VFS layer have raised this limit to 4 TB. Thus, it is now possible to use big disks without the need of creating many partitions.

Ext2fs provides long file names. It uses variable length directory entries. The maximal file name size is 255 characters. This limit could be extended to 1012 if needed.

Ext2fs reserves some blocks for the super user (`root`). Normally, 5% of the blocks are reserved. This allows the administrator to recover easily from situations where user processes fill up filesystems.

A.4.3 "Advanced" Ext2fs features

In addition to the standard Unix features, Ext2fs supports some extensions which are not usually present in Unix filesystems.

File attributes allow the users to modify the kernel behavior when acting on a set of files. One can set attributes on a file or on a directory. In the later case, new files created in the directory inherit these attributes.

BSD or System V Release 4 semantics can be selected at mount time. A mount option allows the administrator to choose the file creation semantics. On a filesystem mounted with BSD semantics, files are created with the same group id as their parent directory. System V semantics are a bit more complex: if a directory has the setgid bit set, new files inherit the group id of the directory and subdirectories inherit the group id and the setgid bit; in the other case, files and subdirectories are created with the primary group id of the calling process.

BSD-like synchronous updates can be used in Ext2fs. A mount option allows the administrator to request that metadata (inodes, bitmap blocks, indirect blocks and directory blocks) be written synchronously on the disk when they are modified. This can be useful to maintain a strict metadata consistency but this leads to poor performances. Actually, this feature is not normally used, since in addition to the performance loss associated with using synchronous updates of the metadata, it can cause corruption in the user data which will not be flagged by the filesystem checker.

Ext2fs allows the administrator to choose the logical block size when creating the filesystem. Block sizes can typically be 1024, 2048 and 4096 bytes. Using big block sizes can speed up I/O since fewer I/O requests, and thus fewer disk head seeks, need to be done to access a file. On the other hand, big blocks waste more disk space: on the average, the last block allocated to a file is only half full, so as blocks get bigger, more space is wasted in the last block of each file. In addition, most of the advantages of larger block sizes are obtained by Ext2 filesystem's preallocation techniques (see section A.4.5).

Ext2fs implements fast symbolic links. A fast symbolic link does not use any data block on the filesystem. The target name is not stored in a data block but in the inode itself. This policy can save some disk space (no data block needs to be allocated) and speeds up link operations (there is no need to read a data block when accessing such a link). Of course, the space available in the inode is limited so not every link can be implemented as a fast symbolic link. The maximal size of the target name in a fast symbolic link is 60 characters. We plan to extend this scheme to small files in a near future.

Ext2fs keeps track of the filesystem state. A special field in the superblock is used by the kernel code to indicate the status of the file system. When a filesystem is mounted in read/write mode, its state is set to "Not Clean". When it is unmounted or remounted in read-only mode, its state is reset to "Clean". At boot time, the filesystem checker uses this information to decide if a filesystem must be checked. The kernel code also records errors in this field. When an inconsistency is detected by the kernel code, the filesystem is marked as "Erroneous". The filesystem checker tests this to force the check of the filesystem regardless of its apparently clean state.

Always skipping filesystem checks may sometimes be dangerous so Ext2fs provides two ways to force checks at regular intervals. A mount counter is maintained in the superblock. Each time the filesystem is mounted in read/write mode, this counter is incremented. When it reaches a maximal value (also recorded in the superblock), the filesystem checker forces the check even if the filesystem is "Clean". A last check time and a maximal check interval are also maintained in the superblock. These two fields allow the administrator to request periodical checks. When the maximal check interval has been reached, the checker ignores the filesystem state and forces a filesystem check.

Ext2fs offers tools to tune the filesystem behavior. The `tune2fs` program can be used to modify:

- the error behavior. When an inconsistency is detected by the kernel code, the filesystem is marked as "Erroneous" and one of the three following actions can

be done: continue normal execution, remount the filesystem in read-only mode to avoid corrupting the filesystem, make the kernel panic and reboot to run the filesystem checker.
- the maximal mount count.
- the maximal check interval.
- the number of logical blocks reserved for the super user.

Mount options can also be used to change the kernel error behavior.

An attribute allows the users to request secure deletion on files. When such a file is deleted, random data is written in the disk blocks previously allocated to the file. This prevents malicious people from gaining access to the previous content of the file by using a disk editor.

