Previous parts on the DVD

The Mike Saunders SCHOOL of LINUX

Part 3: Moving on from hardware and the boot process, our class now turns to the filesystem layout, partitioning and shared libraries. And yes, there's a test at the end!

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.

Guess how many files you end up with on your hard drive after a single-CD Debian 6 installation. Go on – we'll wait. Well, the answer is 82,698. That seems almost unbelievable – and impossible to manage – but fortunately the Linux filesystem layout handles this vast number of files well, providing everything with a sensible place to live. You don't need to know what each individual file does in great detail, but from its location you can determine its overall purpose in the grand scheme of things.

This month, we're looking at how the Linux filesystem fits together, partitioning your hard drive and modifying the *Grub* bootloader's configuration. We'll also look at how shared libraries improve security and reduce disk space requirements. As with the other tutorials in this series, some filesystem locations and commands may vary depending on the distro you're using. However, for training purposes we recommend one used in enterprise, such as Debian, which this tutorial is based on. You can install it from your **LXFDVD**.

Section 1: The Linux filesystem layout

First, let's look at how a Linux installation is organised on a hard drive. As opposed to Windows, which has different 'starting points' or drive letters for each device, in Linux there's one single source of everything – like a Big Bang of data. This is **/**, or the **root** directory, which is not to be confused with the root user, aka administrator. All of the other directories stem from this, such as **/home/username**, your home directory. In other words, **root** is the top-level directory, and everything on the system is a subdirectory of it. Here are the items you'll find in the **root** directory:

» **/bin** This holds binary files – that is, executable programs. These are critical system tools and utilities, such as **ls**, **df**, **rm** and so forth. Anything that's needed to boot and fix the system should be in here, whereas **/usr/bin** has a different purpose – as we'll see in a moment.

» **/boot** This contains the kernel image file (**vmlinuz** – the z is because it's compressed), which is the program loaded and executed by the bootloader. It also contains a RAM disk image (**initrd**), which provides the kernel with a minimal filesystem and set of drivers to get the system running. Many distros drop a file called **config** here – which contains the settings used to build the kernel – and there's a **grub** subdirectory for bootloader configuration, too.

» **/dev** Device nodes. You can access most hardware devices in Linux as if they were files, reading and writing bytes from them. See part one of this series for more on **/dev**.

» **/etc** Primarily configuration files in plain text format, although there are exceptions. Boot scripts (see part two of this series) also live here. These are system-wide configuration files for programs such as *Apache*; settings for desktop apps typically live in a user's **home** directory.

» **/home** Where **home** directories usually reside. Each user account has a directory here for personal files and settings.

» **initrd.img** A symbolic link (not a real file, more like a Windows shortcut) to the aforementioned RAM disk file in **/boot**. You can see its full link target with **ls -l**.

» **Last month** We shone the light of understanding onto the boot process.

» **/lib** Shared libraries; see the *What are shared libraries?* box at the bottom of this page to learn more. Like **/bin,** these are critical libraries used to boot and run the system at a basic level. **/lib** also contains kernel modules (see part one of this series for more).

» **/lost+found** If your PC crashes or loses power during a heavy disk write operation and does a disk check (**fsck**) on next boot, pieces of partially lost files are deposited here.

» **/media** When external drives, such as USB keys, are plugged in, the auto-mounting process will give them a directory here from which you can access the files.

» **/mnt** A bit like **/media**, except this is usually used for manually mounted, long-term storage, including hard drives and network shares.

» **/opt** Optional software. This is quite rarely used, but in some distros and packages you'll find large suites such as KDE and *OpenOffice.org* placed here in order to keep everything neatly together (and therefore easy to remove or upgrade outside a package manager).

» **/proc** Access to process information. Each process (running task) on the system can be examined here, maintaining Unix's 'everything is a file' philosophy.

» **/root** Personal files used by the superuser or root account. Many administrators will keep backups of config files here too. When it comes to multiple-user installations, it's vital that normal user accounts can't poke around here.

