Linux System Administrator's Guide 0.3

Linux System Administrator's Guide 0.3

Lars Wirzenius

The Linux Documentation Project

This is version 0.3 of the Linux System Administrators' Guide.
Published August 6, 1995.

The LaTeX source code and other machine readable formats can be found on the Internet via anonymous ftp on `sunsite.unc.edu`, in the directory `/pub/Linux/docs/LDP`. Also available are Postscript and TeX .DVI formats, and possibly a plain text version (to be released after the other formats). HTML versions may also be forthcoming.

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

This page is dedicated to a future dedication.

Contents

Introduction to the ALPHA Versions

In the beginning, the file was without form, and void; and emptiness was upon the face of the bits. And the Fingers of the Author moved upon the face of the keyboard. And the Author said, Let there be words, and there were words.

This is an ALPHA version of the Linux System Administrators' 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 (on 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.

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.

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 LDP mailing list by Matt Welsh.

Chapter 1

Introduction

I pride myself on the fact that my work has
no socially redeeming value.
(John Waters)

This manual, the Linux System Administrators' 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 have already mastered at least the basics of normal usage, which means roughly the material covered by the Linux Users' Guide. This manual also doesn't tell you how to install Linux; that is described in the Installation and Getting Started document. There is some overlap between all the Linux Documentation Project manuals, 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, say, a house, 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

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

that chapter.[2] 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. 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 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.

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 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, I 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, I 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: I did it because I thought it might be fun and because I felt it should be done. However, like all volunteer work, there is a limit to how much effort I have been able to spend, and also on how much knowledge and experience I 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. I think, of course, that it is pretty nice, but be warned.

One particular point where I have cut corners is that I 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)`. I only describe the purpose of the program, and as much

[2] If you happen to be reading a version that has a chapter on backups, that is.

of its usage as is necessary for the purposes of this manual. For further information, I refer the gentle reader to these other manuals. Usually, all of the referred to documentation is part of the full Linux documentation set.

While I have tried to make this manual as good as possible, I 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, I am interested in all of it. You can contact me 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

Many people have helped me with this book, directly or indirectly. I would like to especially thank Matt Welsh for inspiration and LDP leadership, Andy Oram for igniting an almost dead spark again with much-valued feedback, Olaf Kirch for showing me that it can be done, and Adam Richter at Yggdrasil and others for showing me that other people can find it interesting as well.

H. Peter Anvin, Rémy Card, Theodore Ts'o, and Stephen Tweedie have let me borrow their work (and thus make the book look thicker and much more impressive). I am most grateful for this, and very apologetic for the earlier versions that sometimes lacked proper attribution. Stephen Tweedie also let me borrow his comparison of the xia and ext2 filesystems, but that has since been dropped, since xia is no longer very popular.

In addition, I would like to thank Mark Komarinski for sending his material in 1993 and the many system administration columns in Linux Journal. They are quite informative.

Thanks to Erik Troan at Red Hat, for promising to make a plain text version of this book available.[3]

A minor accusation goes to Linus Torvalds for writing the damn system to write about in the first place. That applies for the rest of `/usr/src/linux/CREDITS` as well. Be ashamed, be ver ashamed.

Many useful comments have been sent by a large number of people. My miniature black hole of an archive doesn't let me find all their names, but some of them are in

[3]Erik, you can color yourself pressurized.

alphabetical order: Paul Caprioli, Ales Cepek, Marie-France Declerfayt, Olaf Flebbe, Helmut Geyer, Larry Greenfield and his father, Stephen Harris, Jyrki Havia, Jim Haynes, York Lam, Timothy Andrew Lister, Jim Lynch, Dan Poirier, Daniel Quinlan, Philippe Steindl. My apologies to anyone I have forgotten.

1.1 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 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 `sunsite.unc.edu`, in the directory `/pub/Linux/docs/LDP`, 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 contact Matt Welsh at `mdw@sunsite.unc.edu`.

Chapter 2

Overview of a Linux System

A quote is needed.

This chapter gives an overivew of a Linux system. First, the major services provided by the operating system are described. Then, the programs that implement these services are described with a considerable lack of detail. The purpose of this chapter is to give an understanding of the system as a whole, so each part is described in detail elsewhere.

2.1 Various parts of an operating system

A UNIX operating system consists of a **kernel** and some **system programs**. There also some **application programs** for doing work. The kernel is the heart of the operating system[1]. It keeps track of files on the disk, starts programs and multiplexes the processor and other hardware between them to provide multitasking, assigns memory and other resources to various processes, receives packets from and sends packets to the network, and so on. The kernel does very little by itself, but it provides tools with which all services can be built. It also prevents anyone from accessing the hardware directly, forcing everyone to use the tools it provides. This way the kernel can control who gets to do what and can provide some protection for users from each other. The tools provided by the kernel are used via **system calls**; see manual page section 2 for more information on these.

The system programs use the tools provided by the kernel to implement the var-

[1] In fact, it is often mistakenly considered to be the operating system itself, but it is not. An operating system provides many more services than a plain kernel.

ious services required from an operating system. System programs, and all other programs, run 'on top of the kernel', in what is called the user mode. The difference between system and application programs is one of intent: applications are intended for getting useful things done (or for playing, if it happens to be a game), whereas system programs are needed to get the system working. A word processor is an application; `telnet` is a system program. The difference is often somewhat blurry, however, and is important only to compulsive categorizers.

An operating system can also contain compilers and their corresponding libraries (GCC and the C library in particular under Linux), although not all programming languages need be part of the operating system. Documentation, and sometimes even games, can also be part of it. Traditionally, the oeprating system has been defined by the contents of the installation tape or disks; with Linux it is not as clear since the stupid thing is spread all over the FTP sites of the world.

2.2 Important parts of the kernel

The Linux kernel consists of several important parts: process management, memory management, hardware device drivers, filesystem drivers, network management, and various other bits and pieces. Figure 2.1 shows some of them.

Probably the most important parts of the kernel (nothing else works without them) are the memory management and the process management. Memory management takes care of assigning memory areas and swap space areas to processes, parts of the kernel, and for the buffer cache. Process management creates processes, and implements the multitasking by switching the active process on the processor.

At the lowest level, the kernel contains a hardware device driver for each kind of hardware it supports. Since the world is full of different kinds of hardware, the number of hardware device drivers is large. There are often many otherwise similar pieces of hardware that differ in how they are controlled by software. The similarities make it possible to have general classes of drivers that support similar operations; each member of the class has the same interface to the rest of the kernel but differs in what it needs to do to implement them. For example, all hard disk drivers look alike to the rest of the kernel, i.e., they all have operations like 'initialize the drive', 'read sector N', and 'write sector N'.

Some software services provided by the kernel itself have similar properties. For example, the various network protocols have been abstracted into one programming interface, the BSD socket library. Another example are the various filesystems Linux

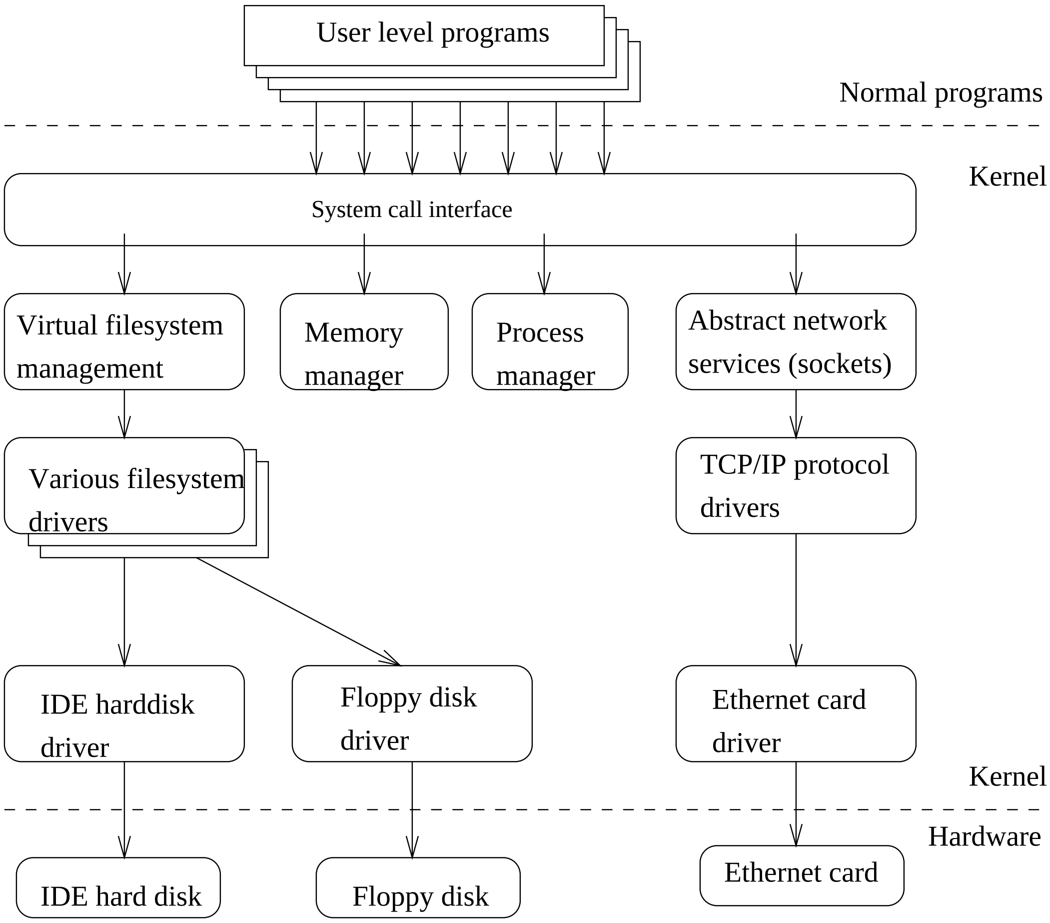

Figure 2.1: Some of the more important parts of the Linux kernel.

supports: the kernel contains a **virtual filesystem** (VFS) that contains all the operations for a filesystem, and a filesystem driver for each supported filesystem. When some entity tries to use a filesystem, the request goes via the VFS, which routes the request to the proper filesystem driver.

2.3 Major services in a UNIX system

This section describes some of the more important UNIX services, but without much detail. They are described more thorougly in later chapters.

2.3.1 `init`

The single most important service in a UNIX system is provided by `init`. `init` is started as the first process of every UNIX system, as the last thing the kernel does when it boots. When `init` starts, it continues the boot process by doing various startup chores (checking and mounting filesystems, starting daemons, etc).

The exact list of things that `init` does depends on which flavor it is; there are several to choose from. `init` usually provides the concept of **single user mode**, in which no one can log in and `root` uses a shell at the console; the usual mode is called **multiuser mode**. Some flavors generalize this as **run levels**; single and multiuser modes are considered to be two run levels, and there can be additional ones as well, for example, to run X on the console.

