

From Power Up To Bash Prompt

Table of Contents

<u>From Power Up To Bash Prompt</u>	1
Greg O'Keefe, gcokeefe@postoffice.utas.edu.au	1
1.Introduction	1
2.Hardware	1
3.Lilo	1
4.The Linux Kernel	1
5.The GNU C Library	2
6.Init	2
7.The Filesystem	2
8.Kernel Daemons	2
9.System Logger	2
10.Getty and Login	2
11.Bash	3
12.Basic Commands	3
13.Building Software From Source	3
14.Conclusion	3
15.Administrivia	3
1. Introduction	3
10. Getty and Login	4
10.1 Configuration	5
10.2 Exercises	5
10.3 More Information	5
11. Bash	5
11.1 Configuration	6
11.2 Exercises	6
11.3 More Information	7
12. Basic Commands	7
13. Building Software From Source	8
13.1 How I Built My System	8
13.2 Random Tips	9
13.3 More Information	10
14. Conclusion	10
15. Administrivia	10
15.1 Copyright	11
15.2 Homepage	11
15.3 Feedback	11
15.4 Acknowledgements	11
15.5 Change History	12
0.5 → 0.6	12
15.6 TODO	12
2. Hardware	12
2.1 Configuration	13
2.2 Exercises	13
2.3 More Information	13
3. Lilo	14
3.1 Configuration	14
3.2 Exercises	14

Table of Contents

3.3 More Information	15
4. The Linux Kernel	15
4.1 Configuration	16
4.2 Exercises	16
4.3 More Information	17
5. The GNU C Library	17
5.1 Configuration	18
5.2 Exercises	18
5.3 More Information	18
6. Init	19
6.1 Configuration	20
6.2 Exercises	20
6.3 More Information	21
7. The Filesystem	21
7.1 Configuration	22
7.2 Exercises	22
7.3 More Information	22
8. Kernel Daemons	23
8.1 Configuration	24
8.2 Exercises	24
8.3 More Information	24
9. System Logger	24
9.1 Configuration	24
9.2 Exercises	25
9.3 More Information	25

From Power Up To Bash Prompt

Greg O'Keefe, gcokeefe@postoffice.utas.edu.au

v0.6, February 2000

This is a brief description of what happens in a Linux system, from the time that you turn on the power, to the time that you log in and get a bash prompt. It is organised by package to make it easier for people who want to build a system from source code. Understanding this will be helpful when you need to solve problems or configure your system.

[1.Introduction](#)

[2.Hardware](#)

- [2.1 Configuration](#)
- [2.2 Exercises](#)
- [2.3 More Information](#)

[3.Lilo](#)

- [3.1 Configuration](#)
- [3.2 Exercises](#)
- [3.3 More Information](#)

[4.The Linux Kernel](#)

- [4.1 Configuration](#)
- [4.2 Exercises](#)
- [4.3 More Information](#)

5.The GNU C Library

- [5.1 Configuration](#)
- [5.2 Exercises](#)
- [5.3 More Information](#)

6.Init

- [6.1 Configuration](#)
- [6.2 Exercises](#)
- [6.3 More Information](#)

7.The Filesystem

- [7.1 Configuration](#)
- [7.2 Exercises](#)
- [7.3 More Information](#)

8.Kernel Daemons

- [8.1 Configuration](#)
- [8.2 Exercises](#)
- [8.3 More Information](#)

9.System Logger

- [9.1 Configuration](#)
- [9.2 Exercises](#)
- [9.3 More Information](#)

10.Getty and Login

- [10.1 Configuration](#)
- [10.2 Exercises](#)
- [10.3 More Information](#)

11. [Bash](#)

- [11.1 Configuration](#)
- [11.2 Exercises](#)
- [11.3 More Information](#)

12. [Basic Commands](#)

13. [Building Software From Source](#)

- [13.1 How I Built My System](#)
- [13.2 Random Tips](#)
- [13.3 More Information](#)

14. [Conclusion](#)

15. [Administrivia](#)

- [15.1 Copyright](#)
- [15.2 Homepage](#)
- [15.3 Feedback](#)
- [15.4 Acknowledgements](#)
- [15.5 Change History](#)
- [15.6 TODO](#)

[Next](#) [Previous](#) [Contents](#) [Next](#) [Previous](#) [Contents](#)

1. Introduction

I find it frustrating that many things happen inside my Linux machine that I do not understand. If, like me, you want to really understand your system rather than just knowing how to use it, this document should be a good place to start. This kind of background knowledge is also needed if you want to be a top notch Linux problem solver.

I assume that you have a working Linux box, and understand some basic things about Unix and PC hardware. If not, an excellent place to start learning is Eric S. Raymond's [The Unix and Internet Fundamentals HOWTO](#) It is short, very readable and covers all the basics.

The main thread in this document is how Linux starts itself up. But it also tries to be a more comprehensive learning resource. I have included exercises in each section. If you actually do some of these, you will learn much more than you could by just reading. There are also links to source code downloads. The reason for this is that I hope some readers will undertake the best Linux learning exercise that I know of, which is building a system from source code, or "rolling your own". Giambattista Vico, an Italian philosopher (1668–1744) said "verum ipsum factum", understanding arises through making. Thanks to Alex (see [acknowledge](#)) for this quote.

