

LPI certification 101 exam prep, Part 4

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. About this tutorial	2
2. Linux filesystems	4
3. Booting the system	14
4. Runlevels	20
5. Filesystem quotas	23
6. System logs	29
7. Resources and feedback	33

Section 1. About this tutorial

What does this tutorial cover?

Welcome to "Advanced Administration", the last of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll bolster your knowledge of important Linux administration skills by covering a variety of topics including: Linux filesystems, the Linux boot process, runlevels, filesystem quotas, and system logs.

By the end of this *series* of tutorials (eight in all), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

The LPI logo is a trademark of Linux Professional Institute.

Should I take this tutorial?

This Advanced administration tutorial (Part 4) is ideal for those who want to learn about or improve their Linux filesystem-related skills. This tutorial is particularly appropriate for someone who may be serving as the primary sysadmin for the first time, since we cover a lot of low-level issues that all system administrators should know. For many, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of "rounding out" their important Linux system administration skills and preparing for the next LPI certification level.

If you are new to Linux, we recommend that you complete [Part 1: Linux fundamentals](#), [Part 2: Basic administration](#), and [Part 3: Intermediate administration](#) of this tutorial series before continuing.

About the authors

For technical questions about the content of this tutorial, contact the authors:

- * Daniel Robbins, at drobbins@gentoo.org
- * Chris Houser, at chouser@gentoo.org
- * Aron Griffis, at agriffis@gentoo.org

Daniel Robbins lives in Albuquerque, New Mexico, and is the President/CEO of [Gentoo Technologies, Inc.](#), the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending

time with his wife, Mary, and their daughter, Hadassah.

Chris Houser, known to many of his friends as "Chouser", has been a UNIX proponent since 1994 when joined the administration team for the computer science network at Taylor University in Indiana, where he earned his Bachelor's degree in Computer Science and Mathematics. Since then, he has gone on to work in Web application programming, user interface design, professional video software support, and now Tru64 UNIX device driver programming at [Compaq](#). He has also contributed to various free software projects, most recently to [Gentoo Linux](#). He lives with his wife and two cats in New Hampshire.

Aron Griffis graduated from Taylor University with a degree in Computer Science and an award that proclaimed him the "Future Founder of a Utopian UNIX Commune". Working towards that goal, Aron is employed by [Compaq](#) writing network drivers for Tru64 UNIX, and spending his spare time plunking out tunes on the piano or developing [Gentoo Linux](#). He lives with his wife Amy (also a UNIX engineer) in Nashua, NH.

Section 2. Linux filesystems

Block devices

In this section, we'll take a good look at Linux filesystems so that you're familiar with all the nitty-gritty details that an administrator needs to know. To begin, I'll introduce "block devices". The most famous block device is probably the one that represents the first IDE drive in a Linux system:

```
/dev/hda
```

If your system uses SCSI drives, then your first hard drive will be:

```
/dev/sda
```

The block devices above represent an *abstract* interface to the disk. User programs can use these block devices to interact with your disk without worrying about whether your drivers are IDE, SCSI or something else. The program can simply address the storage on the disk as a bunch of contiguous, randomly-accessible 512-byte blocks.

Whole disks and partitions

Under Linux, we create filesystems by using a special command called "mkfs", specifying a particular block device as a command-line argument.

However, although it is theoretically possible to use a "whole disk" block device (one that represents the *entire* disk) like `/dev/hda` or `/dev/sda` to house a single filesystem, this is almost never done in practice. Instead, full disk block devices are split up into smaller, more manageable block devices called "partitions". Partitions are created using a tool called **fdisk**, which is used to create and edit the partition table that's stored on each disk. The partition table defines exactly how to split up the full disk.

Introducing fdisk

We can take a look at a disk's partition table by running **fdisk**, specifying a block device that represents a full disk as an argument:

```
# fdisk /dev/hda
```

```
# fdisk /dev/sda
```

Note that you should *not* save or make any changes to a disk's partition table if any of its partitions contain filesystems that are in use or contain important data. Doing so will

generally cause data on the disk to be lost.

Using fdisk

Once in fdisk, you'll be greeted with a prompt that looks like this:

```
Command (m for help):
```

Type **p** to display your disk's current partition configuration:

```
Command (m for help): p
Disk /dev/hda: 240 heads, 63 sectors, 2184 cylinders
Units = cylinders of 15120 * 512 bytes
Device Boot      Start         End      Blocks   Id  System
/dev/hda1                1          14        105808+   83  Linux
/dev/hda2                15          49        264600    82  Linux swap
/dev/hda3                50          70        158760    83  Linux
/dev/hda4                71         2184    15981840    5  Extended
/dev/hda5                71         209        1050808+   83  Linux
/dev/hda6               210         348        1050808+   83  Linux
/dev/hda7               349         626        2101648+   83  Linux
/dev/hda8               627         904        2101648+   83  Linux
/dev/hda9               905        2184        9676768+   83  Linux
Command (m for help):
```

This particular disk is configured to house seven Linux filesystems (listed as "Linux") as well as a swap partition (listed as "Linux swap"). Notice the name of the corresponding partition block devices on the left side, starting with /dev/hda1 and going up to /dev/hda9. In the early days of the PC, partitioning software only allowed a maximum of four partitions (called "primary" partitions). This was too limiting, so a workaround called an *extended partitioning* was created. An extended partition is very similar to a primary partition, and counts towards the primary partition limit of four. However, extended partitions can hold any number of so-called *logical* partitions inside them, providing an effective means of working around the four partition limit.

Using fdisk, continued

```
Command (m for help): p
Disk /dev/hda: 240 heads, 63 sectors, 2184 cylinders
Units = cylinders of 15120 * 512 bytes
Device Boot      Start         End      Blocks   Id  System
/dev/hda1                1          14        105808+   83  Linux
/dev/hda2                15          49        264600    82  Linux swap
/dev/hda3                50          70        158760    83  Linux
/dev/hda4                71         2184    15981840    5  Extended
/dev/hda5                71         209        1050808+   83  Linux
/dev/hda6               210         348        1050808+   83  Linux
/dev/hda7               349         626        2101648+   83  Linux
/dev/hda8               627         904        2101648+   83  Linux
/dev/hda9               905        2184        9676768+   83  Linux
```

Command (m for help):

In our example, hda1 through hda3 are primary partitions. hda4 is an extended partition that contains logical partitions hda5 through hda9. So, in this example, you would never actually *use* /dev/hda4 for storing any filesystems directly -- it simply acts as a container for partitions hda5 through hda9. Also, notice that each partition has an "Id", also called a "partition type". Whenever you create a new partition, you should ensure that the partition type is set correctly. '83' is the correct partition type for partitions that will be housing Linux filesystems, and '82' is the correct partition type for Linux swap partitions. You set the partition type using the **t** option in fdisk. The Linux kernel uses the partition type setting to auto-detect filesystems and swap devices on the disk at boot-time.