When the system is running, the two most important tasks of `init` is to make sure `getty`s are working (to make sure logins work), that various daemons are running, and to adopt orphan processes (processes whose parent has died; in UNIX *all* processes *must* be in a single tree, so orphans must be adopted).

When the system is shut down, it is `init` that is in charge of killing all other processes, unmounting all filesystems and stopping the processor, along with anything else that it feels like doing.

2.3.2 Logins from terminals

Logins from terminals (via serial lines) and the console (when not running X) are provided by the `getty` program. `init` starts a separate instance of `getty` for each terminal for which logins are to be allowed. `getty` reads the username and runs the `login` program, which reads the password. If the username and password match, `login` runs the shell. When the shell terminates, i.e., the user logs out, or when `login` terminated because the username and password didn't match, `init` notices this and starts a new instance of `getty`. The kernel has no notion of logins, this is all handled by the system programs.

2.3.3 Syslog

The kernel and many system programs produce error, warning, and other messages. It is often important that these messages can be viewed later, even much later, so they should be written to a file. The program doing this is `syslog`. It can be configured to sort the messages to different files according to writer or degree of importance.

For example, kernel messages are often directed to a separate file from the others, since kernel messages are often more important and need to be read regularly to spot problems.

2.3.4 Periodic command execution: `cron` and `at`

Both users and the system administrator often need to run specific commands periodically. For example, the system administrator might want to run a command to clean the directories with temporary files (`/tmp` and `/var/tmp`) from old files, to keep the disks from filling up, since not all programs clean up after themselves correctly.

The `cron` service is set up to do this. Each user has a `crontab`, where he lists the commands he wants to execute and the times they should be executed. The `crond` daemon takes care of starting the commands when specified.

The `at` service is similar to `cron`, but it is once only: the command is executed at the given time, but it is not repeated.

2.3.5 Graphical user interface

UNIX and Linux don't incorporate the user interface into the kernel; instead, they let it be implemented by user level programs. This applies for both text mode and graphical environments.

This arrangement makes the system more flexible, but has the disadvantage that it is simple to implement a different user interface for each program, making the system harder to learn.

The graphical environment primarily used with Linux is called the X Window System (X for short). X also does not implement a user interface; it only implements a window system, i.e., tools with which a graphical user interface can be implemented. The three most popular user interface styles implemented over X are Athena, Motif, and Open Look.

2.3.6 Networking

Networking is the act of connecting two or more computers so that the can communicate with each other. The actual methods of connecting and communicating are slightly complicated, but the end result is very attractive.

UNIX operating systems have many networking features. Most basic services—filesystems, printing, backups, etc—can be done over the network. This can make system administration easier, since it allows centralized administration, while still reaping in the benefits of microcomputing and distributed computing, such as lower costs and better fault tolerance.

However, this book merely glances at networking; see the Linux Network Administrators' Guide for more information, including a basic descriptions of how networks operate.

2.3.7 Network logins

Network logins work a little differently than normal logins. There is a separate physical serial line for each terminal via which it is possible to log in. For each person logging in via the network, there is a separate virtual network connection, and there can be any number of these[2]. It is therefore not possible to run a separate `getty` for each possible virtual connection. There are also several different ways to log in via network, `telnet` and `rlogin` being the major ones in TCP/IP networks.

Network logins have, instead of a herd of `getty`s, a single daemon (per way of logging in; `telnet` and `rlogin` have separate daemons) that listens for all incoming login attempts. When it notices one, it starts a new instance of itself to handle that single attempt; the original instance continues to listen for other attempts. The new instance works similarly to `getty`.

2.3.8 Network file systems

One of the more useful things that can be done with networking services is sharing files via a **network file system**. The one usually used is called the Network File System, or NFS, developed by Sun.

With a network file system any file operations done by a program on one machine are sent over the network to another computer. This fools the program to think that all the files on the other computer are actually on the computer the program is running on. This makes information sharing extremely simple, since it requires no modifications to programs.

[2] Well, at least there can be many. Network bandwidth still being a scarce resource, there is still some practical upper limit to the number of concurrent logins via one network connection.

2.3.9 Mail

Electronic mail is usually the most important method for communicating via computer. An electronic letter is stored in a file using a special format, and special mail programs are used to send and read the letters.

Each user has an **incoming mailbox** (a file in the special format), where all new mail is stored. When someone sends a mail, the mail program locates the receiver's mailbox and appends the letter to the mailbox file. If the receiver's mailbox is in an another machine, the letter is sent to the other machine, which delivers it to the mailbox as it best sees fit.

The mail system consists of many programs. The delivery of mail to local or remote mailboxes is done by one program (e.g., `sendmail` or `smail`), while the programs users use are many and varied (e.g., `Pine` or `elm`). The mailboxes are usually stored in `/var/spool/mail`.

2.3.10 Printing

Only one person can use a printer at one time, but it is uneconomical not to share printers between users. The printer is therefore managed by software that implements a **print queue**: all print jobs are put into a queue and whenever the printer is done with one job, the next one is sent to it automatically. This relieves the users from organizing the print queue and fighting over control of the printer.[3]

The print queue software also **spools** the printouts on disk, i.e., the text is kept in a file while the job is in the queue. This allows an application program to spit out the print jobs quickly to the print queue software; the application does not have to wait until the job is actually printed to continue. This is really convenient, since it allows one to print out one version, and not have to wait for it to be printed before one can make a completely revised new version.

2.4 The filesystem layout

The filesystem is divided into many parts; usually along the lines of a root filesystem with `/bin`, `/lib`, `/etc`, `/dev`, and a few others; a `/usr` filesystem with programs and unchanging data; a `/var` filesystem with changing data (such as log files); and a `/home`

[3] Instead, they form a new queue *at* the printer, waiting for their printouts, since no-one ever seems to be able to get the queue software to know exactly when anyone's printout is really finished. This is a great boot for intra-office social relations.

filesystem for everyone's personal files. Depending on the hardware configuration and the decisions of the system administrator, the division can be different; it can even be all in one filesystem.

Chapter 5 describes the filesystem layout in some detail; the Linux Filesystem Standard covers it in somewhat more detail.

Chapter 3

Boots And Shutdowns

This chapter needs a quote. Suggestions, anyone?

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

3.1 An overview of boots and shutdowns

The act of turning on a computer system and making its operating system to be loaded[1] is called **booting**. The name comes from an image of the computer pulling itself up from its bootstraps, but the act itself slightly more realistic.

During bootstrapping the computer first loads a small piece of code called the **bootstrap loader**, which in turn loads and starts the operating system. The bootstrap loader is usually stored in a fixed location on a hard disk or a floppy. The reason for this two step process is that the operating system is big and complicated, but the first piece of code that the computer loads must be very small (a few hundred bytes), to avoid making the hardware unnecessarily complicated.

Different computers do the bootstrapping differently. For PC's, the computer (well, it's BIOS) reads in the first sector (called the boot sector) of a floppy or hard disk. The bootstrap loader is contained withing this sector. It loads the operating system from elsewhere on the disk (or from some other place).

After Linux has been loaded, it initializes the hardware and device drivers, and

[1] On early computers, it wasn't enough to merely turn on the computer, you had to manually load the operating system as well. These new-fangled thing-a-ma-gigs do it all by themselves.

then runs `init(8)`. `init` starts other processes to allow users to log in, and do things. The details of this part will be discussed below.

In order to shut down a Linux system, first all processes are told to terminate (this makes them close any files and do other necessary things to keep things tidy), then filesystems and swap areas are unmounted, and finally a message is printed to the console that the power can be turned off. If the proper procedure is not followed, terrible things can and will happen; most importantly, the filesystem buffer cache might not be flushed, which means that all data in it is lost and the filesystem on disk is inconsistent, and therefore possibly unusable.

3.2 The boot process in closer look

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,[2] 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; the order might be configurable, however) and will then 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 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 few hundred blocks (depending on the actual kernel size, of course) 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. It is possible, however, to boot from a floppy with a filesystem, by using LILO.

When booting from the hard disk, the code in the master boot record will examine the partition table (also in the master boot record), 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

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

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, it is most thorough.)

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.[3]

Booting from floppy and from hard disk have both their advantages, but generally booting from the hard disk is nicer, 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:

- The Linux kernel is installed compressed, so it will first uncompress itself. The beginning of the compressed kernel contains a small uncompressed program that does this.
- If you have a super-VGA card that Linux recognizes and that has some special text modes (such as 100 columns by 40 rows), 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 or `rdev(8)`.

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

- 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.

- Then the kernel will try to mount the root filesystem. The place is configurable at compilation time, or any time with `rdev` or LILO. The filesystem type is detected automatically. If the mounting of the root filesystem fails, for example because you didn't remember to include the corresponding filesystem driver in the kernel, the kernel panics and halts the system (there isn't much it can do, anyway).

  The root filesystem is usually mounted read-only (this can be set in the same way as the place). This makes it possible to check the filesystem while it is mounted; it is not a good idea to check a filesystem that is mounted read-write.

- After this, the kernel starts the program `init(8)` (located in `/sbin/init`) in the background (this will always become process number 1). `init` does various startup chores. The exact things it does depends on the version being used; see chapter **??** for more information.

- `init` then starts a `getty(8)` for virtual consoles and serial lines. `getty` is the program which lets people log in via virtual consoles and serial terminals. `init` may also start some other programs, depending on how it is configured.

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

3.3 More about shutdowns

META: two different implemetnations of shutdown? one that uses reboot/halt as internal binaries that shouldn't be run by hand?

It is important to follow the correct procedures when you shut down a Linux system. If you fail do so, your filesystems probably will become trashed and the files probably will 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.

The commands for properly shutting down a Linux system are `shutdown(8)` and `halt(8)` (both are located in `/sbin`). There are two usual ways of using them.

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 `halt` or `shutdown -h now` (substitute `now` with a plus sign and a number in minutes if you want a delay, though you usually don't on a single user system) or `halt`.

Alternatively, if your system has many users, use the command `shutdown -h` *+time message*, where *time* is the time in minutes until the system is halted, and *message* is a short explanation of why the system is shutting down.

```
root   # shutdown -h +10 'We will install a new disk.  System should
> be back on-line in three hours.'
```

This will warn everybody that the system will shut down in ten minutes, and that they'd better get lost or loose data. The warning is printed to every terminal on which someone is logged in, including all `xterm`s.

```
Broadcast message from root (ttyp0) Wed Aug 2 01:03:25 1995...


We will install a new disk.  System should
```

```
be back on-line in three hours.
The system is going DOWN for system halt in 10 minutes !!
```

The warning is automatically repeated a few times before the boot, with shorter and shorter intervals as the time runs out. You can't use a delay with `halt`; it is seldom appropriate to use `halt` on a multiuser system.