Packages are presented in the order in which they appear in the system startup process. This means that if you install the packages in this order you can reboot after each installation, and see the system get a little closer to giving you a bash prompt each time. There is a reassuring sense of progress in this.

There are choices to make when putting together a Linux system. These are the same choices that the likes of Red Hat and Debian must make when creating a distribution. There are often several free packages available to do the same job. What I describe here are the components that are part of the major Linux distributions. I choose GNU software if it is available because it is better documented, and causes less headaches. It would be possible to get a bash prompt without installing everything I mention here. However I want to describe a base system that can be built on easily, without nasty kludges. For example, one of the init scripts uses awk. (Don't worry if you don't know what awk is) Rather than hack this out of the script, I just install awk.

I recommend that you first read the main text of each section, skipping the exercises and references. Then decide how deep an understanding you want to develop, and how much effort you are prepared to put in. Then start at the beginning again, doing the exercises and additional reading as you go.

[Next](#) [Previous](#) [Contents](#)[Next](#)[Previous](#)[Contents](#)

10. Getty and Login

Getty is the program that enables you to log in through a serial device such as a virtual terminal, a text terminal, or a modem. It displays the login prompt. Once you enter your username, getty hands this over to `login` which asks for a password, checks it out and gives you a shell.

There are many getty's available, but the `util-linux` package, which includes `login` has one called `agetty`, which works fine. This package also contains `clock`, `fdformat`, `mkswap`, `fdisk`, `passwd`, `kill`, `setterm`, `mount`, `swapon`, `rdev`, `renice`, `hexdump`, `more` (the program) and `more` (ie more programs). To keep things simple by minimising the number of packages you have to install, I recommend using `agetty`.

10.1 Configuration

The message that comes on the top of your screen with your login prompt comes from `/etc/issue`. Gettys are usually started in `/etc/inittab`. Login checks user details in `/etc/passwd`, and if you have password shadowing, `/etc/shadow`.

10.2 Exercises

Create a `/etc/passwd` by hand. Passwords can be set to null, and changed with the program `passwd` once you log on. See the man page for this file Use `man 5 passwd` to get the man page for the file rather than the man page for the program.

10.3 More Information

The [util-linux](#) package contains `login` and `agetty`, and lots of other stuff that you will need.

There are lots of other getty's at [Many getty's!](#). `getty_ps` is the most general one, `mingetty` is for virtual terminals only. However, the `util-linux` package which you need for `logon` also contains `agetty`, which works fine.

[NextPreviousContentsNextPreviousContents](#)

11. Bash

If you give `login` a valid username and password combination, it will check in `/etc/passwd` to see which shell to give you. In most cases on a Linux system this will be `bash`. It is `bash`'s job to read your commands and see that they are acted on. It is simultaneously a user interface, and a programming language interpreter.

As a user interface it reads your commands, and executes them itself if they are "internal" commands like `cd`, or finds and executes a program if they are "external" commands like `cp` or `startx`. It also does groovy stuff like keeping a command history, and completing filenames.

We have already seen `bash` in action as a programming language interpreter. The scripts that `init` runs to start the system up are usually shell scripts, and are executed by `bash`. Having a proper programming language, along with the usual system utilities available at the command line makes a very powerful combination, if you know what you are doing. For example (smug mode on) I needed to apply a whole stack

of ``patches" to a directory of source code the other day. I was able to do this with the following single command:

```
for f in /home/greg/sh-utils-1.16*.patch; do patch -p0 < $f; done;
```

This looks at all the files in my home directory whose names start with `sh-utils-1.16` and end with `.patch`. It then takes each of these in turn, and sets the variable `f` to it and executes the commands between `do` and `done`. In this case there were 11 patch files, but there could just as easily have been 3000.

11.1 Configuration

The file `/etc/profile` controls the system-wide behaviour of bash. What you put in here will affect everybody who uses bash on your system. It will do things like add directories to the `PATH`, set your `MAIL` directory variable.

The default behaviour of the keyboard often leaves a lot to be desired. It is actually `readline` that handles this. `Readline` is a separate package that handles command line interfaces, providing the command history and filename completion, as well as some advanced line editing features. It is compiled into bash. By default, `readline` is configured using the file `.inputrc` in your home directory. The bash variable `INPUTRC` can be used to override this for bash. For example in Red Hat 6, `INPUTRC` is set to `/etc/inputrc` in `/etc/profile`. This means that backspace, delete, home and end keys work nicely for everyone.

Once bash has read the system-wide configuration file, it looks for your personal configuration file. It checks in your home directory for `.bash_profile`, `.bash_login` and `.profile`. It runs the first one of these it finds. If you want to change the way bash behaves for you, without changing the way it works for others, do it here. For example, many applications use environment variables to control how they work. I have the variable `EDITOR` set to `vi` so that I can use `vi` in Midnight Commander (an excellent console based file manager) instead of its editor.

11.2 Exercises

The basics of bash are easy to learn. But don't stop there: there is an incredible depth to it. get into the habit of looking for better ways to do things.