Fdisk and beyond

There's more to fdisk than we have room to cover here, including the creation of new partitions (with the **n** command) and writing changes to disk (with the **w** command). Remember that you can type **m** for help. If you're new to **fdisk**, I recommend that you get the hang of the program by creating some partitions on spare disk where no data is at risk. Once you create your partitions and write them to disk, your new partition block devices are ready for use. In a bit, we'll use these new block devices to store new Linux filesystems.

For more information on partitioning, take a look at the following partitioning tips:

- * [Partition planning tips](#)
- * [Partitioning in action: consolidating data](#)
- * [Partitioning in action: moving /home](#)

Creating filesystems

Before a new block device can be used for storing files, we need to create a new filesystem on it. We do this by using a **mkfs** command -- the particular **mkfs** we use depends on the type of filesystem that we'd like to create. In this example, we use **mke2fs** to create an ext2 filesystem on /dev/hdc6, an empty and unused partition block device:

```
# mke2fs /dev/hdc6
mke2fs 1.25 (20-Sep-2001)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
1537088 inodes, 3072423 blocks
153621 blocks (5.00%) reserved for the super user
First data block=0
94 block groups
32768 blocks per group, 32768 fragments per group
16352 inodes per group
Superblock backups stored on blocks:
```

```
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 22 mounts or 180 days,
whichever comes first. Use tune2fs -c or -i to override.
```

Above, **mke2fs** created an empty ext2 filesystem on `/dev/hda6`.

Mounting filesystems

Once the filesystem is created, we can mount it using the **mount** command:

```
# mount /dev/hdc6 /mnt
```

To mount a filesystem, specify the partition block device as a first argument and a "mountpoint" as a second argument. The new filesystem will be "grafted in" at the mountpoint. This also has the effect of "hiding" any files that were in the `/mnt` directory on the parent filesystem. Later, when the filesystem is unmounted, these files will reappear. After executing the mount command, any files added to `/mnt` will be stored on the new ext2 filesystem.

Creating and using swap

If we had just created a partition that is intended to be used as a swap device, we would initialize it by using the **mkswap** command, specifying the partition block device as an argument:

```
# mkswap /dev/hdc6
```

Unlike regular filesystems, swap partitions aren't mounted. Instead, they are enabled using the **swapon** command:

```
# swapon /dev/hdc6
```

Normally, your Linux system's startup scripts will take care of automatically enabling your swap partitions. Therefore, the **swapon** command is generally only needed when you need to immediately add some swap that you just created. To view the swap devices currently enabled, type **cat /proc/swaps**.

Viewing mounted filesystems

To see what filesystems are mounted, type **mount** by itself:

```
# mount
```

```
/dev/ide/host0/bus1/target0/lun0/part7 on / type xfs (rw,noatime,nodiratime)
proc on /proc type proc (rw)
none on /dev type devfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/hdc6 on /mnt type ext2 (rw)
```

You can also view similar information by typing **cat /proc/mounts**. Since my Linux system uses devfs, the first line of my mount output lists a long path for the "root" partition block device. The "root" filesystem will get mounted automatically by the kernel at boot-time. Systems that use the new devfs device-management filesystem for /dev have longer official names for the partition and disk block devices than Linux used to have in the past. For example, /dev/ide/host0/bus1/target0/lun0/part7 is the official name for /dev/hdc7, and /dev/hdc7 itself is just a symlink pointing to the official block device. You can determine if your system is using devfs by checking to see if the /dev/.devfsd file exists; if so, then devfs is active.

Mount options

It's possible to customize various attributes of the to-be-mounted filesystem by specifying *mount options*. For example, you can mount a filesystem as "read-only" by using the "ro" option:

```
# mount /dev/hdc6 /mnt -o ro
```

With /dev/hdc6 mounted read-only, no files can be modified in /mnt -- only read. If your filesystem is already mounted "read/write" and you want to switch it to read-only mode, you can use the "remount" option to avoid having to unmount and remount the filesystem again:

```
# mount /mnt -o remount,ro
```

Notice that we didn't need to specify the partition block device because the filesystem is already mounted and **mount** knows that /mnt is associated with /dev/hdc6. To make the filesystem writeable again, we can remount it as read-write:

```
# mount /mnt -o remount,rw
```

Note that these remount commands will not complete successfully if any process has opened any files or directories in /mnt. To familiarize yourself with all the mount options available under Linux, type **man mount**.

Introducing fstab

So far, we've seen how to mount filesystems manually. In general, if you have a filesystem that you need to mount on a regular basis, manual mounting tends to be a bit cumbersome. And for essential filesystems, such as a separate /var filesystem, manual mounting is not possible.

These filesystems need to be mounted automatically at boot time, and we can tell the system to do just that by adding the appropriate entries to the `/etc/fstab` file. Even if you don't want the filesystem to be mounted automatically at boot, an `/etc/fstab` entry can make mounting easier, as we'll see in a bit.

A sample fstab

Let's take a look at a sample `/etc/fstab` file:

```
# <fs>                <mountpoint>    <type>          <opts>
<dump/pass>
/dev/hda1             /boot           ext2            noauto,noatime          1 2
/dev/hdc7             /               xfs             noatime,osyncisdsync,nodiratime 0 1
/dev/hdc5             none            swap            sw                       0 0
/dev/cdrom /mnt/cdrom iso9660          noauto,ro,user         0 0
# /proc should always be enabled
proc                 /proc           proc            defaults                  0 0
```

Above, each non-commented line in `/etc/fstab` specifies a partition block device, a mountpoint, a filesystem type, the filesystem options to use when mounting the filesystem, and two numeric fields. The first numeric field is used to tell the **dump** backup command the filesystems that should be backed up. Of course, if you are not planning to use **dump** on your system, then you can safely ignore this field. The last field is used by the **fsck** filesystem integrity checking program, and tells it the order in which your filesystems should be checked at boot. We'll touch on **fsck** again in a few panels.

Look at the `/dev/hda1` line; you'll see that `/dev/hda1` is an ext2 filesystem that should be mounted at the `/boot` mountpoint. Now, look at `/dev/hda1`'s mount options in the `<opts>` column. The **noauto** option tells the system to *not* mount `/dev/hda1` automatically at boot time; without this option, `/dev/hda1` would be automatically mounted to `/boot` at system boot time.

Also note the **noatime** option, which turns off the recording of **atime** (last access time) information on the disk. This information is generally not needed, and turning off atime updates has a positive effect on filesystem performance. You can also turn off directory atime updates by supplying the **nodiratime** mount option.

