LPI exam 201 prep: System startup

Intermediate Level Administration (LPIC-2) topic 202

Skill Level: Intermediate

David Mertz, Ph.D. (mertz@gnosis.cx)
Developer
Gnosis Software

31 Aug 2005

In this tutorial, David Mertz begins continues you to take the Linux Professional Institute® Intermediate Level Administration (LPIC-2) Exam 201. In this second of eight tutorials, you learn the steps a Linux™ system goes through during system initialization, and how to modify and customize those behaviors for your specific needs.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The Linux Professional Institute (LPI) certifies Linux system administrators at junior and intermediate levels. To attain each level of certification, you must pass two LPI exams.

Each exam covers several topics, and each topic has a weight. The weights indicate the relative importance of each topic. Very roughly, expect more questions on the exam for topics with higher weight. The topics and their weights for LPI exam 201 are:

Topic 201
Linux kernel (weight 5).

Topic 202

System startup (weight 5). The focus of this tutorial.

Topic 203
File systems (weight 10).

Topic 204
Hardware (weight 8).

Topic 209
File and service sharing (weight 8).

Topic 211
System maintenance (weight 4).

Topic 213
System customization and automation (weight 3).

Topic 214
Troubleshooting (weight 6).

The Linux Professional Institute does not endorse any third-party exam preparation material or techniques in particular. For details, please contact info@lpi.org.

About this tutorial

Welcome to "System startup," the second of eight tutorials designed to prepare you for LPI exam 201. In this tutorial, you will learn the steps a Linux system goes through during system initialization and how to modify and customize those behaviors for your specific needs.

The tutorial is organized according to the LPI objectives for this topic, as follows:

2.201.1 Customizing system startup and boot processes (weight 2)
You will learn to edit appropriate system startup scripts to customize standard system run levels and boot processes. This objective includes interacting with run levels and creating custom `initrd` images as needed.

2.201.2 System recovery (weight 3)
You will be able to properly manipulate a Linux system during the boot process and during recovery mode. This objective includes using both the `init` utility and `init= kernel` options.

This tutorial is at the border of Linux, strictly speaking. The previous tutorial (on topic 201) addressed the kernel, which is the core of Linux. This tutorial moves on to the ancillary tools and scripts that are necessary to get the kernel running and to initialize a system to the point where it does something meaningful. Note that the scripts and tools associated with initialization are maintained by the creators of Linux distributions or individualized by system administrators rather than developed as part of the Linux kernel per se. Still, every Linux system -- even an embedded one --

requires some basic initialization steps. We'll review those steps here.

In later tutorials, we'll look at a variety of tools for networking, system maintenance, manipulating files and data, and so on, which are important for a working Linux installation and part of almost every Linux distribution, but are even less part of Linux per se than are initialization scripts.

Prerequisites

To get the most from this tutorial, you should already have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial.

Section 2. System startup and boot processes

What happens when you turn a Linux computer on?

Let's break the Linux boot process into nine steps that occur in almost every Linux configuration:

1. Hardware/firmware: The BIOS or firmware system reads the master boot record on the harddisk or other boot device (for example, CD, floppy, netboot, etc.).
2. A boot loader runs. Linux systems on x86 systems typically use LILO or GRUB. Some older systems might use loadlin to boot via an intermediate DOS partition. On Power PC® systems, this might be BootX or yaboot. In general, a *boot loader* is a simple program that knows where to look for the Linux kernel, perhaps choosing among several versions or even selecting other operating systems on the same machine.
3. The kernel loads.
4. The root filesystem is mounted. In some cases, a temporary ramdisk image is loaded before the true root filesystem to enable special drivers or modules that might be necessary for the true root filesystem.
   One we have a root filesystem in place, we are ready for initialization proper.
5. Start the process `init`, the parent of all other Linux processes.
6. Read the contents of /etc/inittab to configure the remaining boot steps. Of

special importance is the line in /etc/inittab that controls the runlevel the system will boot to (and therefore which further steps will be taken during initialization).
Actually, everything after this point is completely controlled by the content of the file /etc/inittab. Specifically, the scripts and tools that run generally follow some conventions, but in theory you could completely change /etc/inittab to run different scripts.