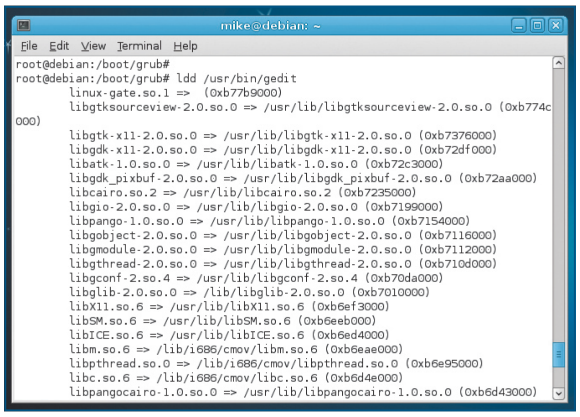
» **/sbin** Binary executable files, similar to in **/bin**, but explicitly for use by the superuser. This contains programs that normal users shouldn't run, such as network configuration tools, partition formatting tools and so on.

» **/selinux** A placeholder for files used by the Security-Enhanced Linux framework.

» **/srv** This is intended to be used for data served by the system (for example, a web server). However, most programs use **/var** instead.

» **/sys** Like a more modern **/dev**, with extra capabilities. You can get lots of information about hardware and the kernel here, but it's not important for normal admin jobs.

» **/tmp** Temporary files. Any program can write here, so you'll see random bits and bobs from background services, web browsers and so on. Most distros clean it at boot.



```
root@debian:/boot/grub#
root@debian:/boot/grub# ldd /usr/bin/gedit
        linux-gate.so.1 =>  (0xb77b9000)
        libgtksourceview-2.0.so.0 => /usr/lib/libgtksourceview-2.0.so.0 (0xb774c
000)
        libgtk-x11-2.0.so.0 => /usr/lib/libgtk-x11-2.0.so.0 (0xb7376000)
        libgdk-x11-2.0.so.0 => /usr/lib/libgdk-x11-2.0.so.0 (0xb72df000)
        libatk-1.0.so.0 => /usr/lib/libatk-1.0.so.0 (0xb72c3000)
        libgdk_pixbuf-2.0.so.0 => /usr/lib/libgdk_pixbuf-2.0.so.0 (0xb72aa000)
        libcairo.so.2 => /usr/lib/libcairo.so.2 (0xb7235000)
        libgio-2.0.so.0 => /usr/lib/libgio-2.0.so.0 (0xb7199000)
        libpango-1.0.so.0 => /usr/lib/libpango-1.0.so.0 (0xb7154000)
        libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0xb7116000)
        libgmodule-2.0.so.0 => /usr/lib/libgmodule-2.0.so.0 (0xb7112000)
        libgthread-2.0.so.0 => /usr/lib/libgthread-2.0.so.0 (0xb710d000)
        libgconf-2.so.4 => /usr/lib/libgconf-2.so.4 (0xb70da000)
        libglib-2.0.so.0 => /lib/libglib-2.0.so.0 (0xb7010000)
        libX11.so.6 => /usr/lib/libX11.so.6 (0xb6ef3000)
        libSM.so.6 => /usr/lib/libSM.so.6 (0xb6eeb000)
        libICE.so.6 => /usr/lib/libICE.so.6 (0xb6ed4000)
        libm.so.6 => /lib/i686/cmov/libm.so.6 (0xb6eae000)
        libpthread.so.0 => /lib/i686/cmov/libpthread.so.0 (0xb6e95000)
        libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb6d4e000)
        libpangocairo-1.0.so.0 => /usr/lib/libpangocairo-1.0.so.0 (0xb6d43000)
```

› Most programs derive a good chunk of their functionality from shared libraries, as running the ldd command shows.

» **/usr** This is a different world. **/usr** contains its own versions of the **bin**, **sbin** and **lib** directories, but these are for applications that exist outside of the base system. Anything that's vital to get the machine running should be in **/bin**, **/sbin** and **/lib**, whereas nonessential programs such as *Firefox* and *Emacs* should live here. There's a good reason for this: you can have the important base system on one partition (**/**) and add-on programs on another (**/usr**), providing more flexibility. **/usr**, for example, might be mounted over the network. In here, there's also **/usr/local**, which is typically used for programs you've compiled yourself, keeping them away from the package manager.