A sample fstab, continued

```
# <fs>                <mountpoint>    <type>          <opts>
<dump/pass>
/dev/hda1             /boot           ext2            noauto,noatime          1 1
/dev/hdc7             /               xfs             noatime,osyncisdsync,nodiratime 0 0
/dev/hdc5             none            swap            sw                       0 0
/dev/cdrom /mnt/cdrom iso9660          noauto,ro,user         0 0
# /proc should always be enabled
proc                 /proc           proc            defaults                  0 0
```

Now, take a look at the `/proc` line and notice the **defaults** option. Use **defaults** whenever you want a filesystem to be mounted with just the standard mount options. Since `/etc/fstab` has multiple fields, we can't simply leave the option field blank.

Also notice the `/etc/fstab` line for `/dev/hdc5`. This line defines `/dev/hdc5` as a swap device. Since swap devices aren't mounted like filesystems, **none** is specified in the mountpoint field. Thanks to this `/etc/fstab` entry, our `/dev/hdc5` swap device will be enabled automatically when the system starts up.

With an `/etc/fstab` entry for `/dev/cdrom` like the one above, mounting the CD-ROM drive becomes easier. Instead of typing:

```
# mount -t iso9660 /dev/cdrom /mnt/cdrom -o ro
```

We can now type:

```
# mount /dev/cdrom
```

In fact, using `/etc/fstab` allows us to take advantage of the **user** option. The **user** mount option tells the system to allow this particular filesystem to be mounted by any user. This comes in handy for removable media devices like CD-ROM drives. Without this `fstab` mount option, only the **root** user would be able to use the CD-ROM drive.

Unmounting filesystems

Generally, all mounted filesystems are unmounted automatically by the system when it is rebooted or shut down. When a filesystem is unmounted, any cached filesystem data in memory is flushed to the disk.

However, it's also possible to unmount filesystems manually. Before a filesystem can be unmounted, you first need to ensure that there are no processes running that have open files on the filesystem in question. Then, use the **umount** command, specifying *either* the device name or mount point as an argument:

```
# umount /mnt
```

or

```
# umount /dev/hdc6
```

Once unmounted, any files in `/mnt` that were "covered" by the previously-mounted filesystem will now reappear.

Introducing fsck

If your system crashes or locks up for some reason, the system won't have an opportunity to cleanly unmount your filesystems. When this happens, the filesystems are left in an inconsistent (unpredictable) state. When the system reboots, the **fsck** program will detect that the filesystems were not cleanly unmounted and will want to perform a consistency check of filesystems listed in `/etc/fstab`.

An important note -- for a filesystem to be checked by **fsck**, it must have a *non-zero* number in the "pass" field (the last field) in `/etc/fstab`. Typically, the root filesystem is set to a passno of **1**, specifying that it should be checked first. All other filesystems that should be checked at startup time should have a passno of **2** or higher.

Sometimes, you may find that after a reboot **fsck** is unable to fully repair a partially damaged filesystem. In these instances, all you need to do is to bring your system down to single-user mode and run **fsck** manually, supplying the partition block device as an argument. As **fsck** performs its filesystem repairs, it may ask you whether to fix particular filesystem defects. In general, you should say **y** (yes) to all these questions and allow **fsck** to do its thing.

Problems with fsck

One of the problems with **fsck** scans is that they can take quite a while to complete, since the entirety of a filesystem's *metadata* (internal data structure) needs to be scanned in order to ensure that it's consistent. With extremely large filesystems, it is not unusual for an exhaustive **fsck** to take more than an hour.

In order to solve this problem, a new type of filesystem was designed, called a *journaling filesystem*. Journaling filesystems record an on-disk log of recent changes to the filesystem metadata. In the event of a crash, the filesystem driver inspects the log. Because the log contains an accurate account of recent changes on disk, only these parts of the filesystem metadata need to be checked for errors. Thanks to this important design difference, checking a journalled filesystem for consistency typically takes just a matter of seconds, regardless of filesystem size. For this reason, journaling filesystems are gaining popularity in the Linux community. For more information on journaling filesystems, see the [Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS](#).

Now, let's take a look at the various filesystems available for Linux.

The ext2 filesystem

The ext2 filesystem has been the standard Linux filesystem for many years. It has generally good performance for most applications, but it does not offer any journaling capability. This makes it unsuitable for very large filesystems, since fscks can take an extremely long time. In addition, ext2 has some built-in limitations due to the fact that every ext2 filesystem has a fixed number of inodes that it can hold. That being said, ext2 is generally considered to be an

extremely robust and efficient non-journalled filesystem.

- * In kernels: 2.0+
- * journaling: no
- * mkfs command: mke2fs
- * mkfs example: mke2fs /dev/hdc7
- * related commands: debugfs, tune2fs, chattr
- * performance-related mount options: noatime, nodiratime

The ext3 filesystem

The ext3 filesystem uses the same on-disk format as ext2, but adds journaling capabilities. In fact, of all the Linux filesystems, ext3 has the most extensive journaling support, supporting not only metadata journaling but also ordered journaling (the default) and full metadata+data journaling. These "special" journaling modes help to ensure data integrity, not just short fscks like other journaling implementations. For this reason, ext3 is the best filesystem to use if data integrity is an absolute first priority. However, these data integrity features do impact performance somewhat. In addition, because ext3 uses the same on-disk format as ext2, it still suffers from the same scalability limitations as its non-journalled cousin. Ext3 is a good choice if you're looking for a good general-purpose journalled filesystem that is also very robust.

- * In kernels: 2.4.16+
- * journaling: metadata, ordered data writes, full metadata+data
- * mkfs command: mke2fs -j
- * mkfs example: mke2fs -j /dev/hdc7
- * related commands: debugfs, tune2fs, chattr
- * performance-related mount options: noatime, nodiratime
- * other mount options:
 - * data=writeback (disable journaling)
 - * data=ordered (the default, metadata journaling and data is written out to disk with metadata)
 - * data=journal (full data journaling for data *and* metadata integrity. Halves write performance.)
- * ext3 resources:
 - * [Andrew Morton's ext3 page](#)
 - * [Andrew Morton's excellent ext3 usage documentation \(recommended\)](#)
 - * [Advanced filesystem implementor's guide, part 7: Introducing ext3](#)
 - * [Advanced filesystem implementor's guide, part 8: Surprises in ext3](#)

The ReiserFS filesystem

ReiserFS is a relatively new filesystem that has been designed with the goal of providing very good small file performance, very good general performance and being very scalable. ReiserFS uses a metadata journal to avoid long fscks, but the journal implementation allows recently-modified data to become corrupted in the event of a system lockup. In general, ReiserFS offers very good performance, but may exhibit certain performance quirks under

specific kinds of filesystem loads. In addition, ReiserFS's **fsck** tool is in its infancy so you may have difficulty recovering data from a corrupted filesystem. A number of these issues are due to the fact that ReiserFS is a relatively new, still evolving filesystem. ReiserFS is preferred by many for its speed and scalability.

