

LPI

Learn Linux and get certified

Table of Contents

LPI -Part 1: Hardware.....	3
Section 1: Listing hardware.....	3
Section 2: Driver modules.....	4
Section 3: USB peripherals.....	9
LPI- Part 2: booting.....	11
Section 1: From power up to desktop.....	11
Section 2: Editing Grub settings.....	13
Section 3: Viewing log files.....	15
Section 4: Runlevels and the magic of /etc/init.d/.....	15
Shutting down the system safely.....	17
LPI Part3: FS layout, partitioning and shared libraries.....	19
Section 1: The Linux FS layout.....	19
What are shared libraries?.....	21
Section 2: Partitioning schemes.....	22
Section 3: Configuring the boot loader.....	23
LPI: Part 4: Package Management.....	25
Section 1: The Debian way.....	25
Building packages from source code.....	28
Section 2: The RPM way.....	29
LPI Part 5: The Command Line.....	31
Section 1: Getting orientated.....	31
Section 2: Delving deeper.....	31
Section 3: Understanding the environment.....	33
Test yourself!.....	33
LPI Part 6: Advanced Command Line.....	34
Section 1: Redirecting output.....	34
Section 2: Processing text.....	35
Test yourself!.....	37
LPI Part 7: Processes and Filesystems.....	38
Section 1: Managing processes.....	38
Section 2: Creating new filesystems.....	41
Test yourself!.....	42
LPI Part: 8 Links and Permissions.....	43
Section 1: Restricting disk space with quotas.....	45

LPI -Part 1: Hardware

Certified and bonafide

LPI stands for the Linux Professional Institute, a non-profit group that provides exams and qualifications for those seeking to work with Linux systems. There are three levels of certification available, the first of which covers general system administration, including configuring hardware, working at the command line, package management and handling processes. In this series, we're going to set you up with all you need to know for the LPI 101 exam.

Live and learn

LPI training based on long-life distros that don't change drastically every six months. Red Hat Enterprise Linux (RHEL) is a good example, but it's not cheap, so CentOS is an excellent base for your training. Another good choice is Debian, which tries to adhere to standards and remains stable for years at a time. Here, we'll use CentOS 5.5.

Section 1: Listing hardware

/proc and /sys directories: These are not real folders in the same sense as your home directory, but rather virtual directories created by the kernel, which contains information about running processes and hardware devices. What are they for? Well, /proc is largely focused on supplying information about processes (running programs on the system), whereas /sys primarily covers hardware devices. However, there is a bit of overlap between the two.

/sbin/lspci # obtain information about hardware devices

/sbin/lspci -vv # More detail output (include interrupts and I/O)

With these commands you should be able to see your video card, Ethernet adapter and other devices

Section 2: Driver modules

What if you want to disable a device? Well, first of all we need to identify what enables a device in the first place: the hardware driver. In Linux, drivers can be enabled in two ways when compiling the kernel. The distro maker can either compile them directly into the kernel, or as standalone module files that the kernel loads when necessary. The latter approach is the norm, since it makes the kernel smaller, speeds up booting and makes the OS much more flexible too.

You can find your modules in `/lib/modules/Kernel Version /kernel`.

These are KO files, and you'll see that they're sorted into directories for sound, FS and so on. It's important to make a distinction here between block and char devices. The former is for hardware where data is transmitted in large blocks, such as hard drivers, whereas character devices stream data a byte or so at a time (mice and serial ports).

What is /dev?

In Linux everything is a file. Not just your documents and images, but hardware too. However, there are some devices (such as random number generators) that normally work on one way only: they can be read, for instance, but you have no ability to send anything back. The /dev directory contains hardware device nodes - files representing the devices. For example, `/dev/dvd` is your DVD-ROM drive. You can enter `cat /dev/dvd` and it would spew out the binary data to your terminal. Device nodes are created automatically by the kernel. Running `strings /dev/mem | less` is a fascinating way to see what text your RAM holds.

The Linux kernel is clever, and can load modules when it detects certain pieces of hardware (it can load modules on demand when USB devices are plugged in). To get a list of all the modules that the kernel has currently loaded, enter the command:

`/sbin/lsmmod`

You'll see a list of modules. The exact contents of which will vary from system to system, depending on the hardware that you have installed.

You know my name

The names of some of these modules will be immediately obvious to you, such as `cdrom` and `battery`. For certain modules, you'll see a list of **Used By modules** in the right column. These are like dependencies in the package management world, and show which modules need the other ones to be loaded before. What about those modules with cryptic names? What do they do exactly? Use `modinfo` to find it:

```
/sbin/modinfo dm_mod
```

We get a bunch of information. This is largely technical, but comes with a handy `Description` line that provides information about what the module does. Unfortunately, not every module has anything useful in this field, but it's worth trying to check it. As mentioned, many modules are loaded by the kernel automatically. You can also force one to be loaded with the `modprobe` command. This small utility is responsible for both loading and removing modules from the kernel, and is a very handy way to disable and enable kernel functionality on the fly. For instance, in our module list we see that there's `lp`, `parport` and `parport_pc`. These are for printers parallel port, which hardly anyone uses these days, so we can disable this functionality to free up a bit of RAM with:

```
/sbin/modprobe -r lp parport_pc parport
```

How do we know the right order to enter these? We can work it out using the `Used By` field mentioned before.

Probe deeper

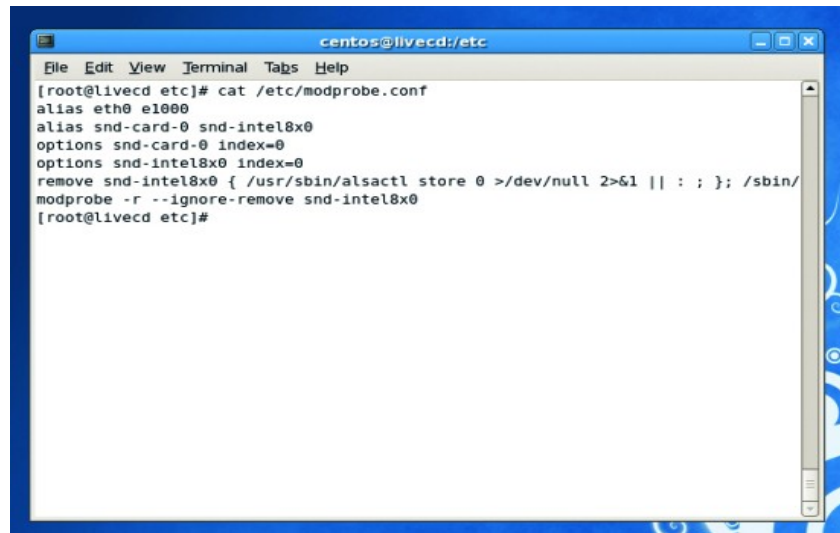
Similarly, we can enable these modules again by using same `modprobe` command (without the `-r` remove flag). Because of the dependencies system, we need only specify the first in the list, and `modprobe` will work out what else it needs:

```
/sbin/modprobe lp
```