One specific setting in /etc/inittab is particularly crucial. A line similar to:

```
id:5:initdefault:
```

generally occurs near the top of the file, and sets the runlevel. This runlevel controls what actions are taken in the remainder on the /etc/inittab script.

Just what happens as an /etc/inittab script is processed? And specifically, what conventional files and directories are involved in the process?

7. Runlevel-neutral system initialization. Generally there are some initialization actions that are performed regardless of runlevel. These steps are indicated in /etc/inittab with a line like:
```
# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
```

   On some Linux systems (mostly Debian-based systems), you will see something more like:

```
si::sysinit:/etc/init.d/rcS
```

   If the latter case, /etc/init.d/rcS is a script that simply runs each of the scripts matching /etc/rcS.d/[Ss]??*. On the other hand, if your system uses /etc/rc.d/rc.sysinit, that file contains a single long script to perform *all* the initialization.

8. Runlevel-specific system initialization. You may actually define as many actions as you like related to runlevel, and each action may pertain to one or more runlevels. As a rule, /etc/inittab will contain some lines like:
```
l0:0:wait:/etc/rc.d/rc 0
# ...
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```

   In turn, the script /etc/rc.d/rc will run all the files matched by the pattern /etc/rc$1.d/[KkSs]??*. For example, on the sample system described that starts at runlevel 5, we would run (in order):

```
/etc/rc5.d/K15postgresql
/etc/rc5.d/S01switchprofile
```

```
/etc/rc5.d/S05harddrake
...
/etc/rc5.d/S55sshd
...
/etc/rc5.d/S99linuxconf
/etc/rc5.d/S99local
```

The files(s) starting with "K" or "k" are *kill scripts* that end processes or clean up their actions. Those starting with "S" or "s" are *startup scripts* that generally launch new processes or otherwise prepare the system to run at that runlevel. Most of these files will be shell scripts, and most will be links (often to files in /etc/init.d/).

Most of the time, once a Linux system is running at a runlevel, you want to log into the system as a user. To let that happen, a program called getty runs to handle the login process. A number of variations on the basic getty are used by distribution creators, such as agetty, mgetty, and mingetty. All do basically the same thing.

9. Log in at the prompt. Our good friend /etc/inittab usually launches getty programs on one or more virtual terminals and does so for several different runlevels. Those are configured with lines like:
```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

   The first number reminds us of the virtual terminal where the getty runs; the next set of numbers are the several runlevels where this will happen (for example, launching mingetty in all of the runlevels 2, 3, 4, and 5).

Next steps might involve launching additional services, logging into a graphical environment, restoring UI settings, or other more personalized details that are outside the scope of this tutorial.

Understanding runlevels

The concept of runlevel is somewhat arbitrary, or at least it is not hardcoded into a Linux kernel. Valid runlevel numbers to set with the `initdefault` option (or override otherwise) are 0 - 6. By convention, the following meanings are given to each number:

Listing 1. Runlevels, defined

```
# Default runlevel. The runlevels used by Mandrake Linux are:
```

```
#   0 - Halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you don't have networking)
#   3 - Full multiuser mode
#   4 - Unused
#   5 - X11
#   6 - Reboot (Do NOT set initdefault to this)
```

This convention, as you can see, is as used in the Mandrake Linux distribution, but most distributions obey the same convention. Text-only or embedded distributions may not use some of the levels, but will still reserve those numbers.