- * In kernels: 2.4.0+ (2.4.16+ recommended)
- * journaling: metadata
- * mkfs command: mkreiserfs
- * mkfs example: mkreiserfs /dev/hdc7
- * performance-related mount options: noatime, nodiratime, notail
- * ReiserFS Resources:
 - * [The home of ReiserFS](#)
 - * [Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS](#)
 - * [Advanced filesystem implementor's guide, Part 2: Using ReiserFS and Linux 2.4](#)

The XFS filesystem

The XFS filesystem is an enterprise-class journaling filesystem being ported to Linux by [SGI](#). XFS isn't in the stock kernel yet, but more information on XFS can be found at <http://oss.sgi.com/projects/xfs>. For an introduction to XFS, read [Advanced filesystem implementor's guide, Part 9: Introducing XFS](#).

The JFS filesystem

JFS is a high-performance journaling filesystem being ported to Linux by IBM. JFS is used by IBM enterprise servers and is designed for high-performance applications. JFS isn't in the stock kernel yet. You can learn more about JFS at [the JFS project Web site](#).

VFAT

The VFAT filesystem isn't really a filesystem that you would choose for storing Linux files. Instead, it's a DOS-compatible filesystem driver that allows you to mount and exchange data with DOS and Windows FAT-based filesystems. The VFAT filesystem driver is present in the standard Linux kernel.

network adapters, sound cards, and possibly others will each in turn report their status.

/sbin/init

When the kernel finishes loading, it starts a program called **init**. This program remains running until the system is shut down. It is always assigned process ID 1, as you can see:

```
$ ps --pid 1
  PID TTY          TIME CMD
   1  ?            00:00:04 init.system
```

The **init** program boots the rest of your distribution by running a series of scripts. These scripts typically live in `/etc/rc.d/init.d` or `/etc/init.d`, and they perform services such as setting the system's hostname, checking the filesystem for errors, mounting additional filesystems, enabling networking, starting print services, etc. When the scripts complete, **init** starts a program called **getty** which displays the login prompt, and you're good to go!

Digging in: LILO

Now that we've taken a quick tour through the booting process, let's look more closely at the first part: the MBR and loading the kernel. The maintenance of the MBR is the responsibility of the "boot loader". The two most popular boot loaders for x86-based Linux are "LILO" (Linux LOader) and "GRUB" (GRand Unified Bootloader).

Of the two, LILO is the older and more common boot loader. LILO's presence on your system is reported at boot, with the short "LILO boot:" prompt. Note that you may need to hold down the shift key during boot to get the prompt, since often a system is configured to whiz straight through without stopping.

There's not much fanfare at the LILO prompt, but if you press the <tab> key, you'll be presented with a list of potential kernels (or other operating systems) to boot. Often there's only one in the list. You can boot one of them by typing it and pressing <enter>. Alternatively you can simply press <enter> and the first item on the list will boot by default.

Using LILO

Occasionally you may want to pass an option to the kernel at boot time. Some of the more common options are **root=** to specify an alternative root filesystem, **init=** to specify an alternative **init** program (such as **init=/bin/sh** to rescue a misconfigured system), and **mem=** to specify the amount of memory in the system (for example **mem=512M** in the case that Linux only autodetects 128M). You could pass these to the kernel at the LILO boot prompt:

```
LILO boot: linux root=/dev/hdb2 init=/bin/sh mem=512M
```

If you need to regularly specify command-line options, you might consider adding them to `/etc/lilo.conf`. The format of that file is described in the `lilo.conf(5)` man page.

An important LILO gotcha

Before moving on to GRUB, there is an important gotcha to LILO. Whenever you make changes to `/etc/lilo.conf`, or whenever you install a new kernel, you *must* run **lilo**. The **lilo** program rewrites the MBR to reflect the changes you made, including recording the absolute disk location of the kernel. The example here makes use of the `-v` flag for verbosity:

```
# lilo -v
LILO version 21.4-4, Copyright (C) 1992-1998 Werner Almesberger
'lba32' extensions Copyright (C) 1999,2000 John Coffman
Reading boot sector from /dev/hda
Merging with /boot/boot.b
Mapping message file /boot/message
Boot image: /boot/vmlinuz-2.2.16-22
Added linux *
/boot/boot.0300 exists - no backup copy made.
Writing boot sector.
```

Digging in: GRUB

The GRUB boot loader could be considered the next generation of boot loader, following LILO. Most visibly to users, it provides a menu interface instead of LILO's primitive prompt. For system administrators, the changes are more significant. GRUB supports more operating systems than LILO, provides some password-based security in the boot menu, and is easier to administrate.

GRUB is usually installed with the **grub-install** command. Once installed, GRUB's menu is administrated by editing the file `/boot/grub/menu.lst`. Both of these tasks are beyond the scope of this document; you should read the GRUB info pages before attempting to install or administrate GRUB.

Using GRUB

To give parameters to the kernel, you can press **e** at the boot menu. This provides you with the opportunity to edit (by again pressing **e**) either the name of the kernel to load or the parameters passed to it. When you're finished editing, press `<enter>` then **b** to boot with your changes.

A significant difference between LILO and GRUB that bears mentioning is that GRUB does not need to re-install its boot loader each time that the configuration changes, or a new kernel is installed. This is because GRUB understands the Linux filesystem, whereas LILO just stores the absolute disk location of the kernel to load. This single fact about GRUB alleviates the

frustration system administrators feel when they forget to type **lilo** after installing a new kernel!

dmesg

The boot messages from the kernel and init scripts typically scroll by quickly. You might notice an error, but it's gone before you can properly read it. In that case, there are two places you can look after the system boots to see what went wrong (and hopefully get an idea how to fix it).

If the error occurred while the kernel was loading or probing hardware devices, you can retrieve a copy of the kernel's log using the **dmesg** command:

```
# dmesg | head -1
Linux version 2.4.16 (root@time.flatmonk.org) (gcc version 2.95.3 20010315 (release)) #1
```

Hey, we recognize that line! It's the first line the kernel prints when it loads. Indeed, if you pipe the output of **dmesg** into a pager, you can view all of the messages the kernel printed on boot, plus any messages the kernel has printed to the console in the meantime.

/var/log/messages

The second place to look for information is in the `/var/log/messages` file. This file is recorded by the `syslog` daemon, which accepts input from libraries, daemons, and the kernel. Each line in the messages file is timestamped. This file is a good place to look for errors that occurred during the init scripts stage of booting. For example, to see the last few messages from the `nameserver`:

```
# grep named /var/log/messages | tail -3
Jan 12 20:17:41 time /usr/sbin/named[350]: listening on IPv4 interface lo, 127.0.0.1#53
Jan 12 20:17:41 time /usr/sbin/named[350]: listening on IPv4 interface eth0, 10.0.0.1#53
Jan 12 20:17:41 time /usr/sbin/named[350]: running
```

Single-user mode

We know that it's possible to pass parameters to the kernel when it boots. One of the most often used parameters is **s** which causes the system to start in "single-user" mode. This mode usually mounts only the root filesystem, starts a minimal subset of the init scripts, and starts a shell rather than providing a login prompt. Additionally, networking is not configured, so there is no chance of external factors affecting your work.

