Contents

Chapter 1

Introduction to the ALPHA Versions

This is an ALPHA version of the Linux System Administrator's Guide. That means that I don't even pretend it contains anything useful, or that anything contained within it is factually correct. In fact, if you believe anything that I say in this version, and you are 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 than just the author 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 (comp.os.linux 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 one of 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).

Note, however, that I will be mostly out of touch with the net until the beginning of September. I will only read my mail every now and then, so do not be alarmed if I do not answer at once.

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 other things like that. They are mostly for my own benefit and for your amusement, they are not things that I am hoping someone else will write for me.

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

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

For reference: This is ALPHA version 1, hopefully released 1993-07-21.

Chapter 2

Introduction

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but who already master at least the basics of normal usage, which means roughly the material covered by the (as yet unpublished) Linux User's Guide.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

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

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

This manual is not intended to be used completely by itself. Plenty of the rest of the Linux documentation is also important for system administrators. After all, a system administrator is just a user with special privileges. A very important resource are the man pages, which should always be consulted when a command is not familiar. (Unfortunately, as of this writing, the state of the Linux man pages is not very encouraging. Things are improving, however.)

While this manual is targeted at Linux, a general principle has been that it should be useful

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

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

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

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

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

Lars Wirzenius / Linux docs
Ohratie 16 C 198
SF-01370 VANTAA
Finland

(this contact information should be valid in 1993 and for at least a couple of years more, but probably won't be after five years or so).

The LaTeX source code for this manual and other manuals of the Linux documentation set, as well as pre-formatted versions for popular output media, are available in electronic form on a number of anonymous FTP sites on the Internet. The most important ones are `tsx-11.mit.edu`, `sunsite.unc.edu`, and `nic.funet.fi`. The Getting Started guide has a more complete list, as well as a list of other sources, and instructions on how to get the files.

2.1 Typographical Conventions

Bold Used to mark **new concepts**, **WARNINGS**, and **keywords** in a language.

italics Used for *emphasis* in text, and occasionally for quotes or introductions at the beginning of a section.

slanted Used to mark **meta-variables** in the text, especially in representations of the command line. For example,

`ls -l` *foo*

where *foo* would "stand for" a filename, such as `/bin/cp`.

`Typewriter` Used to represent screen interaction, as in

```
$ ls -l /bin/cp
-rwxr-xr-x  1 root    wheel    12104 Sep 25 15:53 /bin/cp
```

Also used for code examples, whether it is "C" code, a shell script, or something else, and to display general files, such as configuration files. When necessary for clarity's sake, these examples or figures will be enclosed in thin boxes.

Key Represents a key to press. You will often see it in this form:

Press return to continue.

◇ A diamond in the margin, like a black diamond on a ski hill, marks "danger" or "caution." Read paragraphs marked this way carefully.

2.2 The Linux Documentation Project

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

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

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

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

with the line

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

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

2.3 Copyright, Trademarks, Other Legalese

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

Ohratie 16 C 198, SF-01370 Vantaa, FINLAND
`lars.wirzenius@helsinki.fi`

Chapter 3

A Short Tutorial

This chapter is a quick introduction to and overview of LINUX system administration. It appears in two places in the LINUX documentation: in the System Administrator's Guide, and in the Getting Started manual. The reason for this is not because the authors have discovered cut-and-paste in the editor, but because we think that it serves a useful purpose in both places, and because we do not want readers of either manual to have to find the other manual in order to read this part.

We have tried to cover here the most important things about system administration you need to know when you use LINUX, in sufficient detail to get you comfortably started. In order to keep it short and sweet, we have only covered the very basics, and have skipped many an important detail, so you really should read the whole System Administrator's Guide, if you are serious about using LINUX. It will help you understand better how things work, and how they hang toghether. At least skim through it so that you know what it contains and know what kind of help you can expect from it.

This chapter assumes that your system is up and running, so there are no installation instructions as such here. See the Getting Started manual for those.

3.1 About `root`, Hats, and the Feeling of Power

LINUX differentiates between different users, so that what they do to each other and to the system can be regulated (one wouldn't want anybody to be able to read one's love letters, for instance). Each user is given an **account**, which includes a username, home directory, and so on. In addition to the real people, there are special accounts, or special imaginary users, which have special privileges. The most important of these is the **root account**, for an imaginary user called `root`. Ordinary users are generally restricted so that they can't do harm to anybody else on the system, just to themselves. [1] They are, for example, not allowed to remove programs installed in `/bin` or `/usr/bin`, or to change the programs there, or to look at other people's files (unless those people explicitly allow it by setting

[1] It *does* happen that an ordinary user can cause a lot of trouble. Usually this is due to some bug, or misconfigured system software.

the permissions of their files and directories to a friendlier level).

