From−PowerUp−To−Bash−Prompt−HOWTO

Table of Contents

Table of Contents

From Power Up To Bash Prompt

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

v0.7, April 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.

5.The GNU C Library

6.Init

7.The Filesystem

8.Kernel Daemons

9.System Logger

10.Getty and Login

11.Bash

12.Commands

13.Building A Minimal Linux System From Source

14.Conclusion

15.Administrivia

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. Giambattista Vico, an Italian philosopher (1668−1744) said ``verum ipsum factum", which means ``understanding arises through making". Thanks to Alex (see Acknowledgements) for this quote.

If you want to ``roll your own", you should also see Gerard Beekmans' Linux From Scratch HOWTO (LFS). LFS has detailed instructions on building a complete useable system from source code. On the LFS website, you will also find a mailing list for people building systems this way. What I have included in this document, is instructions (see Building a Minimal Linux System From Source) for building a ``toy" system, purely as a learning exercise.

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.

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.

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 = 510. That's the last two bytes of the sector. 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. (They used to have a home page at http://www.unios.org, but it disappeared) 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.

Open the boot disk image for unios with a hex editor. This image is 512 bytes long, exactly one sector. Find the magic number 0xAA55. Do the same for the boot sector from a bootable floppy disk or your own computer. You can use the `dd` command to copy it to a file: `dd if=/dev/fd0 of=boot.sector`. Be *very* careful to get `if` (input file) and `of` (output file) the right way round!

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

2.3 More Information

- The Unix and Internet Fundamentals HOWTO by Eric S. Raymond, 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 section 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

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 who's 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 who's 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.
- The Lilo package (see downloads) contains 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 there abouts. 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

---

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 acheives 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 see downloads

---

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

- source code, see section downloads

6.Init

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

It is `init`'s 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' (for kill), 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 lots of 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, ..., 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

- see downloads for source code download url's
- 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.

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` in the contents of `/home`, 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 section 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 downloads.
- 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''
- Unix File System Standard Another link to the 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.

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 processess 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!

The following ramblings were contributed by David Leadbeater:

These processes seem to take care of disk reads and writes, they seem to be started by the kernel but after it runs the init process, it seems that being run as kernel processes rather than seperate processess they are

protected from being killed (kill −9 dosen't stop them), I am not sure why they are run as seperate threads (it seems to be something with disk access)

kflushd and kupdate These two processes are started to flush dirty (changed) buffers back to disk. kflushd is run when the buffers are full and kupdate runs periodically (5 seconds?) to sync the disk and the buffers in memory.

kpiod and kswapd These deal with paging out pages (sections) of memory into the swap file so main memory never gets exhausted, these are similar to kflushd and kupdate in that one is run when needed kpiod and the other kswapd is run peridically (1 second intervals)

Other Kernel Daemons On a default install of RH6 kupdate is missing but update is running as a user space daemon so it seems it needs to be run! Also another daemon mdrecoveryd is there, this seems to be dealing with software RAID, looking at the kernel source it seems that some SCSI drivers also start seperate processes.

I am still unsure of the meaning of the brackets but it seems that they appear when the RSS of a process is 0 meaning it isn't using any memory?

(end of ramble, thanks David)

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 section The Linux Kernel for a url), and the kernel source code are all I can think of.

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, or whether it just uses `/etc/services` to convert the service names you type `/etc/syslog.conf` into port numbers.

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

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. Some distributions, including Red Hat use a very small one called `mingetty` that only works with virtual terminals.

The `login` program is part of the util−linux package, which also contains a getty called `agetty`, which works fine. This package also contains `mkswap`, `fdisk`, `passwd`, `kill`, `setterm`, `mount`, `swapon`, `rdev`, `renice`, `more` (the program) and more (ie more programs).

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.

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

- source code download see downloads
- There is a ``Bash Reference Manual'' with this, which is comprehensive, but heavy going.
- There is an O'Rielly book on Bash, not sure if it's good.
- I don't know of any good free up to date bash tutorials. If you do, please email me a url.