This also loads up `parport_pc` and `parport`, which we can confirm with a quick `lsmod` command. While Linux typically handles modules automatically sometimes it's useful to have a bit of manual input in the process. We can do this via the `/etc/modprobe.conf` file. First up is `aliases`, a way to

provide a shorthand term for a list of modules. For instance, you might want to be able to disable and enable your sound card manually, but you can't always remember the specific module that it uses. You can add an alias line like this:

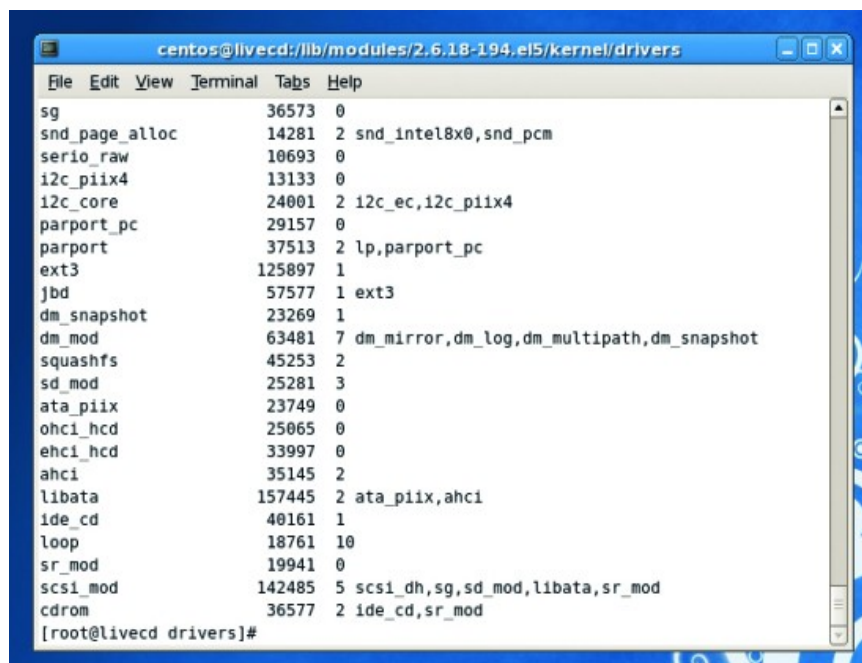
alias sound snd-ens1371



```
centos@livecd:/etc
File Edit View Terminal Tabs Help
[root@livecd etc]# cat /etc/modprobe.conf
alias eth0 e1000
alias snd-card-0 snd-intel8x0
options snd-card-0 index=0
options snd-intel8x0 index=0
remove snd-intel8x0 { /usr/sbin/alsactl store 0 >/dev/null 2>&1 || : ; } /sbin/
modprobe -r --ignore-remove snd-intel8x0
[root@livecd etc]#
```

An example /etc/modprobe.conf file in CentOS 5.5

Now you can just enter `modprobe sound` and have your card working without having to remember the specific driver. Using this system, you can unify the commands you use across different machines. Then there's options, which enables you to pass settings to a module to configure the way it works. Use *modinfo* command and look for parm sections:



```
centos@livecd:/lib/modules/2.6.18-194.el5/kernel/drivers
File Edit View Terminal Tabs Help
sg 36573 0
snd_page_alloc 14281 2 snd_intel8x0,snd_pcm
serio_raw 10693 0
i2c_piix4 13133 0
i2c_core 24001 2 i2c_ec,i2c_piix4
parport_pc 29157 0
parport 37513 2 lp,parport_pc
ext3 125897 1
jbd 57577 1 ext3
dm_snapshot 23269 1
dm_mod 63481 7 dm_mirror,dm_log,dm_multipath,dm_snapshot
squashfs 45253 2
sd_mod 25281 3
ata_piix 23749 0
ohci_hcd 25065 0
ehci_hcd 33997 0
ahci 35145 2
libata 157445 2 ata_piix,ahci
ide_cd 40161 1
loop 18761 10
sr_mod 19941 0
scsi_mod 142485 5 scsi_dh,sg,sd_mod,libata,sr_mod
cdrom 36577 2 ide_cd,sr_mod
[root@livecd drivers]#
```

For instance, when running `modinfo snd-intel8x0` we can see a list of parm sections that show options available for this sound chip module. One is called `index`. Our CentOS on VirtualBox `/etc/modprobe.conf` shows this in action with:

`options snd-intel8x0 index=0`

Custom commands

Lastly, we have the install and remove facilities. These are really powerful: they enable you to replace commands with different ones. For instance, in CentOS on VirtualBox we see:

`remove snd-intel8x0 { /usr/bin/alsactl store 0... }`

The full line is much longer, but essentially it says: 'When the user or system runs `modprobe -r snd-intel8x0`, execute this command instead, beginning with `alsactl` - a volume control utility.' In this way, you can perform clean up and logging operations before the module removal takes place. To prevent a module from loading entirely, simply alias it to off in `/etc/modprobe.conf`:

`alias parport off`

This will stop the module from ever being loaded, and therefore usually stop the hardware from being activated.

What are HAL, udev, D-Bus?

Desktop environments, such as Gnome and KDE, are abstracted from the hardware management. After all, Gnome hackers working on a photo management app don't want to write code to poke bytes down a USB cable to a camera - they want the OS to handle it. This makes sense and enables Gnome to run on other OSes.

The HAL (hardware abstraction layer) daemon once provided this abstraction, but it's been replaced by udev, a background process that creates device nodes in `/dev` and interfaces with hardware. How do programs interact with udev? They do this primarily via D-Bus, an inter-process communication (IPC) system which helps programs send messages to one another. For instance, a desktop environment can ask D-Bus to inform it if a new device is plugged in. D-Bus gets this

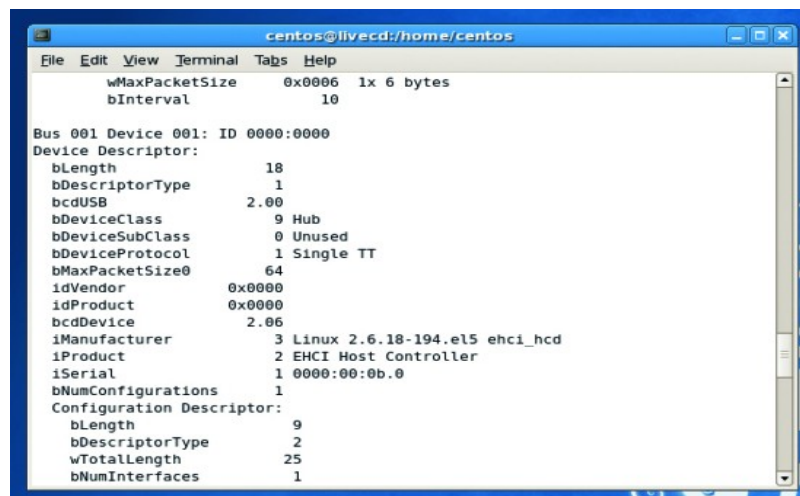
information from udev when the user plugs in hardware and then informs the desktop so that it can pop up a dialog or launch an app.