META: /etc/inittab can give commands to execute when halting/rebooting

When the real shutting down starts after any delays, all filesystems (except the root one) are unmounted, user processes (if anybody is still logged in) are killed, daemons are shut down, all filesystem are unmounted, 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 power switch.

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(8)` a chance to flush the buffer cache, and only cut the power after that.

Some people like to shut down using the command `sync(8)`[4] 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 and 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.)

3.4 Rebooting

Rebooting means booting the system again. This can be accomplished by first shutting it down completely, turning power off, and then turning it back on. A simpler

[4] `sync` flushes the buffer cache.

way is to ask `shutdown` to reboot the system, instead of merely halting it. This is accomplished by using the `-r` option to shutdown, for example, by giving the command `shutdown -r now`. You can also use the `reboot` command (which, like `halt`, doesn't wait until it perpetrates its foul deed).

3.5 Single user mode

The `shutdown` command can also be used to bring the system down to single user mode, in which no one can log in, but `root` can use the console. This is useful for system administration tasks that can't be done while the system is running normally. Single user mode is discussed more thoroughly in chapter **??**.

3.6 Emergency boot floppies

It is not always possible to boot a computer from the hard disk. For example, if you make a mistake in configuring LILO, you might make your system unbootable. For these situations, you need an alternative way of booting that will always work (as long as the hardware works). For typical PC's, this means booting from the floppy drive.

Most Linux distributions allow one to create an **emergency boot floppy** during installation. It is a good idea to do this. However, many such boot disks contain only the kernel, and assume you will be using the programs on the distributions' installation disks to fix whatever problem you have. Sometimes those programs aren't enough; for example, you might have to restore some files from backups made with software not on the installation disks.

Thus, it might be necessary to create a custom root floppy as well. The *Bootdisk HOWTO* by Graham Chapman contains instructions for doing this. You must, of course, remember to keep your emergency boot and root floppies up to date.

You can't use the floppy drive you use to mount the root floppy for anything else. This can be inconvenient if you only have one floppy. However, if you have enough memory, you can configure your boot floppy to load the root disk to a ramdisk (the boot floppy's kernel needs to be specially configured for this). This frees the floppy drive after the root floppy has been loaded to a ramdisk.

Chapter 4

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 to form 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.

4.1 Two kinds of devices

UNIX, and therefore Linux, recognizes two different kinds of devices: random-access block devices (such as disks), and character devices (such as tapes and serial lines), some of which may be serial, and some random-access. 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).

4.2 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 4.1 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

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

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.

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.

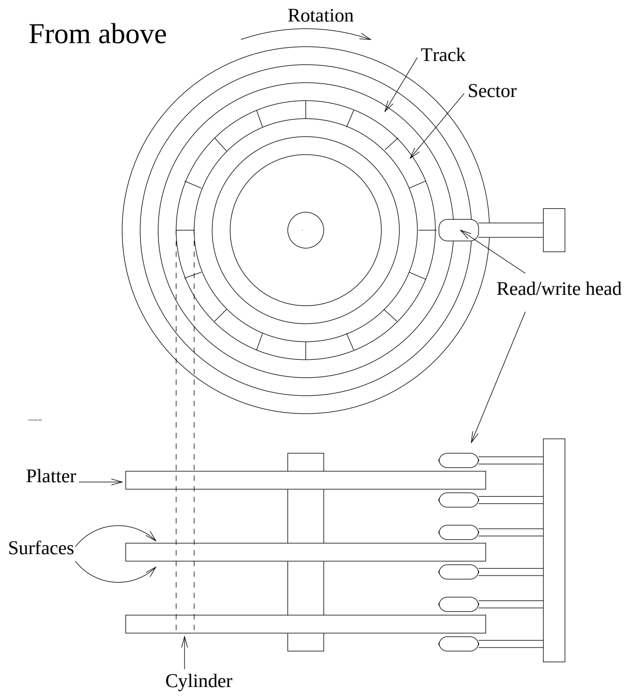

Figure 4.1: A schematic picture of a hard disk.

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 during bootup or driver initialization.

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, its filesystems don'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

[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.

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).

4.3 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.

4.4 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.

4.5 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.

4.5.1 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`).

4.5.2 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 4.2. 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.

4.5.3 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.

[4] Illogical?

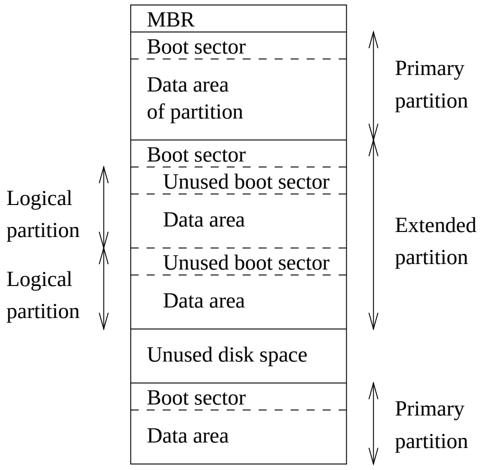

Figure 4.2: A sample hard disk partitioning.

There is no standardization agency to specify what each byte value means, but some commonly accepted ones are included in the table in table 4.1. The same list is available in the Linux `fdisk(8)` program.

Table 4.1: 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		

4.5.4 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.

4.5.5 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.

4.6 Filesystems

4.6.1 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 will 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 block numbers 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.

4.6.2 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 extension to the CD-ROM standard that allow longer file names is 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.

META: ifs, userfs 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.

4.6.3 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.

4.6.4 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.

4.6.5 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 4.3 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 4.4.

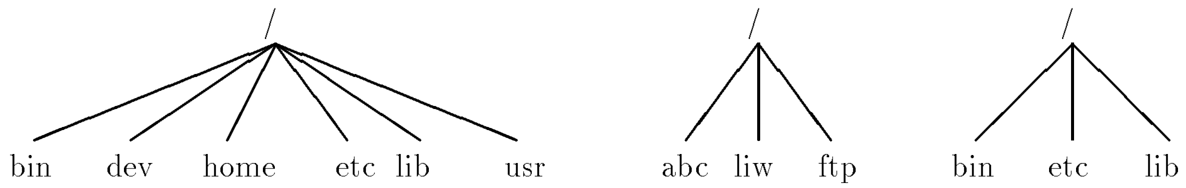

Figure 4.3: Three separate filesystems.

The mounts could be done as in the following example:

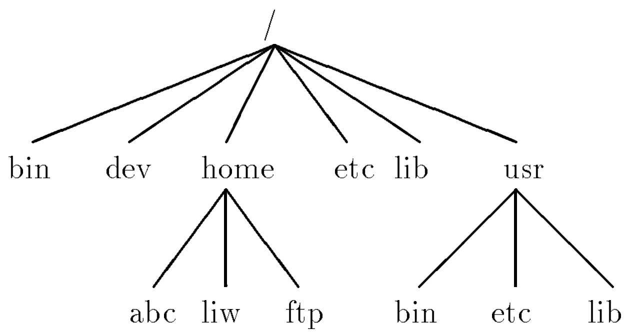

Figure 4.4: `/home` and `/usr` have been mounted.

```
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` must 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, since the remount operation will flush any metadata that the filesystem keeps in memory.

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.

META: What to do if several types of floppies are needed?

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

4.6.6 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 ~ $
```

4.7 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 things 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.)

4.8 Allocating disk space

4.8.1 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.

4.8.2 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.

4.8.3 Examples of hard disk allocation

I used to have a 109 MB hard disk. Now I am 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).

4.8.4 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.

4.8.5 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 5

Directory Tree Overview

This chapter needs a quote. Suggestions, anyone?

This chapter describes the important parts of a standard Linux directory tree, based on the FSSTND filesystem standard. It outlines the normal way of breaking the directory tree into separate filesystems with different purposes and gives the motivation behind this particular split. Some alternative ways of splitting are also described.

META: The next version of the FSSTND (1.3?) will cause many minor changes, and some new ones, due to work to make the FSSTND work for BSD systems as well.

5.1 Background

This chapter is loosely based on the Linux filesystem standard, FSSTND, version 1.2 (see the bibliography), which attempts to set a standard for how the directory tree in a Linux system is organized. Such a standard has the advantage that it will be easier to write or port software for Linux, and to administer Linux machines, since everything will be in their usual places. There is no authority behind the standard that forces anyone to comply to it, but it has got the support of most, if not all Linux distributions. It is not a good idea to break with the FSSTND without very compelling reasons. The FSSTND attempts to follow Unix tradition and current trends, making Linux systems familiar to those with experience with other Unix systems, and vice versa.

This chapter is not as detailed as the FSSTND. A system administrator should also read the FSSTND for a complete understanding.

This chapter does not explain all files in detail. The intention is not to describe every file, but to give an overview of the system from a filesystem point of view. Further information of each file is available elsewhere in this manual or the manual pages.

The full directory tree is intended to be breakable into smaller parts, each on its own disk or partition, to accomodate to disk size limits and to ease backup and other system administration. The major parts are the root, `/usr`, `/var`, and `/home` filesystems. Each part has a different purpose. The directory tree has been designed so that it works well in a network of Linux machines which may share some parts of the filesystems over a read-only device (e.g., a CD-ROM), or over the network with NFS.

The roles of the different parts of the directory tree are described below.

- The root filesystem is specific for each machine (it is generally stored on a local disk, although it could possibly be downloaded to a ramdisk during bootup) and contains the files that are necessary for booting the system up, and to bring it up to such a state that the other filesystems may be mounted. The contents of the root filesystem will therefore be sufficient for the single user state. It will also contain tools for fixing a broken system, and for recovering lost files from backups.

- The `/usr` filesystem contains all commands, libraries, manual pages, and other unchanging files needed during normal operation. No files in `/usr` should be specific for any given machine, nor should they be modified during normal use. This allows the files to be shared over the network, which can be cost-effective since it saves disk space (there can easily be hundreds of megabytes in `/usr`), and can make administration easier (only the master `/usr` needs to be changed when updating an application, not each machine separately). Even if the filesystem is on a local disk, it could be mounted read-only, to lessen the chance of filesystem corruption during a crash.

- The `/var` filesystem contains files that change, such as spool directories (for mail, news, printers, etc), log files, formatted manual pages, and temporary files. Traditionally everything in `/var` has been somewhere below `/usr`, but that made it impossible to mount `/usr` read-only.

- The `/home` filesystem contains the users' home directories, i.e., all the real data on the system. Separating home directories to their own directory tree or filesystem makes backups easier; the other parts often do not have to be backed up, or at least not as often (they seldom change). A big `/home` might have to be broken on several filesystems, which requires adding an extra naming level below `/home`, e.g., `/home/students` and `/home/staff`.