Using single-user mode

So what "work" can be done with the system in such a state? To answer this question, we

have to realize a vast difference between Linux and Windows. Windows is designed to normally be used by one person at a time, sitting at the console. It is effectively always in "single-user" mode. Linux, on the other hand, is used more often to serve network applications, or provide shell or X sessions to remote users on the network. These additional variables are not desirable when you want to perform maintenance operations such as restoring from backup, creating or modifying filesystems, upgrading the system from CD, etc. In these cases you should use single-user mode.

Changing runlevels

In fact, it's not actually necessary to reboot in order to reach single-user mode. The **init** program manages the current mode, or "runlevel", for the system. The standard runlevels for a Linux system are labeled and defined as follows:

- * **0**: Halt the computer
- * **1 or s**: Single-user mode
- * **2**: Multi-user, no network
- * **3**: Multi-user, text console
- * **4**: Multi-user, graphical console
- * **5**: Same as 4
- * **6**: Reboot the computer

These runlevels vary between distributions, so be sure to consult your distro's documentation. To change to single-user mode, use the **telinit** command, which instructs init to change runlevels:

```
# telinit 1
```

You can see from the table above that you can also shutdown or reboot the system in this manner. **telinit 0** will halt the computer; **telinit 6** will reboot the computer. When you issue the **telinit** command to change runlevels, a subset of the init scripts will run to either shutdown or startup system services.

Shutting down gracefully

However, note that this is rather rude if there are users on the system at the time (who may be quite angry with you). The **shutdown** command provides a method for changing runlevels in a way that treats users reasonably. Similarly to the **kill** command's ability to send a variety of signals to a process, **shutdown** can be used to halt, reboot, or change to single-user mode. For example, to change to single-user mode in 5 minutes:

```
# shutdown 5
Broadcast message from root (pts/2) (Tue Jan 15 19:40:02 2002):
The system is going DOWN to maintenance mode in 5 minutes!
```

If you press control-c at this point, you can cancel the pending switch to single-user mode. The message above would appear on all terminals on the system, so users have a reasonable amount of time to save their work and log off. (Some might argue whether or not 5 minutes is "reasonable.")

Shutting down immediately

If you're the only person on the system, you can use "now" instead of an argument in minutes. For example, to reboot the system right now:

```
# shutdown -r now
```

No chance to hit control-c in this case; the system is already on its way down. Finally, the **-h** option halts the system:

```
# shutdown -h 1
Broadcast message from root (pts/2) (Tue Jan 15 19:50:58 2002):
The system is going DOWN for system halt in 1 minute!
```

The default runlevel

You've probably gathered at this point that the **init** program is quite important on a Linux system. You can configure init by editing the file `/etc/inittab`, which is described in the *inittab(5)* man page. We'll just touch on one key line in this file:

```
# grep ^id: /etc/inittab
id:3:initdefault:
```

On my system, runlevel 3 is the default runlevel. It can be useful to change this value if you prefer your system to boot immediately into a graphical login (usually runlevel 4 or 5). To do so, simply edit the file and change the value on that line. But be careful! If you change it to something invalid, you'll probably have to employ the **init=/bin/sh** trick we mentioned earlier.

Additional information

Additional information related to this section can be found at:

- * IBM developerWorks' [Getting to know GRUB tutorial](#)
- * [LILO Mini-HOWTO](#)
- * [GRUB home](#)
- * Kernel command-line options in `/usr/src/linux/Documentation/kernel-parameters.txt`
- * [Sysvinit docs at Redhat](#)

Section 4. Runlevels

Single-user mode

Recall from the section regarding boot loaders that it's possible to pass parameters to the kernel when it boots. One of the most often used parameters is **s**, which causes the system to start in "single-user" mode. This mode usually mounts only the root filesystem, starts a minimal subset of the init scripts, and starts a shell rather than providing a login prompt. Additionally, networking is not configured, so there is no chance of external factors affecting your work.

Single-user mode, continued

So what "work" can be done with the system in such a state? To answer this question, we have to realize a vast difference between Linux and Windows. Windows is designed to normally be used by one person at a time, sitting at the console. It is effectively always in "single-user" mode. Linux, on the other hand, is used more often to serve network applications, or provide shell or X sessions to remote users on the network. These additional variables are not desirable when you want to perform maintenance operations such as restoring from backup, creating or modifying filesystems, upgrading the system from CD, etc. In these cases you should use single-user mode.

Runlevels

In fact, it's not actually necessary to reboot in order to reach single-user mode. The **init** program manages the current mode, or "runlevel", for the system. The standard runlevels for a Linux system are labeled and defined as follows:

- * **0**: Halt the computer
- * **1 or s**: Single-user mode
- * **2**: Multi-user, no network
- * **3**: Multi-user, text console
- * **4**: Multi-user, graphical console
- * **5**: Same as 4
- * **6**: Reboot the computer

These runlevels vary between distributions, so be sure to consult your distro's documentation.

telinit

To change to single-user mode, use the **telinit** command, which instructs **init** to change runlevels:

```
# telinit 1
```

You can see from the table above that you can also shutdown or reboot the system in this manner. **telinit 0** will halt the computer; **telinit 6** will reboot the computer. When you issue the **telinit** command to change runlevels, a subset of the init scripts will run to either shutdown or startup system services.

Runlevel etiquette

However, note that this is rather rude if there are users on the system at the time (who may be quite angry with you). The **shutdown** command provides a method for changing runlevels in a way that treats users reasonably. Similarly to the **kill** command's ability to send a variety of signals to a process, **shutdown** can be used to halt, reboot, or change to single-user mode. For example, to change to single-user mode in 5 minutes:

```
# shutdown 5
Broadcast message from root (pts/2) (Tue Jan 15 19:40:02 2002):
The system is going DOWN to maintenance mode in 5 minutes!
```

If you press control-c at this point, you can cancel the pending switch to single-user mode. The message above would appear on all terminals on the system, so users have a reasonable amount of time to save their work and log off. (Some might argue whether or not 5 minutes is "reasonable.")