Read shell scripts, look up stuff you don't understand.

11.3 More Information

- Australian Bash [mirror](#)
- There is a ``Bash Reference Manual" with this, which is comprehensive, but heavy going.
- Australian readline [mirror](#) You need to download readline separately ((is it an addon type thingy?))
- ((bash tutorials? – if there isn't one around, make one!))
- There is an O'Reilly book on Bash, not sure if it's good.

[NextPreviousContentsNextPreviousContents](#)

12. Basic Commands

You do most things in bash by issuing commands. Most of these commands are small programs. I won't say too much about these. I have just listed the packages that I found I needed. I fear that I may have lost track slightly of what was really necessary and what wasn't. I will fix this when I rebuild my system to test this document. There isn't too much baggage in the list. Most of it is needed for a fully functional Linux system anyway.

Ideally, this list should include all commands specified in The Unix [File Heirarchy Standard](#) and everthing needed to run the basic initscripts that come with the sysvinit dist.

- GNU fileutils [fileutils](#) commands such as cp, dd, ls, ln, mkdir and so on.
- GNU findutils [findutils](#) find and locate commands. Find is needed in an init script ((check your notes on this)).
- textutils? was that needed?? it contains `cat`, which is nice to have so you can look a files. its GNU
- Gawk [gawk](#) GNU's implementation of the awk language. Awk is good for processing records in text files like the system log. It is needed in an init script ((check your notes on this)).
- grep [grep](#) It is needed in an init script ((check your notes on this)).
- sed?? was that needed?? it's a GNU package
- sh-utils contains `hostname`, `stty`, `true`, `false`, `yes`, `who`, `sleep`
- Net Tools: is probably not completely necessary, since `hostname` is in `sh-utils` [net-tools](#). The only thing you *really* need from this package, to get a really basic system up is `hostname`. But the other stuff in here such as, `ifconfig`, `netstat` and `route` will be needed when you want to connect your system to anything else.
- Process Monitoring [procps](#). The main commands in here are `ps` and `top`. They enable you to see what is running on your system. This is useful for a learning exercise.

[NextPreviousContentsNextPreviousContents](#)

13. Building Software From Source

So far I have focussed on what the packages do. Here I will offer what clues I can about making a minimal Linux system from source.

13.1 How I Built My System

There is more than one way to go about building a system. But the way I did it seems to have worked out ok, so this account may be helpful to you.

I used a dedicated machine – an old Wang 386sx of almost zero dollar value. I did a minimal install of Red Hat 6.0 to be the ``source" system, and allocated a ``target" partition where I built the system. In the old Wang, I have a 3G hard disk partitioned as follows:

```
hda1    480M   where I built the system (``target'')
hda2     20M   boot partition for the Red Hat system
hda3     50M   swap
hda4   2500M  extended partition containing hda5
hda5   2500M  Red Hat 6.0 root file system (``source'')
```

There is no real point in having the logical partition hda5 inside an extended partition, hda4. That's just what Red Hat's Disk Druid did when I installed. You only need the base Red Hat system, plus development tools and libraries. It used about 250M of disk space. You could do this exercise with a 1G disk, or a pair of 500M disks.

Older PC hardware, mostly 486's and earlier, have an annoying limitation in their bios. They can not read from a hard disk past the first 512M. This is not too much of a problem for Linux, because once it is up, it does its own disk io. But for Linux to get loaded by these old machines, it has to reside somewhere below 512M. This is why I have both the whole target partition, and the small boot partition for the source system below the 512M mark.

You may wish to actually use the target system, rather than simply build it for a learning exercise. In this case, you would need to go a bit beyond what is described in this document. You would need to install `gcc` and other development tools so that the system could build itself. Once this was done, you could wipe the ``source" system and use its space on the target. Perhaps you could move the `/usr` directory to it.

The Wang only has 8M of RAM in it. I think this is the main reason that the compile of `glibc` took 90 hours (and spanned millenia). It ``only" took 6 hours on my 486 with 32M. My guess is that if I had 16M in the Wang, it would have taken 24 to 48 hours. Kernel compiles take about 8 hours.

From Power Up To Bash Prompt

I made an ext2 file system on the target partition using `mke2fs`, and created directories by hand using `mkdir`. I didn't have it at the time, but the [Filesystem Heirarchy Standard](#) would have been a good thing to follow.

In the `fstab` of the source system, I set up the target partition to be mounted at `/mnt/target`. Most of the packages have a configuration option for where they are to be installed. By default, the ``base" directory for a package installation is `/`, ie you want to install it on the system where it is being built. I used these options to set the base install directory to `/mnt/target`. For example, to install a GNU package to `/mnt/target`, you configure as follows

```
./configure --prefix=/mnt/target
```

There is a problem with this approach if some of the packages of the target system are more recent than their equivalents on the source system. For example, I installed ncurses 5 on the target system, but the source had 4. When compiling, by default the headers and libraries of the source system are used. To fix this you need to set variables or configuration parameters to tell it where the headers and libraries that you want it to use are. Sometimes all you can do is hack the `Makefile`. If you look at the output that is produced while a program is being compiled, the `-I` flags tell it where to look for headers, and the `-L` flags tell it where to look for libraries. Look for a variable called `LD_FLAGS`. This is probably where you can slip a couple extra of these flags in, and make it look where you want. For example in the `Makefile` for the `procps` package, I got it to use the right libraries by adding

```
-L /mnt/target/lib
```