12.Commands

You do most things in bash by issuing commands like `cp`. Most of these commands are small programs, though some, like `cd` are built into the shell.

The commands come in packages, most of them from the Free Software Foundation (or GNU). Rather than list the packages here, I'll direct you to the Linux From Scratch HOWTO. It has a full and up to date list of the packages that go into a Linux system as well as instructions on how to build them.

13.Building A Minimal Linux System 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. This is a toy system we are making here. If you want to build a real system to be used for real work, see the Linux From Scratch HOWTO.

It is possible to get a bash prompt without installing everything I mention here. What I describe is a base system, without nasty kludges, that can be built on easily.

13.1 What You Will Need

We will install a Linux distribution like Red Hat in one partition, and use that to build a new Linux system in another partition. I will call the system we are building the ``target'' and the system we are using to build it with, the ``source'' (not to be confused with *source code* which we will also be using.)

So you are going to need a machine with two spare partitions on it. If you can, use a machine with nothing important on it. You could use an existing Linux installation as the source system, but I wouldn't recommend that. If you leave a parameter out of one of the commands we will issue, you could accidentally install stuff to this system. This could lead to incompatibilites and strife.

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, bypassing the bios. But for Linux to get loaded by these old machines, the kernel has to reside somewhere below 512M. If you have one of these machines you will need to have a separate partition completely below the 512M mark, to mount as `/boot` for any partitions that are over that 512M mark.

Last time I did this, I used Red Hat 6.1 as a source system. I installed the base system plus

- cpp
- egcs
- egcs−c++
- patch
- make
- dev86
- ncurses−devel
- glibc−devel
- kernel−headers

I also had X−window and Mozilla so I could read documentation easily, but that's not really necessary. By the time I had finished working, it had used about 350M of disk space. (Seems a bit high, I wonder why?)

The finished target system took 650M, but that includes all the source code and intermediate build files. If space is tight, you should do a `make clean` after each package is built. Still, this mind boggling bloat is a bit of a worry.

Finally, you are going to need the source code for the system we are going to build. These are the ``packages'' that I have discussed in this document. These can be obtained from a source cd, or from the internet. I'll give URL's for the USA sites and for Australian mirrors.

- MAKEDEV USA Another USA site
- Lilo USA, Australia.
- Linux Kernel Use one of the mirrors listed at home page rather than USA because they are always overloaded. Australia
- GNU libc itself, and the linuxthreads addon are at USAAustralia
- GNU libc addons You will also need the linuxthreads and libcrypt addons. If libcrypt is not there it is because of some US export laws. You can get it at libcrypt The linuxthreads addon is in the same places as libc itself
- GNU ncurses USAAustralia
- SysVinit USAAustralia
- GNU Bash USAAustralia
- GNU sh−utils USAAustralia
- util−linux Somewhere elseAustralia This package contains `agetty` and `login`.

To sum up then, you will need:

- A machine with two spare partitions of about 400M and 700M respectively though you could probably get away with less
- A Linux distribution (eg. a Red Hat cd) and a way of installing it (eg. a cdrom drive)
- The source code tarballs listed above

I'm assuming that you can install the source system yourself, without any help from me. From here on, I'll assume that its done.

The first milestone in this little project is getting the kernel to boot up and panic because it can't find an `init`. This means we are going to have to install a kernel, and install lilo. To install lilo nicely though, we will need the device files in the target `/dev` directory. Lilo needs them to do the low level disk access necessary to write the boot sector. MAKEDEV is the script that creates these device files. (You can just copy them from the source system of course, but that's cheating!) But first of all, we need a filesystem to put all of this into.

13.2 The Filesystem

Our new system is going to live in a file system. So first, we have to make that file system using `mke2fs`. Then mount it somewhere. I'd suggest `/mnt/target`. In what follows, I'll assume that this is where it is. You could save yourself a bit of time by putting an entry in `/etc/fstab` so that it mounts there automatically when the source system comes up.