Although the different parts have been called filesystems above, there is no requirement that they actually be on separate filesystems. They could easily be kept in a single one if the system is a small single-user system and he wants to keep things simple. The directory tree might also be divided into filesystems differently, depending on how large the disks are, and how space is allocated for various purposes. The important part, though, is that all the standard *names* work; even if, say, `/var` and `/usr` are actually on the same partition, the names `/usr/lib/libc.a` and `/var/adm/messages` must work, for example by moving files below `/var` into `/usr/var`, and making `/var` a symlink to `/usr/var`.

The Unix filesystem structure groups files according to purpose, i.e., all commands are in one place, all data files in another, documentation in a third, and so on. An alternative would be to group files files according to the program they belong to, i.e., all Emacs files would be in one directory, all TEX in another, and so on. The problem with the latter approach is that it makes it difficult to share files (the program directory often contains both static and shareable and changing and non-shareable files), and sometimes to even find the files (e.g., manual pages in a huge number of places, and making the manual page programs find all of them is a maintenance nightmare).

5.2 The root filesystem

The root filesystem should generally be small, since it contains very critical files and a small, infrequently modified filesystem has a better chance of not getting corrupted. A corrupted root filesystem will generally mean that the system becomes unbootable except with special measures (e.g., from a floppy), so you don't want to risk it.

The root directory generally doesn't contain any files, except perhaps the standard boot image for the system, usually called `/vmlinuz`. All other files are in subdirectories in the root filesystems:

`/bin` Commands needed during bootup that might be used by normal users (probably after bootup).

`/sbin` Like `/bin`, but the commands are not intended for normal users, although they may use them if necessary and allowed.

`/etc` Configuration files specific to the machine.

`/root` The home directory for user `root`.

`/lib` Shared libraries needed by the programs on the root filesystem.

`/lib/modules` Loadable kernel modules, especially those that are needed to boot the system when recovering from disasters (e.g., network and filesystem drivers).

`/dev` Device files.

`/tmp` Temporary files. Programs running after bootup should use `/var/tmp`, not `/tmp`, since the former is probably on a disk with more space.

`/boot` Files used by the bootstrap loader, e.g., LILO. Kernel images are often kept here instead of in the root directory. If there are many kernel images, the directory can easily grow rather big, and it might be better to keep it in a separate filesystem. Another reason would be to make sure the kernel images are within the first 1024 cylinders of an IDE disk.

`/mnt` Mount point for temporary mounts by the system administrator. Programs aren't supposed to mount on `/mnt` automatically. `/mnt` might be divided into subdirectories (e.g., `/mnt/dosa` might be the floppy drive using an MS-DOS filesystem, and `/mnt/exta` might be the same with an ext2 filesystem).

`/proc`, `/usr`, `/var`, `/home` Mount points for the other filesystems.

5.2.1 The `/etc` directory

The `/etc` directory contains a lot of files. Some of them are described below. For others, you should determine which program they belong to and read the manual page for that program. Many networking configuration files are in `/etc` as well, and are described in the Networking Administrators' Guide.

`/etc/rc` or `/etc/rc.d` or `/etc/rc?.d` Scripts or directories of scripts to run at startup or when changing the run level. See the chapter on `init` for further information.

`/etc/passwd` The user database, with fields giving the username, real name, home directory, encrypted password, and other information about each user. The format is documented in the `passwd(5)` manual page.

`/etc/fdprm` Floppy disk parameter table. Describes what different floppy disk formats look like. Used by `setfdprm(1)`. See the `setfdprm(8)` manual page for more information.

`/etc/fstab` Lists the filesystems mounted automatically at startup by the `mount -a` command (in `/etc/rc` or equivalent startup file). Under Linux, also contains information about swap areas used automatically by `swapon -a`. See section 4.6.5 and the `mount(8)` manual page for more information.

`/etc/group` Similar to `/etc/passwd`, but describes groups instead of users. See the `group(5)` manual page for more information.

`/etc/inittab` Configuration file for `init(8)`.

`/etc/issue` Output by `getty` before the login prompt. Usually contains a short description or welcoming message to the system. The contents are up to the system administrator.

`/etc/magic` The configuration file for `file(1)`. Contains the descriptions of various file formats based on which `file` guesses the type of the file. See the `magic(8)` and `file(1)` manual pages for more information.

`/etc/motd` The **message of the day**, automatically output after a successful login. Contents are up to the system administrator. Often used for getting information to every user, such as warnings about planned downtimes.

`/etc/mtab` List of currently mounted filesystems. Initially set up by the scripts, and updated automatically by the `mount` command. Used when a list of mounted filesystems is needed, e.g., by the `df(1)` command.

`/etc/shadow` Shadow password file on systems with shadow password software installed. Shadow passwords move the encrypted password from `/etc/passwd` into `/etc/shadow`; the latter is not readable by anyone except `root`. This makes it harder to crack passwords.

`/etc/login.defs` Configuration file for the `login(1)` command.

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

`/etc/profile`, `/etc/csh.login`, `/etc/csh.cshrc` Files executed at login or startup time by the Bourne or C shells. These allow the system administrator to set global defaults for all users. See the manual pages for the respective shells.

`/etc/securetty` Identifies secure terminals, i.e., the terminals from which `root` is allowed to log in. 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/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. See the `termcap(5)`, `curs_termcap(3)`, and `terminfo(5)` manual pages for more information.

META: HOSTNAME, adjtime, disktab, gettydefs, networking (exports, host.conf, hosts, hosts.equiv, inetd.conf, named.*, networks, ntp.conf, protocols, resolv.conf, rpc, services, syslog.conf), mtools, and so forth.

5.2.2 The `/dev` directory

The `/dev` directory contains the special device files for all the devices. The device files are named using special conventions; these are described in appendix C. The device files are created during installation, and later with the `/dev/MAKEDEV` script. The `/dev/MAKEDEV.local` is a script written by the system administrator that creates local-only device files or links (i.e., those that are not part of the standard `MAKEDEV`, such as device files for some non-standard device driver).

5.3 The `/usr` filesystem

The `/usr` filesystem is often large, since all programs are installed there. All files in `/usr` usually come from a Linux distribution; locally installed programs and other stuff goes below `/usr/local`. This makes it possible to update the system from a new version of the distribution, or even a completely new distribution, without having to install all programs again. Some of the subdirectories of `/usr` are listed below (some of the less important directories have been dropped; see the FSSTND for more information).

`/usr/X11R6` The X Window System, all files. To simplify the development and installation of X, the X files have not been integrated into the rest of the system. There is a directory tree below `/usr/X11R6` similar to that below `/usr` itself.

`/usr/X386` Similar to `/usr/X11R6`, but for X11 Release 5.

`/usr/bin` Almost all user commands. Some commands are in `/bin` or in `/usr/local/bin`.

`/usr/sbin` System administration commands that are not needed on the root filesystem, e.g., most server programs.

`/usr/man`, `/usr/info`, `/usr/doc` Manual pages, GNU Info documents, and miscellaneous other documentation files, respectively.

`/usr/include` Header files for the C programming language. This should actually be below `/usr/lib` for consistency, but the tradition is overwhelmingly in support for this name.

`/usr/lib` Unchanging data files for programs and subsystems, including some site-wide configuration files. The name `lib` comes from library; originally libraries of programming subroutines were stored in `/usr/lib`.

`/usr/local` The place for locally installed software and other files.

5.4 The /var filesystem

The `/var` contains data that is changed when the system is running normally. It is specific for each system, i.e., not shared over the network with other computers.

`/var/catman` A cache for man pages that are formatted on demand. The source for manual pages is usually stored in `/usr/man/man*`; some manual pages might come with a pre-formatted version, which is stored in `/usr/man/cat*`. Other manual pages need to be formatted when they are first viewed; the formatted version is then stored in `/var/man` so that the next person to view the same page won't have to wait for it to be formatted. (`/var/catman` is often cleaned in the same way temporary directories are cleaned.)

`/var/lib` Files that change while the system is running normally.

`/var/local` Variable data for programs that are installed in `/usr/local` (i.e., programs that have been installed by the system administrator). Note that even locally installed programs should use the other `/var` directories if they are appropriate, e.g., `/var/lock`.

`/var/lock` Lock files. Many programs follow a convention to create a lock file in `/var/lock` to indicate that they are using a particular device or file. Other programs will notice the lock file and won't attempt to use the device or file.

`/var/log` Log files from various programs, especially `login` (`/var/log/wtmp`, which logs all logins ans logouts into the system) and `syslog` (`/var/log/messages`, where all kernel and system program message are usually stored). File in `/var/log` can often grow indefinitely, and may require cleaning at regular intervals.

`/var/run` Files that contain information about the system that is valid until the system is next booted. For example, `/var/run/utmp` contains information about people currently logged in.

`/var/spool` Directories for mail, news, printer queues, and other queued work. Each different spool has its own subdirectory below `/var/spool`, e.g., the mailboxes of the users are in `/var/spool/mail`.

`/var/tmp` Temporary files that are large or that need to exist for a longer time than what is allowed for `/tmp`. (Although the system administrator might not allow very old files in `/var/tmp` either.)

5.5 The /proc filesystem

The `/proc` filesystem contains a illusionary filesystem. It does not exist on a disk. Instead, the kernel creates it in memory. It is used to provide information about the system (originally about processes, hence the name). Some of the more important files and directories are explained below. The `/proc` filesystem is described in more detail in the `proc(5)` manual page.

`/proc/1` A directory with information about process number 1. Each process has a directory below `/proc` with the name being its process identification number.

`/proc/cpuinfo` Information about the processor, such as its type, make, model, and perfomance.

`/proc/devices` List of device drivers configured into the currently running kernel.

`/proc/dma` Shows which DMA channels are being used at the moment.

`/proc/filesystems` Filesystems configured into the kernel.

`/proc/interrupts` Shows which interrupts are in use, and how many of each there have been.

`/proc/ioports` Which I/O ports are in use at the moment.

`/proc/kcore` An image of the physical memory of the system. This is exactly the same size as your physical memory, but does not really take up that much memory; it is generated on the fly as programs access it. (Remember: unless you copy it elsewhere, nothing under `/proc` takes up any disk space at all.)

`/proc/kmsg` Messages output by the kernel. These are also routed to `syslog`.

`/proc/ksyms` Symbol table for the kernel.

`/proc/loadavg` The 'load average' of the system; three meaningless indicators of how much work the system has to do at the moment.

`/proc/meminfo` Information about memory usage, both physical and swap.

`/proc/modules` Which kernel modules are loaded at the moment.

`/proc/net` Status information about network protocols.

`/proc/self` A symbolic link to the process directory of the program that is looking at `/proc`. When two processes look at `/proc`, they get different links. This is mainly a convenience to make it easier for programs to get at their process directory.