» **/var** Here be files that vary. In other words, files that change a lot, such as log files, databases and mail spools. Most distros place *Apache*'s document root here too (**/var/www**). On busy servers, where this directory is accessed often with lots of write operations, it's frequently given its own partition with filesystem tweaks for fast performance.

» **vmlinuz** A symbolic link to the kernel image file in **/boot**.

Depending on your distro, you may find additional items in the **root** directory. However, most distros try to abide by the guidelines of the Filesystem Hierarchy Standard (FHS). This is an attempt to standardise the filesystem layout across distros. You can delve deeper into **/var**, **/usr** and other directories by looking at the hierarchy manual page – just enter **man hier** in a terminal.

»

What are shared libraries?

A library is a piece of code that doesn't run on its own, but can be used by other programs. For instance, you might be writing an application that needs to parse XML, but don't want to create a whole XML parser. Instead, you can use *libxml*, the XML handling library, which someone else has already written. There are hundreds of libraries like this in a typical Linux installation, including ones for basic C functions (*libc*) and graphical interfaces (*libgtk*, *libqt*).

Libraries can be statically linked to a program – rolled into the final executable – but usually they're provided as shared entities in **/lib**, **/usr/lib** and **/usr/local/lib** with **.so** in the filename, which stands for shared object. This means multiple programs can share the same library, so if a security hole is discovered in it, only one fix is needed to cover all the apps that use it. Shared libraries also mean program binary sizes are smaller, saving disk space.

You can find out what libraries are used by a program with **ldd**. For instance, **ldd /usr/bin/gedit** shows a list of libraries including:

```
libgtk-x11-2.0.so.0 => /usr/lib/libgtk-x11-2.0.so.0
(0xb7476000)
```

Gedit depends on *GTK*, so it needs *libgtk-x11*, and on the right you can see where the library's found on the system. What determines the locations for libraries, though? The answer is in **/etc/ld.so.conf**, which nowadays points to all files in **/etc/ld.so.conf.d**. These files contain plain text lines of locations in the filesystem where libraries can be found, such as **/usr/local/lib**. You can add new files with locations here if you install libraries elsewhere, but must run **ldconfig** (as root) afterwards to update a cache that's used by the program loader.

Sometimes you might want to run a program that needs a library in a specific place that's not part of the usual locations. You can use the **LD_LIBRARY_PATH** environment variable for this. For instance, entering the following will run the **myprog** executable that's in the current directory, and temporarily add **mylibs** to the list of library locations as well:

```
LD_LIBRARY_PATH=/path/to/mylibs ./myprog.
```

Many games use this method to bundle libraries alongside a binary, without needing the binaries.

Section 2: Partitioning schemes

Drive partitioning is one of those tasks that an administrator rarely has to perform, but one that can have huge long-term consequences. Allocate the wrong amount of space for a particular partition and everything can get very messy later on. How you go about partitioning depends on the installer your distro uses, so we won't list thousands of keybindings here. Instead, we'll look at a partitioning tool common to all distros, and the options you have when divvying up a drive.

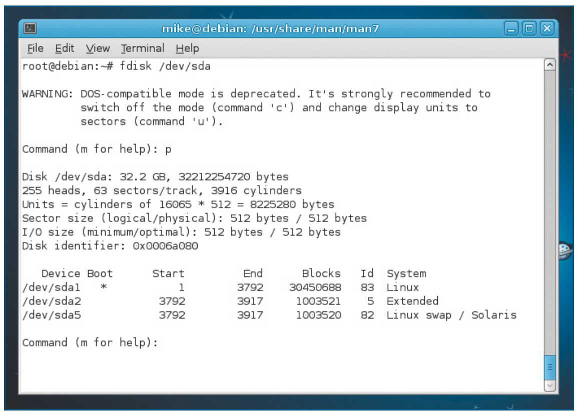
Open up a terminal, switch to root and enter:

```
fdisk /dev/sda
```