Last, new types of files inspired from the 4.4 BSD filesystem have recently been added to Ext2fs. Immutable files can only be read: nobody can write or delete them. This can be used to protect sensitive configuration files. Append-only files can be opened in write mode but data is always appended at the end of the file. Like immutable files, they cannot be deleted or renamed. This is especially useful for log files which can only grow.

A.4.4 Physical Structure

The physical structure of Ext2 filesystems has been strongly influenced by the layout of the BSD filesystem [6]. A filesystem is made up of block groups. Block groups are analogous to BSD FFS's cylinder groups. However, block groups are not tied to the physical layout of the blocks on the disk, since modern drives tend to be optimized for sequential access and hide their physical geometry to the operating system.

The physical structure of a filesystem is represented on figure A.4.

Boot Sector	Block Group 1	Block Group 2	... ...	Block Group N

Figure A.4: Physical structure of an Ext2 filesystem

Each block group contains a redundant copy of crucial filesystem control informations (superblock and the filesystem descriptors) and also contains a part of the filesystem (a block bitmap, an inode bitmap, a piece of the inode table, and data blocks). The structure of a block group is represented on figure A.5.

Super Block	FS descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

Figure A.5: Structure of a block group

Using block groups is a big win in terms of reliability: since the control structures are replicated in each block group, it is easy to recover from a filesystem where the superblock has been corrupted. This structure also helps to get good performances: by reducing the distance between the inode table and the data blocks, it is possible to reduce the disk head seeks during I/O on files.

In Ext2fs, directories are managed as linked lists of variable length entries. Each entry contains the inode number, the entry length, the file name and its length. By using variable length entries, it is possible to implement long file names without wasting disk space in directories. The structure of a directory entry is shown on figure A.6.

inode number	entry length	name length	filename

Figure A.6: Structure of a directory entry

As an example, figure A.7 represents the structure of a directory containing three files: `file1`, `long_file_name`, and `f2`.

i1	16	05	`file1`	i2	40	14	`long_file_name`	i3	12	02	`f2`

Figure A.7: Example of directory

A.4.5 Performance optimizations

The Ext2fs kernel code contains many performance optimizations, which tend to improve I/O speed when reading and writing files.

Ext2fs takes advantage of the buffer cache management by performing readaheads: when a block has to be read, the kernel code requests the I/O on several contiguous blocks. This way, it tries to ensure that the next block to read will already be loaded into the buffer cache. Readaheads are normally performed during sequential reads on files and Ext2fs extends them to directory reads, either explicit reads (`readdir(2)` calls) or implicit ones (`namei` kernel directory lookup).

Ext2fs also contains many allocation optimizations. Block groups are used to cluster together related inodes and data: the kernel code always tries to allocate data blocks for a file in the same group as its inode. This is intended to reduce the disk head seeks made when the kernel reads an inode and its data blocks.

When writing data to a file, Ext2fs preallocates up to 8 adjacent blocks when allocating a new block. Preallocation hit rates are around 75% even on very full filesystems. This preallocation achieves good write performances under heavy load. It also allows contiguous blocks to be allocated to files, thus it speeds up the future sequential reads.

These two allocation optimizations produce a very good locality of:

- related files through block groups
- related blocks through the 8 bits clustering of block allocations.

A.5 The Ext2fs library

To allow user mode programs to manipulate the control structures of an Ext2 filesystem, the libext2fs library was developed. This library provides routines which can be used to examine and modify the data of an Ext2 filesystem, by accessing the filesystem directly through the physical device.

The Ext2fs library was designed to allow maximal code reuse through the use of software abstraction techniques. For example, several different iterators are provided. A program can simply pass in a function to `ext2fs_block_interate()`, which will be called for each block in an inode. Another iterator function allows an user-provided function to be called for each file in a directory.

Many of the Ext2fs utilities (`mke2fs`, `e2fsck`, `tune2fs`, `dumpe2fs`, and `debugfs`) use the Ext2fs library. This greatly simplifies the maintainance of these utilities, since any changes to reflect new features in the Ext2 filesystem format need only be made in one place — in the Ext2fs library. This code reuse also results in smaller binaries, since the Ext2fs library can be built as a shared library image.

Because the interfaces of the Ext2fs library are so abstract and general, new programs which require direct access to the Ext2fs filesystem can very easily be written. For example, the Ext2fs library was used during the port of the 4.4BSD dump and restore backup utilities. Very few changes were needed to adapt these tools to Linux: only a few filesystem dependent functions had to be replaced by calls to the Ext2fs library.