`/proc/stat` Various statistics about the system, such as the number of page faults since the system was booted.

`/proc/uptime` The time the system has been up.

`/proc/version` The kernel version.

Note that while the above files tend to be easily readable text files, they can sometimes be formatted in a way that is not easily digestable. There are many commands that do little more than read the above files and format them for easier understanding. For example, the `free` program reads `/proc/meminfo` and converts the amounts given in bytes to kilobytes (and adds a little more information, as well).

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)`. It must reside on a local disk, however; it can't reside in a filesystem that has been mounted over NFS.

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.)

4. If your calculated swap area is very much larger than your physical memory (more than a couple times larger), you should probably invest in more physical memory, otherwise performance will be too low.

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

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

one wants to be sure 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

Logging In And Out

This chapter needs a quote. Suggestions, anyone?

This section describes what happens when a user logs in or out. The various interactions of background processes, log files, configuration files, and so on are described in some detail.

7.1 Logins via terminals

Figure 7.1 shows how logins happen via terminals. First, `init` makes sure there is a `getty` program for the terminal connection (or console). `getty` listens at the terminal and waits for the user to notify that he is ready to login in (this usually means that the user must type something). When it notices a user, `getty` outputs a welcome message (stored in `/etc/issue`), and prompts for the username, and finally runs the `login` program. `login` gets the username as a parameter, and prompts the user for the password. If these match, `login` starts the shell configured for the user; else it just exits and terminates the process (perhaps after giving the user another chance at entering the username and password). `init` notices that the process terminated, and starts a new `getty` for the terminal.

Note that the only process new process is created by `init` (using the `fork(2)` system call); `getty` and `login` only replace the program running in the process (using the `exec(3)` system call).

A separate program for noticing the user is needed for serial lines, since it can be (and traditionally was) complicated to notice when a terminal becomes active. `getty` also adapts to the speed and other settings of the connection, which is important especially for dial-in connections, where these parameters may change from call to call.

There are several versions of `getty` and `init` in use, all with their good and bad points. It is a good idea to learn about the versions on your system, and also about the other versions (you could use the Linux Software Map to search them). If you don't have dial-in's, you probably don't have to worry about `getty`, but `init` is still important.

7.2 Logins via the network

Two computers in the same network are usually linked via a single physical cable. When they communicate over the network, the programs in each computer that take part in the communication are linked via a **virtual connection**, a sort of imaginary cable. As far as the programs at either end of the virtual connection are concerned, they have a monopoly on their own cable. However, since the cable is not real, only imaginary, the operating systems of both computers can have several virtual connections share the same physical cable. This way, using just a single cable, several programs can communicate without having to know of or care about the other communications. It is even possible to have several computers use the same cable; the virtual connections exist between two computers, and the other computers ignore those connections that they don't take part in.

That's a complicated and over-abstracted description of the reality. It might, however, be good enough to understand the important reason why network logins are somewhat different from normal logins. The virtual connections are established when there are two programs on different computers that wish to communicate. Since it is in principle possible to login from any computer in a network to any other computer, there is a huge number of potential virtual communications. Because of this, it is not practical to start a `getty` for each potential login.

There is a single process corresponding to `getty` that handles *all* network logins. When it notices an incoming network login (i.e., it notices that it gets a new virtual connection to some other computer), it starts a new process to handle that single login. The original process remains and continues to listen for new logins.

To make things a bit more complicated, there is more than one communication protocol for network logins. The two most important ones are `telnet` and `rlogin`. In addition to logins, there are many other virtual connections that may be made (for

FTP, Gopher, HTTP, and other network services). It would be ineffective to have a separate process listening for a particular type of connection, so instead there is only one listener that can recognize the type of the connection and can start the correct type of program to provide the service. This single listener is called `inetd`; see the "Linux Network Administrators' Guide" for more information.

7.3 What `login` does

The `login` program takes care of authenticating the user (making sure that the username and password match), and of setting up an initial environment for the user by setting permissions for the serial line and starting the shell.

Part of the initial setup is outputting the contents of the file `/etc/motd` (short for message of the day) and checking for electronic mail. These can be disabled by creating a file called `.hushlogin` in the user's home directory.

If the file `/etc/nologin` exists, logins are disabled. That file is typically created by `shutdown(8)` and relatives. `login` checks for this file, and will refuse to accept a login if it exists. If it does exist, `login` outputs it contents to the terminal before it quits.

`login` logs all failed login attempts in a system log file (via `syslog`). It also logs *all* logins by `root`. Both of these can be useful when tracking down intruders.

Currently logged in people are listed in `/var/run/utmp`. This file is valid only until the system is next rebooted or shut down; it is cleared when the system is booted. It lists each user and the terminal (or network connection) he is using, along with some other useful information. The `who`, `w`, and other similar commands look in `utmp` to see who are logged in.

All successful logins are recorded into `/var/log/wtmp`. This file will grow without limit, so it must be cleaned regularly, for example by having a weekly `cron` job to clear it.[1] The `last` command browses `wtmp`.

Both `utmp` and `wtmp` are in a binary format (see the `utmp(5)` manual page); it is unfortunately not convenient to examine them without special programs.

[1] Good Linux distributions do this out of the box.

7.4 X and xdm

META: X implements logins via xdm; also: xterm -ls

7.5 Access control

The user database is traditionally contained in the `/etc/passwd` file. Some systems use **shadow passwords**, and have moved the passwords to `/etc/shadow`. Sites with many computers that share the accounts use NIS or some other method to store the user database; they might also automatically copy the database from one central location to all other computers.

The user database contains not only the passwords, but also some additional information about the users, such as their real names, home directories, and login shells. This other information needs to be public, so that anyone can read it. Therefore the password is stored encrypted. This does have the drawback that anyone with access to the encrypted password can use various cryptographical methods to guess it, without trying to actually log into the computer. Shadow passwords try to avoid this by moving the password into another file, which only `root` can read (the password is still stored encrypted). However, installing shadow passwords later onto a system that did not support them can be difficult.

With or without passwords, it is important to make sure that all passwords in a system are good, i.e., not easily guessable. The `crack` program can be used to crack passwords; any password it can find is by definition not a good one. While `crack` can be run be intruders, it can also be run by the system adminstrator to avoid bad passwords. Good passwords can also be enforced by the `passwd(1)` program; this is in fact more effective in CPU cycles, since cracking passwords requires quite a lot of computation.

The user group database is kept in `/etc/group`; for systems with shadow passwords, there can be a `/etc/shadow.group`.

`root` usually can't login via most terminals or the network, only via terminals listed in the `/etc/securetty` file. This makes it necessary to get physical access to one of these terminals. It is, however, possible to log in via any terminal as any other user, and use the `su` command to become `root`.

7.6 Shell startup

When an interactive login shell starts, it automatically executes one or more predefined files. Different shells execute different files; see the documentation of each shell for further information.

Most shells first run some global file, for example, the Bourne shell (`/bin/sh`) and its derivatives execute `/etc/profile`; in addition, they execute `~/.profile`. `/etc/profile` allows the system administrator to have set up a common user environment, especially by setting the PATH to include local command directories in addition to the normal ones. On the other hand, `~/.profile` allows the user to customize the environment to his own tastes by overriding, if necessary, the default environment.

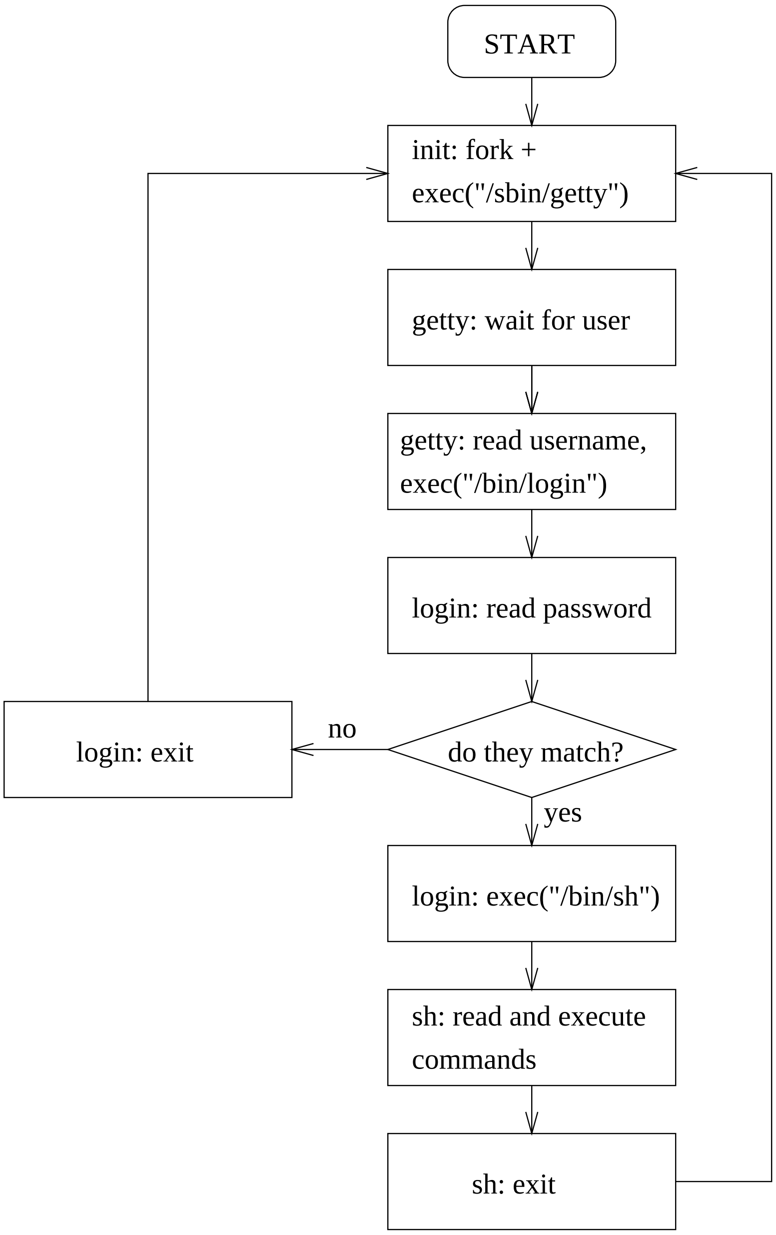

Figure 7.1: Logins via terminals: the interaction of `init`, `getty`, `login`, and the shell (here, `/bin/sh`).

Appendix A

Design and Implementation of the Second Extended Filesystem

This appendix is a paper written by Rémy Card (`card@masi.ibp.fr`), Theodore Ts'o (`tytso@mit.edu`), and Stephen Tweedie (`sct@dcs.ed.ac.uk`), the designers and implementors of the ext2 filesystem. It was first published in the Proceedings of 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.