LILO is installed in the master boot record by Red Hat. I installed LILO for the target system in the boot sector of the target partition. I then added the following to `/etc/lilo.conf` in the source system

```
other=/dev/hda1
    label=target
```

and reran `lilo`. This has the effect that when you first boot, one of the options LILO gives you is ``target". If you choose this, you get a second instance of LILO which boots the target system. This might seem crazy, but it allows the separation of the system you are building from the system you are using to build it with.

13.2 Random Tips

If you have a command called `thingy` on a Linux system with RPM, and want a clue about where to get the source from, you can use the command:

```
rpm -qif `which thingy`
```

And if you have a Red Hat source CD, you can install the source code using

From Power Up To Bash Prompt

```
rpm -i /mnt/cdrom/SRPMS/what.it.just.said-1.2.srpm
```

Once you have a bash prompt, the next stage is to get your system able to self replicate. I have not done this yet, but the following are some of the things you will need to install to do this.

- GNU make
- GNU egcs
- gdb
- binutils – assembler, linker, etc; bin86 – intel specific versions
- tar, gzip, bzip2
- diff comes from diffutils, patch comes from patch, hehe

13.3 More Information

- There is a mini-howto on building software from source, the [Software Building mini-HOWTO](#).
- There is also a HOWTO on building a Linux system from scratch. It focuses much more on getting the system built so it can be used, rather than just doing it as a learning exercise. [The Linux From Scratch HOWTO](#)

[NextPreviousContentsNextPreviousContents](#)

14. Conclusion

One of the best things about Linux, in my humble opinion, is that you can get inside it and really find out how it all works. I hope that you enjoy this as much as I do. And I hope that this little note has helped you do it.

[NextPreviousContents](#) Next [PreviousContents](#)

15. Administrivia

15.1 Copyright

This document is copyright (c) 1999, 2000 Greg O'Keefe. You are welcome to use, copy, distribute or modify it, without charge, under the terms of the [GNU General Public Licence](#). Please acknowledge me if you use part of this in another document.

15.2 Homepage

The latest version of this document lives at [From Powerup To Bash Prompt](#)

15.3 Feedback

I would like to hear any comments, criticisms and suggestions for improvement that you have. Please send them to me [Greg O'Keefe](#)

15.4 Acknowledgements

Product names are trademarks of the respective holders, and are hereby considered properly acknowledged.

There are some people I want to say thanks to, for helping to make this happen.

Everyone on the learning@TasLUG mailing list

Thanks for reading all my mails and asking interesting questions. You can join this list by sending a message to [majordomo](#) with
subscribe learning in the message body.

Michael Emery

For reminding me about Unios.

Tim Little

For some good clues about /etc/passwd

sPaKr on #linux in efnet

Who sussed out that syslogd needs /etc/services, and introduced me to the phrase

"rolling your own" to describe building a system from source code.

Alex Aitkin

For bringing Vico and his ``verum ipsum factum" (understanding arises through making) to my attention.

15.5 Change History

0.5 → 0.6

- added change history
- added some todos

15.6 TODO

- add links to home sites, not just the aarnet mirror
- add more exercises
- wipe the target system on the Wang, and rebuild, closely following these notes as a test
- check that all packages specified as kernel requirements (state version) are included

Next [PreviousContentsNextPreviousContents](#)

2. Hardware

When you first turn on your computer it tests itself to make sure everything is in working order. This is called the ``Power on self test". Then a program called the bootstrap loader, located in the ROM BIOS, looks for a boot sector. A boot sector is the first sector of a disk and has a small program that can load an operating system. Boot sectors are marked with a magic number $0xAA55 = 43603$ at byte $0x1FE = 500$. This is how the hardware can tell whether the sector is a boot sector or not.

The bootstrap loader has a list of places to look for a boot sector. My old machine looks in the primary floppy drive, then the primary hard drive. More modern machines can also look for a boot sector on a CD-ROM. If it finds a boot sector, it loads it into memory and passes control to the program that loads the operating system. On a typical Linux system, this program will be LILO's first stage boot loader. There are many different ways of setting your system up to boot though. See the *LILO User's Guide* for details. See section [LILO](#) for a URL.

Obviously there is a lot more to say about what PC hardware does. But this is not the place to say it. See one of the many good books about PC hardware.

2.1 Configuration

The machine stores some information about itself in its CMOS. This includes what disks and RAM are in the system. The machine's BIOS contains a program to let you modify these settings. Check the messages on your screen as the machine is turned on to see how to access it. On my machine, you press the delete key before it begins loading its operating system.

2.2 Exercises

A good way to learn about PC hardware is to build a machine out of second hand parts. Get at least a 386 so you can easily run Linux on it. It won't cost much. Ask around, someone might give you some of the parts you need.