"Now" and halt

If you're the only person on the system, you can use **now** instead of an argument in minutes. For example, to reboot the system right now:

```
# shutdown -r now
```

No chance to hit control-c in this case; the system is already on its way down. Finally, the **-h** option halts the system:

```
# shutdown -h 1
Broadcast message from root (pts/2) (Tue Jan 15 19:50:58 2002):
The system is going DOWN for system halt in 1 minute!
```

Configuring init

You've probably gathered at this point that the **init** program is quite important on a Linux system. You can configure init by editing the file `/etc/inittab`, which is described in the **inittab(5)** man page. We'll just touch on one key line in this file:

```
# grep ^id: /etc/inittab
id:3:initdefault:
```

On my system, runlevel 3 is the default runlevel. It can be useful to change this value if you prefer your system to boot immediately into a graphical login (usually runlevel 4 or 5). To do so, simply edit the file and change the value on that line. But be careful! If you change it to something invalid, you'll probably have to employ the **init=/bin/sh** trick we mentioned earlier.

Section 5. Filesystem quotas

Introducing quotas

Quotas are a feature of Linux that let you track disk usage by user or by group. They're useful for preventing any single user or group from using an unfair portion of a filesystem, or from filling it up altogether. Quotas can only be enabled and managed by the root user. In this section, I'll describe how to set up quotas on your Linux system and manage them effectively.

Kernel support

Quotas are a feature of the filesystem; therefore, they require kernel support. The first thing you'll need to do is verify that you have quota support in your kernel. You can do this using `grep`:

```
# cd /usr/src/linux
# grep -i quota .config
CONFIG_QUOTA=y
CONFIG_XFS_QUOTA=y
```

If this command returns something less conclusive (such as **CONFIG_QUOTA is not set**) then you should rebuild your kernel to include quota support. This is not a difficult process, but is outside of the scope of this section of the tutorial. If you're unfamiliar with the steps to build and install a new kernel, you might consider referencing [this tutorial](#).

Filesystem support

Before diving into the administration of quotas, please note that quota support on Linux as of the 2.4.x kernel series is not complete. There are currently problems with quotas in the ext2 and ext3 filesystems, and ReiserFS does not appear to support quotas at all. This tutorial bases its examples on XFS, which [seems to properly support quotas](#).

Configuring quotas

To begin configuring quotas on your system, you should edit `/etc/fstab` to mount the affected filesystems with quotas enabled. For our example, we use an XFS filesystem mounted with user and group quotas enabled:

```
# grep quota /etc/fstab
/usr/users /mnt/hdcl xfs usrquota,grpquota,noauto 0 0
# mount /usr/users
```

Configuring quotas, continued

Note that the `usrquota` and `grpquota` options don't necessarily enable quotas on a filesystem. You can make sure quotas are enabled using the **quotaon** command:

```
# quotaon /usr/users
```

There is a corresponding **quotaoff** command should you desire to disable quotas in the future:

```
# quotaoff /usr/users
```

But for the moment, if you're trying some of the examples in this tutorial, be sure to have quotas enabled.

The quota command

The **quota** command displays a user's disk usage and limits for all of the filesystems currently mounted. The **-v** option includes in the list filesystems where quotas are enabled, but no storage is currently allocated to the user.

```
# quota -v
Disk quotas for user root (uid 0):
Filesystem blocks  quota  limit  grace  files  quota  limit  grace
/dev/hdc1    0      0      0      3      0      0
```

The first column, `blocks`, shows how much disk space the root user is currently using on each filesystem listed. The following columns, `quota` and `limit`, refer to the limits currently in place for disk space. We will explain the difference between `quota` and `limit`, and the meaning of the `grace` column later on. The `files` column shows how many files the root user owns on the particular filesystem. The following `quota` and `limit` columns refer to the limits for files.

Viewing quota

Any user can use the **quota** command to view their own quota report as shown in the previous example. However only the root user can look at the quotas for other users and groups. For example, say we have a filesystem, `/dev/hdc1` mounted on `/usr/users`, with two users: **jane** and **john**. First, let's look at jane's disk usage and limits.

```
# quota -v jane
Disk quotas for user jane (uid 1003):
Filesystem blocks  quota  limit  grace  files  quota  limit  grace
/dev/hdc1   4100    0      0      6      0      0
```

In this example, we see that **jane**'s quotas are set to zero, which indicates no limit.

edquota

Now let's say we want to give the user **jane** a quota. We do this with the **edquota** command. Before we start editing quotas, let's see how much space we have available on `/usr/users`:

```
# df /usr/users
Filesystem          1k-blocks      Used Available Use% Mounted on
/dev/hdc1            610048         4276    605772    1% /usr/users
```

This isn't a particularly large filesystem, only 600M or so. It seems prudent to give **jane** a quota so that she can't use more than her fair share. When you run **edquota**, a temporary file is created for each user or group you specify on the command line.

edquota, continued

The **edquota** command puts you in an editor, which enables you to add and/or modify quotas via this temporary file.

```
# edquota jane
Disk quotas for user jane (uid 1003):
Filesystem      blocks      soft      hard      inodes      soft      hard
/dev/hdc1       4100         0         0         6           0         0
```

Similar to the output from the `quota` command above, the `blocks` and `inodes` columns in this temporary file refer to the disk space and number of files `jane` is currently using. You cannot modify the number of blocks or inodes; any attempt to do so will be summarily discarded by the system. The `soft` and `hard` columns show `jane`'s quota, which we can see is currently unlimited (zero indicates no quota).

Understanding edquota

The `soft` limit is the maximum amount of disk usage that `jane` has allocated to her on the filesystem (in other words, her quota). If `jane` uses more disk space than is allocated in her `soft` limit, she will be issued warnings about her quota violation via e-mail. The `hard` limit indicates the *absolute* limit on disk usage, which a user can't exceed. If `jane` tries to use more disk space than is specified in the `hard` limit, she will get a "Disk quota exceeded" error and will not be able to complete the operation.

Making changes

So here we change `jane`'s `soft` and `hard` limits and save the file:

```
Disk quotas for user jane (uid 1003):
```

```
Filesystem      blocks      soft      hard      inodes      soft      hard
/dev/hdc1       4100       10000    11500     6           2000     2500
```

Running the quota command, we can inspect our modifications:

```
# quota jane
Disk quotas for user jane (uid 1003):
Filesystem blocks  quota  limit  grace  files  quota  limit  grace
/dev/hdc1  4100  10000  11500  6      2000  2500
```

Copying quotas

You'll remember that we also have another user john on this filesystem. If we want to give john the same quota as jane, we can use the **-p** option to edquota, which uses jane's quotas as a prototype for all following users on the command line. This is an easy way to set up quotas for groups of users.

```
# edquota -p jane john
# quota john
Disk quotas for user john (uid 1003):
Filesystem blocks  quota  limit  grace  files  quota  limit  grace
/dev/hdc1    0     10000  11500  1      2000  2500
```

Group restrictions

We can also use edquota to restrict the allocation of disk space based on the group ownership of files. For example, to edit the quotas for the users group:

```
# edquota -g users Disk quotas for group users (gid 100): Filesystem blocks soft hard inodes
soft hard /dev/hdc1 4100 500000 510000 7 100000 125000
```

