Part one on the DVD

The Mike Saunders SCHOOL of LINUX

Part 2: After last month's foray into the world of hardware, in this class we'll prep your Linux skills with a detailed look at the boot process. Oh, and spit out that gum!

Mike

Linux Professional Institute

Our expert

Mike Saunders has been writing about Linux for over a decade, and has installed more distros than he's had hot dinners.

You press the power button on your PC. A bunch of messages scroll by, or perhaps a flashy animation if you're using a desktop-oriented distro, and finally you arrive at a login prompt. What exactly happens in the mean time? That's what we'll be explaining in this instalment of our School of Linux series. Like last month, which focused on identifying and managing hardware on your Linux system, this tutorial will help you prepare for Linux Professional Institute (LPI) certification. So it's useful if you want to get a job in the Linux world, or even if you just want to learn a bit more about your operating system.

Linux certification avoids whizz-bang, rapidly updated distros and teaches skills applicable to the more stable, enterprise-friendly flavours such as Red Hat Enterprise Linux (RHEL), CentOS and Debian. We used CentOS for last month's guide – this time it's the turn of Debian (version 5). While distros vary in the way they implement certain features, much here will be applicable across the board.

Section 1: From power up to desktop

The Linux boot process is an intricate collection of processes and scripts that turn your PC, initially nothing more than a lump of cold metal, into a powerful workstation or server. Let's go through the key steps in order.

BIOS

The BIOS (Basic Input/Output System) is a small program that lives in a chip on your motherboard. When you hit your PC's power button, the CPU starts executing BIOS code. You've no doubt seen the 'Hit F2 for setup' messages that appear when your system starts, providing you with access to the BIOS for changing settings such as the disk drive boot order. Typically, the BIOS performs a quick check of your hardware – making sure the RAM chips are working, for example – and then tries to find a bootloader. It attempts to load the first 512 bytes from a floppy drive or hard drive into RAM and, if this works, executes the contents.

Bootloader

So the BIOS has handed over control to the first part of the OS: the half-kilobyte bootloader. Back in the 1980s, such a tiny amount of memory was fine for loading an OS kernel. However, modern bootloaders must support many different filesystems, OSes and graphics modes, so 512 bytes isn't enough. In the case of *Grub*, as used by most Linux distros, the half-k loader then loads another program called *Stage 1.5*.

This is a slightly larger bootloader that's located towards the start of the drive so that it can be found easily. It then loads *Grub Stage 2*, a fully fledged bootloader that provides all of the features you're used to. *Grub* reads a configuration file, loads the Linux kernel into RAM and starts executing it.

Linux kernel and Init

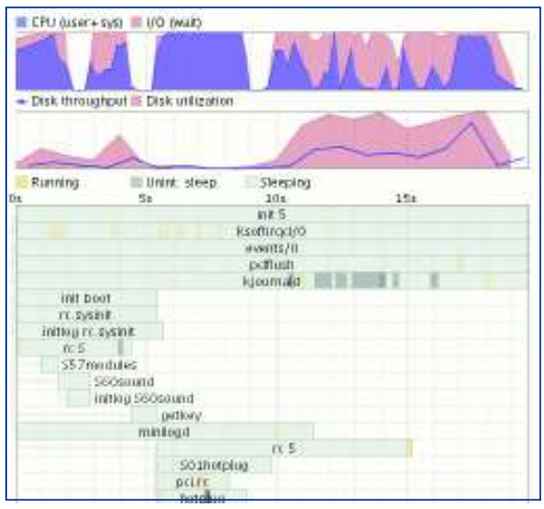
When the very first bytes of the Linux kernel begin executing, it's like a newborn, unaware of the outside world of your system. First, it tries to work out what processor and features are available, sees how much RAM is installed and gets an overall picture of the system. It can then allocate itself a safe place in memory – so that other programs can't overwrite it and cause spectacular crashes – and starts enabling features such as hardware drivers, networking protocols and so forth.