Check out, download compile and make a boot disk for [Unios](#). This is just a bootable "Hello World!" program, consisting of just over 100 lines of assembler code. It would be good to see it converted to a format that the GNU assembler `as` can understand.

Check out the source code for LILO's boot loader.

2.3 More Information

- *The Unix and Internet Fundamentals HOWTO*, by Eric S. Raymond, <http://mirror.aarnet.edu.au/linux/LDP/HOWTO/Unix-Internet-Fundamentals-HOWTO.html> especially section 3, *What happens when you switch on a computer?*
- The first chapter of *The LILO User's Guide* gives an excellent explanation of PC disk partitions and booting. See [LILO](#) for a URL.
- *The NEW Peter Norton Programmer's Guide to the IBM PC & PS/2*, by Peter Norton and Richard Wilton, Microsoft Press 1988 There is a newer Norton book, which looks good, but I can't afford it right now!
- One of the many books available on upgrading PC's

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

3. Lilo

When the computer loads a boot sector on a normal Linux system, what it loads is actually a part of lilo, called the "first stage boot loader". This is a tiny program whose only job in life is to load and run the "second stage boot loader".

The second stage loader gives you a prompt (if it was installed that way) and loads the operating system you choose.

When your system is up and running, and you run `lilo`, what you are actually running is the "map installer". This reads the configuration file `/etc/lilo.conf` and writes the boot loaders, and information about the operating systems it can load, to the hard disk.

There are lots of different ways to set your system up to boot. What I have just explained is the most obvious and "normal" way, at least for a system whose main operating system is Linux. The Lilo Users' Guide explains several examples of "boot concepts". It is worth reading these, and trying some of them out.

3.1 Configuration

The configuration file for lilo is `/etc/lilo.conf`. There is a manual page for it: type `man lilo.conf` into a shell to see it. The main thing in `lilo.conf` is one entry for each thing that lilo is set up to boot. For a Linux entry, this includes where the kernel is, and what disk partition to mount as the root filesystem. For other operating systems, the main piece of information is which partition to boot from.

3.2 Exercises

DANGER: take care with these exercises. It is easy enough to get something wrong and screw up your master boot record and make your system unuseable. Make sure you have a working rescue disk, and know how to use it to fix things up again. See below for a link to `tomsrtbt`, the rescue disk I use and recommend. The best precaution is to use a machine that doesn't matter.

Set up lilo on a floppy disk. It doesn't matter if there is nothing other than a kernel on the floppy – you will get a "kernel panic" when the kernel is ready to load `init`, but at least you will know that lilo is working.

If you like you can press on and see how much of a system you can get going on the floppy. This is probably the second best Linux learning activity around. See the Bootdisk HOWTO ([url below](#)), and `tomsrtbt` ([url below](#)) for clues.

Get lilo to boot `unios` (see section [hardware exercises](#) for a URL). As an extra challenge, see if you can do this on a floppy disk.

Make a boot-loop. Get lilo in the master boot record to boot lilo in one of the primary partition boot sectors, and have that boot lilo in the master boot record... Or perhaps use the master boot record and all four primary partitions to make a five point loop. Fun!

3.3 More Information

- The lilo man page.
- Lilo [download](#), Australian [mirror](#). These directories also contain the "LILO User's Guide" `lilo-u-21.ps.gz` (or a later version). You may already have this document though. Check `/usr/doc/lilo` or thereabouts. The postscript version is better than the plain text, since it contains diagrams and tables.
- [tomsrtbt](#) the coolest single floppy linux. Makes a great rescue disk.
- [The Bootdisk HOWTO](#)

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

4. The Linux Kernel

The kernel does quite a lot really. I think a fair way of summing it up is that it makes the hardware do what the programs want, fairly and efficiently.

The processor can only execute one instruction at a time, but Linux systems appear to be running lots of things simultaneously. The kernel achieves this by switching from task to task really quickly. It makes the best use of the processor by keeping track of which processes are ready to go, and which ones are waiting for something like a record from a hard disk file, or some keyboard input. This kernel task is called scheduling.

If a program isn't doing anything, then it doesn't need to be in RAM. Even a program that is doing something, might have parts that aren't doing anything. The address space of each process is divided into pages. The Kernel keeps track of which pages of which processes are being used the most. The pages that aren't used so much can be moved out to the swap partition. When they are needed again, another unused page can be paged out to make way for it. This is virtual memory management.

If you have ever compiled your own Kernel, you will have noticed that there are many many options for specific devices. The kernel contains a lot of specific code to talk to diverse kinds of hardware, and present it all in a nice uniform way to the application programs.

The Kernel also manages the filesystem, interprocess communication, and a lot of networking stuff.

Once the kernel is loaded, the first thing it does is look for an `init` program to run.

4.1 Configuration

Most of the configuration of the kernel is done when you build it, using `make menuconfig`, or `make xconfig` in `/usr/src/linux/` (or wherever your Linux kernel source is). You can reset the default video mode, root filesystem, swap device and RAM disk size using `rdev`. These parameters and more can also be passed to the kernel from `lilo`. You can give `lilo` parameters to pass to the kernel either in `lilo.conf`, or at the `lilo` prompt. For example if you wanted to use `hda3` as your root file system instead of `hda2`, you might type

```
LILO: linux root=/dev/hda3
```

If you are building a system from source, you can make life a lot simpler by creating a "monolithic" kernel. That is one with no modules. Then you don't have to copy kernel modules to the target system.