Section 3: USB peripherals

USB simplify the access to external hardware devices. When you plug in a USB device, the kernel probes it to find out what class it belongs to. There are classes for audio devices, printers, webcams and more.

ls + usb = lsusb

/sbin/lspci | grep -i usb



```
centos@livecd:/home/centos
File Edit View Terminal Tabs Help

wMaxPacketSize 0x0006 1x 6 bytes
bInterval 10

Bus 001 Device 001: ID 0000:0000
Device Descriptor:
  bLength           18
  bDescriptorType   1
  bcdUSB            2.00
  bDeviceClass       9 Hub
  bDeviceSubClass    0 Unused
  bDeviceProtocol    1 Single TT
  bMaxPacketSize0    64
  idVendor           0x0000
  idProduct          0x0000
  bcdDevice          2.06
  iManufacturer      3 Linux 2.6.18-194.el5 ehci_hcd
  iProduct            2 EHCI Host Controller
  iSerial             1 0000:00:0b.0
  bNumConfigurations 1
Configuration Descriptor:
  bLength           9
  bDescriptorType   2
  wTotalLength       25
  bNumInterfaces     1
```

After you've run the command, a few lines of information should appear, telling you the vendor and type of USB controller that you have. Slightly confusingly, there are two standard controller types for USB 1: UHCI and OHCI. USB 2.0 created EHCI, which layers on top of one of those. You don't need to worry about the differences between them, since the kernel handles this itself, but be aware that there's a bit of fragmentation in the USB world.

Dmesg in a bottle

A good way to determine how the kernel is recognize USB devices is with the *dmesg* command. Run *dmesg*, then plug in a USB device, wait a few seconds for it to be recognized, and run *dmesg* again... Extra lines will be added to the bottom of the *dmesg* output, showing that the kernel has (hopefully) recognized the device and activated it.

LPI- Part 2: booting

You press the power button on your PC. A bunch of messages scroll by and finally you arrive at a login prompt. What exactly happens in the mean time? That's what we'll be explaining now. Last chapter we have used CentOS, this time it's Debian (version 5) turn.

Section 1: From power up to desktop

The Linux boot process is an intricate collection of processes and scripts that turn your PC. 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. Typically, the BIOS performs quick checks of your hardware (for example: making sure the RAM chips are working), 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 try to load 512 byte bootloader. In 1980 such a tiny amount of memory was fine for loading the OS kernel. However, modern bootloaders must support many different file-systems, OS's and graphics modes, so 512 bytes isn't enough. In the case of Grub, as used by most Linux distros, the 512 bytes of bootloader loads another program called Grub Stage 1.5 that load, afterward, 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.

Booting into the future

Traditional Linux boot scripts run sequentially: one follows another. This is simple, and guarantees that certain bits of hardware and features will be enabled at certain points in the boot process. However, it's an

inefficient way of doing things and leads to long boot-up time, especially on older hardware. Much of the time, the scripts are waiting for something to happen (like DHCP server on the network instance). Wouldn't it be great if other things could be done in that delays? That's the aim of parallelized init scripts. While your network script is waiting for DHCP, another script can clean /tmp or start the X Window System. You can't just put '&' on the end of every script and run them in parallel, though since some scripts depend on certain facilities being available. For instance, a boot script that gets an IP address via DHCP needs to assume the network is already enable by another script. *InitNG* is a parallelized boot system in which scripts have dependencies to sort out their order. *Upstart*, as used by Ubuntu, starts scripts based on system events (such when hardware device is detected). Then there's *System D* (Fedora) and other approaches. Let's hope that the Linux world will eventually settle on one system, but in any case the move towards parallelism is hugely speeding up the Linux boot process.

Linux kernel and Init

When the very first bytes of the Linux kernel begin executing, it's like a newborn, unaware 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 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 focus on the user land: The kernel isn't interested in running Bash, GDM 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. The main config file for init is */etc/inittab*, a plain text file you can edit.

This file is based on the concept runlevels: the different states the system can be in, such as single user, multiuser and shutdown. 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 (establish the network, start system loggers, launch the X Window System and login manager). This entire process, from the power button to clicking icons, require a lot of work, but is generally well-shielded from the user.

Section 2: Editing Grub settings

Now let's look at the bootloader in more detail. In most cases, this will be Grub. 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:

```
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 (for FS checks if it should run), but it will be remounted as read-write shortly after. Meanwhile, quiet tells the kernel that we don't want to see lot of messages, making the boot cleaner and easier to follow. We can modify these options by using the Down cursor key and hit 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 initialized and mounted the root partition, and you'll be asked for the root user password.

Alerting users to runlevel changes

Changing runlevels on a single-user machine is no problem, But what about on a multiuser machine? What if you have other users logged in via SSH and running programs? You can mail the users in order to alert them:

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.

Section 3: Viewing log files

Look the file */var/log/messages* 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 warning messages such 'warning: strange, CPU MTRRs'.

Kernel Saunders explains

Roughly, the order in which the kernel works is like this::

- Get hardware Information from the BIOS (It's not always reliable).
- Find out how many CPUs/COREs exist and learn any CPU features.
- Get ACPI Information and probe the PCI bus for devices.
- Initialize the TCP/IP Networking Stack.
- Look for Floppy,CDROM Drives and USB controllers

Once the kernel is happy with the state of the system, it mounts the root FS 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. If you want to get kernel-only messages, run *dmesg* command. While most of the messages are from the early parts of the boot process, this information will be updated if you plug in new hardware, USB for instance

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

A runlevel defines a state for your system - specifically, which processes are running and which resources are available. It's not secret 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:** To this runlevel the machine enters in shutdown
- **1 Single user mode:** Normal user logins are not allowed.
- **2 to 5 Multiuser mode:** All the same in a Debian installation
- **6 Reboot:** Very similar to runlevel 0.
- **S Single user mode.** 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. Although runlevels 2 to 5 are identical on Debian (so you can customize one of them), in some other distros there are specific runlevels in this range. For instance, many distros use runlevel 3 for a multiuser, text-mode login setup, and runlevel 5 for a graphical login (Gdm/Kdm). To find out which runlevel you're currently using run: `/sbin/runlevel`. To switch to another runlevel, use `/sbin/telinit` command, as root, like this: `/sbin/telinit 2`

You can find out which runlevel your distro runs by default by viewing `/etc/inittab` file. Near the top, you'll see lines like this: `id:2:initdefault:` Lines beginning with `#` are comments. This line tells init that runlevel 2 is the default. Slightly further down in `/etc/inittab`, you'll see a bunch of lines like these:

```
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 organized 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: 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 on these scripts with *ls -l*, you'll notice that 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). In a Debian system, most of these scripts can be called with parameters: run */etc/init.d/gdm* and you'll see a line like this:

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

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

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 (*respawn* means that it restarts every time it quits).

LPI Part3: FS layout, partitioning and shared libraries

In this chapter we're looking at how the Linux FS 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 FS 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, on which this tutorial is based.