The Ext2fs library provides access to several classes of operations. The first class are the filesystem-oriented operations. A program can open and close a filesystem, read and write the bitmaps, and create a new filesystem on the disk. Functions are also available to manipulate the filesystem's bad blocks list.

The second class of operations affect directories. A caller of the Ext2fs library can create and expand directories, as well as add and remove directory entries. Functions are also provided to both resolve a pathname to an inode number, and to determine a pathname of an inode given its inode number.

The final class of operations are oriented around inodes. It is possible to scan the inode table, read and write inodes, and scan through all of the blocks in an inode. Allocation and deallocation routines are also available and allow user mode programs to allocate and free blocks and inodes.

A.6 The Ext2fs tools

Powerful management tools have been developed for Ext2fs. These utilities are used to create, modify, and correct any inconsistencies in Ext2 filesystems. The `mke2fs` program is used to initialize a partition to contain an empty Ext2 filesystem.

The `tune2fs` program can be used to modify the filesystem parameters. As explained in section A.4.3, it can change the error behavior, the maximal mount count, the maximal check interval, and the number of logical blocks reserved for the super user.

The most interesting tool is probably the filesystem checker. `E2fsck` is intended to repair filesystem inconsistencies after an unclean shutdown of the system. The original version of `e2fsck` was based on Linus Torvald's fsck program for the Minix filesystem. However, the current version of `e2fsck` was rewritten from scratch, using the Ext2fs library, and is much faster and can correct more filesystem inconsistencies than the original version.

The `e2fsck` program is designed to run as quickly as possible. Since filesystem checkers tend to be disk bound, this was done by optimizing the algorithms used by `e2fsck` so that filesystem structures are not repeatedly accessed from the disk. In addition, the order in which inodes and directories are checked are sorted by block number to reduce the amount of time in disk seeks. Many of these ideas were originally explored by [3] although they have since been further refined by the authors.

In pass 1, `e2fsck` iterates over all of the inodes in the filesystem and performs checks over each inode as an unconnected object in the filesystem. That is, these checks do not require any cross-checks to other filesystem objects. Examples of such checks include making sure the file mode is legal, and that all of the blocks in the inode are valid block numbers. During pass 1, bitmaps indicating which blocks and inodes are in use are compiled.

If `e2fsck` notices data blocks which are claimed by more than one inode, it invokes passes 1B through 1D to resolve these conflicts, either by cloning the shared blocks so that each inode has its own copy of the shared block, or by deallocating one or more of the inodes.

Pass 1 takes the longest time to execute, since all of the inodes have to be read into memory and checked. To reduce the I/O time necessary in future passes, critical filesystem information is cached in memory. The most important example of this technique is the location on disk of all of the directory blocks on the filesystem. This obviates the need to re-read the directory inodes structures during pass 2 to obtain this information.

Pass 2 checks directories as unconnected objects. Since directory entries do not span disk blocks, each directory block can be checked individually without reference to other directory blocks. This allows `e2fsck` to sort all of the directory blocks by block number, and check directory blocks in ascending order, thus decreasing disk seek time. The directory blocks are checked to make sure that the directory entries are valid, and contain references to inode numbers which are in use (as determined by pass 1).

For the first directory block in each directory inode, the '.' and '..' entries are checked to make sure they exist, and that the inode number for the '.' entry matches the current directory. (The inode number for the '..' entry is not checked until pass 3.)

Pass 2 also caches information concerning the parent directory in which each directory is linked. (If a directory is referenced by more than one directory, the second reference of the directory is treated as an illegal hard link, and it is removed).

It is noteworthy to note that at the end of pass 2, nearly all of the disk I/O which `e2fsck` needs to perform is complete. Information required by passes 3, 4 and 5 are cached in memory; hence, the remaining passes of `e2fsck` are largely CPU bound, and take less than 5-10% of the total running time of `e2fsck`.

In pass 3, the directory connectivity is checked. `E2fsck` traces the path of each directory back to the root, using information that was cached during pass 2. At this time, the '..' entry for each directory is also checked to make sure it is valid. Any directories which can not be traced back to the root are linked to the `/lost+found` directory.