NOTE: The `System.map` file is used by the kernel logger to determine the module names generating messages. The program `top` also uses this information. When you copy the kernel to the target system, copy `System.map` too.

4.2 Exercises

Think about this: `/dev/hda3` is a special type of file that describes a hard disk partition. But it lives on a file system just like all other files. The kernel wants to know which partition to mount as the root filesystem – it doesn't have a file system yet. So how can it read `/dev/hda3` to find out which partition to mount?

If you haven't already: build your own kernel. Read all the help information for each option.

See how small a kernel you can make that still works. You can learn a lot by leaving the wrong things out!

Read "The Linux Kernel" (URL below) and as you do, find the parts of the source code that it refers to. The book (as I write) refers to kernel version 2.0.33, which is pretty out of date. It might be easier to follow if you download this old version and read the source there. Its amazing to find bits of C code called "process" and "page".

Hack! See if you can make it spit out some extra messages or something.

4.3 More Information

- `/usr/src/linux/README` and the contents of `/usr/src/linux/Documentation/` (These may be in some other place on your system)
- [The Kernel HOWTO](#)
- The help available when you configure a kernel using `make menuconfig` or `make xconfig`
- [The Linux Kernel \(and other LDP Guides\)](#)
- Kernel source download – Australian [mirror](#)
- The Linux Kernel [home pagedownload](#) Use one of the mirrors listed at `kernel.org`, because they are always overloaded.

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

5. The GNU C Library

The next thing that happens as your computer starts up is that `init` is loaded and run. However, `init`, like almost all programs, uses functions from libraries.

You may have seen an example C program like this:

```
main() {
    printf("Hello World!\n");
}
```

The program contains no definition of `printf`, so where does it come from? It comes from the standard C libraries, on a GNU/Linux system, `glibc`. If you compile it under Visual C++, then it comes from a Microsoft implementation of the same standard functions. There are zillions of these standard functions, for math, string, dates/times memory allocation and so on. Everything in Unix (including Linux) is either written in C or has to try hard to pretend it is, so everything uses these functions.

If you look in `/lib` on your linux system you will see lots of files called `libsomething.so` or `libsomething.a` etc. They are libraries of these functions. `Glibc` is just the GNU implementation of these functions.

There are two ways programs can use these library functions. If you *statically* link a program, these library functions are copied into the executable that gets created. This is what the `libsomething.a` libraries are for. If you *dynamically* link a program (and this is the default), then when the program is running and needs the library code, it is called from the `libsomething.so` file.