When we boot up the target system, the stuff that's now in `/mnt/target` will be in `/`.

We need a directory structure on target. Have a look at the File Heirarchy Standard (see section Filesystem) to work out what this should be, or just `cd` to where the target is mounted and blindly do

```
mkdir bin boot dev etc home lib mnt root sbin tmp usr var
cd var; mkdir lock log run spool
cd ../usr; mkdir bin include lib local sbin share src
cd share/; mkdir man; cd man
mkdir man1 man2 man3 ... man9
```

Since the FHS and most packages disagree about where man pages should go, we need a symlink

```
cd ..; ln −s share/man man
```

13.3 MAKEDEV

We will put the source code in the target `/usr/src` directory. So for example, if your target file system is mounted on `/mnt/target` and your tarballs are in `/root`, you would do

```
cd /mnt/target/usr/src
tar −xzvf /root/MAKEDEV−2.5.tar.gz
```

Don't be completely lame and copy the tarball to the place where you are going to extract it ;−>

Normally when you install software, you are installing it onto the system that is running. We don't want to do that though, we want to install it as though `/mnt/target` is the root filesystem. Different packages have different ways of letting you do this. For MAKEDEV you do

```
ROOT=/mnt/target make install
```

You need to look out for these options in the README and INSTALL files or by doing a `./configure --help`.

Have a look in MAKEDEV's `Makefile` to see what it does with the `ROOT` varible that we set in that command. Then have a look in the man page by doing `man ./MAKEDEV.man` to see how it works. You'll find that the way to make our device files is to `cd /mnt/target/dev` and do `./MAKEDEV generic`. Do an `ls` to see all the wonderful device files it has made for you.

13.4 Kernel

Next we make a kernel. I presume you've done this before, so I'll be brief. It is easier to install lilo if the kernel it is meant to boot is already there. Go back to the target `usr/src` directory, and unpack the linux kernel source there. Enter the linux source tree (`cd linux`) and configure the kernel using your favourite method, for example `make menuconfig`. You can make life slightly easier for yourself by configuring a kernel without modules. If you configure any modules, then you will have to edit the `Makefile`, find `INSTALL_MOD_PATH` and set it to `/mnt/target`.

Now you can `make dep`, `make bzImage`, and if you configured modules: `make modules`, `make modules_install`. Copy the kernel `arch/i386/boot/bzImage` and the system map `System.map` to the target boot directory `/mnt/target/boot`, and we are ready to install lilo.

13.5 Lilo

Lilo comes with a neat script called `QuickInst`. Unpack the lilo source into the target source directory, run this script with the command `ROOT=/mnt/target ./QuickInst`. It will ask you questions about how you want lilo installed.

Remember, since we have set `ROOT`, to the target partition, you tell it file names relative to that. So when it asks what kernel you want to boot by default, answer `/boot/bzImage`*not*`/mnt/target/boot/bzImage`. I found a little bug in the script, so it said

```
        ./QuickInst: /boot/bzImage: no such file
```

But if you just ignore it, it's ok.

Where should we get `QuickInst` to put the boot sector? When we reboot we want to have the choice of booting into the source system or the target system, or any other systems that are on this box. And we want the instance of lilo that we are building now to load the kernel of our new system. How are we going achieve both of these things? Let's digress a little and look at how lilo boots DOS on a dual boot Linux system. The `lilo.conf` file on such a system probably looks something like this:

```
prompt
timeout = 50
default = linux

image = /boot/bzImage
        label  = linux
        root   = /dev/hda1
        read−only

other = /dev/hda2
        label = dos
```

If the machine is set up this way, then the master boot record gets read and loaded by the bios, and it loads the lilo bootloader, which gives a prompt. If you type in `dos` at the prompt, lilo loads the boot sector from hda2, and it loads DOS.

What we are going to do is just the same, except that the boot sector in hda2 is going to be another lilo boot sector − the one that `QuickInst` is going to install. So the lilo from the Linux distribution will load the lilo that we have built, and that will load the kernel that we have built. You will see two lilo prompts when you reboot.