There are no such restrictions on **root**. He (the person using that account, that is) can read any file, change any file, remove any file, change permissions and ownerships of any file to anything, format the hard disk, and so on. The basic idea is that the person or persons who take care of a system logs in as **root** whenever it is necessary to do system administration work that can't be done as a normal user. However, since **root** can do anything, it is easy for him to make mistakes that have catastrophic consequences. A simple, innocent

```
rm -rf *                    DON'T DO THIS
```

which for an ordinary user might only wipe out some unimportant thesis (or whatever he's been doing the last five years), could remove every single file in the system if given by **root**. This possibility of truly horrendous mistakes means two things: don't make mistakes, and be prepared to fix those that you do anyway. The best way to avoid doing really bad mistakes is to not do things as **root** if you can avoid it.

That last advice is so good it deserves to be repeated and highlighted:

Don't do things as `root` if you can avoid it!

Put another way, if you picture using the **root** account as wearing a special magic hat that gives you lots of power, so that you can, by waving your hand, destroy entire cities, it is a good idea to be a bit careful about what you do with your hands. Since it is easy to move your hand in a destructive way by accident, it is not a good idea to wear the magic hat when it is not needed, despite the wonderful feeling.

3.2 Booting and Shutting Down

The simplest way to boot is to use a boot floppy. This involves having the kernel on a floppy and have that floppy in the first floppy drive when you turn on the power (or after you do a reset or ctrl-alt-del). The kernel will then automagically load itself from the floppy and that's that.

Another way is to use LILO, or another boot loader, a program that resides in the boot sector of your hard disk (or hard disk partition), and is automatically started if there is no floppy in the first floppy drive. LILO (or whatever) will then load Linux from the hard disk and everything will be jolly.

Shutting down a running Linux system is a bit trickier than shutting down an MS-DOS system, but not much. The best way is to use the **shutdown(8)** command. First quit all programs you run and log out from all sessions, then log in as root and give the command

shutdown *when*

where *when* is be the number of minutes to wait before shutting down, or the word **now**, which means just that. **shutdown** will ask for a message to be displayed on terminals where people are still logged in (this is useful on machines where there are many users at a time, but pretty useless for single user machines), wait the specified time, and then kill all system processes, write out any unwritten data in the disk cache, and do other things to ensure that the system goes down properly, without doing harm. After this is done, `shutdown` prints a message that you can cut the power. *Don't do it before this message has been printed out!* If you cut the power too early, not everything may be ready for it, and you might cause the disk contents to become garbled (not always in immediately visible ways, either).

3.3 Creating and Removing Users

Each person using the system should have his or her own account. It is seldom a good idea to have several people use the same account.

The easy way to add users is to use an interactive command that asks for the required information and then updates the system files automatically. The command is usually called `adduser`, or `useradd`, depending on what software is installed. You should be able to just run them and answer the questions. See the man pages for more information. The chapter "Users" in the System Administrator's Guide describes the process in more detail, including what files should be modified and how.

Similarly, to remove users, use the command `deluser` or `userdel`, and answer the questions.

3.4 Using Floppies and Making Backups

Floppies are usually used pseudo-tapes under Linux. This means that the sectors on the floppy are arranged in some linear order, and those are treated as if they were back to back on a tape. Typically, you use `tar(1)` to handle the floppies. It is also possible to use the floppies as disks, or course, in which case you need to create a filesystem on them with `mkfs(8)`, just as on the hard disk. Then you mount the floppy when you want to use it, and umount it when you are done.

Backups are often done on floppies using the command

`tar -cf` */dev/fd0* `-M`

where */dev/fd0* is the name of the device for the floppy you want to use. `tar` will prompt you for a new floppies when the previous one fills up. The same method can be used for tapes, just substitute the name of the tape for */dev/fd0*. Other ways are described in chapter on backups in the Administrator's Guide.

3.5 Installing New Software

The SLS distribution of Linux has the command `sysinstall` for installing new packages on the system. It is usually used as

`sysinstall -install` *package.tgz*

where *package.tgz* is the name of the compressed tar file that contains the package. This method can be used both for SLS packages, and other packages that follow the same conventions: the package should be unpacked as root in the root directory, and all files will then drop into the correct places. **sysinstall** also keeps a log of what files have been installed by it, so that the whole package can be uninstalled by

`sysinstall -remove` *packagename*

See the man page for **sysinstall** for more information.