Replace **sda** with the device node for your drive. (This should be **sda** on single-hard drive machines, or **sdb** if you're booting Linux from a second drive. Consult **dmesg**'s output if you're unsure.) **fdisk** itself is a simple command-driven partitioning utility. Like *Vi*, it's austere in appearance, but it's ubiquitous. Enter **p** and you'll see a list of partitions on the drive, as in the screenshot, below. Type **m** to list the available commands: you can delete partitions (**d**), create new ones (**n**), save changes to the drive (**w**) and so forth.

This is a powerful tool, but it assumes that you know what you're doing and won't mollycoddle you. **fdisk** also doesn't format partitions – it just gives them a type number. To format, type in **mkfs** and hit Tab to show the possible completion options. You'll see that there are commands to format partitions in typical Linux formats (such as **mkfs.ext4**) plus Windows FAT32 (**mkfs.vfat**) and more.

Along with filesystem partitions, there's also the **swap** partition to be aware of. This is used for virtual memory. In other words, when a program can no longer fit in the RAM chips because other apps are eating up memory, the kernel can push that program out to the **swap** partition by writing the memory as data there. When the program becomes active again, it's pulled off the disk and back into RAM. There's no magical formula for exactly how big a **swap** partition should be, but most administrators recommend twice the size of the RAM, but no bigger than 2GB. You can format a partition as **swap** with **mkswap** followed by the device node (**/dev/sda5**, for instance), and activate it with **swapon** plus the node. You can also use a single file as swap space – see the **mkswap** and **swapon** man pages for more information.



```
root@debian:~# fdisk /dev/sda

WARNING: DOS-compatible mode is deprecated. It's strongly recommended to
         switch off the mode (command 'c') and change display units to
         sectors (command 'u').

Command (m for help): p

Disk /dev/sda: 32.2 GB, 32212254720 bytes
255 heads, 63 sectors/track, 3916 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0006a080

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        3792    30450688   83  Linux
/dev/sda2            3792        3917     1003521    5  Extended
/dev/sda5            3792        3917     1003520   82  Linux swap / Solaris

Command (m for help):
```

mike@debian: /usr/share/man/man7
File Edit View Terminal Help

› Most distros have their own graphical partitioning tools, but wherever you are, you'll always find the trusty fdisk.

Quick tip

Want to stop other users from fiddling around with options at the *Grub* boot screen? You can password-protect your boot entries so that only you can edit them – see *Grub*'s online documentation at **http://tinyurl.com/6czhkn8** for a guide.

Partition approaches

Now, what approach do you take when partitioning a drive? There are three general approaches:

1 All-in-one. This is a large single partition that contains the OS files, **home** directory data, temporary files and server data, plus everything else. This isn't the most efficient route in some cases, but it's by far the easiest, and means that each directory has equal right to space in the whole partition. Many desktop-oriented distros take this approach by default.

2 Splitting **root** and **home**. A slightly more complex design, this puts **/home** in its own partition, keeping it separate from the **root** (**/**) partition. The big advantage here is that you can upgrade, reinstall and change distros – completely wiping out all the OS files if necessary – while the personal data and settings in **/home** remain intact. For a more detailed look at the benefits of creating a separate **/home** partition, turn to our tutorial on page 80.

3 Partitions for all. If you're working on a critical machine, such as a live internet-facing server that needs to be up 24-7, you can develop some very efficient partitioning schemes. For instance, say your box has two hard drives: one that's slow and one that's fast. If you're running a busy mail server, you can put the **root** directory on the slow drive, since it's only used for booting and the odd bit of loading. **/var/spool**, however, could go on the faster drive, since it could see hundreds of read and write operations every minute.

This flexibility in partitioning is a great strength of Linux and Unix, and it just keeps getting more useful. Consider, for example, the latest fast SSD drives: you could put **/home** on a traditional hard drive to give yourself plenty of room at a cheap price, but put the **root** directory onto an SSD so that your system boots and runs programs at light speed.