To cut a long story short, when `QuickInst` asks you where to put the boot sector, tell it the device where your target filesystem is, eg. `/dev/hda2`.

Now modify the `lilo.conf` on your source system, so it has a line like

```
other = /dev/hda2
        label = target
```

run lilo, and we should be able to do our first boot into the target system.

13.6 Glibc

Next we want to install `init`, but like almost every program that runs under Linux, `init` uses library functions provided by the GNU C library, glibc. So we will install that first.

Glibc is a very large and complicated package. It took 90 hours to build on my old 386sx/16 with 8M RAM. But it only took 33 minutes on my Celeron 433 with 64M. I think memory is the main issue here. If you only have 8M of RAM (or, shudder, less!) be prepared for a long build.

The glibc install documentation recommends building in a separate directory. This enables you to start again easily, by just blowing that directory away. You might also want to do that to save yourself about 265M of disk space!

Unpack the `glibc−2.1.3.tar.gz` (or whatever version) tarball into `/mnt/target/usr/src` as usual. Now, we need to unpack the ``add−ons'' into glibc's directory. So `cd glibc−2.1.3`, and then unpack the `glibc−crypt−2.1.3.tar.gz` and `glibc−linuxthreads−2.1.3.tar.gz` tarballs there.

Now we can create the build directory, configure, make and install glibc. These are the commands I used, but read the documentation yourself and make sure you do what is best for your circumstances. Before you do though, you might want to do a `df` command to see how much free space you have. You can do another after you've built and installed glibc, to see what a space−hog it is.

```
cd ..
mkdir glibc−build
../glibc−2.1.3/configure −−enable−add−ons −−prefix=/usr
make
make install_root=/mnt/target install
```

Notice that we have yet another way of telling a package where to install.

13.7 SysVinit

Making and installing the SysVinit binaries is pretty straight forward. I'll just be lazy and give you the commands, assuming that you have unpacked and entered the SysVinit source code directory:

```
cd src
make
ROOT=/mnt/target make install
```

There are also a lot of scripts associated with `init`. There are example scripts with the SysVinit package, which work fine. But you have to install them manually. They are set up in a heirarchy under `debian/etc` in the SysVinit source code tree. You can just copy them straight across into the target `etc` directory, with something like `cd ../debian/etc; cp −r * /mnt/target/etc`. Obviously you will want to have a look before you copy them across!

Everything is in place now for the target kernel to load up `init` when we reboot. The problem this time should be that the scripts won't run, becasue `bash` isn't there to interpret them. Also, `init` will try to run `getty`'s, but there is no `getty` for it to run. Reboot now and make sure there is nothing else wrong.

13.8 Ncurses

The next thing we need is Bash, but bash needs ncurses, so we'll install it first. Ncurses replaces termcap as the way of handling text screens, but it can also provide backwards compatibility by supporting the termcap calls. In the interests of having a clean simple modern system, I think its best to disable the old termcap method. You might strike trouble later on if you are compiling an older application that uses termcap. But at least you will know what is using what. If you need to you can recompile ncurses with termcap support.

The commands I used are

```
./configure −−prefix=/usr −−with−install−prefix=/mnt/target −−with−shared −−disable−termc
make
make install
```

13.9 Bash

It me took quite a lot of reading and thinking and trial and error to get Bash to install itself where I thought it should go. The configuration options I used are

```
./configure −−prefix=/mnt/target/usr/local −−exec−prefix=/mnt/target −−with−curses
```

Once you have made and installed Bash, you need to make a symlink like this `cd /mnt/target/bin; ln −s bash sh`. This is because scripts usually have a first line like this

```
#!/bin/sh
```

If you don't have the symlink, your scripts won't be able to run, because they will be looking for `/bin/sh` not `/bin/bash`.

You could reboot again at this point if you like. You should notice that the scripts actually run this time, though you still can't login, because there are no `getty` or `login` programs.