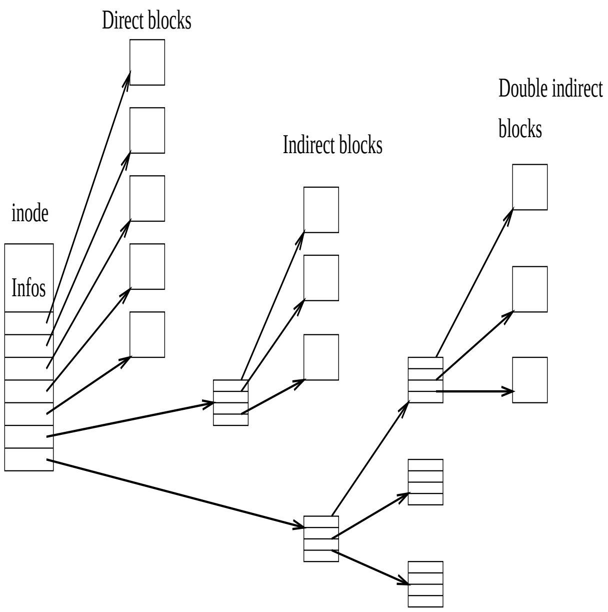

Figure A.1: Structure of an inode

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 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.

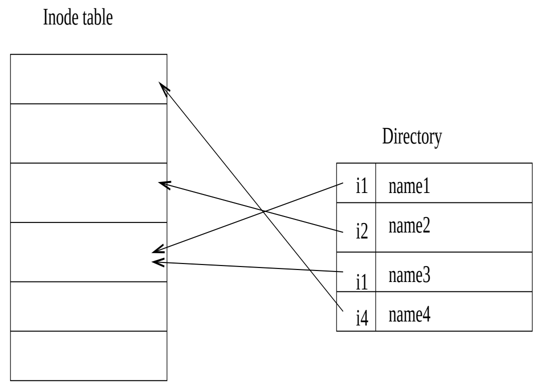

Figure A.2: Structure of a directory

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 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.

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.

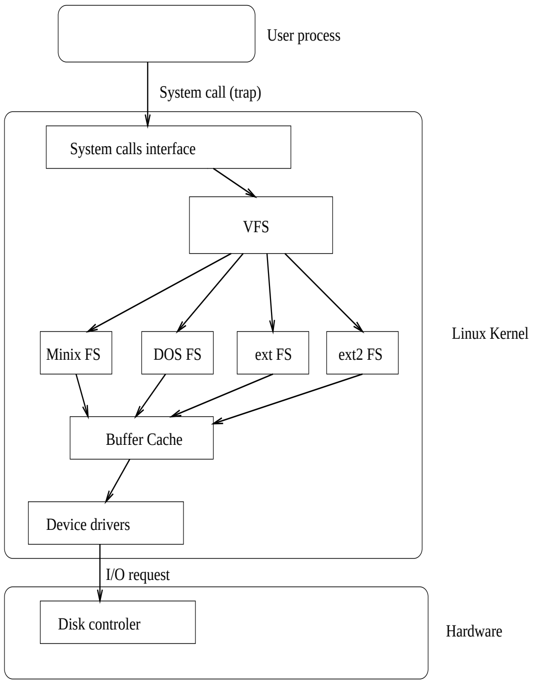

Figure A.3: The VFS Layer

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.

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.

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

Figure A.4: Physical structure of an Ext2 filesystem

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`.

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

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

Figure A.7: Example of directory

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.

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

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.