Then to view the modified quotas for the users group:

```
# quota -g users Disk quotas for group users (gid 100): Filesystem blocks quota limit grace
files quota limit grace /dev/hdc1 4100 500000 510000 7 100000 125000
```

The repquota command

Looking at each users' quotas using the quota command can be tedious if you have many users on a filesystem. The **repquota** command summarizes the quotas for a filesystem into a nice report. For example, to see the quotas for all users and groups on /usr/users:

```
# repquota -ug /usr/users
*** Report for user quotas on device /dev/hdc1
Block grace time: 7days; Inode grace time: 7days
User          used      Block limits          File limits
              used      soft   hard  grace  used      soft   hard  grace
```

```

-----
root      --      0      0      0      3      0      0
john      --      0     10000    11500    1     2000    2500
jane      --     4100    10000    11500    6     2000    2500
*** Report for group quotas on device /dev/hdc1
Block grace time: 7days; Inode grace time: 7days
Group          used      Block limits      File limits
              used      soft  hard  grace  used  soft  hard  grace
-----
root      --      0      0      0      3      0      0
users     --     4100  500000  510000  7     100000  125000

```

Repquota options

There are a couple other options to repquota that are worth mentioning. **repquota -a** will report on all currently-mounted read-write filesystems that have quotas enabled. **repquota -n** will not resolve uids and gids to names. This can speed up the output for large lists.

Monitoring quotas

If you are a system administrator, you will want to have a way to monitor quotas to ensure that they are not being exceeded. An easy way to do this is to use **warnquota**. The **warnquota** command sends e-mail to users who have exceeded their soft limit. Typically **warnquota** is run as a cron-job.

When a user exceeds their soft limit, the grace column in the output from the **quota** command will indicate the grace period -- how long before the soft limit is enforced for that filesystem.

```

Disk quotas for user jane (uid 1003):
Filesystem  blocks  quota  limit  grace  files  quota  limit  grace
/dev/hdc1   10800* 10000  11500  7days    7     2000  2500

```

By default, the grace period for blocks and inodes is 7 days.

Modifying the grace period

You can modify the grace period for filesystems using equota:

```
# edquota -t
```

This puts you in an editor of a temporary file that looks like this:

```

Grace period before enforcing soft limits for users:
Time units may be: days, hours, minutes, or seconds
Filesystem          Block grace period      Inode grace period

```

`/dev/hdc1``7days``7days`

The text in the file is nicely explanatory. Be sure to leave your users enough time to receive their warning e-mail and find some files to delete!

Checking quotas on boot

You may also want to check quotas on boot. You can do this using a script to run the **quotacheck** command; there is an example script in the [Quota Mini HOWTO](#). The **quotacheck** command also has the ability to repair damaged quota files; familiarize yourself with it by reading the `quotacheck(8)` man page.

Also remember what I mentioned previously regarding **quotaon** and **quotaoff**. You should incorporate **quotaon** into your boot script so that quotas are enabled. To enable quotas on all filesystems where quotas are supported, use the **-a** option:

```
# quotaon -a
```

Section 6. System logs

Introducing syslogd

The syslog daemon provides a mature client-server mechanism for logging messages from programs running on the system. Syslog receives a message from a daemon or program, categorizes the message by priority and type, then logs it according to administrator-configurable rules. The result is a robust and unified approach to managing logs.

Reading logs

Let's jump right in and look at the contents of a syslog-recorded log file. Afterward we can come back to syslog configuration. The FHS (see [Part 2](#) of this tutorial series) mandates that log files be placed in `/var/log`. Here we use the `tail` command to display the last 10 lines in the "messages" file:

```
# cd /var/log
# tail messages
Jan 12 20:17:39 bilbo init: Entering runlevel: 3
Jan 12 20:17:40 bilbo /usr/sbin/named[337]: starting BIND 9.1.3
Jan 12 20:17:40 bilbo /usr/sbin/named[337]: using 1 CPU
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: loading configuration from '/etc/bind/named.conf'
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: no IPv6 interfaces found
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: listening on IPv4 interface lo, 127.0.0.1#53
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: listening on IPv4 interface eth0, 10.0.0.1#53
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: running
Jan 12 20:41:58 bilbo gnome-name-server[11288]: starting
Jan 12 20:41:58 bilbo gnome-name-server[11288]: name server starting
```

You hopefully remember from the text-processing whirlwind that the **tail** command displays the last lines in a file. In this case, we can see that the nameserver **named** was recently started on this system, which is named **bilbo**. If we were deploying IPv6, we might notice that **named** found no IPv6 interfaces, indicating a potential problem. Additionally, we can see that a user may have recently started **GNOME**, indicated by the presence of **gnome-name-server**.

Tailing log files

An experienced system administrator might use **tail -f** to follow the output to a log file as it occurs:

```
# tail -f /var/log/messages
```

For example, in the case of debugging our theoretical IPv6 problem, running the above command in one terminal while stopping and starting **named** would immediately display the messages from that daemon. This can be a useful technique when debugging. Some administrators even like to keep a constantly running **tail -f messages** in a terminal where they

can keep an eye on system events.

Grepping logs

Another useful technique is to search a log file using the **grep** utility, described in [Part 2](#) of this tutorial series. In the above case, we might use **grep** to find where "named" behavior has changed:

```
# grep named /var/log/messages
```

Log overview

The following summarizes the log files typically found in /var/log and maintained by syslog:

- * **messages**: Informational and error messages from general system programs and daemons
- * **secure**: Authentication messages and errors, kept separate from "messages" for extra security
- * **maillog**: Mail-related messages and errors
- * **cron**: Cron-related messages and errors
- * **spooler**: UUCP and news-related messages and errors

syslog.conf

As a matter of fact, now would be a good time to investigate the syslog configuration file, /etc/syslog.conf. (Note: If you don't have syslog.conf, keep reading for the sake of information, but you may be using an alternative syslog daemon.) Browsing that file, we see there are entries for each of the common log files mentioned above, plus possibly some other entries. The file has the format **facility.priority action**, where those fields are defined as follows:

syslog.conf, continued

facility

Specifies the subsystem that produced the message. The valid keywords for facility are auth, authpriv, cron, daemon, kern, lpr, mail, news, syslog, user, uucp, and local0 through local7.

priority

Specifies the minimum severity of the message, meaning that messages of this priority and higher will be matched by this rule. The valid keywords for priority are debug, info, notice, warning, err, crit, alert, and emerg.

action

The action field should be either a filename, tty (such as /dev/console), remote machine prefixed by @ , comma-separated list of users, or * to send the message to everybody logged on. The most common action is a simple filename.

Reloading and additional information

Hopefully this overview of the configuration file helps you to get a feel for the strength of the syslog system. You should read the **syslog.conf(5)** man page for more information prior to making changes. Additionally the **syslogd(8)** man page supplies lots more detailed information.