Once the kernel has done everything it needs to, it's time to hand control over to userland: the place where programs are run. The kernel isn't interested in running *Bash*, *Gdm*, and

so on directly, so it runs a single master program: **/sbin/init**. This is the first proper process on the system. Init is responsible for starting the boot scripts that get the system running, but needs to know what to run. The main config file for init is **/etc/inittab**, a plain text file you can edit.

This file is based around a concept called runlevels – the different running states the system can be in, such as single user, multiuser and shutting down. We'll cover these later, but for now you need to know that **/etc/inittab** tells init to run the **/etc/init.d/rc** script, with the runlevel as a parameter.

This script calls other scripts to set up various parts of the system – to establish a network connection, start system loggers and, on a desktop machine, launch the *X Window System* and login manager. Once you've given your details, the login manager launches your desktop or window manager of choice and you're ready to go. This entire process – from power button to clicking icons – involves a lot of work, but is generally well-shielded from the user.

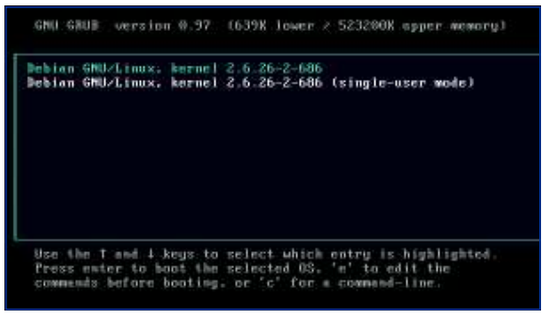


› Want to see how much time is being spent by your boot scripts? Get a graphical representation with *Bootchart* (it's in most distros' package repositories).

Section 2: Editing Grub settings

Now let's look at the bootloader in more detail. In most cases, this will be *Grub*, a powerful program that can launch a range of OSes and enables you to make configuration changes at startup. We're going to cover *Grub*'s configuration files and related utilities in a future tutorial – for now, we'll focus on making changes at boot time on our Debian installation.

When *Grub* appears, just after the BIOS screen, you're given a list of boot options. You can hit Enter to start one of these, but you can edit them in place as well. Select the entry you want to edit and hit E. After this, you'll be taken to another screen with three lines that look like the following:



› *Grub* is a hugely flexible bootloader. With a tap of E, you can edit its options before starting the boot sequence.

```
root  (hd0,0)
kernel  /boot/vmlinuz-2.6.26-2-686 root=/dev/hda1 ro quiet
initrd  /boot/initrd.img-2.6.26-2-686
```

Have a look at the second line. This tells *Grub* where to find the Linux kernel, and what options to pass over to it. In this case, we tell the kernel where the root partition (**/**) device is, then **ro** says the partition should be mounted as read-only. This is so that filesystem checks can be run if needed – but it will be remounted as read-write shortly after. Meanwhile, **quiet** tells the kernel that we don't want it to spit out lots of messages, making the boot cleaner and easier to follow.

We can modify these options by using the Down cursor key to select that second line and hitting E again. The screen will switch to a plain editing mode, where you can add and remove options. The cursor keys move around in the line. So let's try something: after **quiet**, add a single **s** (with a space separating them). What we're doing is specifying the runlevel we want the kernel to boot in – **s** means single user.

Hit Enter to return to the *Grub* screen, then press B to start the boot process. Since we're booting into single user mode, the normal process stops after the kernel's initialised and mounted the root partition, and you'll be asked for the root user password. Provide it and you'll enter a prompt. This is a limited mode of operation, but handy for sorting boot problems – you can edit and fix scripts unhindered. »

Booting into the future

Traditionally, Linux (and Unix) boot scripts have run sequentially – that is, one follows the other. This is simple, and guarantees that certain bits of hardware and features will be enabled by certain points in the boot process. However, it's an inefficient way of doing things and leads to long bootup times, especially on older hardware. Much of the time, the scripts are waiting for something to happen: for a piece of hardware to activate itself, or for a DHCP server on the network to send a lease, for instance.