Section 1: The Linux FS layout

Opposed to Windows, Linux has one single source of everything: `/` 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. 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
- `/boot` - This contains the kernel image file (`vmlinuz` - 'z' 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 FS and set of drivers to get the system running. Many distros drop the config file here (it 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 like a file: reading and writing bytes from them.
- `/etc` - Primarily configuration files (generally plain text files). Boot scripts also live here. These are system-wide configuration files for programs such as Apache.
- `/home` - The user directory for personal files and settings
- `initrd.img` - A symbolic link (not a real file) to the RAM disk file in `/boot`
- `/lib` - Shared libraries (see below)
- `/lost+found` - pieces of fsck lost files are deposited here.
- `/media` - External drives (like USB keys) the auto-mounting here
- `/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 you'll find large suites such OpenOffice.org placed here
- **/proc** - Access to process information. Each process on the system can be examined here - maintain everything is a file philosophy.
- **/root** - Personal files used by the root user. Many administrators will keep backups of config files here too.
- **/sbin** - Binary executable files, similar to /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** - Place for the Security-Enhanced Linux framework.
- **/sys** - More modern of /dev, with extra capabilities. You can get lots of information about hardware\kernel here.
- **/tmp** - Temporary files (any program can write here). Most distros clean it at boot.
- **/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 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.
- **/var** - Files that vary\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.

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 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 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?

The answer is in `/etc/ld.so.conf`, which 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 if you install libraries elsewhere, but you must run as root:

`ldconfig` to update the cache 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 the

Section 2: Partitioning schemes

Drive partitioning is one of those tasks that an administrator rarely has to perform, but allocating the wrong amount of space for a particular partition and everything can get very messy later on.

we'll look at a partitioning tool common to all distros:

Open up a terminal, switch to root and enter: `fdisk /dev/sdax`

- Enter **'p'** to see list of partitions on the drive
- Type **'m'** to list the available commands: you can delete partitions (d), creating new ones (n), save changes to the drive (w) and so forth...

fdisk doesn't format partitions. 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).

Partition approaches

There are three general approaches to partitioning a drive:

1. All-in-one: This is a large single partition that contains the OS files, **/home**, **/tmp**, **/bin**... 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, while the personal data and settings in **/home** remain. If you're working on a critical machine that needs to be up 24-7, you can develop a 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. Consider, for example, to use fast SSD drives and 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.

The magic of /etc/fstab

Have a look inside /etc/fstab and you'll see various lines that look like:

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

This is split into five fields:

- The first is the device (/dev/sdax or with UUID string (use *blkid* command with a device node to get its UUID).
- The mount point (in this case the root directory)
- 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.

Section 3: Configuring the boot loader

Almost all Linux distributions today use Grub 2. 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. A permanent 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"*. 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*

To update */boot/grub/grub.cfg* with this options.

If you're using an older distro with Grub 1, the setup will be slightly different.

The file `/boot/grub/menu.lst`:

```
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 quietinitrd
/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. If grub get corrupted or removed by another bootloader, you can reinstall it by running:

```
grub-install /dev/sdx
```

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 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.

Test yourself!

- Where are the kernel image and RAM disk files located?
- Explain the difference between `/lib`, `/usr/lib` and `/usr/local/lib`
- Explain the available Linux partitioning schemes.
- 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.

LPI: Part 4: Package Management

Package is a single compressed file that expands into multiple files and directories. Many packages contain programs, but some contain artwork and documentation. Large projects (such as KDE) are split into a range of packages, so that when one small program has a security fix, you don't need to download the entire desktop. Packages are typically more than simple archives. For instance, they can depend on other packages or include scripts that should be run when they're installed and removed. Making a high-quality package can be a lot of work, but it does make life easier for users.

Section 1: The Debian way

Let's start with Deb packages. Here's the filename for a typical Debian package: *nano_2.2.4-1_i386.deb*

There are five components to this filename:

- The name of the application
- Version number (2.2.4)
- -1 is the distro's own revision of the package,
- i386 identifies the CPU architecture that this package runs on
- Finally the .deb identifier suffix.

To install a deb package, enter:

dpkg -i nano_2.2.4-1_i386.deb

If you have multiple packages to install, use `dpkg -i *.deb`.

To remove a package run:

dpkg -r nano

will remove the program, but will leave any configuration files intact (in this case, /etc/nanorc).

If you want to get rid of everything, run:

dpkg --purge nano

APT is the Advanced Package Tool, and provides facilities beyond simple package installation and removal. If you want to install Vim editor, but you don't have the relevant Deb packages Enter:

apt-get install vim

APT will retrieve the correct packages for your current distro version from the Internet and install them.

apt-cache search chess

Search available packages with the word chess in their title or desc.

aptitude: is a program that provides various GUI-like features in a text mode environment, such as menus, dialog boxes and so on...

You can browse lists of packages using the cursor keys and Enter, and the available key press operations are displayed at the top.

Hit Ctrl+T to bring up a menu. Aptitude is great when you're logged into a remote machine via SSH and need to perform a certain job but can't remember the exact command for it.

back to the Dpkg tool : Dpkg can be used to query the database of installed packages. If you want to list all files included in nano package:

dpkg -L nano

To get a more detailed list of information about a package run:

dpkg -s nano

This will provide everything you need to know about the package: its version, size, architecture, dependencies and even the email address of the maintainer, in case you wish to report any problems.

Another useful command is

dpkg -S filename

This searches for files matching this filename on the system, and tells you which package provides them. For instance

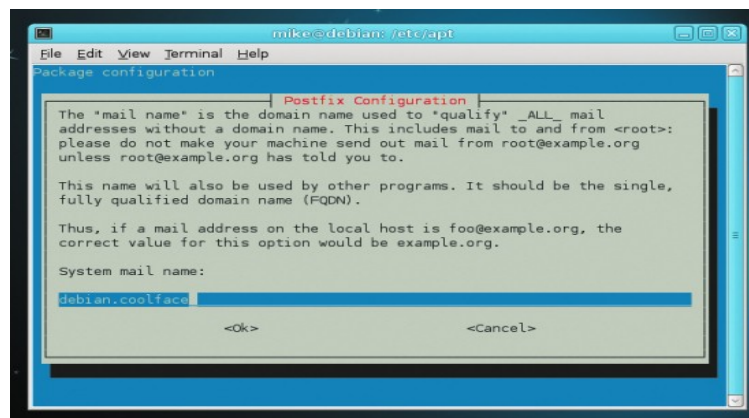
dpkg -S vmlinuz

will locate the vmlinuz kernel file on the system and show you which package originally carried out its installation.

Finally, a word about package configuration:

As you know, many programs have text-based configuration files in the /etc directory that you can modify by hand. That's all fine, but many Deb packages try to make things easier for the administrator by providing a certain level of automation. Install the Postfix mail server via apt-get, for instance, and a dialog box will pop up, offering to guide you through the server setup process. This way you don't have to learn the format of a specific configuration file. If you do ever need to change the configuration and want to do it the Debian way, simply use this command:

dpkg-reconfigure postfix



Building packages from source code

The process of creating packages is rather more involved than just zipping it, so scripts and configuration files are required. On Debian based distros, the first step is to install the required tools:

apt-get install dpkg-dev build-essential fakeroot

Next, tell Debian that you want access to source code and not just binary Debs by opening up ***/etc/apt/sources.list*** and duplicating lines that start with deb to deb-src. For example:

deb-src <http://ftp.uk.debian.org/debian/> squeeze main

You can then get the source for a program with **apt-get source package**

The original upstream source code will be downloaded, extracted and patched with any distro-specific changes. You can now go about making any source code customisations you need, or changing the optimization options for the compiler in the CFLAGS line in debian/rules. Then build the package using:

dpkg-buildpackage

Once the build process is complete, enter `cd ..` to go to the directory above the current one, then `ls`. You'll see that there are one or more freshly built Deb packages, which you can now distribute.

For RPM systems, you can install the yumdownloader tool, which lets you grab SRPM (source RPM packages) via **yumdownloader -source package**

An SRPM contains the source code along with specifications for building the code (a SPEC file), plus any distro-specific tweaks and patches. You can then build binary packages from them with

rpmbuild --rebuild filename.src.rpm

Converting packages with Alien:

RPM and Deb are the big two package formats in the Linux world, and they don't play nicely with one another: You can install the Dpkg tools on an RPM box (or the rpm command on a Debian box) and try to force packages to install that way, but the results won't be pretty and you can expect a lot of breakage. A slightly saner option is to use the Alien tool, which is available in the Debian repositories. This handy utility converts Deb files to RPMs, and vice versa. For instance:

alien --to-deb nasm-2.07-1.i386.rpm

All that Alien does is modify the compression format and metadata formats to fit a particular packaging system; it can't guarantee that the package will adhere to the filesystem layout guidelines of the distro, or that pre- and post-installation scripts will function correctly.

Over the years, we've had reasonable success when using Alien to convert small, standalone programs that have minimal dependencies. You might be in luck, too. Large apps are generally out of the question, though, and trying to replace critical system files (such as glibc) with those from another distro would be a very unwise move indeed.

Section 2: The RPM way

Look at this package:

rpm -Uvh nasm-0.98.39-1.i386.rpm

The filename structure is the same like Deb package:

- The name of the package
- Its version (0.98.39 in this case)
- Package maintainer's own version (1)
- The architecture

The flags that can be used in this command:

- U*** is particularly important, because it means 'upgrade'. You can use rpm ***-i*** to install a package
- q*** to get information about a package
- e*** remove a package from system

rpm --checksig - To check that the file isn't corrupt

rpm -qpi - to find out information about a package

rpm -qR nasm - For dealing with packages that are already installed,

rpm -ql nasm - To get a list of files installed by a package

rpm -qf /path/to/file - To find out which package a file belongs.

Yum - Yellowdog Updater Modified:

yum install zsh

yum update To grab information about operating system updates

In ***/etc/yum.repos.d*** you'll find text files ending in ***.repo***, which contain repository information. For instance, the stock CentOS installation that we're using for this tutorial contains repositories for all of the main CentOS packages and their relevant updates. You can add your own entries here if you find a program on the internet that has an

appropriate repository for your distro version, but make sure that you run `yum makecache` afterwards to update the locally stored information. Yum is highly configurable: see `/etc/yum.conf` for settings to play with.

Test yourself

- What's the command used to remove a Deb package, including its configuration files?
- Which file contains a list of repositories used in Debian-based distros?
- Which command provides a detailed list of information about a package in Debian?
- Which command would you use to convert an RPM file to a Deb?
- How do you remove a package from an RPM-based system?
- Where do Yum's repositories live?
- How do you refresh the cache of packages with Yum?

Answers: 1. `dpkg --purge` 2. `/etc/apt/sources.list` 3. `dpkg -s` 4. `alien --to-deb` 5. `rpm -e` 6. `/etc/yum.repos.d` 7. `yum makecache`

LPI Part 5: The Command Line

Section 1: Getting orientated

For most commands you can see which options are available using the `--help` flag, eg `uname --help`.

You might have installed something in `/opt` which needs to be added to your `$PATH` to function correctly. To do this, use the `export` command:

```
export PATH=$PATH:/opt/newprog
```

Now, when you do `echo $PATH` you'll see the previous locations along with `/opt/newprog` added to the end

Section 2: Delving deeper

If you see a file that you can't identify you can use the command:

```
file /usr/bin/emacs
```

This tool probes the first few bytes of a file to determine its type (if possible). For instance, if it spots a JPEG header, it'll tell you that it's a JPEG file. Of course, this isn't always 100% accurate, and you might find a plain text file identified as 'Microsoft FoxPro Database' or something crazy like that.

There are two ways to locate files: *locate* and *find*. They sound the same, but there's a fundamental difference: if you do *locate foobar.txt* it will consult a pre-made database of files on the system and tell you where it is at light speed. This database is typically updated every day by Cron, so it can be out-of-date. For more to-the-second results, use *find*:

```
find /home/mike -name hamster
```

```
find . -size +100k locates all files bigger than 100 kilobytes (use M for megabytes and G for gigabytes)
```

find . -type f will only show files, whereas *-type d* shows only directories. You can mix *-name*, *-size* and *-type* options to create very specific searches.

Creating and expanding archives

Quick reference for compressed files:

- *.gz* - A single compressed file
Extract: *gunzip foo.gz*
Compress: *gzip foo*
- *.bz2* - Like the above, but with stronger and slower compression.
Extract: *bunzip2 filename*
Compress: *use bzip2*
- *.tar* A tape archive. It use to bundle multiple files together into a single file (without compression).
Extract: *tar xfv foo.tar*
Join: *tar cfv foo.tar file1 file2 dir3*
that creates a new archive called *foo.tar* with the files
- *.tar.gz* / *.tar.bz2* - A combination of the previous formats and the most common way for distributing source code. Files are gathered together with *tar*, and then compressed with *gzip* or *bzip2*.
Extract: *tar xfv filename.tar.gz* or *tar xfv filename.tar.bz2*
Cmpress: *tar cfvz foo.tar.gz file1 file2* (for *.tar.gz*) or
tar cfvj foo.tar.bz2 file1 file2 (for *.tar.bz2*).
- *.cpio* - A relatively rare format that bundles files together into a single file (without compression).
Extract: *cpio -id foo.cpio*
- *dd* - Copy data from one source to another:
dd if=/dev/cdrom of=myfile.iso you end up with an ISO image

Section 3: Understanding the environment

There are environment variables which store bits of information such as options and settings to determine how commands operate. Environment variables are usually in capital letters and begin with a dollar sign. For instance, try this:

```
echo $BASH_VERSION
```

You'll see a number such as 3.2.25. Scripts can probe this variable to determine if the script can run in this version. To see a full list of the global environment variables in use, along with their contents, enter *env*. You can set up your own environment variables in this way:

```
export FOO="bar"  
echo $FOO
```

This new \$FOO variable will last as long as the terminal session is open. To make it permanent add it to *.bashrc* in your home directory (which contains variable definitions and other settings that are read when a command line session starts). Save your changes, restart the terminal and it will take effect.

env - get a full list of Environment variables

set - To see list of local variable. To remove it type

unset FOO

Test yourself!

- What does the tilde (~) sign in a command prompt mean?
- How would you list all files in the current directory, in detailed mode?
- Which command to find files uses a pre-made database?
- How would you set the environment variable \$WM to icewm?
- How would you add /opt/kde/bin to your \$PATH?
- How would you make a .tar.bz2 archive of the directory myfiles?
- You want to run a version of Nano from your current directory, not in your \$PATH. How?

1 - Home directory. 2 - ls -la. 3 - locate. 4 - export WM="icewm". 5 - export PATH=\$PATH:/opt/kde/bin. 6 - tar cfvj archive.tar.bz2 myfiles. 7 - ./nano.

LPI Part 6: Advanced Command Line

Section 1: Redirecting output

ls -la | less

Pipe: send the output of one command as input to another.

What are regular expressions?

In LPIC 1 training you don't need to be a regular expression (regexp) guru - just be aware of them. The most you're likely to come across is an expression for replacing text, typically in conjunction with sed, the streamed text editor. sed operates on input, does edits in place, and then sends the output. You can use it with the regular expression to replace text like this:

cat file.txt | sed s/apple/banana/g > file2.txt

Here we send the contents of file.txt to sed, telling it to use a substitution regular expression, changing all instances of the word apple to banana. Then we redirect the output to another file. This is by far the most common use of regular expressions for most administrators, and gives you a taste of what it's all about.

In certain situations, you might want to use the output of one command as a series of arguments for another. For instance, imagine that you want Gimp to open up all JPEG images in the current directory and any subdirectories. The first stage of this operation is to build up a list, which we can do with the find command:

find . -name ".jpg"*

We can't just pipe this information directly to Gimp, as it's just raw data when sent through a pipe, whereas Gimp expects filenames to be specified as arguments. We do this using *xargs*, a very useful utility that builds up argument lists from sources and passes them onto the program. So the command we need is:

find . -name ".jpg" | xargs gimp*

Tee: display the output of a command on the screen, and also redirect its output to a file:

free -m | tee output.txt

Here, the output of the *free -m* command (shows memory usage in megabytes) is displayed on the screen, but also sent to the file.

tee -a for appending data rather than overwriting it.

Section 2: Processing text

If you want a certain portion of a command's output, and you can trim it down with the cut command, like this: *cat words.txt | cut -c 5-7*

This will show characters 5 through, including 7.

cut can use any number of ways to break up text. Look at this command:

cat words.txt | cut -d " " -f 2

cut -d " " use space characters as the delimiter.

ls -lSh | head -n 6

S - sort by file size

There are a couple of characters we use in regexps to identify the start and end of a line: Create a plain text file containing three lines: bird, badger, hamster. Then run: *cat file.txt | grep -e ^b*

we tell grep to use a regular expression search, and the ^ character refers to the start of the line. So here, we just get the lines that begin with b - bird and badger. If we want to do our searches around the end of lines, we use the \$ character like this: *cat file.txt | grep -e r\$*

In this instance, we're searching for lines that end in the r character - so the result is badger and hamster. You can use multiple grep operations in sequence, separated by pipes, in order to build advanced searches.

Occasionally you'll see references to egrep and fgrep commands - they used to be variants of the grep tool, but now they're just shortcuts to specify certain options to grep command. See man page for more info.

cat list.txt | sort | uniq

uniq filters out repeated consecutive lines in a text stream, leaving just the first original intact.

uniq is tremendously powerful and has a bag of options for modifying the output further: for instance, try `uniq -u` to only show lines that are never repeated, or `uniq -c` to show a line count number next to each line. You'll find uniq very useful when you're processing log files and trying to filter out a lot of duplicate output.

Let's move on to reformatting text. Create a file name list.txt, and copy and paste its contents several times so that it's about 100 lines long.

Save it and then enter this command: `cat list.txt | fmt`

Here, the fmt utility formats text into different shapes and styles. By default, it takes our list - separated by newline characters - and writes out the result like a regular block of text, wrapping it to the width of the terminal window. We can control where it wraps the text using the -w flag, eg `cat list.txt | fmt -w 30`. Now the lines will be, at most, 30 characters wide.

If you love gathering statistics, then you'll need a way to count lines in an output stream. There are two ways to do this, using `nl` and `wc`. The first is a very immediate method which simply adds line numbers to the start of a stream, for instance: `cat /var/log/messages | nl`

This outputs the textual content of /var/log/messages, but with line numbers inserted at the start of each line. If you only want to count the line numbers use: `cat /var/log/messages | wc -l`

If you want to compare files you can use the diff utility, but a simpler tool to show which lines match in two files is join. Create a text file called file1 with the lines bird, cat and dog. Then create file2 with adder, cat and horse. Then run: `join file1 file2`

You'll see that the word cat is output to the screen, as it's the only word that matches in the files. If you want to make the matches case-insensitive, use the -i flag.

For splitting up files, there's the `split` command, which is useful for both textual content and binary files. For the former, you can specify how many lines you want to split a file into using the `-l` flag: `split -l 10 file.txt`. This will take `file.txt` and split it into separate 10-line files, starting with `xaa`, then `xab`, `xac` and so forth - how many files are produced will depend on the size of the original file. You can also do this with binary files. If you have a 6GB file and you run `split -b 4096m largefile` it will split into two parts: the first, `xaa`, is 4GB (4096MB) and the second, `xab`, contains the remainder. Once you've transferred these chunks to the target machine, you can reassemble them by appending the second file onto the first like this: `cat xab >> xaa`.

Finally, a mention of a few other utilities that may pop up if you take an LPI exam. If you want to see the raw byte data in a file, you can use the `hd` and `od` tools to generate hexadecimal and octal dumps respectively.

Test yourself!

- You have a file called `data.txt`, and you want to append the output of the `uname` command to it. How?
- How would you display the output of `df` and simultaneously write it to `myfile.txt`?
- You have `file.txt` containing this line: `bird,badger,hamster`. How would you chop out the second word?
- You have a 500-line file that you want to split into two 250-line chunks. How?
- And how do you reassemble the two parts?
- You have `file1.txt`, and you want to change all instances of the word `Windows` to `MikeOS`. How?
- And finally, take `myfile.txt`, sort it, remove duplicates, and output it with prefixed line numbers.

1 - `uname > data.txt`. 2 - `df | tee myfile.txt`. 3 - `cat file.txt | cut -d "," -f 2`. 4 - `split -l 250 file.txt`. 5 - `cat xab >> xaa`. 6 - `cat file1.txt | sed s/Windows/MikeOS/g > output.txt`. 7 - `cat myfile.txt | sort | uniq | nl`

LPI Part 7: Processes and Filesystems

Section 1: Managing processes

Ctrl+C stop running process (*Try it with `ls -R /`*)

By default, a process doesn't have more rights to resources than any other process on the system. If process A and process B are started, and they're both taxing the CPU, the Linux kernel scheduler will split time evenly between them. However, this isn't always desirable, especially when you have many processes running in the background. For instance, you might have a cron job set up to compress old archive files on a desktop machine: if the user is doing something important, you don't want them to suddenly lose 50% of their processing power whenever that cron job comes up. To combat this, there's a system of priorities. Each process has a nice value, which sets how the OS should treat it, with 19 the lowest priority, counting upwards to zero for the default, and -20 for the top priority. For instance, if you want to start a program with the lowest priority, use:

`nice -n 19 programname`

This will run the program, and if nothing else is happening on the system, it should complete in normal time. However, if the system gets under load from other processes, it will deal with them first. For nice values above zero, you have to be root:

`sudo nice -n -10 programname`

You can change a process's nice value using the renice command - see its manual page for more information

What if you want to merely pause the program's execution for later? Say, for instance, you've just entered `man gcc`. You've scrolled around and found an interesting point - so you want to try out some things, without losing your position. Hit *Ctrl+Z*, and the manual page viewer will disappear into the background, putting you at the command prompt. You can do your work and then type *fg* (for foreground) to bring the manual page viewer to the front, exactly where you left it.

It's possible to start a program in non-viewing mode, so you can switch to it when you're ready. This is done by appending (&) char like this:

```
man df &
```

Here, we start the man page of df command, but in the background. We get a line of feedback on the screen:

```
[1] 3192
```

The second number is the process ID. You can now go about doing your work, and when you're ready just enter *fg*. This system becomes especially useful when you combine multiple programs. For instance, enter:

```
nano &
```

```
man df &
```

Here we've started two programs in the background. If we enter *jobs*, we get a list of them. We can resume specific programs using a number - for instance, *fg 1* will switch to Nano, *fg 2* to the man page viewer.

Let's move on to processes. Ultimately, a process is an instance of execution by a program: most simple programs provide one process, which is the program itself. More complicated suites of software, such as a desktop environment, start many processes - file monitoring daemons, window managers and so forth. To show a list of processes of the current user enter *ps*. To view all processes running on the machine, enter *ps ax*.

Typically, this will be very long, so you can pipe it to the less viewer:

```
ps ax | less
```

here's an example line:

```
2972 pts/0 Ss 0:00 bash
```

The 2972 here is the process ID (PID). Every process has a unique ID, starting from 1, which is the */sbin/init* program that the kernel runs on boot. After that, the pts/0 bit shows from which virtual terminal the

command was run - if you see a question mark here, it's a process that was started outside of a terminal, eg by the kernel or a boot script. The **Ss** says that the process is sleeping (not doing any active processing), then there's a time indicator showing how much CPU time the process has consumed so far, followed by the command line used to start the process.

```
mike@localhost:~$ ps ax
2852 ?        Ss          0:00 eggccups --sm-client-id default4
2853 ?        Ss          0:00 gnome-volume-manager --sm-client-id default5
2866 ?        Ss          0:00 bt-applet --sm-disable
2874 ?        S           0:00 /usr/libexec/gam_server
2875 ?        Ss          0:00 /usr/bin/python -tt /usr/bin/puplet
2879 ?        S           0:00 /usr/libexec/wnck-applet --oaf-activate-iid=OAFIID:GN
2881 ?        S           0:00 /usr/libexec/trashapplet --oaf-activate-iid=OAFIID:GN
2883 ?        Ss          0:00 nm-applet --sm-disable
2900 ?        Ss          0:00 pam-panel-icon --sm-client-id default0
2901 ?        Ss          0:00 gnome-power-manager
2904 ?        S           0:00 /sbin/pam_timestamp_check -d root
2906 ?        S           0:00 /usr/libexec/mapping-daemon
2907 ?        Sl          0:00 ./escd --key_Inserted="/usr/bin/esc" --on_Signal="/us
2911 ?        S           0:00 /usr/sbin/nm-system-settings --config /etc/NetworkMan
2919 ?        S           0:00 /usr/libexec/notification-area-applet --oaf-activate-
2921 ?        S           0:00 /usr/libexec/clock-applet --oaf-activate-iid=OAFIID:G
2923 ?        Sl          0:00 /usr/libexec/mixer_applet2 --oaf-activate-iid=OAFIID:
2963 ?        S           0:00 /usr/libexec/notification-daemon
2967 ?        Ss          0:00 gnome-screensaver
2969 ?        Sl          0:02 gnome-terminal
2971 ?        S           0:00 gnome-pty-helper
2972 pts/0    Ss          0:00 bash
3382 pts/0    R+          0:00 ps ax
[mike@localhost ~]$
```

Here's the output of `ps ax`, showing all running processes on the system.

top - is an alternative way to get a list of processes. It's sort by default by CPU usage. Note that while there are various columns for memory usage, the most important is RES (resident), which shows exactly how much real memory the process is currently using up. To exit top, press **Q**.