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 (sag/measure-holes/measure-holes.c).

```
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

The Linux Device List

This is the device list, maintained by H. Peter Anvin (`Peter.Anvin@linux.org`), at `ftp://ftp.yggdrasil.com/pub/device-list/devices.tex`. The rest of this text is by Peter.

C.1 Introduction

This list is the successor to Rick Miller's Linux Device List, which he stopped maintaining when he lost network access in 1993. It is a registry of allocated major device numbers, as well as the recommended `/dev` directory nodes for these devices.

This list is available via FTP from ftp.yggdrasil.com in the directory `/pub/device-list`; filename is `devices.`*format* where *format* is `txt` (ASCII), `tex` (LaTeX), `dvi` (DVI) or `ps` (PostScript). In cases of discrepancy, the LaTeX version has priority.

This document is included by reference into the Linux Filesystem Standard (FSSTND). The FSSTND is available via FTP from tsx-11.mit.edu in the directory `/pub/linux/docs/linux-standards/fsstnd`.

To have a major number allocated, or a minor number in situations where that applies (e.g. busmice), please contact me. Also, if you have additional information regarding any of the devices listed below, I would like to know.

Allocations marked (68k) apply to Linux/68k only.

C.2 Major numbers

0		Unnamed devices (NFS mounts, loopback devices)
1	char	Memory devices
	block	RAM disk
2	char	Reserved for PTY's <tytso@athena.mit.edu>
	block	Floppy disks
3	char	Reserved for PTY's <tytso@athena.mit.edu>
	block	First MFM, RLL and IDE hard disk/CD-ROM interface
4	char	TTY devices
5	char	Alternate TTY devices
6	char	Parallel printer devices
7	char	Virtual console access devices
8	block	SCSI disk devices
9	char	SCSI tape devices
	block	Multiple disk devices
10	char	Non-serial mice, misc features
11	block	SCSI CD-ROM devices
12	char	QIC-02 tape
	block	MSCDEX CD-ROM callback support
13	char	PC speaker
	block	8-bit MFM/RLL/IDE controller
14	char	Sound card
	block	BIOS harddrive callback support
15	char	Joystick
	block	Sony CDU-31A/CDU-33A CD-ROM
16	char	Reserved for scanners
	block	GoldStar CD-ROM
17	char	Chase serial card (Under development)
	block	Optics Storage CD-ROM (Under development)
18	char	Chase serial card – alternate devices
	block	Sanyo CD-ROM (Under development)
19	char	Cyclades serial card
	block	Double compressed disk
20	char	Cyclades serial card – alternate devices
	block	Hitachi CD-ROM (Under development)
21	char	Generic SCSI access
22	char	Digiboard serial card

	block	Second MFM, RLL and IDE hard disk/CD-ROM interface
23	char	Digiboard serial card – alternate devices
	block	Mitsumi proprietary CD-ROM
24	char	Stallion serial card
	block	Sony CDU-535 CD-ROM
25	char	Stallion serial card – alternate devices
	block	First Matsushita (Panasonic/SoundBlaster) CD-ROM
26	block	Second Matsushita (Panasonic/SoundBlaster) CD-ROM
27	char	QIC-117 tape
	block	Third Matsushita (Panasonic/SoundBlaster) CD-ROM
28	char	Stallion serial card – card programming
	block	Fourth Matsushita (Panasonic/SoundBlaster) CD-ROM
	block	ACSI disk (68k)
29	char	Universal frame buffer
	block	Aztech/Orchid/Okano/Wearnes CD-ROM
30	char	iBCS-2
	block	Philips LMS-205 CD-ROM
31	char	MPU-401 MIDI
	block	ROM/flash memory card
32	block	Philips LMS-206 CD-ROM
33	block	Modular RAM disk
34–223		Unallocated
224–254		Local use
255		Reserved

C.3 Minor numbers

0 **Unnamed devices (NFS mounts, loopback devices)**

0	reserved as null device number

1 **char** **Memory devices**

1 `/dev/mem`	Physical memory access
2 `/dev/kmem`	Kernel virtual memory access
3 `/dev/null`	Null device
4 `/dev/port`	I/O port access

		5	`/dev/zero`	Null byte source
		6	`/dev/core`	OBSOLETE – should be a link to `/proc/kcore`
		7	`/dev/full`	Returns ENOSPC on write
	block	**RAM disk**		
		1	`/dev/ramdisk`	RAM disk
2	**char**	**Reserved for PTY's <tytso@athena.mit.edu>**		
	block	**Floppy disks**		
		0	`/dev/fd0`	Controller 1, drive 1 autodetect
		1	`/dev/fd1`	Controller 1, drive 2 autodetect
		2	`/dev/fd2`	Controller 1, drive 3 autodetect
		3	`/dev/fd3`	Controller 1, drive 4 autodetect
		128	`/dev/fd4`	Controller 2, drive 1 autodetect
		129	`/dev/fd5`	Controller 2, drive 2 autodetect
		130	`/dev/fd6`	Controller 2, drive 3 autodetect
		131	`/dev/fd7`	Controller 2, drive 4 autodetect
		To specify format, add to the autodetect device number		
		0	`/dev/fd?`	Autodetect format
		4	`/dev/fd?d360`	5.25” 360K in a 360K drive[1]
		20	`/dev/fd?h360`	5.25” 360K in a 1200K drive[1]
		48	`/dev/fd?h410`	5.25” 410K in a 1200K drive
		64	`/dev/fd?h420`	5.25” 420K in a 1200K drive
		24	`/dev/fd?h720`	5.25” 720K in a 1200K drive
		80	`/dev/fd?h880`	5.25” 880K in a 1200K drive[1]
		8	`/dev/fd?h1200`	5.25” 1200K in a 1200K drive[1]
		40	`/dev/fd?h1440`	5.25” 1440K in a 1200K drive[1]
		56	`/dev/fd?h1476`	5.25” 1476K in a 1200K drive
		72	`/dev/fd?h1494`	5.25” 1494K in a 1200K drive
		92	`/dev/fd?h1600`	5.25” 1600K in a 1200K drive[1]
		12	`/dev/fd?u360`	3.5” 360K Double Density
		16	`/dev/fd?u720`	3.5” 720K Double Density[1]
		120	`/dev/fd?u800`	3.5” 800K Double Density[2]
		52	`/dev/fd?u820`	3.5” 820K Double Density
		68	`/dev/fd?u830`	3.5” 830K Double Density

84	`/dev/fd?u1040`	3.5" 1040K Double Density[1]
88	`/dev/fd?u1120`	3.5" 1120K Double Density[1]
28	`/dev/fd?u1440`	3.5" 1440K High Density[1]
124	`/dev/fd?u1600`	3.5" 1600K High Density[1]
44	`/dev/fd?u1680`	3.5" 1680K High Density[3]
60	`/dev/fd?u1722`	3.5" 1722K High Density
76	`/dev/fd?u1743`	3.5" 1743K High Density
96	`/dev/fd?u1760`	3.5" 1760K High Density
116	`/dev/fd?u1840`	3.5" 1840K High Density[3]
100	`/dev/fd?u1920`	3.5" 1920K High Density[1]
32	`/dev/fd?u2880`	3.5" 2880K Extra Density[1]
104	`/dev/fd?u3200`	3.5" 3200K Extra Density
108	`/dev/fd?u3520`	3.5" 3520K Extra Density
112	`/dev/fd?u3840`	3.5" 3840K Extra Density[1]
36	`/dev/fd?CompaQ`	Compaq 2880K drive; probably obsolete

[1] **Autodetectable format**
[2] **Autodetectable format in a Double Density (720K) drive only**
[3] **Autodetectable format in a High Density (1440K) drive only**

NOTE: The letter in the device name (`d`, `q`, `h` or `u`) signifies the type of drive supported: 5.25" Double Density (`d`), 5.25" Quad Density (`q`), 5.25" High Density (`h`) or 3.5" (any type, `u`). The capital letters `D`, `H`, or `E` for the 3.5" models have been deprecated, since the drive type is insignificant for these devices.

3 **char** **Reserved for PTY's <tytso@athena.mit.edu>**
block **First MFM, RLL and IDE hard disk/CD-ROM interface**

0	`/dev/hda`	Master: whole disk (or CD-ROM)
64	`/dev/hdb`	Slave: whole disk (or CD-ROM)

For partitions, add to the whole disk device number

0	`/dev/hd?`	Whole disk
1	`/dev/hd?1`	First primary partition
2	`/dev/hd?2`	Second primary partition
3	`/dev/hd?3`	Third primary partition
4	`/dev/hd?4`	Fourth primary partition

		5 `/dev/hd?5`	First logical partition
		6 `/dev/hd?6`	Second logical partition
		7 `/dev/hd?7`	Third logical partition
		...	
		63 `/dev/hd?63`	59th logical partition
4	**char**	**TTY devices**	
		0 `/dev/console`	Console device
		1 `/dev/tty1`	First virtual console
		...	
		63 `/dev/tty63`	63rd virtual console
		64 `/dev/ttyS0`	First serial port
		...	
		127 `/dev/ttyS63`	64th serial port
		128 `/dev/ptyp0`	First pseudo-tty master
		...	
		191 `/dev/ptysf`	64th pseudo-tty master
		192 `/dev/ttyp0`	First pseudo-tty slave
		...	
		255 `/dev/ttysf`	64th pseudo-tty slave

Pseudo-tty's are named as follows:

- Masters are `pty`, slaves are `tty`;
- the fourth letter is one of `pqrs` indicating the 1st, 2nd, 3rd, 4th series of 16 pseudo-ttys each, and
- the fifth letter is one of `0123456789abcdef` indicating the position within the series.

5	**char**	**Alternate TTY devices**	
		0 `/dev/tty`	Current TTY device
		64 `/dev/cua0`	Callout device corresponding to `ttyS0`
		...	
		127 `/dev/cua63`	Callout device corresponding to `ttyS63`

6	**char**	**Parallel printer devices**	
		0 `/dev/lp0`	First parallel printer (`0x3bc`)
		1 `/dev/lp1`	Second parallel printer (`0x378`)
		2 `/dev/lp2`	Third parallel printer (`0x278`)

Not all computers have the `0x3bc` parallel port, hence the "first" printer may be either `/dev/lp0` or `/dev/lp1`.

7	**char**	**Virtual console access devices**	
		0 `/dev/vcs`	Current vc text access
		1 `/dev/vcs1`	tty1 text access
		...	
		63 `/dev/vcs63`	tty63 text access
		128 `/dev/vcsa`	Current vc text/attribute access
		129 `/dev/vcsa1`	tty1 text/attribute access
		...	
		191 `/dev/vcsa63`	tty63 text/attribute access

NOTE: These devices permit both read and write access.

8	**block**	**SCSI disk devices**	
		0 `/dev/sda`	First SCSI disk whole disk
		16 `/dev/sdb`	Second SCSI disk whole disk
		32 `/dev/sdc`	Third SCSI disk whole disk
		...	
		240 `/dev/sdp`	Sixteenth SCSI disk whole disk

Partitions are handled in the same way as for IDE disks (see major number 3) except that the limit on logical partitions is 11 rather than 59 per disk.

9	**char**	**SCSI tape devices**	
		0 `/dev/st0`	First SCSI tape
		1 `/dev/st1`	Second SCSI tape
		...	
		128 `/dev/nst0`	First SCSI tape, no rewind-on-close
		129 `/dev/nst1`	Second SCSI tape, no rewind-on-close

...

	block	**Multiple disk devices**	
		0 /dev/md0	First device group
		1 /dev/md1	Second device group
		...	

The multiple device driver is used to span a filesystem across multiple physical disks.

10	**char**	**Non-serial mice, misc features**	
		0 /dev/logibm	Logitech bus mouse
		1 /dev/psaux	PS/2-style mouse port
		2 /dev/inportbm	Microsoft Inport bus mouse
		3 /dev/atibm	ATI XL bus mouse
		4 /dev/jbm	J-mouse
		4 /dev/amigamouse	Amiga Mouse (68k)
		5 /dev/atarimouse	Atari Mouse (68k)
		128 /dev/beep	Fancy beep device
		129 /dev/modreq	Kernel module load request

11	**block**	**SCSI CD-ROM devices**	
		0 /dev/sr0	First SCSI CD-ROM
		1 /dev/sr1	Second SCSI CD-ROM
		...	

The prefix `/dev/scd` instead of `/dev/sr` has been used as well, and might make more sense.

12	**char**	**QIC-02 tape**	
		2 /dev/ntpqic11	QIC-11, no rewind-on-close
		3 /dev/tpqic11	QIC-11, rewind-on-close
		4 /dev/ntpqic24	QIC-24, no rewind-on-close
		5 /dev/tpqic24	QIC-24, rewind-on-close
		6 /dev/ntpqic120	QIC-120, no rewind-on-close
		7 /dev/tpqic120	QIC-120, rewind-on-close
		8 /dev/ntpqic150	QIC-150, no rewind-on-close

		9 `/dev/tpqic150`	QIC-150, rewind-on-close

The device names specified are proposed – if there are "standard" names for these devices, please let me know.

	block	**MSCDEX CD-ROM callback support**	
		0 `/dev/dos_cd0`	First MSCDEX CD-ROM
		1 `/dev/dos_cd1`	Second MSCDEX CD-ROM
		...	
13	**char**	**PC speaker**	
		0 `/dev/pcmixer`	Emulates `/dev/mixer`
		3 `/dev/pcsp`	Emulates `/dev/dsp` (8-bit)
		4 `/dev/pcaudio`	Emulates `/dev/audio`
		5 `/dev/pcsp16`	Emulates `/dev/dsp` (16-bit)
	block	**8-bit MFM/RLL/IDE controller**	
		0 `/dev/xda`	First XT disk whole disk
		64 `/dev/xdb`	Second XT disk whole disk

Partitions are handled in the same way as IDE disks (see major number 3).

14	**char**	**Sound card**	
		0 `/dev/mixer`	Mixer control
		1 `/dev/sequencer`	Audio sequencer
		2 `/dev/midi00`	First MIDI port
		3 `/dev/dsp`	Digital audio
		4 `/dev/audio`	Sun-compatible digital audio
		6 `/dev/sndstat`	Sound card status information
		8 `/dev/sequencer2`	Sequencer – alternate device
		16 `/dev/mixer1`	Second soundcard mixer control
		17 `/dev/patmgr0`	Sequencer patch manager
		18 `/dev/midi01`	Second MIDI port
		19 `/dev/dsp1`	Second soundcard digital audio
		20 `/dev/audio1`	Second soundcard Sun digital audio
		33 `/dev/patmgr1`	Sequencer patch manager

		34	/dev/midi02	Third MIDI port