Wouldn't it be great if other things could be done in the delays? That's the aim of parallelised init scripts. While your network script is waiting for DHCP, another script can clean **/tmp** or start up the *X Window System*. You can't just put ampersands on the end of every script call and run them all in parallel, though; some scripts depend on certain facilities being available. For instance, a boot script that gets an IP address via DHCP needs to assume that networking's already been enabled by another script.

InitNG is a parallelised boot system in which scripts have dependencies to sort out their order. *Upstart*, as used by Ubuntu, starts scripts based on system events, such as when a hardware device is detected. Then there's *System D* (due to be in Fedora 15) and other approaches. For the sake of documenters and administrators, let's hope the Linux world will eventually settle on one system, but in any case the move towards parallelisation is hugely speeding up the Linux boot process.

» **Last month** We got to grips with hardware listing and driver modules.

Section 3: Viewing log files

With the aforementioned **quiet** option and the overall speed of modern PCs, it's quite possible that the photons from the boot messages will barely have time to reach your retinas before the boot process is finished. Fortunately, then, we can read them in peace once the system is fully up and running. Look in the file **/var/log/messages** (you'll need to be root to view this) and you'll see everything generated by the kernel, right from the moment it begins execution. However, since the kernel is trying to find out what hardware it lives in, it's sometimes surprised by what it finds, so don't panic if you see entertaining warning messages such as 'warning: strange, CPU MTRRs all blank?'.

Quick tip

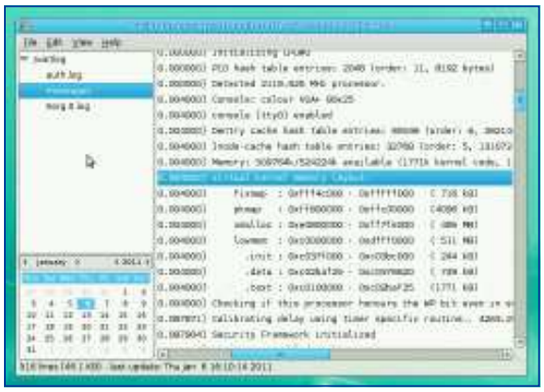
A security note: anyone with access to your machine can reboot it and play around with the *Grub* options, no matter how secure the OS is. In a later tutorial, when we cover *Grub*'s configuration file, we'll show you how to password protect the bootloader to stop such nefarious antics happening.

Kernel Saunders explains

Roughly, the order in which the kernel works is like this, although there's some overlap:

- Get hardware information from the BIOS (note that this is not always reliable).
- Find out how many CPUs/cores there are, and learn of any CPU features.
- Get ACPI information and probe the PCI bus for devices.
- Initialise the TCP/IP networking stack.
- Look for hard, floppy and CD-ROM drives.
- Probe for USB controllers and connected devices.

Once the kernel is happy with the state of the system, it mounts the root filesystem and runs **/sbin/init**, as described before. While **/var/log/messages** is a valuable resource for finding out what the kernel has done since it booted, it can become cluttered with lines from other programs as well. For instance, on our installation many lines include **debian:**, but there are others with **dhcbd** and the like.



Try gnome-system-log or KDE's *Ksystemlog* for a slightly more attractive view of your log messages.

If you want to get a kernel-only list of messages, run the **dmesg** command. (You can redirect this into a text file for easier reading with **dmesg > listing.txt**.) While most of the messages contained therein will be from the early parts of the boot process, this information will be updated if you plug in new hardware. Add a USB flash key and then run the command, for instance, and you'll see new lines describing how the kernel detected the device.