```
mike@localhost:~$ top
top - 19:21:43 up 2:47, 2 users, load average: 0.26, 0.33, 0.32
Tasks: 114 total, 1 running, 112 sleeping, 0 stopped, 1 zombie
Cpu(s): 1.0%us, 1.3%sy, 0.0%ni, 96.7%id, 0.0%wa, 0.0%hi, 1.0%si, 0.0%st
Mem: 515316k total, 461756k used, 53560k free, 36100k buffers
Swap: 1048568k total, 0k used, 1048568k free, 281492k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 2720 root        15   0 52720 10m 5636 S   1.0   2.1   0:11.45 Xorg
 2323 root        18   0 1976   636 556 S   0.3   0.1   0:13.32 hald-addon-stor
 2818 mike        15   0 33704 7880 6524 S   0.3   1.5   0:00.62 gnome-settings-
 2838 mike        15   0 16596 7304 5776 S   0.3   1.4   0:00.43 metacity
 2901 mike        15   0 44884 6152 4648 S   0.3   1.2   0:02.23 gnome-power-man
 2907 mike        17   0 18208 2424 1984 S   0.3   0.5   0:00.99 escd
 3398 mike        15   0 39992 12m 9080 S   0.3   2.5   0:03.30 gnome-terminal
    1 root        15   0 2072   656 564 S   0.0   0.1   0:01.53 init
    2 root        RT  -5    0     0   0 S   0.0   0.0   0:00.00 migration/0
    3 root        34  19    0     0   0 S   0.0   0.0   0:00.03 ksoftirqd/0
    4 root        RT  -5    0     0   0 S   0.0   0.0   0:00.00 watchdog/0
    5 root        10  -5    0     0   0 S   0.0   0.0   0:01.64 events/0
    6 root        10  -5    0     0   0 S   0.0   0.0   0:03.16 khelper
    7 root        10  -5    0     0   0 S   0.0   0.0   0:00.13 kthread
   10 root        10  -5    0     0   0 S   0.0   0.0   0:00.17 kblockd/0
   11 root        20  -5    0     0   0 S   0.0   0.0   0:00.00 kacpid
   48 root        19  -5    0     0   0 S   0.0   0.0   0:00.00 cqueue/0
```

If you want to kill a process you have to find out its PID using the previous methods, and then enter: **kill NUM_OF_PID**.

kill sends a friendly message ("Would you mind shutting down?") to the process, which the process can then deal with (cleaning temp files before shutting down). Sometimes this will stop a process, but if that process is too messed up to deal with it, you're stuck. This is when kill starts to justify its name. Enter: *kill -9*

This doesn't even bother asking the program if it's OK - it just stops it immediately. If the process is halfway through writing a file, the results could be very messy, so this should be used with extreme care, when no other option is available. Sometimes you might have multiple processes with the same name, or you just don't want to look up its PID. In this case, you can use the *killall* command (*killall -9 app_name*)

Another useful signal which isn't destructive but informs a program to restart itself or re-read its configuration files is *SIGHUP*. Many programs will ignore this, but it works well on certain daemons:

killall -HUP sendmail

This tells all sendmail processes running to slow down a second, re-read the config files and then carry on. This is very useful when you want to make a quick change to a config file, and not take down the whole program.

Section 2: Creating new filesystems

The most basic tool for partitioning from the command line is fdisk:

#!/sbin/fdisk /dev/sda

Enter p to get a list of partitions on your hard drive, and m to get help.

There you'll see which commands delete partitions, create new partitions and so forth. Any changes you make aren't actually committed until you enter w to write them to disk. Some distros include cfdisk, a based version of fdisk that makes things a bit easier: there are simple menus and you move around with the cursor keys.

After you've made a partition, you need to format it. This is where the mkfs tools come into play. Type /sbin/mkfs and then hit tab to show possible options - you'll see there's mkfs.ext3, mkfs.vfat (for FAT32 partitions) and more. To format a partition, just provide its device node:

/sbin/mkfs.ext3 /dev/sda2

For swap partitions, use the *mkswap* command. You can then enable and deactivate swap space with the *swapon* and *swapoff* commands.

Testing filesystem integrity:

Reboot your distro into single-user mode (the method for this varies between distros, but usually involves editing the kernel line in the boot loader, and adding S to the end.) When you reach the command prompt, enter:

/sbin/fsck sda1

fsck is actually a front-end to various filesystem checking tools, and on most Linux boxes runs */sbin/e2fsck*, which handles ext2/3/4 filesystems. During the checking process, if problems are found, fsck will ask you what you want to do. After checking, you can run

/sbin/dumpe2fs sda1 to get more information about the partition, which helps if you need to report a problem in an online forum.

You may notice that many Linux distributions automatically run fsck filesystem checks after every 30 boots, or every 100 days, or a combination. You can change how this works with the *tune2fs* tool and its *-c* and *-C* options. There are also settings for how the kernel should treat a filesystem if it spots errors, and many other features. It's well worth reading the manual page, especially the information about the first five options.

Test yourself!

- How would you run the command *exim -q* in the background?
- You have several programs running in the background. How do you bring up a list of them?
- How do you generate a list of all processes?
- Exim has gone haywire, and you need to completely terminate all instances of it. What's the command?
- You've just created a new partition, */dev/sda2*, and you want to format it as FAT32. How?
- Provide a way to start *myprog* with the lowest priority.

1 - *exim -q &*. 2 - *jobs*. 3 - *ps ax*. 4 - *killall -9 exim*. 5 - */sbin/mkfs.vfat /dev/sda2*. 6 - *nice -n 19 myprog*.

LPI Part: 8 Links and Permissions

This is the final part of this series. You are invited to: www.linuxformat.com/files/lpquiz.txt to test yourself

Find \ Locate:

Example:

find . -name "linux"

This command has one major flaw: it's slow. There's an answer for this:

updatedb

locate linux

updatedb typically run by a daily Cron job (/etc/cron.daily)

Have a look at /etc/updatedb.conf for settings, and note in particular that you can avoid having certain filesystems and mountpoints added to the database.

Section 1: Creating links

Symbolic links are pointers to another file on the filesystem - they're a lot like 'shortcuts' in the Windows world. Symbolic links exist on their own, independent of the file they point to, so there's no harm in removing them. First let's set up a link as an example:

touch myfile

ln -s myfile mylink

-s states that the link is symbolic. If you enter *ls -l --color* at this stage, you will see that the link file has a different colour.

Caution:

rm mylink erase the symbolic link (not the file it points to)

rm myfile - The file that mylink is pointing to doesn't exist (it's a broken)

Enter *ls -l --color* and you can see it's a warning shade of red, and .file mylink tells you that it's broken.

Symlinks provide a great deal of flexibility and customisation, and you can see this by looking in */etc/alternatives* (if your distro has it). That directory is full of symbolic links to programs, and lets you have multiple versions of software on the same machine. For instance, there are different versions of Java out there, but with symlinks you can point the

generic java command on your machine for the particular version you want to us

hard links

There's another type of link, though, and it's considerably more powerful: a hard link. Whereas a symbolic link contains a filename for its target, being a separate file on the filesystem with a pointer inside, a hard link is an extra entry in the filesystem data.

If this is a bit hard to understand: imagine that myfile is a plain text file whose data begins at position 63813 on the hard drive. If you create a symbolic link to myfile, a whole new file is created with a shortcut inside that says "Hello, I actually point to myfile, so go over to that one".

Conversely, when you create a hard link, no new files are created.

Instead, the filesystem table is updated so that mylink also points to position 63813 on the drive. There aren't two separate files here; just two filenames that correspond to the exact same place on the drive.

In summary: symbolic links point to other files; hard links point to other data areas on the disk. In practice, this means they work quite differently to symbolic links. For instance, create a text file called foo with some text in and make a hard link to it like this:

In foo bar

Enter `rm foo` to remove the original file, and then `ls -l`. You'll notice that bar still seems to be OK, and if you `cat bar` the text contents are there. How can that be? Well, when you removed foo you just removed its entry from the filesystem table; you didn't wipe the data from the disk itself. That's because another entry in the filesystem table is also pointing to that data - bar.

Make sense? Hard links have limitations (they can't span across different filesystems) and are rarely used, so you don't have to wrack your brains with the technical underpinnings. But it's worth having an overview of what's going on. Symbolic links are far more commonly used in Linux installations, however, and being able to create them means you can remove a lot of fluff and duplication from your setups. You'll see many of them in your Linux travels, as lots of packages set them up.

Section 1: Restricting disk space with quotas

On a multi-user machine, you may have a scenario where you want to restrict disk space usage. A quick-hack way to do this would be to limit the size of the /home partition, but that's very inflexible. A better approach is to use disk quotas. Setting these up is a rather involving process and varies from distro to distro - so we'll explain the essentials so you know what to expect. You can then Google specific instructions for your distro.

Quotas can restrict disk usage for users or groups. You will need to edit */etc/fstab* - the file that controls how filesystems are mounted - to add *usrquota* and *grpquota* options for the relevant filesystem. You mount the filesystem in single-user mode, create *aquota.user* and *aquota.group* control files, and then add a *quotacheck* command to Cron, to periodically check if users are overrunning their allowance. Finally, *edquota* sets up the specific quotas for users. You can use *quotaon* and *quotaoff* to enable and disable quotas, and *repquota* to get a report of disk usage.

Where to go from here

Open up the quiz in Magazine/LPI_quiz and test your knowledge. Don't fret if you get anything wrong - there's a lot to take in, so just go back to the relevant tutorial and try the commands again.

When you're happy with your progress, visit www.lpi.org and look at the certification options. The LPI has various partners performing exams over the internet or in person. If you want to get a job in Linux, this is the way to go. Good luck, and let us know how you get on!