Instead of using **sysinstall**, you can also just use the command

`cd /; tar xzvf` *package.tar.Z*

which is the essence of **sysinstall**. This becomes necessary for those packages that should not be unpacked in the root directory, but in some place else instead, typically **/usr** or **/usr/local**.

However, before you start wildly unpacking things, you should look at the contents of a package with

`tar tzvf` *package.tar.Z*

just in case it is funny (perhaps it overwrites **/etc/passwd**, and then what do you do?).

The above methods are good for binary distributions. Some programs are distributed in source code only, and they have to be installed in a different way. Put simply, you first unpack all the sources in a directory dedicated for them, compile the sources, then copy the executable files and other files to their proper places. In practice, things can get much more tricky than this, especially if the program was not written for the operating system, compiler and libraries you use. If there are problems, you often have to be a programmer to solve them. Before you start porting, ask around (in **comp.os.linux**, for example) to see if anyone already has ported the program, or something similar.

3.6 What To Do In An Emergency?

META: Using boot and root disks. Running fsck manually, from root disk for root fs on hard disk.

META: what to do if you have lost the root password

What should you do if something really bad happens? To begin with, keep calm. Panic and desperation don't help. It is much easier to be calm if you have little to lose when the computer goes crazy, of course, so being prepared and making it impossible to lose everything in one disaster is a good way to keep one's calm. Refer to a previous section in this chapter for information about making backups. If you always have backups of everything, the worst thing that can happen is that you have to install everything from scratch.

It is not always necessary to install from scratch, of course. Often one can fix the problem by editing some file, or installing only a few programs anew. It is fairly easy to make it impossible to boot or login as root—by removing **/etc/getty**, for example—so an alternative, known-to-work way to boot and login is a good thing to have. The best way is to prepare a special floppy or two that can be booted and that has enough tools that you can repair any damage. See the appropriate chapter in the Administrator's Guide for instructions on how to create one. The installation floppies of most LINUX distributions work also.

Chapter 4

Using Disks and Other Storage Media

This chapter tells about different types of storage media (hard disks and floppies) and about some ways of using them. Unfortunately, because I lack the equipment, I cannot tell you much about using other types of media, such as tapes or CD-ROMs.

4.1 Definitions of Terms

We'll begin by explaining the important concepts. If you already know what a partition and a filesystem are, you can probably skip this section. Even if you do read this section, be aware that there is a lot of detail here, and that it is sufficient to understand the principles.

4.1.1 Hard Disks, Floppies, Other Types of Media

META: hd: cylinders, tracks, sectors, heads, platters, disk surfaces; translations

4.1.2 Formatting

META: why is it necessary; what does it do; what kinds of disks do not need formatting

4.1.3 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 and 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. Usually only hard disks are partitioned, floppies are not (do not know about CD-ROMs and other disks).

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** 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 which 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 boot sector contains another small program which 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, however, 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.

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

META: explain extended partitions: a way to circumvent the maximum of 4 partitions in the original design; partition id

META: explain partition types (the partition type byte in the partition table) and that it is ignored by Linux but that some other OS's do not ignore it; DR-DOS strips hibit.

META: figure: show a hard disk layout, with the first sector in detail

4.1.4 Filesystems

A **filesystem** is the methods and data structures that an operating uses to keep track of files on a partition (or disk, if it is not or cannot be partitioned), 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: programs which operate on a disk or partition will usually operate on the raw disk sectors directly, completely disregarding whatever filesystem there might be. Programs and operations which require a filesystem won't work on a partition that doesn't contain one (or that contains one of the wrong type). Also, using a partition at the same time both as a filesystem and as a raw bunch of sectors will often lead to trouble.

Before a partition or disk can be used to store files, it needs to be initialized, the bookkeeping data structures need to be written to the disk. This process is usually called **making a filesystem** in UNIX circles, and **formatting** in—among others—MS-DOS circles (although, strictly speaking, formatting is the task of magnetically initializing a disk, and is actually separate from making a filesystem).

META: Explain basics of Unix filesystem structure (superblock, inodes, data blocks, distributed superblocks and inodes) and that DOS is different

4.1.5 Disk Buffering

Reading from a disk, of any type (except a RAM disk), is very slow, when compared to accessing RAM memory. If one happens to have some extra memory, it is usually worth it to use that to remember to the data one just read from the disk. This way, if one happens to need the same information twice, one can just fetch the remembered data instead of having to go all the way to the hard disk to read it. This is the principle of disk buffering. It is described in more detail below, in section 4.7, "The Buffer Cache".