Test yourself!

As you progress through this series of tutorials, you may want to assess your knowledge along the way. After all, if you go for full-on LPI certification when our time together is over, you'll need to be able to use your knowledge on the spot without consulting the guides. We're planning to include a comprehensive set of example questions when this series concludes, but for now here are some tasks to try and questions to answer based on the three sections in these pages:

1 Where are the kernel image and RAM disk files located?

2 Explain the difference between **/lib**, **/usr/lib** and **/usr/local/lib**.

3 Explain the available Linux partitioning schemes. Why would you put **/home** on a separate partition?

4 Describe how to add a new location for libraries on the system.

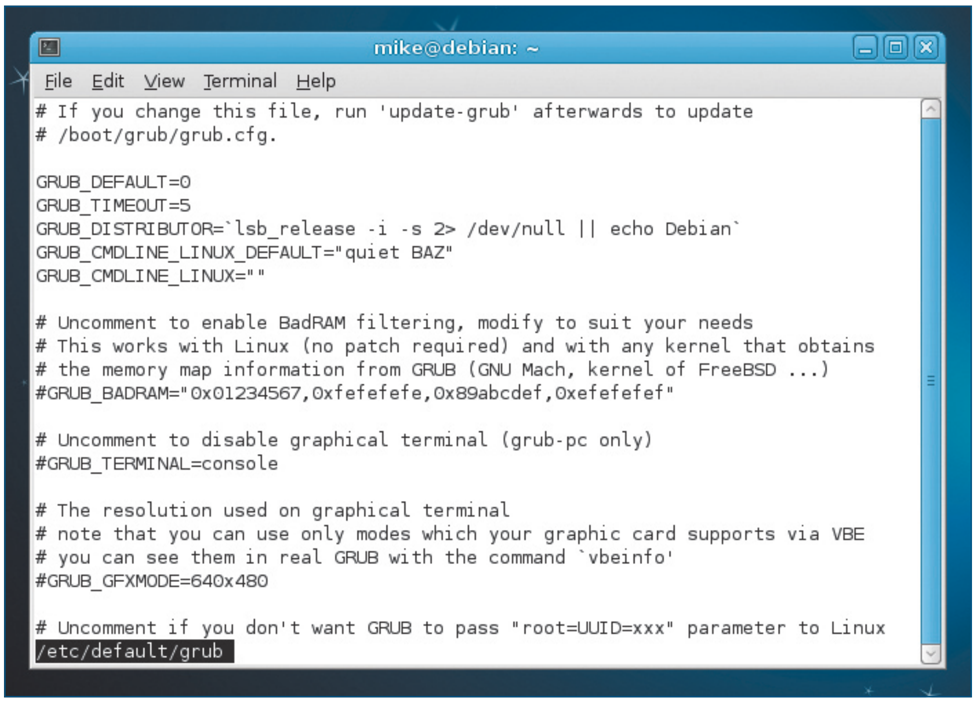
See if you can answer these without having to turn back to the relevant sections. If you struggle, no worries – just go back and read it again. The best way to learn is to take the information we've provided and experiment on your machine (or a distro in *VirtualBox* if you don't want to risk breaking an installation).

Section 3: Configuring the boot loader

Almost all Linux distributions today use *Grub 2* (the *Grand Unified Bootloader* to give it its full name) to get the kernel loaded and start the whole Linux boot process. We looked at *Grub* in last issue's tutorial, and specifically how to edit its options from inside *Grub* itself. However, such edits are only temporary. If there's a change you need to do every time, it'd be a pain to interrupt the boot sequence at each startup. The solution is to edit the **/etc/default/grub** file.