Section 4: Runlevels and the magic of /etc/init.d/

Earlier, we mentioned runlevels, which play a massively important role in the workings of your Linux installation, even if you've never heard of them. A runlevel defines a state for your system – specifically, which processes are running and which resources are available. It's not some secret, inner-kernel black magic, but merely a system whereby **/sbin/init** runs scripts to turn functionality on and off. There are eight runlevels, seven of them with numbers:

- **0** Halt the system. This is the runlevel that the machine enters when it shuts down. Switching to this runlevel starts scripts to end processes and cleanly halt the system.
- **1** Single user mode. Normal user logins are not allowed.
- **2 to 5** Multiuser mode. These are all the same in a Debian installation, so you can customise one of them if you need to. This is the normal mode of operation, allowing multiple users to log in, with all features enabled.
- **6** Reboot. Very similar to runlevel 0.

Then there's runlevel S, for single user mode, which we enabled before when editing *Grub*'s boot parameters. This is quite similar to runlevel 1, but there are subtle differences: S is the runlevel you use when booting the system and you need to be in a safe recovery mode. In contrast, you use runlevel 1 when the system is already running and you need to switch to a single user mode to do some maintenance work. Don't worry, though – already logged-in users won't be kicked off.

Although runlevels 2 to 5 are identical on Debian, in some other distros there are specific runlevels in this range. For instance, many distros use runlevel 3 for a multiuser, text-

Alerting users to runlevel changes

Changing runlevels on a single-user machine is no problem – you're already prepared for it. But what about on a multiuser machine? What if you have other users logged in via SSH and running programs? They don't want everything to disappear from under their feet in an instant. Fortunately, there are a couple of ways you can alert them about the changes to come. First, if you log in as root and enter **wall**, you can type a message and hit Ctrl+D to finish. This message will then be displayed on the terminals of every currently logged-in user. So you could, for instance, broadcast, "Shutdown in 10 minutes." Normal users can run **wall** too, but they can also disable messages from other normal users with the **mesg** command.

An alternative way to contact users in order to alert them is to mail them. This is similarly simple and can be done in a single command, such as the following:

```
echo "Reboot in 10 minutes" | mail -s "Reboot notice" user@localhost
```

If the users are running an email notification tool, they'll see the new message immediately. If you've got a big installation, with hundreds of logged-in users, you'll want to give several hours rather than minutes of advance notice.

mode login setup, and runlevel 5 for a graphical login (such as *Xdm/Gdm/Kdm*). To find out which runlevel you're currently using, run **/sbin/runlevel**. To switch to another runlevel, use the **/sbin/telinit** command, as root, like this:

```
/sbin/telinit 2
```

Now how do you find out which runlevel your distro runs by default? The magic here lies in the **/etc/inittab** file. Towards the top, you'll see lines like this:

```
# The default runlevel.
id:2:initdefault:
```

Lines beginning with hash marks are comments, while the lower line tells init that runlevel 2 is to be the default. If you create your own custom runlevel using the scripts for runlevel 3 and want to boot into it all the time, you can simply edit this file as root, change the number and restart your machine.

Slightly further down in **/etc/inittab**, you'll see a bunch of lines like the following:

```
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
...
```

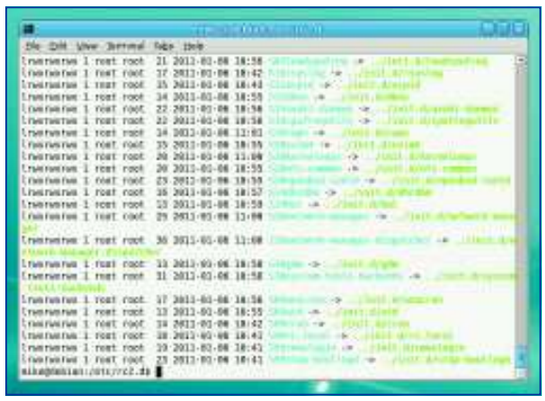
These will continue all the way to 6. These tell init what to do in each runlevel: run the script **/etc/init.d/rc** with the number of the runlevel as a parameter. Then **/etc/init.d/rc** will work out which scripts it needs to execute for the current runlevel. These are neatly organised in numbered directories inside **/etc**. So you'll find **/etc/rc0.d**, **/etc/rc1.d** and so on.