4.1.6 Virtual Memory and Swap Space

LINUX supports virtual memory, i.e. using a part of the hard disk as an extension of RAM so that the effective size of usable memory grows correspondingly. The kernel will write the contents of a not-right-now used part 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 (not necessarily to the same place). This is all 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 hard 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** (as this is called) is called the **swap space**.

4.2 Preparing a Hard Disk For Use

4.2.1 Formatting

META: IDE/SCSI: not really needed; is there a program for HD formatting?; bad blocks are handled by the controller or mkfs or mkswap

4.2.2 Partitioning

META: fdisks for different operating systems: use each one's own; beware of starting partitions at non-track boundaries; non-destructive re-partitioning

4.3 Preparing a Floppy For Use

4.3.1 Selecting Floppy Type

META: proper device file; automatic selection; setfdprm

4.3.2 Formatting

META: fdformat

4.4 Using a Filesystem

4.4.1 Types of Filesystems

LINUX supports several types of filesystems. As of this writing they are:

minix-1
: The 'original' LINUX filesystem. LINUX was first developed on Andrew Tanenbaum's Minix operating system, and it was natural for Linus to use the Minix filesystem layout instead of inventing his own (the code was all his, only the layout was the same)[1]. The '-1' is used here to differentiate between other versions (some developed for LINUX, others for Minix). The minix-1 filesystem is usually considered to be the most stable one, as it has been in use longer than the others. It is fast, but only supports one of the time fields for files (the time of the last modification; the times for the last access and last modification of the inode are missing), partitions up to 65 megabytes, and, perhaps most importantly, limits filenames to 14 characters. LINUX extends the Minix version by adding symbolic links.

minix-30
: This is an enhanced version of the minix-1 filesystem, developed for LINUX. The only difference is that filenames can be 30 characters long. (Actually, minix-1 and minix-30 are the same filesystem for the kernel, and the kernel allows any length, up to 30, for the filenames. Since other programs—most notably Minix itself—only understand 14 character names, that case is kept separate and is given a separate name.)

extfs
: The 'extended filesystem', developed by Remy Card. It is originally based on the minix-1 filesystem by Linus, but doesn't have much to do with it anymore. extfs adds filenames up to 255 characters, the two missing times, and support for partitions up to XXX gigabytes. It is slower than the minix filesystems, ext2fs or xiafs.

ext2fs
: The 'second extended filesystem', also developed by Remy Card. It is a rewrite of extfs, with improved algorithms (greatly improving the speed over extfs) and a modified layout on the disk that allows for future growth (the layout has room for a lot of things that are at most planned). Ext2fs is the most featureful of the Linux filesystems.

xiafs
: This is a modification of the minix-1 filesystem that allows for longer filenames (up to 255 characters) and larger partitions (up to XXX megabytes), but doesn't have much more new things. Written by Frank Xia. It is perhaps the fastest filesystem,

[1] Especially since he only had 40 MB of hard disk, and didn't have room for another partition...

but still suffers from the single time field in the inode that the minix-1 filesystem allows.

procfs
: This is not really a filesystem in the traditional sense at all, in that it doesn't give access to files stored on a disk. Instead, it is a 'phantom' filesystem, where files and directories are created by the kernel (more accurately, the procfs driver in the kernel) based on various data structures in the kernel. Most importantly, it creates a directory for each running process, each directory containing a few files that contain information about the process. This way it is easy to access information about processes. In traditional Unices, one has to access the `/dev/kmem` special file (a special file that is an image of all the memory from the kernel's point of view), which is both cumbersome and a security risk. With the procfs, the access to the information is controlled.

  In addition to providing information about processes, the procfs also provides access to various other information that the kernel maintains. See the `procfs(5)` man page for more information.

dosfs
: This gives access to the FAT filesystem used by MS-DOS. It is only useful if you need to access floppies or hard disk partitions that have that filesystem.

xenixfs
: Another filesystem for accessing files from another operating system. This time the Xenix operating system.

A note about the two extended filesystems, and extended partitions: *Do not confuse the two!* An extended partition, and the two extended filesystems have nothing in common. In particular, the extended filesystems do not need to be installed on an extended partition.

META: explain 90% full in ext2fs

4.4.2 Selecting a Filesystem

META: give advice about which filesystem should be used

META: give some speed comparisons (iozone?)