This isn't actually *Grub*'s own configuration file – that's located at **/boot/grub/grub.cfg**. However, that file is automatically generated by scripts after kernel updates, so it's not something you should ever have to change by hand. In most cases, you'll want to add an option to the kernel boot line, such as one to disable a piece of hardware or boot in a certain mode. You can add these by opening up **/etc/default/grub** as root in a text editor, and looking at this line:

```
GRUB_CMDLINE_LINUX_
DEFAULT="quiet"
```



```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.

GRUB_DEFAULT=0
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet BAZ"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
# This works with Linux (no patch required) and with any kernel that obtains
# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
#GRUB_BADRAM="0x01234567,0xfefefefe,0x89abcdef,0xefefefef"

# Uncomment to disable graphical terminal (grub-pc only)
#GRUB_TERMINAL=console

# The resolution used on graphical terminal
# note that you can use only modes which your graphic card supports via VBE
# you can see them in real GRUB with the command `vbeinfo'
#GRUB_GFXMODE=640x480

# Uncomment if you don't want GRUB to pass "root=UUID=xxx" parameter to Linux
/etc/default/grub
```

› **The place to add kernel boot options is /etc/default/grub – remember to run update-grub afterwards in order to transfer your changes to /boot/grub/grub.cfg.**

This contains the default options that are passed to the Linux kernel. Add the options you need after **quiet**, separated by spaces and inside the double-quotes. Once done, run:

```
/usr/sbin/update-grub
```

This will update **/boot/grub/grub.cfg** with the new options.

Old and grubby

If you're using an older distro with *Grub 1*, the setup will be slightly different. You'll have a file called **/boot/grub/menu.lst**, which will contain entries such as:

```
title Fedora Core (2.6.20-1.2952.fc6)
root (hd0,0)
kernel /vmlinuz-2.6.20-1.2952.fc6 ro root=/dev/md2 rhgb
quiet
initrd /initrd-2.6.20-1.2952.fc6.img
```

Here, you can add options directly to the end of the **kernel** line, save and reboot for the options to take effect. Should *Grub* get corrupted or removed by another bootloader, you can reinstall it by running the following as root:

```
grub-install /dev/sda
```

Replace **sda** with **sdb** if you want to install it on your second hard drive. This writes the initial part of *Grub* to the first 512 bytes of your hard drive, which is also known as the master boot record (MBR). Note that *Grub* doesn't always need to be installed on the MBR; it can be installed in the superblock (first sector) of a partition, allowing a master bootloader in the MBR to chain-load other bootloaders. That's beyond the scope of LPI 101, however, and you're unlikely to come across it, but it's worth being aware of.

Finally, while *Grub* is used by the vast majority of distros, there are still a few doing the rounds that use the older *LILO* – the *Linux Loader*. Its configuration file is **/etc/lilo.conf**, and after making any changes you should run **/sbin/lilo** to update the settings stored in the boot sector. LXF

Quick tip

A mount point is a place where a partition is attached to the filesystem. Say, for instance, that the **/dev/sdb2** partition on your hard drive is going to be used for the **home** directories: in this case, its mount point will be **/home**. Simple!

The magic of /etc/fstab

After reading about the partitioning schemes available, and how to set them up using **fdisk** and **mkfs**, you may be wondering how they hook into a working Linux installation. The controller of all this is **/etc/fstab**, a plain text file that associates partitions with mount points. Have a look inside and you'll see various lines that look like this:

```
UUID=cb300f2c-6baf-4d3e-85d2-9c965f6327a0 /
ext3    errors=remount-ro 0      1
```

This is split into five fields. The first is the device, which you can specify in **/dev/sda1-type format** or with a more modern UUID string (use the **blkid** command with a device node to get its UUID – it's just a unique way of identifying a device). Then there's its mount point, which in this case is the **root** directory. Following that is the filesystem format, and then options.

Here, we're saying that if errors are spotted when the boot scripts mount the drive, it should be remounted as read-only, so that write operations can't do more damage. Use the man page for **mount** to see all the options available for each filesystem format. The final numbers deal with filesystem checks, and you don't need to change these defaults.

You can add your own mount points to **/etc/fstab** with a text editor, but beware that some distros make automatic changes to the file, so be sure to keep an original copy backed up too.

» **Next month** Package management explained in-depth – both RPM and Deb!