Note that you need to inform the syslog daemon of changes to the configuration file before they are put into effect. Sending it a SIGHUP is the right method, and you can use the **killall** command to do this easily:

```
# killall -HUP syslogd
```

A security note

You should beware that the log files written to by syslogd will be created by the program if they don't exist. Regardless of your current umask setting, the files will be created world-readable. If you're concerned about the security, you should chmod the files to be read-write by root only. Additionally, the **logrotate** program (described below) can be configured to create new log files with the appropriate permissions. The syslog daemon always preserves the current attributes of an existing log file, so you don't need to worry about it once the file is created.

logrotate

The log files in /var/log will grow over time, and potentially could fill the filesystem. It is advisable to employ a program such as **logrotate** to manage the automatic archiving of the logs. The **logrotate** program usually runs as a daily cron job, and can be configured to rotate, compress, remove, or mail the log files.

For example, a default configuration of logrotate might rotate the logs weekly, keeping 4 weeks worth of backlogs (by appending a sequence number to the filename), and compress the backlogs to save space. Additionally, the program can be configured to deliver a **SIGHUP** to **syslogd** so that the daemon will notice the now-empty log files and append to them appropriately.

For more information on **logrotate**, see the **logrotate(8)** man page, which contains a description of the program and the syntax of the configuration file.

Advanced topic -- klogd

Before moving away from syslog, I'd like to note a couple of advanced topics for ambitious readers. These tips may save you some grief when trying to understand syslog-related topics.

First, the syslog daemon is actually part of the `sysklogd` package, which contains a second daemon called **klogd**. It's **klogd**'s job to receive information and error messages from the kernel, and pass them on to `syslogd` for categorization and logging. The messages received by `klogd` are exactly the same as those you can retrieve using the **dmesg** command. The difference is that `dmesg` prints the current contents of a ring buffer in the kernel, whereas `klogd` is passing the messages to `syslogd` so that they won't be lost when the ring wraps around.

Advanced topic -- alternate loggers

Second, there are alternatives to the standard `sysklogd` package. The alternatives attempt to be more efficient, easier to configure, and possibly more feature-rich than `sysklogd`. [Syslog-ng](#) and [Metalog](#) seem to be some of the more popular alternatives; you might investigate them if you find `sysklogd` doesn't provide the level of power you need.

Third, you can log messages in your scripts using the **logger** command. See the **logger(1)** man page for more information.

Section 7. Resources and feedback

We're done... almost!

Congratulations, you've reached the end of this tutorial! Well, almost. We were unable to include a few topics in our first four tutorials due to space limitations. Fortunately, we have a couple of good resources that will help you get up to speed on these topics in no time. Be sure to cover these particular tutorials if you are planning to get your LPIC level 1 certification.

On the important topic of system backups, we refer you to an IBM developerWorks Linux zone tutorial on the subject called [Backing up your Linux machines](#). In this tutorial, you'll learn how to back up Linux systems using a **tar** variant called **star**. You'll also learn how to use the **mt** command to control tape functions.

The second topic that we weren't quite able to fit in was periodic scheduling. Fortunately, there's a good [cron and at tutorial over at thelinuxgurus.org](#). **cron** and **at** are used to schedule jobs to be executed at a specific time, and are important knowledge for any system administrator.

In the next panel, you'll find a number of resources that you will find helpful in learning more about the subjects presented in this tutorial.

Resources

To find out more about quota support under Linux, be sure to check out the [Linux Quota mini-HOWTO](#). Also be sure to consult the `quota(1)`, `edquota(8)`, `repquota(8)`, `quotacheck(8)`, and `quotaon(8)` man pages on your system.

Additional information to the system boot process and boot loaders can be found at:

- * The IBM developerWorks Linux zone's [Getting to know GRUB tutorial](#)
- * [LILO Mini-HOWTO](#)
- * [GRUB home](#)
- * Kernel command-line options in `/usr/src/linux/Documentation/kernel-parameters.txt`
- * [Sysvinit docs at Redhat](#)

To learn more about Linux filesystems, read the multi-part advanced filesystem implementor's guide on the IBM developerWorks Linux zone, covering:

- * [The benefits of journalling and ReiserFS](#) (Part 1)
- * [Setting up a ReiserFS system](#) (Part 2)
- * [Using the tmpfs virtual memory filesystem and bind mounts](#) (Part 3)
- * [The benefits of devfs, the device management filesystem](#) (Part 4)
- * [Beginning the conversion to devfs](#) (Part 5)
- * [Completing the conversion to devfs using an init wrapper](#) (Part 6)
- * [The benefits of the ext3 filesystem](#) (Part 7)

- * [An in-depth look at ext3 and the latest kernel updates](#) (Part 8)
- * [An introduction to XFS](#) (Part 9)

For more information on partitioning, take a look at the following partitioning tips on the IBM developerWorks Linux zone:

- * [Partition planning tips](#)
- * [Partitioning in action: consolidating data](#)
- * [Partitioning in action: moving /home](#)

ReiserFS Resources:

- * [The home of ReiserFS](#)
- * [Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS](#) on developerWorks
- * [Advanced filesystem implementor's guide, Part 2: Using ReiserFS and Linux 2.4](#) on developerWorks

ext3 resources:

- * [Andrew Morton's ext3 page](#)
- * [Andrew Morton's excellent ext3 usage documentation \(recommended\)](#)

XFS and JFS resources:

- * [SGI XFS projects page](#)
- * The IBM [JFS project Web site](#)

Don't forget <http://www.linuxdoc.org>. You'll find linuxdoc's collection of guides, HOWTOs, FAQs, and man pages to be invaluable. Be sure to check out [Linux Gazette](#) and [LinuxFocus](#) as well.

The Linux System Administrators guide, available from [Linuxdoc.org's "Guides" section](#), is a good complement to this series of tutorials -- give it a read! You may also find Eric S. Raymond's [Unix and Internet Fundamentals HOWTO](#) to be helpful.

In the *Bash by example* article series on *developerWorks*, Daniel shows you how to use **bash** programming constructs to write your own **bash** scripts. This bash series (particularly Parts 1 and 2) will be excellent additional preparation for the LPIC Level 1 exam:

- * [Bash by example, Part 1: Fundamental programming in the Bourne-again shell](#)
- * [Bash by example, Part 2: More bash programming fundamentals](#)
- * [Bash by example, Part 3: Exploring the ebuild system](#)

We highly recommend the [Technical FAQ by Linux Users](#) by Mark Chapman, a 50-page

in-depth list of frequently-asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Adobe Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out. We also recommend the [Linux glossary for Linux users](#), also from Mark.

If you're not familiar with the `vi` editor, we strongly recommend that you check out Daniel's [Vi intro -- the cheat sheet method tutorial](#). This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use `vi`.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact Daniel Robbins directly at drobbins@gentoo.org.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial [Building tutorials with the Toot-O-Matic](#) demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.