META: give a feature chart

4.4.3 A Comparison Between Ext2fs and Xiafs

The following is the essentials of a comparison between ext2fs and xiafs posted by Stephen Tweedie (e-mail address `sct@dcs.ed.ac.uk`) to the Usenet newsgroup `comp.os.linux.announce` on May 13, 1993 (so it may not be quite up to date any longer). The rest of the text in this paragraph is from the comparison, with minor edits by the editor of the book.

History

The original Linux filesystem was a simple copy of the Minix operating system's own filesystem. It inherited Minix's restrictions of 14 character filenames and 64 MB partitions, but has since been enhanced to support filenames of up to 30 characters. It is the oldest and most trusted of the Linux filesystems.

In an effort to lift the filename and partition restrictions of the Minix fs, Remy Card developed the extfs. This supports 4 GB files and partitions, and 255 character filenames. Unfortunately, it uses a linked list to hold its free block and free inode information; this is slightly more memory-efficient than using a bitmap, but is a lot more prone to fragmentation. Performance of the extfs drops seriously after a partition has been used for a while. The linked-list technique is now acknowledged to be a bad idea, although extfs is still very useable if the partition is defragmented regularly.

Within a couple of weeks of each other, the xiafs and ext2fs filesystems were announced on an unsuspecting public. Chaos has reigned ever since.

Xiafs and ext2fs were both intended to provide fast and efficient alternative filesystems which did not suffer from the performance problems of the extfs or the restrictions of the minix fs.

Overview

xiafs was based closely on the original minix fs code, but enhanced to be both faster and less restrictive. ext2fs was a more ambitious effort which was designed to add new features and to take filesystems a step forward. It has taken longer to debug ext2fs; they are *probably* both bug-free now, but xiafs has been a stable beta release for a lot longer. They both have good support for bad-block mapping.

The biggest difference between the two filesystems is in their design philosophy. Both filesystems were originally designed to fill the basic requirements of speed and capacity. However, beyond this, the priority for xiafs was to change as little as possibly of the minix fs, and so to produce a stable filesystem; whereas ext2fs was designed from the start to have room to expand.

The net effect is that the xiafs kernel code has had no bugs reported since mid-February, whereas the ext2fs code has taken significantly longer to debug. The ext2fs is now in stable beta release. It has had much less time to prove its reliability in this state than xiafs; but you get more functionality out of ext2fs. This is the crux of the choice between the two.

Looking to the future, ext2fs has much more room to expand in than xiafs. There are several significant enhancements planned to ext2fs, and users will be able to take advantage of the new systems without reformatting any of their old ext2fs partitions. Upgrades either planned or under development include transparently compressed files, POSIX-style access control features, undeletion of lost files and fragments (a technique for reducing the disk usage overheads especially for for small files).

Xiafs and ext2fs are both significantly faster than the previous minix or extfs filesystems (the extfs in particularly would get terribly slow with time). Although tests to date have placed the ext2fs ahead of the xiafs in terms of speed, the differences between them are not generally as significant as the speedup when compared to the older fs's. Ext2fs seems to suffer less from long-term fragmentation.

Ext2fs is the standard enhanced filesystem for SLS; xiafs was adopted as HJ's distribution filesystem for his base system and rootdisk releases.

Xiafs is pretty stable just now. Extensions to ext2fs are still being actively developed.

Limits

The table below summarizes the differences between ext2fs and xiafs.

	ext2fs	xiafs
Partition size	4 GB	4 GB
File size	4 GB	64 MB
Filename length	255	248
Inodes	2^{31}	2^{31}
Inode size	128 B	64 B

Both filesystems may support, sometime in the future, up to 2 TB partitions in the future (and both will be compatible with current filesystem formats). The xiafs file size limit will rise to 1 GB once kernel support for larger block sizes is introduced. The xiafs filename limit can be raised to 255 by changing a constant and recompiling the kernel.

Ext2fs:

- Pros
  - flexible, expandable
  - fast
  - after a crash, lost files can often be automatically recovered (e2fsck will place them in /lost+found)
  - is the only filesystem to support a changeable filesize limit (through the standard kernel resource limit mechanism)
  - supports a "valid" flag which is set when the filesystem is cleanly unmounted, to automatically identify filesystems which may need repair after a crash
- Cons
  - larger inode overhead (though still insignificant against other overheads)
  - some bugs in versions before 0.3 (no bugs known or reported in 0.3, but these are early days yet...)
- Coming soon
  - POSIX access control lists
  - transparent compressed file support

  - fragments
  - large block sizes
  - undelete files
  - recovery from superblock corruption

Xiafs:

- Pros
  - designed from the start as a minimal but reliable enhancement to the robust minix fs, so there have been no bugs reported recently
- Cons
  - not as much room for upgrade.
  - generally lower limits on filesize and name lengths (but see note in the limits section above)
- Coming soon
  - large block sizes

4.4.4 Making a Filesystem

META: mkfs (minix), mkefs (extfs), mke2fs (ext2fs), mkxfs? (xiafs); also, the front-end that allows one to say just mkfs for all types

4.4.5 Mounting and Unmounting

Before one can use the contents of disk or partition as a filesystem, it has to be **mounted**. This means that one tells the operating system that one wants to use the filesystem on the partition (or floppy, or whatever). 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 new filesystem is the contents of an existing subdirectory in some already mounted filesystem. For example, one might mount a floppy so that it looks like the contents of the `/mnt` directory. The command would be

```
mount /dev/fd0 /mnt
```

(`/dev/fd0` being the device file for the floppy drive), and one would then say that "**`/dev/fd0` is mounted on `/mnt`**". Then, to look at the contents of the floppy, one would use the command

```
ls /mnt
```

just as if `/mnt` were any random directory. Note the difference between the device file, `/dev/fd0`, and the mounted-on directory, `/mnt`. 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 alert reader has already noticed that there is a slight logistical problem caused by the need to have an existing directory in an already mounted filesystem before one can mount a new one. 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[2]. 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.

When a filesystem no longer needs to be mounted, it can be unmounted. This is done with the `umount(8)`[3] command. For example, to unmount the floppy in the previous example, one would use the command

```
umount /dev/fd0
```

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 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.

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.

4.5 Using 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 supports using several swap partitions and/or swap files at the same time. This means that if you only occasionally need unusually much swap space, you can set up an extra swap file every such time, instead of keeping the whole amount allocated at the same time.

4.5.1 Creating a Swap Space

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

The bit about holes is important: 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

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

[3] 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.

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 binaries.

The swap file is used to reserve the disk space so that the kernel does not, when it needs to swap out a page, have 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 wayto create the swap file, which avoids creating holes, is using the following command:

```
dd if=/dev/zero of=/swapfile bs=1024 count=N
```

where N is the size of the swap file in kilobytes. It is best for N to be a multiple of 4, because the kernel writes out **memory pages**, which are 4 kilobytes in size.

A swap partition is also not special in any way. You create is 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.

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

```
mkswap swapspace size
```

Substitute the name of the swap file or swap partition for *swapspace* above, and the size, in kilobytes, for *size*. Note that the swap space is still not in use yet.

4.5.2 Putting Swap Space Into Use

META: swapon, swapoff, mount -av and /etc/fstab

4.5.3 Sharing Swap Space With Other Operating Systems

META: mswin, os/2, 386bsd

4.6 Using Raw Floppies

META: actually, any disk; tar, cpio, afio, dd, others; why are raw disks useful?

4.7 The Buffer Cache

META: what it is; how it works, and how it affects the free memory; why it is beneficial; when it is not beneficial; things to be careful with; sync, /etc/update;

4.8 Allocating Disk Space

4.8.1 Partitioning Schemes

In what way should a hard disk be partitioned? This is a tricky question that requires a bit of thinking on your part to answer correctly. There is no universally correct way to do it, there are to many factors involved.

The traditional way is to have a (relatively) small root filesystem, which contains `/bin`, `/etc`, `/dev`, `/tmp`, and other stuff that is needed to boot the system. 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 it is therefore easier to fix any problems caused by the crash. The directory tree below `/usr`, the users' home directories (often under `/home`) and the swap space are each in their own partitions or disks as well.

The problem with having many partitions is that it splits the total amount of free disk space in 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. Additionally, it is less painful to backup (and restore) a small partition.

For a small hard disk (and assuming you don't do kernel development), the best way to go is probably to have just one partition. For large hard disks, when the problem with split up free space is practically non-existant, 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. (If you have more than two, put programs or home directories on them as necessary.)

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 in a good way.

4.8.2 Space For Files

4.8.3 Swap Space

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

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. 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 meg 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.)

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 overdo 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 only need to subtract the size of your physical memory from the total memory need, 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. On these systems, the usable swap area begins after the size of the physical memory.)

4.8.4 Adding More Disk For Linux

META: also taking some away

4.8.5 Tips For Saving Disk Space

Chapter 5

Directory Tree Overview

META: Give sample contents of important files, e.g. /etc/passwd, /etc/group.

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

META: timezone files, /etc/skel.

