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`

v0.9, November 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. Understanding this will be helpful when you need to solve problems or configure your system.

---

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.

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. The instructions that used to be part of this document are now in a separate document ``Building a Minimal Linux System from Source Code'', and can be found at From PowerUp to Bash Prompt home page. They explain how to ``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 (lilo), 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)
- source code, see Building a Minimal Linux System from Source Code for urls

---

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 Building a Minimal Linux System from Source Code for urls

---

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

- 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.
- source code, see Building a Minimal Linux System from Source Code for urls

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)

A completely different kind of filesystem gets mounted on `/proc`. It is really a representation of things in the kernel. There is a directory there for each process running on the system, with the process number as the directory name. There are also files such as `interrupts`, and `meminfo` which tell you about how the hardware is being used. You can learn a lot by exploring `/proc`.

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 Building a Minimal Linux System from Source Code
- man pages for `mount`, `fstab`, `fsck`, `mke2fs` and `proc`
- The file `Documentation/proc.txt` in the Linux source code explains the `/proc` filesystem.
- 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

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. The information comes from the `/proc` filesystem that I mentioned in the previous section. 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 the kernel daemons. Most of 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 kernel daemons are started after init, so they get process numbers like normal processes do. But their code and data lives in the kernel's part of the memory.

There are brackets around the entries in the command column because the `/proc` filesystem does not contain command line information for these processes.

So what are these kernel daemons for? Previous versions of this document had a plea for help, as I didn't know much about the kernel daemons. The following partial story has been patched together from various replies to that plea, for which I am most grateful. Further clues, references and corrections are most welcome!

Input and output is done via *buffers* in memory. 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. The daemons `kflushd` and `kupdate` handle this work: `kupdate` runs periodically (5 seconds?) to check whether there are any dirty buffers. If there are, it gets `kflushd` to flush them to disk.

Processes often have nothing to do, and ones that are running often don't need all of their code and data in memory. This means we can make better use of our memory, by shifting unused parts of running programs out to the swap partition(s) of the hard disk. Moving this data in and out of memory as needed is done by `kpiod` and `kswapd`. Every second or so, `kswapd` wakes up to check out the memory situation, and if something out on the disk is needed in memory, or there is not enough free memory, `kpiod` is called in.

There might also be a `kapmd` daemon running on your system if you have configured automatic power management into your kernel.

8.1 Configuration

The program `update` allows you to configure `kflushd` and `kswapd`. Try `update -h` for some information.

Swap space is turned on by `swapon` and off by `swapoff`. The init script (`/etc/rc.sysinit` or `/etc/rc.d/rc.sysinit`) usually calls `swapon` as the system is coming up. I'm told that `swapoff` is handy for saving power on laptops.

8.2 Exercises

Do an `update -d`, note the blatherings on the last line about ``threshold for buffer fratricide''. Now there's an intriguing concept, go investigate!

Change directory to `/proc/sys/vm` and `cat` the files there. See what you can work out.

8.3 More Information

The Linux Documentation Project's ``The Linux Kernel'' (see section The Linux Kernel for a url)

The Linux kernel source code, if you are brave enough! The `kswapd` code is in `linux/mm/vmscan.c`, and `kflushd` and `kupdate` are in `linux/fs/buffer.c`.

---

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

- 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.
- source code, see Building a Minimal Linux System from Source Code for urls

---

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. 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.

---

14. Administrivia

14.1 Copyright

14.2 Homepage

The lastest version of this document lives at From Powerup To Bash Prompt as does its companion ``Building a Minimal Linux System from Source Code''.

There is a French translation at From Powerup To Bash Prompt thanks to Dominique van den Broeck. A Japanese by Yuji Senda is coming soon, if it's not at Japanese Documentation and FAQ Project already.

14.3 Feedback

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

14.4 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.

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.

Dominique van den Broeck

For translating this doc into French.

Matthieu Peeters

For some good information about kernel deamons.

John Fremlin

For some good information about kernel deamons.

Yuji Senda

For the Japanese translation.

Antonius de Rozari

For contributing a GNU assembler version of UNIOS (see resources section on the home page)

14.5 Change History

0.8 −> 0.9 (November 2000)

- Incorporated some information from Matthieu Peeters and John Fremlin on kernel deamons and the `/proc` filesystem.

0.7 −> 0.8 (September 2000)

- Removed instructions on how to build a system, placing them in a separate document. Adjusted a few links accordingly.
- Changed homepage from learning@TasLUG to my own webspace.
- Completely failed to incorporate a lot of good material contributed by various people. Maybe next time :(

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

14.6 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.
- add makefile hack to bash build instructions − see easter notes.