Inside a runlevel

Let's have a look inside **/etc/rc2.d** for the default Debian runlevel. Inside, you'll find a bunch of scripts with filenames such as **S05loadcpufreq** and **S89cron**. Each of these scripts enables a specific functionality in your Linux installation – have a look inside one and you'll see a description comment describing exactly what it does.

So we have **S30gdm**, which starts the *Gnome Display Manager*. What do the first three characters mean, though? S denotes that it's a script to start something and 30 gives it a position in the boot order. You can see that each script has a number like this and they're executed in numeric order. In this way, important scripts such as **S10rsyslog** are executed early on (to enable logging), while more trivial features such as *Cron* (**S89cron**) are enabled towards the end of the runlevel.

If you look carefully with **ls -l**, though, these scripts aren't actually unique files, but symbolic links to scripts in **/etc/init.d** – that's where the real scripts live. This is because scripts can be shared across runlevels. You might want to change the way *Cron* starts, for instance, so by editing **/etc/init.d/cron** you can make your modifications active across all



› **Each runlevel has a directory (/etc/rcX.d) with symbolic links to scripts in /etc/init.d.**

runlevels that use it. You can have a look inside **/etc/init.d** to see what's available.

These scripts are carefully written wrappers around the programs after which they're named. For instance, **/etc/init.d/gdm** isn't just a single-line text file containing **gdm**; rather, it sets up necessary environmental variables, adds messages to log files and so on. In a Debian system, most of these scripts can be called with parameters. For instance, run **/etc/init.d/gdm** and you'll see a line like this:

```
Usage: /etc/init.d/gdm {start|stop|restart|reload|force-reload|status}
```

So you can run **/etc/init.d/gdm start** to get *Gdm* going, and **/etc/init.d/gdm stop** to halt it. Note that **restart** does a stop and start, whereas **reload** asks the program to reread its configuration files without actually stopping, if this is possible. You can freely use these scripts outside of the whole runlevel system – for instance, to restart *Exim* or *Apache* after you've made changes to their configuration files.

Lastly, let's make a quick mention of something else in **/etc/inittab**, which isn't related to runlevels but is useful nonetheless. Have you ever wondered where the text terminals at bootup come from? The ones you can switch to with Ctrl+Alt+Fx from the *X server*? These are defined towards the bottom of **/etc/inittab** with lines like this:

```
2:23:respawn:/sbin/getty 38400 tty2
```

The **/sbin/getty 38400 tty2** part is simply a command to run a login prompt on the second virtual terminal. You can replace this with anything, so you could even have a virtual terminal devoted to *Tetris*! Meanwhile, **respawn** means that it restarts every time it quits. It's fun to play around with, but be careful – if you make it run a program that hogs all keyboard input, you won't be able to switch to another terminal to kill it. Still, it's not brown trouser time – just reboot into single user mode and revert your edits. LXF

Shutting down the system safely

In the desktop operating systems of the 1980s, you normally had no special process to shut down the computer – you just hit the power button when you were done with your work. This was fine back then, but on today's machines it can be very risky for two reasons. Firstly, some operating systems, including Linux, don't immediately write data to drives when you save a file. They wait until other processes want to save data, then bundle it all together in one big write operation in order to improve performance. You can, however, force Linux to write all data stored in its RAM buffers to disk with the **sync** command.

Secondly, Linux startup scripts also have shutdown equivalents, which make sure processes end safely, temporary files are cleaned up and so forth. Don't worry too much, it's not a massive crisis if they're not run, but it does help to keep your system in a tidy state. Most of us shut down via widgets on our desktop, but if you'd rather do this via the command line, have a look at the manual pages for **shutdown**, **halt** and **reboot**. With **shutdown**, for instance, you can specify a delay, but the most common command to power off a machine immediately is **shutdown -h now**.

» **Next month** We delve into the filesystem layout and shared libraries.