		50	/dev/midi03	Fourth MIDI port
	block	**BIOS harddrive callback support**		
		0	/dev/dos_hda	First BIOS harddrive whole disk
		64	/dev/dos_hdb	Second BIOS harddrive whole disk
		128	/dev/dos_hdc	Third BIOS harddrive whole disk
		192	/dev/dos_hdd	Fourth BIOS harddrive whole disk

Partitions are handled in the same way as IDE disks (see major number 3).

15	**char**	**Joystick**		
		0	/dev/js0	First joystick
		1	/dev/js1	Second joystick
	block	**Sony CDU-31A/CDU-33A CD-ROM**		
		0	/dev/sonycd	Sony CDU-31A CD-ROM
16	**char**	**Reserved for scanners**		
	block	**GoldStar CD-ROM**		
		0	/dev/gscd	GoldStar CD-ROM
17	**char**	**Chase serial card (Under development)**		
		0	/dev/ttyH0	First Chase port
		1	/dev/ttyH1	Second Chase port
			...	
	block	**Optics Storage CD-ROM (Under development)**		
		0	/dev/optcd	Optics Storage CD-ROM
18	**char**	**Chase serial card – alternate devices**		
		0	/dev/cuh0	Callout device corresponding to ttyH0
		1	/dev/cuh1	Callout device corresponding to ttyH1
			...	

	block	**Sanyo CD-ROM (Under development)**		
		0	?	Sanyo CD-ROM
19	char	**Cyclades serial card**		
		32	`/dev/ttyC0`	First Cyclades port
			...	
		63	`/dev/ttyC31`	32nd Cyclades port

It would make more sense for these to start at 0...

	block	**"Double" compressed disk**		
		0	`/dev/double0`	First compressed disk
			...	
		7	`/dev/double7`	Eighth compressed disk
		128	`/dev/cdouble0`	Mirror of first compressed disk
			...	
		135	`/dev/cdouble7`	Mirror of eighth compressed disk

See the Double documentation for an explanation of the "mirror" devices.

20	char	**Cyclades serial card – alternate devices**		
		32	`/dev/cub0`	Callout device corresponding to `ttyC0`
			...	
		63	`/dev/cub31`	Callout device corresponding to `ttyC31`
	block	**Hitachi CD-ROM (Under development)**		
		0	`/dev/hitcd`	Hitachi CD-ROM
21	char	**Generic SCSI access**		
		0	`/dev/sg0`	First generic SCSI device
		1	`/dev/sg1`	Second generic SCSI device
			...	
22	char	**Digiboard serial card**		

		0 `/dev/ttyD0`	First Digiboard port
		1 `/dev/ttyD1`	Second Digiboard port
		...	
	block	**Second MFM, RLL and IDE hard disk/CD-ROM interface**	
		0 `/dev/hdc`	Master: whole disk (or CD-ROM)
		64 `/dev/hdd`	Slave: whole disk (or CD-ROM)

Partitions are handled the same way as for the first interface (see major number 3).

23	**char**	**Digiboard serial card – alternate devices**	
		0 `/dev/cud0`	Callout device corresponding to `ttyD0`
		1 `/dev/cud1`	Callout device corresponding to `ttyD1`
		...	
	block	**Mitsumi proprietary CD-ROM**	
		0 `/dev/mcd`	Mitsumi CD-ROM

24	**char**	**Stallion serial card**	
		0 `/dev/ttyE0`	Stallion port 0 board 0
		1 `/dev/ttyE1`	Stallion port 1 board 0
		...	
		64 `/dev/ttyE64`	Stallion port 0 board 1
		65 `/dev/ttyE65`	Stallion port 1 board 1
		...	
		128 `/dev/ttyE128`	Stallion port 0 board 2
		129 `/dev/ttyE129`	Stallion port 1 board 2
		...	
		192 `/dev/ttyE192`	Stallion port 0 board 3
		193 `/dev/ttyE193`	Stallion port 1 board 3
		...	
	block	**Sony CDU-535 CD-ROM**	
		0 `/dev/cdu535`	Sony CDU-535 CD-ROM

25	**char**	**Stallion serial card – alternate devices**	
		0 `/dev/cue0`	Callout device corresponding to `ttyE0`
		1 `/dev/cue1`	Callout device corresponding to `ttyE1`

		...	
		64 `/dev/cue64`	Callout device corresponding to `ttyE64`
		65 `/dev/cue65`	Callout device corresponding to `ttyE65`
		...	
		128 `/dev/cue128`	Callout device corresponding to `ttyE128`
		129 `/dev/cue129`	Callout device corresponding to `ttyE129`
		...	
		192 `/dev/cue192`	Callout device corresponding to `ttyE192`
		193 `/dev/cue193`	Callout device corresponding to `ttyE193`
		...	
	block	**First Matsushita (Panasonic/SoundBlaster) CD-ROM**	
		0 `/dev/sbpcd0`	Panasonic CD-ROM controller 0 unit 0
		1 `/dev/sbpcd1`	Panasonic CD-ROM controller 0 unit 1
		2 `/dev/sbpcd2`	Panasonic CD-ROM controller 0 unit 2
		3 `/dev/sbpcd3`	Panasonic CD-ROM controller 0 unit 3
26	**char**	**Frame grabbers**	
		0 `/dev/wvisfgrab`	Quanta WinVision frame grabber
	block	**Second Matsushita (Panasonic/SoundBlaster) CD-ROM**	
		0 `/dev/sbpcd4`	Panasonic CD-ROM controller 1 unit 0
		1 `/dev/sbpcd5`	Panasonic CD-ROM controller 1 unit 1
		2 `/dev/sbpcd6`	Panasonic CD-ROM controller 1 unit 2
		3 `/dev/sbpcd7`	Panasonic CD-ROM controller 1 unit 3
27	**char**	**QIC-117 tape**	
		0 `/dev/rft0`	Unit 0, rewind-on-close
		1 `/dev/rft1`	Unit 1, rewind-on-close
		2 `/dev/rft2`	Unit 2, rewind-on-close
		3 `/dev/rft3`	Unit 3, rewind-on-close
		4 `/dev/nrft0`	Unit 0, no rewind-on-close
		5 `/dev/nrft1`	Unit 1, no rewind-on-close
		6 `/dev/nrft2`	Unit 2, no rewind-on-close
		7 `/dev/nrft3`	Unit 3, no rewind-on-close

	block	**Third Matsushita (Panasonic/SoundBlaster) CD-ROM**		
		0	/dev/sbpcd8	Panasonic CD-ROM controller 2 unit 0
		1	/dev/sbpcd9	Panasonic CD-ROM controller 2 unit 1
		2	/dev/sbpcd10	Panasonic CD-ROM controller 2 unit 2
		3	/dev/sbpcd11	Panasonic CD-ROM controller 2 unit 3
28	char	**Stallion serial card – card programming**		
		0	/dev/staliomem0	First Stallion I/O card memory
		1	/dev/staliomem1	Second Stallion I/O card memory
		2	/dev/staliomem2	Third Stallion I/O card memory
		3	/dev/staliomem3	Fourth Stallion I/O card memory
	block	**Fourth Matsushita (Panasonic/SoundBlaster) CD-ROM**		
		0	/dev/sbpcd12	Panasonic CD-ROM controller 3 unit 0
		1	/dev/sbpcd13	Panasonic CD-ROM controller 3 unit 1
		2	/dev/sbpcd14	Panasonic CD-ROM controller 3 unit 2
		3	/dev/sbpcd15	Panasonic CD-ROM controller 3 unit 3
	block	**ACSI disk (68k)**		
		0	/dev/ada	First ACSI disk whole disk
		16	/dev/adb	Second ACSI disk whole disk
		32	/dev/adc	Third ACSI disk whole disk
			...	
		240	/dev/adp	Sixteenth ACSI disk whole disk

Partitions are handled in the same way as for IDE disks (see major number 3) except that the limit on logical partitions is 11 rather than 59 per disk.

29	char	**Universal frame buffer**		
		0	/dev/fb0current	First frame buffer
		1	/dev/fb0autodetect	
			...	
		16	/dev/fb1current	Second frame buffer
		17	/dev/fb1autodetect	
			...	

The universal frame buffer device is currently supported only on Linux/68k. The `current` device accesses the frame buffer at current resolution; the `autodetect` one at bootup (default) resolution. Minor numbers 2–15 within each frame buffer assignment are used for specific device-dependent resolutions. There appears to be no standard naming for these devices.

	block	**Aztech/Orchid/Okano/Wearnes CD-ROM**	
		0 `/dev/aztcd`	Aztech CD-ROM

30	**char**	**iBCS-2 compatibility devices**	
		0 `/dev/socksys`	Socket access
		1 `/dev/spx`	SVR3 local X interface
		2 `/dev/inet/arp`	Network access
		2 `/dev/inet/icmp`	Network access
		2 `/dev/inet/ip`	Network access
		2 `/dev/inet/udp`	Network access
		2 `/dev/inet/tcp`	Network access

iBCS-2 requires `/dev/nfsd` to be a link to `/dev/socksys` and `/dev/X0R` to be a link to `/dev/null`.

	block	**Philips LMS CM-205 CD-ROM**	
		0 `/dev/cm205cd`	Philips LMS CM-205 CD-ROM

`/dev/lmscd` is an older name for this drive. This driver does not work with the CM-205MS CD-ROM.

31	**char**	**MPU-401 MIDI**	
		0 `/dev/mpu401data`	MPU-401 data port
		1 `/dev/mpu401stat`	MPU-401 status port
	block	**ROM/flash memory card**	
		0 `/dev/rom0`	First ROM card (rw)
		...	
		7 `/dev/rom7`	Eighth ROM card (rw)
		8 `/dev/rrom0`	First ROM card (ro)

...		
15	`/dev/rrom0`	Eighth ROM card (ro)
16	`/dev/flash0`	First flash memory card (rw)
...		
23	`/dev/flash7`	Eighth flash memory card (rw)
24	`/dev/rflash0`	First flash memory card (ro)
...		
31	`/dev/rflash7`	Eighth flash memory card (ro)

The read-write (rw) devices support back-caching written data in RAM, as well as writing to flash RAM devices. The read-only devices (ro) support reading only.

32 block Philips LMS CM-206 CD-ROM

0	`/dev/cm206cd`	Philips LMS CM-206 CD-ROM

33 block Modular RAM disk

0	`/dev/ram0`	First modular RAM disk
1	`/dev/ram1`	Second modular RAM disk
...		
255	`/dev/ram255`	256th modular RAM disk

34–223 Unallocated

224–254 Local/experimental use

For devices not assigned official numbers, this range should be used, in order to avoid conflict with future assignments. Please note that `MAX_CHRDEV` and `MAX_BLKDEV` in `linux/include/linux/major.h` must be set to a value greater than the highest used major number. For a kernel using local/experimental devices, it is probably easiest to set both of these equal to 256. The memory cost above using the default value of 64 is 3K.

255 Reserved

C.4 Additional /dev directory entries

This section details additional entries that should or may exist in the `/dev` directory. It is preferred that symbolic links use the same form (absolute or relative) as is indicated here. Links are classified as *hard* or *symbolic* depending on the preferred type of link; if possible, the indicated type of link should be used.

C.4.1 Compulsory links

These links should exist on all systems:

`/dev/fd`	`/proc/self/fd`	symbolic	File descriptors
`/dev/stdin`	`fd/0`	symbolic	Standard input file descriptor
`/dev/stdout`	`fd/1`	symbolic	Standard output file descriptor
`/dev/stderr`	`fd/2`	symbolic	Standard error file descriptor

C.4.2 Recommended links

It is recommended that these links exist on all systems:

`/dev/X0R`	`null`	symbolic	Used by iBCS-2
`/dev/nfsd`	`socksys`	symbolic	Used by iBCS-2
`/dev/core`	`/proc/kcore`	symbolic	Backward compatibility
`/dev/scd?`	`sr?`	hard	Alternate name for CD-ROMs

C.4.3 Locally defined links

The following links may be established locally to conform to the configuration of the system. This is merely a tabulation of existing practice, and does not constitute a recommendation. However, if they exist, they should have the following uses.

`/dev/mouse`	*mouse port*	symbolic	Current mouse device
`/dev/tape`	*tape device*	symbolic	Current tape device
`/dev/cdrom`	*CD-ROM device*	symbolic	Current CD-ROM device
`/dev/modem`	*modem port*	symbolic	Current dialout device
`/dev/root`	*root device*	symbolic	Current root filesystem

`/dev/swap`	*swap device*	symbolic	Current swap device

`/dev/modem` should not be used for a modem which supports dialin as well as dialout, as it tends to cause lock file problems. If it exists, `/dev/modem` should point to the appropriate dialout (alternate) device.

C.4.4 Sockets and pipes

Non-transient sockets or named pipes may exist in `/dev`. Common entries are:

`/dev/printer`	socket	`lpd` local socket
`/dev/log`	socket	`syslog` local socket

Bibliography

[Car95] Rémy Card. The second extended filesystem: current state, future development, 1995. Slides used during presentation at the Second International Linux and Internet Conference, in Berlin, May 1995. Available via anonymous FTP from `ftp.ibp.fr`, in the directory `/pub/linux/packages/ext2fs/slides/berlin`.

[NSS89] Evi Nemeth, Garth Snyder, and Scott Seebass. *UNIX System Administration Handbook*. Prentice-Hall, 1989. 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.

[POL93] Jerry Peek, Tim O'Reilly, and Mike Loukide. *UNIX Power Tools*. Bantam, 1993. 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.

[Qui95] Daniel Quinlan. *Linux Filesystem Structure—Release 1.2*, March 1995. A description of and a proposal for a standard Linux directory tree, with the intention is to make it easier to package software and administer Linux systems by making files appear in standard places. Follows fairly closely traditional Unix practice, and has got support from most Linux distributions. Available via FTP from `ftp.funet.fi`, directory `/pub/Linux/doc/fsstnd`.

[Ray91] Eric Raymond, editor. *The New Hacker's Dictionary*. MIT Press, 1991. 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.