13.10 Util−linux (getty and login)

The util−linux package contains `agetty` and `login`. We need both of these to be able to log in and get a bash prompt. After it is instlalled, make a symlink from `agetty` to `getty` in the target `/sbin` directory. `getty` is one of the programs that is supposed to be there on all Unix−like systems, so the link is a better idea than hacking `inittab` to run `agetty`.

I have one remaining problem with the compilation of util−linux. The package also contains the program `more`, and I have not been able to persuade the `make` process to have `more` link against the ncurses 5 library on the target system rather than the ncurses 4 on the source system. I'll be having a closer look at that.

You will also need a `/etc/passwd` file on the target system. This is where the `login` program will check to find out if you are allowed in. Since this is only a toy system at this stage, we can do outrageous things like setting up only the root user, and not requiring any password!! Just put this in the target `/etc/passwd`

```
root::0:0:root:/root:/bin/bash
```

The fields are separated by colons, and from left to right they are user id, password (encrypted), user number, group number, user's name, home directory and default shell.

13.11 Sh−utils

The last package we need is GNU sh−utils. The only program we need from here at this stage is `stty`, which is used in `/etc/init.d/rc` which is used to change runlevels, and to enter the initial runlevel. I actually have, and used a package that contains only `stty`, but I can't remember where it came from. Its a better idea to use the GNU package, because there is other stuff in there that you will need if you add to the system to make it useable.

Well that's it. You should now have a system that will boot up and prompt you for a login. Type in ``root'', and you should get a shell. You won't be able to do much with it. There isn't even an `ls` command here for you to see your handiwork. Press tab twice so you can see the available commands. This was about the most satisfying thing I found to do with it.

13.12 Towards Useability

It might look like we have made a pretty useless system here. But really, there isn't that far to go before it can do some work. One of the first things you would have to do is have the root filesystem mount read−write. There is a script from the SysVinit package, in `/etc/init.d/mountall.sh` which does this, and issues a `mount −a` so that everything gets mounted the way you specify in `/etc/fstab`. Put a symlink called something like `S05mountall` to it in the target's `etc/rc2.d`.

You may find that this script will use commands that you haven't installed yet. If so, find the package that contains the commands and install it. See section Random Tips for clues on how to find packages.

Look at the other scripts in `/etc/init.d`. Most of them will need to be included in any serious system. Add them in one at a time, make sure everthing is running smoothly before adding more.

Check the File Heirarchy Standard (see section Filesystem). It has lists of the commands that should be in `/bin` and `/sbin`. Make sure that you have all these commands installed. Even better, find the Posix documentation that specifies this stuff.

>From there, it's really just a matter of throwing in more and more packages until everything you want it there. The sooner you can put the build tools such as `gcc` and `make` in the better. Once that is done, you can use the target system to build itself, which is much less complicated.

13.13 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

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

This will put the tarball, and any Red Hat patches into `/usr/src/redhat/SOURCES`.

13.14 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

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.

15.Administrivia

15.1 Copyright

15.2 Homepage

The lastest 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.

Dennis Scott

For correcting my hexidecimal arithmetic.

jdd

For pointing out some typos.

David Leadbeater

For contributing some ``ramblings'' about the kernel deamons.

15.5 Change History

0.6 −> 0.7

- more emphasis on explanation, less on how to build a system, building info gathered together in a separate section and the system built is trimmed down, direct readers to Gerard Beekmans' ``Linux From Scratch'' doc for serious building
- added some ramblings contributed by David Leadbeater
- fixed a couple of url's, added link to unios download at learning.taslug.org.au/resources
- tested and fixed url's
- generally rewrite, tidy up

0.5 −> 0.6

- added change history
- added some todos

15.6 TODO

- explain kernel modules, depmod, modprobe, insmod and all that (I'll have to find out first!)
- mention the /proc filesystem, potential for exercises here
- convert to docbook sgml
- add more exercises, perhaps a whole section on larger exercises, like creating a minimal system file by file from a distro install.