The command `ldd` is your friend when you want to work out which libraries are needed by a particular program. For example, here are the libraries that `bash` uses:

```
[greg@Curry power2bash]$ ldd /bin/bash
        libtermcap.so.2 => /lib/libtermcap.so.2 (0x40019000)
        libc.so.6 => /lib/libc.so.6 (0x4001d000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

5.1 Configuration

Some of the functions in the libraries depend on where you are. For example, in Australia we write dates as `dd/mm/yy`, but Americans write `mm/dd/yy`. There is a program that comes with the `glibc` distribution called `localedef` which enables you to set this up.

5.2 Exercises

Use `ldd` to find out what libraries your favourite applications use.

Use `ldd` to find out what libraries `init` uses.

Make a toy library, with just one or two functions in it. The program `ar` is used to create them, the man page for `ar` might be a good place to start investigating how this is done. Write, compile and link a program that uses this library.

5.3 More Information

- Australian GNU libc [mirror](#). You will also need the `linuxthreads` and `libcrypt` addons. If `libcrypt` is not there it is because of some US export laws. There will be a `README` or some such saying where you can get it from.
- Australian GNU `ncurses` [mirror](#). `ncurses` is a library that provides a lot of text screen capabilities. It includes the terminfo database, which replaces the `termcap` file. You can (and probably should) compile `ncurses` as a `glibc` addon.

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

6. Init

I will only talk about the "System V" style of init that Linux systems mostly use. There are alternatives. Infact, you can put any program you like in `/sbin/init`, and the kernel will run it when it has finished loading.

It is inits job to get everthing running the way it should be. It checks that the file systems are ok and mounts them. It starts up "daemons" to log system messages, do networking, serve web pages, listen to your mouse and so on. It also starts the getty processes that put the login prompts on your virtual terminals.

There is a whole complicated story about switching "run-levels", but I'm going to mostly skip that, and just talk about system start up.

Init reads the file `/etc/inittab`, which tells it what to do. Typically, the first thing it is told to do is to run an initialisation script. The program that executes (or interprets) this script is `bash`, the same program that gives you a command prompt. In Debian systems, the initialisation script is `/etc/init.d/rcS`, on Red Hat, `/etc/rc.d/rc.sysinit`. This is where the filesystems get checked and mounted, the clock set, swap space enabled, hostname gets set etc.

Next, another script is called to take us into the default run-level. This just means a set of subsystems to start up. There is a set of directories `/etc/rc.d/rc0.d`, `/etc/rc.d/rc1.d`, ..., `/etc/rc.d/rc6.d` in Red Hat, or `/etc/rc0.d`, `/etc/rc1.d`, ..., `/etc/rc6.d` in Debian, which correspond to the run-levels. If we are going into runlevel 3 on a Debian system, then the script runs all the scripts in `/etc/rc3.d` that start with 'S' (for start). These scripts are really just links to scripts in another directory usually called `init.d`.

So our run-level script was called by init, and it is looking in a directory for scripts starting with 'S'. It might find `S10syslog` first. The numbers tell the run-level script which order to run them in. So in this case `S10syslog` gets run first, since there were no scripts starting with `S00` ... `S09`. But `S10syslog` is really a link to `/etc/init.d/syslog` which is a script to start and stop the system logger. Because the link starts with an 'S', the run-level script knows to execute the `syslog` script with a "start" parameter. There are corresponding links starting with 'K', which specify what to shut down and in what order when leaving the run-level.

To change what subsystems start up by default, you must set up these links in the `rcN.d` directory, where N is the default runlevel set in your `inittab`.

The last important thing that init does is to start some getty's. These are "respawned" which means that if they stop, init just starts them again. Most distributions come with six virtual terminals. You may want less than this to save memory, or more so you can leave things running and quickly flick to them as you need them. You may also want to run a getty for a text terminal or a dial in modem. In this case you will need to edit the `inittab` file.

6.1 Configuration

`/etc/inittab` is the top level configuration file for `init`.

The `rcN.d` directories, where $N = 0, 1, \dots, 6$ determine what subsystems are started.

Somewhere in one of the scripts invoked by `init`, the `mount -a` command will be issued. This means `mount` all the file systems that are supposed to be mounted. The file `/etc/fstab` defines what is supposed to be mounted. If you want to change what gets mounted where when your system starts up, this is the file you will need to edit. There is a man page for `fstab`.

6.2 Exercises

Find the `rcN.d` directory for the default run-level of your system and do a `ls -l` to see what the files are links to.

Change the number of `gettys` that run on your system.

Remove any subsystems that you don't need from your default run-level.

See how little you can get away with starting.

Set up a floppy disk with `lilo`, a kernel and a statically linked "hello world" program called `/sbin/init` and watch it boot up and say hello.

Watch carefully as your system starts up, and take notes about what it tells you is happening. Or print a section of your system log `/var/log/messages` from start up time. Then starting at `inittab`, walk through all the scripts and see what code does what. You can also put extra start up messages in, such as

```
echo "Hello, I am rc.sysinit"
```

This is a good exercise in learning Bash shell scripting too, some of the scripts are quite complicated. Have a good Bash reference handy.

6.3 More Information

- Australian Sys V init [mirror](#)
- Sys V init [download](#)
- There are man pages for the `inittab` and `fstab` files. Type (eg) `man inittab` into a shell to see it.
- The Linux System Administrators Guide has a good [section](#) on `init`.

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

7. The Filesystem

In this section, I will be using the word "filesystem" in two different ways. There are filesystems on disk partitions and other devices, and there is the filesystem as it is presented to you by a running Linux system. In Linux, you "mount" a disk filesystem onto the system's filesystem.

In the previous section I mentioned that `init` scripts check and mount the filesystems. The commands that do this are `fsck` and `mount` respectively.

A hard disk is just a big space that you can write ones and zeros on. A filesystem imposes some structure on this, and makes it look like files within directories within directories... Each file is represented by an inode, which says who's file it is, when it was created and where to find its contents. Directories are also represented by inodes, but these say where to find the inodes of the files that are in the directory. If the system wants to read `/home/greg/bigboobs.jpeg`, it first finds the inode for the root directory `/` in the "superblock", then finds the inode for the directory `home` in the contents of `/`, then finds the inode for the directory `greg`, then the inode for `bigboobs.jpeg` which will tell it which disk blocks to read.

If we add some data to the end of a file, it could happen that the data is written before the inode is updated to say that the new blocks belong to the file, or vice versa. If the power cuts out at this point, the filesystem will be broken. It is this kind of thing that `fsck` attempts to detect and repair.

The `mount` command takes a filesystem on a device, and adds it to the heirarchy that you see when you use your system. Usually, the kernel mounts its root file system read-only. The `mount` command is used to remount it read-write after `fsck` has checked that it is ok.

Linux supports other kinds of filesystem too: `msdos`, `vfat`, `minix` and so on. The details of the specific kind of filesystem are abstracted away by the virtual file system (VFS). I won't go into any detail on this though. There is a discussion of it in "The Linux Kernel" (see [The Linux Kernel](#) for a url)

7.1 Configuration

There are parameters to the command `mke2fs` which creates ext2 filesystems. These control the size of blocks, the number of inodes and so on. Check the `mke2fs` man page for details.

What gets mounted where on your filesystem is controlled by the `/etc/fstab` file. It also has a man page.

7.2 Exercises

Make a very small filesystem, and view it with a hex viewer. Identify inodes, superblocks and file contents.

I believe there are tools that give you a graphical view of a filesystem. Find one, try it out, and email me the url and a review!

Check out the ext2 filesystem code in the Kernel.

7.3 More Information

- Chapter 9 of the LDP book "The Linux Kernel" is an excellent description of filesystems. You can find it at the Australian LDP [mirror](#)
- The `mount` command is part of the `util-linux` package, there is a link to it in [Login and Getty](#).
- man pages for `mount`, `fstab`, `fsck` and `mke2fs`
- EXT2 File System Utilities [ext2fsprogs](#) home page [ext2fsprogs](#) Australian mirror. There is also a `Ext2fs-overview` document here, although it is out of date, and not as readable as chapter 9 of "The Linux Kernel"
- [MAKEDEV](#) This is a script to make all the device files in `/dev`
- [Unix File System Standard](#) This describes what should go where in a Unix file system, and why. It also has minimum requirements for the contents of `/bin`, `/sbin` and so on. This is a good reference if your goal is to make a minimal yet complete system.

[Next](#)[Previous](#)[Contents](#)[Next](#)[Previous](#)[Contents](#)

8. Kernel Daemons

Unfortunately, this section contains more conjectures and questions than facts. Perhaps you can help?

If you issue the `ps aux` command, you will see something like the following:

```

USER      PID  %CPU  %MEM  SIZE  RSS  TTY  STAT  START  TIME  COMMAND
root      1    0.1   8.0   1284  536  ?   S     07:37  0:04  init [2]
root      2    0.0   0.0    0     0   ?   SW    07:37  0:00  (kflushd)
root      3    0.0   0.0    0     0   ?   SW    07:37  0:00  (kupdate)
root      4    0.0   0.0    0     0   ?   SW    07:37  0:00  (kpiod)
root      5    0.0   0.0    0     0   ?   SW    07:37  0:00  (kswapd)
root     52    0.0  10.7  1552  716  ?   S     07:38  0:01  syslogd -m 0
root     54    0.0   7.1  1276  480  ?   S     07:38  0:00  klogd
root     56    0.3  17.3  2232  1156  1   S     07:38  0:13  -bash
root     57    0.0   7.1  1272  480  2   S     07:38  0:01  /sbin/agetty 38400 tt
root     64    0.1   7.2  1272  484  S1  S     08:16  0:01  /sbin/agetty -L ttyS1
root     70    0.0  10.6  1472  708  1   R     Sep 11  0:01  ps aux

```

This is a list of the processes running on the system. Note that `init` is process number one. Processes 2, 3, 4 and 5 are `kflushd`, `kupdate`, `kpiod` and `kswapd`. There is something strange here though: notice that in both the virtual storage size (`SIZE`) and the Real Storage Size (`RSS`) columns, these processes have zeroes. How can a process use no memory? These processes are really part of the kernel. The kernel does not show up on process lists at all, and you can only work out what memory it is using by subtracting the memory available from the amount on your system. The brackets around the command name could signify that these are kernel processes(?)

`kswapd` moves parts of programs that are not currently being used from real storage (ie RAM) to the swap space (ie hard disk). `kflushd` writes data from buffers to disk. This allows things to run faster. What programs write can be kept in memory, in a buffer, then written to disk in larger more efficient chunks. I don't know what `kupdate` and `kpiod` are for.

This is where my knowledge ends. What do these last two daemons do? Why do kernel daemons get explicit process numbers rather than just being anonymous bits of kernel code? Does `init` actually start them, or are they already running when `init` arrives on the scene?

I put a script to mount `/proc` and do a `ps aux` in `/sbin/init`. Process 1 was the script itself, and processes 2, 3, 4 and 5 were the kernel daemons just as under the real `init`. The kernel must put these processes there, because my script certainly didn't!

8.1 Configuration

I don't know of any configuration for these kernel daemons.

8.2 Exercises

Find out what these processes are for, how they work, and write a new "Kernel Daemons" section for this document and send it to me!

8.3 More Information

The Linux Documentation Project's "The Linux Kernel" (see [The Linux Kernel](#) for url), and the kernel source code are all I can think of.

[NextPreviousContentsNextPreviousContents](#)

9. System Logger

Init starts the `syslogd` and `klogd` daemons. They write messages to logs. The kernel's messages are handled by `klogd`, while `syslogd` handles log messages from other processes. The main log is `/var/log/messages`. This is a good place to look if something is going wrong with your system. Often there will be a valuable clue in there.

9.1 Configuration

The file `/etc/syslog.conf` tells the loggers what messages to put where. Messages are identified by which service they come from, and what priority level they are. This configuration file consists of lines that say messages from service `x` with priority `y` go to `z`, where `z` is a file, tty, printer, remote host or whatever.

NOTE: Syslog requires the `/etc/services` file to be present. The services file allocates ports. I am not sure whether syslog needs a port allocated so that it can do remote logging, or whether even local logging is done through a port.

9.2 Exercises

Have a look at your system log. Find a message you don't understand, and find out what it means.

Send all your log messages to a tty. (set it back to normal once done)

9.3 More Information

Australian sysklogd [Mirror](#)

[NextPreviousContents](#)