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

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

5.1 The `/etc` and `/usr/etc` Directories

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

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

META: the items below should be sorted

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

META: Describe /etc/rc.[0-9] better: which is what, and so on. Give and explain in detail a sample /etc/rc (in a subchapter?).

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

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

`/etc/disktab` I don't know what this is. Presumably some kind of disk parameter table. Doesn't seem to be used in Linux.

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

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

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

`/etc/uugetty` Another version of `getty`. [XXX - what are the differences? Is uugetty better for uucp and dial in/out on the same line? Yes. Is it used anymore? /dev/cua? and so on, after all.]

`/etc/gettydefs` On my system this is just a symlink to `/etc/gettytab`. Presumably some Unices use `gettydefs`, others `gettytab`, and Linux has both names to be compatible with both camps.

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

META: I think this is something like it, gotta find a man page first. (See uucp chapter.)

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

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

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

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

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

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

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

META: .hushlogin?

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

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

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

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

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

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

`/etc/securetty` Identifies secure terminals. `root` can login only from the terminals listed in this file. Typically only the virtual consoles are listed, so that it becomes impossible (or at least much 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.

`/etc/startcons` I don't know, looks like a script to set colors on VCs on my system.

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

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

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

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

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

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

5.2 Devices, `/dev`

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

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

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

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

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

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

See section 5.5 for an example of how the shared libraries might become fouled up.

The drawback of `/sbin` is that statically linked binaries take a lot more disk space. When a binary of `ln` might be a couple of kilobytes when linked with shared libraries, it might be a couple of hundred kilobytes if linked statically. If you are willing to have to boot if you mess up your shared libraries—and have an emergency boot disk always available—it is never necessary to have `/sbin`. If you can't afford to boot, and you can afford the disk space, then `/sbin` is a very good idea. There is not, however, a single answer that is correct for everyone on this issue.

5.4 The `/usr/lib` Directory

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

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

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

[1] SunOS, and some LINUX distributions at least

versions are in `/usr/lib/shlib` (the location has changed during the development of LINUX, and may change again).

5.5 Shared Library Images, /lib

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

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

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

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

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

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

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

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

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

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

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

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

META: fixing the missing link

5.6 C/C++ Programming, `/usr/include`, `/usr/g++include`

META: C++ should be typeset prettier.

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

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

5.7 Source Codes, `/usr/src`

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

5.8 Administrativia, `/usr/adm`

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

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

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

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

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

META: linux has chapter 9 as well?

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

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

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

5.10 More Manuals, `/usr/info`

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

5.11 `/home`, Sweet `/home`

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

5.12 Temporary Files, `/tmp` and `/usr/tmp`

META: /tmp on root can be a bad idea

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

On many systems, `/tmp` is automatically cleaned at boot time, so that the temporary files won't remain and take up space when they are no longer needed. `/usr/tmp` is usually not cleaned that way. This is done in `/etc/rc` or one of the scripts it calls.

5.13 The `/mnt` and `/swap` Directories

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

The `/swap` directory is similarly only a commonly used mount point. As the name implies, it is intended as the mount point for the swap partition.

5.14 Process Information, `/proc`

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

5.15 The /install Directory

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

Chapter 6

Boots, Shutdowns, Logins, and Background Demons

This sections explains what goes on when a LINUX system is turned on and off, and how it should be done properly. Logging in and background demons are somewhat related to this, so they are also covered here.

META: don't power cycle too often

6.1 Booting

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 allright[1], and will then start the actual booting. It will choose a disk drive (typically the first floppy drive, if there is a floppy inserted, otherwise the first hard disk, if one is installed in the computer) and read its very first sector. This is called the **boot sector**; for hard disk, it is also called the **master boot record** (since a hard disk can contain several partitions, each with their own boot sectors).

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

When booting from the hard disk, the code in the master boot record will examine the partition table, identify the active partition (the partition that is marked to be bootable), read the boot sector from that partition, and then start the code in that boot sector. The code in the partition's boot

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

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

When booting with LILO, it will normally go right ahead and read in and boot the default kernel. It is also possible to configure LILO to be able to 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 pass arguments to the kernel at boot time, or overriding some configuration options built into the kernel. Hence, it is usually the best choice. Among the alternatives are bootlin and bootactv[2]

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

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

- If the kernel was installed compressed, it will first uncompress itself. (The beginning of the compressed kernel contains a small, uncompressed program that does this.)
- If you have a super-VGA card that LINUX thinks it recognizes and that has some special text modes (such as 100 columns by 40 rows[3]), LINUX asks you which mode you want to use. (During the kernel compilation, it is possible to preset a video mode, so that this is never asked.)
- 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
```

[2] I don't know much about any of the alternatives. If and when I learn, I will add more descriptions.
[3] Incidentally, this is the mode preferred by Linus himself.