In pass 4, `e2fsck` checks the reference counts for all inodes, by iterating over all the inodes and comparing the link counts (which were cached in pass 1) against internal counters computed during passes 2 and 3. Any undeleted files with a zero link count is also linked to the `/lost+found` directory during this pass.

Finally, in pass 5, `e2fsck` checks the validity of the filesystem summary information. It compares the block and inode bitmaps which were constructed during the previous passes against the actual bitmaps on the filesystem, and corrects the on-disk copies if necessary.

The filesystem debugger is another useful tool. `Debugfs` is a powerful program which can be used to examine and change the state of a filesystem. Basically, it provides an interactive interface to the Ext2fs library: commands typed by the user are translated into calls to the library routines.

`Debugfs` can be used to examine the internal structures of a filesystem, manually repair a corrupted filesystem, or create test cases for `e2fsck`. Unfortunately, this program can be dangerous if it is used by people who do not know what they are doing; it is very easy to destroy a filesystem with this tool. For this reason, `debugfs` opens filesytems for read-only access by default. The user must explicitly specify the `-w` flag in order to use `debugfs` to open a filesystem for read/wite access.

A.7 Performance Measurements

A.7.1 Description of the benchmarks

We have run benchmarks to measure filesystem performances. Benchmarks have been made on a middle-end PC, based on a i486DX2 processor, using 16 MB of memory and two 420 MB IDE disks. The tests were run on Ext2 fs and Xia fs (Linux 1.1.62) and on the BSD Fast filesystem in asynchronous and synchronous mode (FreeBSD 2.0 Alpha — based on the 4.4BSD Lite distribution).

We have run two different benchmarks. The Bonnie benchmark tests I/O speed on a big file — the file size was set to 60 MB during the tests. It writes data to the file using character based I/O, rewrites the contents of the whole file, writes data using block based I/O, reads the file using character I/O and block I/O, and seeks into the file. The Andrew Benchmark was developed at Carneggie Mellon University and has been used at the University of Berkeley to benchmark BSD FFS and LFS. It runs in five phases: it creates a directory hierarchy, makes a copy of the data, recursively examine the status of every file, examine every byte of every file, and compile several of the files.

A.7.2 Results of the Bonnie benchmark

The results of the Bonnie benchmark are presented in table A.2.

Table A.2: Results of the Bonnie benchmark

	Char Write (KB/s)	Block Write (KB/s)	Rewrite (KB/s)	Char Read (KB/s)	Block Read (KB/s)
BSD Async	710	684	401	721	888
BSD Sync	699	677	400	710	878
Ext2 fs	452	1237	536	397	1033
Xia fs	440	704	380	366	895

The results are very good in block oriented I/O: Ext2 fs outperforms other filesystems. This is clearly a benefit of the optimizations included in the allocation routines. Writes are fast because data is written in cluster mode. Reads are fast because contiguous blocks have been allocated to the file. Thus there is no head seek between two reads and the readahead optimizations can be fully used.

On the other hand, performance is better in the FreeBSD operating system in character oriented I/O. This is probably due to the fact that FreeBSD and Linux do not use the same stdio routines in their respective C libraries. It seems that FreeBSD has a more optimized character I/O library and its performance is better.

A.7.3 Results of the Andrew benchmark

The results of the Andrew benchmark are presented in table A.3.

The results of the two first passes show that Linux benefits from its asynchronous metadata I/O. In passes 1 and 2, directories and files are created and BSD synchronously writes inodes and directory entries. There is an anomaly, though: even in asynchronous mode, the performance under BSD is poor. We suspect that the asynchronous support under FreeBSD is not fully implemented.

Table A.3: Results of the Andrew benchmark

	P1 Create (ms)	P2 Copy (ms)	P3 Stat (ms)	P4 Grep (ms)	P5 Compile (ms)
BSD Async	2203	7391	6319	17466	75314
BSD Sync	2330	7732	6317	17499	75681
Ext2 fs	790	4791	7235	11685	63210
Xia fs	934	5402	8400	12912	66997

In pass 3, the Linux and BSD times are very similar. This is a big progress against the same benchmark run six months ago. While BSD used to outperform Linux by a factor of 3 in this test, the addition of a file name cache in the VFS has fixed this performance problem.

In passes 4 and 5, Linux is faster than FreeBSD mainly because it uses an unified buffer cache management. The buffer cache space can grow when needed and use more memory than the one in FreeBSD, which uses a fixed size buffer cache. Comparison of the Ext2fs and Xiafs results shows that the optimizations included in Ext2fs are really useful: the performance gain between Ext2fs and Xiafs is around 5–10 %.

A.8 Conclusion

The Second Extended File System is probably the most widely used filesystem in the Linux community. It provides standard Unix file semantics and advanced features. Moreover, thanks to the optimizations included in the kernel code, it is robust and offers excellent performance.

Since Ext2fs has been designed with evolution in mind, it contains hooks that can be used to add new features. Some people are working on extensions to the current filesystem: access control lists conforming to the Posix semantics [7], undelete, and on the fly file compression.

Ext2fs was first developed and integrated in the Linux kernel and is now actively being ported to other operating systems. An Ext2fs server running on top of the GNU Hurd has been implemented. People are also working on an Ext2fs port in the LITES server, running on top of the Mach microkernel [1], and in the VSTa operating system. Last, but not least, Ext2fs is an important part of the Masix operating system [4], currently under development by one of the authors.

Acknowledgments

The Ext2fs kernel code and tools have been written mostly by the authors of this paper. Some other people have also contributed to the development of Ext2fs either by suggesting new features or by sending patches. We want to thank these contributors for their help.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the USENIX 1986 Summer Conference*, June 1986.

[2] M. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[3] E. Bina and P. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Conference*, January 1986.

[4] R. Card, E. Commelin, S. Dayras, and F. Mével. The MASIX Multi-Server Operating System. In *OSF Workshop on Microkernel Technology for Distributed Systems*, June 1993.

[5] S. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*, pages 260–269, June 1986.

[6] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 3:181–197, August 1984.

[7] Institute of Electrical and Inc Electronics Engineers. Security interface for the portable operating system interface for computer environments - draft 13, 1992.

[8] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Conference*, January 1993.

[9] A. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

Appendix B

Measuring Holes

This appendix contains the interesting part of the program used to measure the potential for holes in a filesystem. The source distribution of the book contains the full source code.

```
int process(FILE *f, char *filename) {
        static char *buf = NULL;
        static long prev_block_size = -1;
        long zeroes;
        char *p;

        if (buf == NULL || prev_block_size != block_size) {
                free(buf);
                buf = xmalloc(block_size + 1);
                buf[block_size] = 1;
                prev_block_size = block_size;
        }
        zeroes = 0;
        while (fread(buf, block_size, 1, f) == 1) {
                for (p = buf; *p == '\0'; )
                        ++p;
                if (p == buf+block_size)
                        zeroes += block_size;
        }
        if (zeroes > 0)
                printf("%ld %s\n", zeroes, filename);
        if (ferror(f)) {
                errormsg(0, -1, "read failed for `%s'", filename);
                return -1;
        }
        return 0;
}
```

Appendix C

Bibliography

This chapter contains a bibliography of all the books referenced elsewhere in this manual, as well as a number of other books. Where possible, some comments on each book are given. Since the author of this book is unfortunately unable to afford all the books, the comments come from several sources (source is indicated, but withheld if so requested).

META: the c/unix book list should be a good source

META: HD-list

Title:	*The New Hacker's Dictionary*
Author:	Raymond, E. (ed)
Publisher:	MIT Press, 1991
ISBN:	0-262-18145-2 (hc), 0-262-68069-6 (pbk), $10.95

From Lars Wirzenius: A dictionary of the slang and jargon used by hackers. A book version of the *Jargon File*, which contains all the text of the book (typically in a more up-to-date form), and which is in the public domain.

Title:	*UNIX System Administration Handbook*
Author:	Evi Nemeth, Garth Snyder and Scott Seebass
Publisher:	Prentice-Hall, 1989

ISBN: 0-13-933441-6, $?

From Anonymous: I haven't seen any others to compare this one to, so I don't know that I'd particularly recommend it. It does cover both BSD and SYSV, though, so it might be more useful to a Linux sysadmin than a single book that focussed on BSD or SYSV exclusively.

Title: *UNIX Power Tools*
Author: Jerry Peek, Tim O'Reilly, and Mike Loukide
Publisher: Bantam, 1993
ISBN: 0-679-79073-X, $price

From Anonymous: Not a comprehensive guide to much of anything, but it does include a LOT of hints and tips at the sysadmin level. This comes with a CD-ROM full of useful Unix programs, too.