```
tty01 at 0x02f8 (irq = 3) is a 16450
lp_init: lp1 exists (0), using polling driver
Memory: 7332k/8192k available (300k kernel code, 384k reserved, 176k data)
Floppy drive(s): fd0 is 1.44M, fd1 is 1.2M
Loopback device init
Warning WD8013 board not found at i/o = 280.
Math coprocessor using irq13 error reporting.
Partition check:
  hda: hda1 hda2 hda3
VFS: Mounted root (ext filesystem).
Linux version 0.99.pl9-1 (root@haven) 05/01/93 14:12:20
```

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

- After all this configration business is complete, LINUX switches the processor into protected mode. The switch is not visible to the user, but is an important step for the kernel. A big leap for the kernel, a small step for the userkind.
- Then the kernel will try to mount the root filesystem. The place—which floppy or partition—is configurable at compilation time, with `rdev` (see man page), or with LILO (see LILO documentation). The filesystem type is detected automatically. If the mounting of the root filesystem fails, the kernel panics and halts the system (there isn't much one can do, anyway).
- After this the kernel starts the program `/etc/init` in the background (this will always become process number 1), which runs the shell script `/etc/rc`. The script will start all the background programs that take care of things like printer queues and such. It runs some other important commands as well.

  META: Need to discuss various flavor of init.
- The init program then starts a `getty` for virtual consoles and serial lines as configured in `/etc/gettytabs`. `getty` is the program which lets people log in via virtual consoles and serial terminals.
- After this, the boot is complete, and the system is up and running normally.

6.2 Shutting Down

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

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

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

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

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

META: /etc/shutdown.rc

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

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

META: halt, fastboot, options to shutdown

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

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

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

6.3 Background Processes and Demons

A **demon** or **daemon** is a program running in the background, invisibly to users, that provides the system with some sort of useful service. From *The Jargon File* (also published as a book under the name *The New Hacker's Dictionary*):

> **daemon** /day'mn/ or /dee'mn/ [from the mythological meaning, later rationalized as the acronym 'Disk And Execution MONitor'] n. A program that is not invoked explicitly, but lies dormant waiting for some condition(s) to occur. The idea is that the perpetrator of the condition need not be aware that a daemon is lurking (though often a program will commit an action only because it knows that it will implicitly invoke a daemon). For example, under **ITS** writing a file on the **LPT** spooler's directory would invoke the spooling daemon, which would then print the file. The advantage is that programs wanting (in this example) files printed need neither compete for access to nor understand any idiosyncrasies of the **LPT**. They simply enter their implicit requests and let the daemon decide what to do with them. Daemons are usually spawned automatically by the system, and may either live forever or be regenerated at intervals.
>
> Daemon and **demon** are often used interchangeably, but seem to have distinct connotations. The term 'daemon' was introduced to computing by **CTSS** people (who pronounced it /dee'mon/) and used it to refer to what ITS called a **dragon**. Although the meaning and the pronunciation have drifted, we think this glossary reflects current (1993) usage.

> **demon** n. 1. [MIT] A portion of a program that is not invoked explicitly, but that lies dormant waiting for some condition(s) to occur. See **daemon**. The distinction is that demons are usually processes within a program, while daemons are usually programs running on an operating system. 2. [outside MIT] Often used equivalently to **daemon** — especially in the **UNIX** world, where the latter spelling and pronunciation is considered mildly archaic.
>
> Demons in sense 1 are particularly common in AI programs. For example, a knowledge-manipulation program might implement inference rules as demons. Whenever a new piece of knowledge was added, various demons would activate (which demons depends on the particular piece of data) and would create additional pieces of knowledge by applying their respective inference rules to the original piece. These new pieces could in turn activate more demons as the inferences filtered down through chains of logic. Meanwhile, the main program could continue with whatever its primary task was.

The most basic background process, but still an important one, is `/etc/update`, which does a `sync(2)` every 30 seconds.

Other important demons are `crond` and `syslogd`. See their man pages for more information.

META: should give more examples

6.4 Logging In

The `login:` prompt on the terminal is written by getty. It reads the username, then executes the `login(1)` program, which reads the password (if a password is required), and starts the shell, and the shell will do whatever it will do. If the file `/etc/nologin` exists, logins are disabled. That file is typically created by `shutdown(8)` and relatives.

META: much more detail