Configuration lines in /etc/inittab

You have seen a number of /etc/inittab lines in examples, but it is worth understanding explicitly what these lines do. Each one has the format:

```
id:runlevels:action:process
```

The `id` field is a short abbreviation naming the configuration line (1 - 4 characters in recent versions of `init`; 1 - 2 in ancient ones). The `runlevels` is explained already. Next is the action taken by the line. Some actions are "special," such as:

```
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

This action traps the Ctrl-Alt-Delete key sequence (regardless of runlevel). But most actions simply relate to spawning. A partial list of actions includes:

- `respawn`: The process will be restarted whenever it terminates (as with getty).
- `wait`: The process will be started once when the specified runlevel is entered, and `init` will wait for its termination.
- `once`: The process will be executed once when the specified runlevel is entered.
- `boot`: The process will be executed during system boot (but after sysinit). The runlevels field is ignored.

Section 3. Customizing system startup and boot processes

What is a boot loader?

A few years ago, a program called LILO was pretty much universally used to boot Linux on x86 systems. The name LILO is short for "LInux LOader." Nowadays,

another program called GRUB (GRand Unified Bootloader) is more popular. On non-x86 Linux systems, other boot loaders are used, but they are generally configured in the same manner as LILO or GRUB.

While there are differences in their configuration syntaxes, LILO and GRUB perform largely the same task. Basically, each presents a choice of operating systems (including, perhaps, multiple Linux kernels) and loads the selected OS kernel into memory. Both programs let you pass arguments on to a Linux kernel along the way, and both can be configured to boot non-Linux operating systems on the same machine.

Either LILO or GRUB (or other boot loaders) generally lives in the MBR (Master Boot Record) of the primary hard disk, which is automatically loaded by system BIOS. LILO was restricted to loading a specific raw sector from a harddisk. GRUB is more sophisticated in that it understands a number of filesystem types such as ext2/3, ReiserFS, VFAT, and UFS. This means that GRUB doesn't need to rewrite the MBR every time a configuration file is changed (the way LILO does).

Configuring the LILO boot loader

The LILO boot loader is configured with the contents of the file /etc/lilo.conf. For full details on configuration options, read the manpage on lilo.conf. Several initial options control general behavior. For example, you will often see `boot=/dev/hda` or similar; this installs LILO to the MBR of the first IDE hard disk. You might also install LILO within a particular partition, usually because you use a different main boot loader. For example, `boot=/dev/sda3` installs LILO to the third partition of the first SCSI disk. Other options control the appearance and wait time of LILO.

Remember that after you have edited a /etc/lilo.conf configuration, you need to run LILO to actually install a new boot sector used during initialization. It is easy to forget to install new settings, but the boot loader itself cannot read the configuration except as encoded as raw sector offsets (which LILO calculates when run).

When using LILO you are mainly interested in the one or more `image=` lines and perhaps in some `other=` lines if you multiboot to other operating systems. A sample /etc/lilo.conf might contain:

Listing 2. Sample LILO configuration

```
image=/boot/bzImage-2.7.4
label="experimental"
image=/boot/vmlinuz
label="linux"
initrd=/boot/initrd.img
append="devfs=mount acpi=off quiet"
vga=788
read-only
other=/dev/hda3
label=dos
```

This configuration would allow you to choose at runtime either a 2.7.4 development kernel or a stable kernel (the latter happens to utilize an initial ramdrive during boot). You can also select a DOS installation installed to partition 3 on the first IDE drive.

Configuring the GRUB boot loader

A nice thing about GRUB is that it does not need to be reinstalled each time you change boot configuration. Of course, you *do* need to install it once in the first place, usually using a command like `grub-install /dev/hda`. Generally, distributions will do this for you during installation, so you may never explicitly run this.

However, since GRUB knows how to read many filesystems, normally you can simply change the contents of /boot/grub/menu.lst to change the options for the next bootup. Let's look at a sample configuration:

Listing 3. Sample GRUB configuration

```
timeout 5
color black/yellow yellow/black
default 0
password secretword

title linux
kernel (hd0,1)/boot/vmlinuz root=/dev/hda2 quiet
vga=788 acpi=off
initrd (hd0,1)/boot/initrd.img

title experimental
kernel (hd0,1)/boot/bzImage-2.7.4 root=/dev/hda2 quiet

title dos
root (hd0,4)
makeactive
chainloader +1
```

Changing options within the boot loader (LILO)

Both LILO and GRUB allow you to pass special parameters to the kernel you select. If you use LILO, you may pass boot prompt arguments by appending them to your kernel selection. For example, for a custom boot setting, you might type:

```
LILO: linux ether=9,0x300,0xd0000 root=/dev/ha2 vga=791
acpi=on
```

This line passes special parameters to the Ethernet module, specifies the root partition, chooses video mode, etc. Of course, it is not all that friendly, since you need to know the exact options available *and* type them correctly.

Of particular importance is the option to change the runlevel from the boot loader. For example, for recovery purposes, you may want to run in single-user mode, which you can do with the following:

```
LILO: experimental single
```

or:

```
LILO: linux 1
```

Another special option is the `init=` argument, which lets you use a program other than init as the first process. An option for a fallback situation might be `init=/bin/sh`, which at least gets you a Linux shell if init fails catastrophically.

Changing options within the boot loader (GRUB)

With GRUB, you have even more flexibility. In fact, GRUB is a whole basic shell that lets you change boot loader configurations and even read filesystems. For custom boot options, press "e" in the GRUB shell, then add options (such as a numeric runlevel or the keyword "single" as with LILO). All the other boot prompt arguments you might type under LILO can be edited in a GRUB boot command, using simple readlines-style editing.

For some real sense of the power, you can open a GRUB command line. For example, suppose you think your /etc/inittab might be misconfigured, and you want to examine it before booting. You might type:

```
grub> cat (hd0,2)/etc/inittab
```

This would let you manually view your initialization without even launching any operating system. If there were a problem there, you might want to boot into single-user mode and fix it.

Customizing what comes after the boot loader

Once you understand the steps in a Linux post-kernel boot process (in other words, the `init` process and everything it calls), you also understand how to customize it. Basically, customization is just a matter of editing /etc/inittab and the various scripts in /etc/rc?.d/ directories.

For example, I recently needed to customize the video bios on a Debian-based Linux laptop using a third-party tool. If this didn't run before X11 ran, my XOrg driver would not detect the correct video modes. Once I figured out what the issue was, the solution was as simple as creating the script /etc/rcS.d/S56-resolution.sh. In other words, I began running an extra script during every system startup.

Notably, I made sure this ran before /etc/rcS.d/S70xorg-common by the simple convention that scripts run in alphabetical order (if I wanted it to run later, I might have named it S98-resolution.sh instead). Arguably, I might have put this script only in the /etc/rc5.d/ directory to run when X11 does -- but my way lets me manually run `startx` from other runlevels.

Everything in the initialization process is out in the open, right in the filesystem; almost all of it is in editable text scripts.

Section 4. System recovery

About recovery

The nicest thing about Linux from a maintenance perspective is that everything is a file. Of course, it can be perplexing at times to know *which* file something lives in. But as a rule, Linux recovery amounts to using basic filesystem utilities like `cp`, `mv`, and `rm`, and a text editor like vi. Of course, to automate these activities, tools like grep, awk, and bash are helpful; or at a higher level, perl or python. This tutorial does not address basic file manipulation.

Assuming you know how to manipulate and edit files, the only "gotcha" perhaps remaining for a broken system is not being able to use the filesystems at all.

Fixing a filesystem with fsck

Your best friend in repairing a broken filesystem is `fsck`. The next tutorial (on topic 203) has more information, so we will just introduce the tool here.

The tool called `fsck` is actually just a front-end for a number of more narrow `fsck.*` tools -- `fsck.ext2`, `fsck.ext3`, or `fsck.reiser`. You may specify the type explicitly using the `-t` option, but `fsck` will make an effort to figure it out on its own. Read the manpage for `fsck` or `fsck.*` for more details. The main thing you want to know is that the `-a` option will try to fix everything it can automatically.

You can check an unmounted filesystem by mentioning its raw device. For example, use `fsck /dev/hda8` to check a partition not in use. You can also check a rooted filesystem such as `fsck /home`, but generally do that only if the filesystem is already mounted as read-only, not as read-write.

Mounting and unmounting with mount and umount

A flexible feature of Linux systems is the fine-tuned control you have over mounting and unmounting filesystems. Unlike under Windows and some other operating systems, partitions are not automatically assigned locations by the Linux kernel, but are instead attached to the single `/` root hierarchy by the `mount` command. Moreover, different filesystem types (on different drives, even) may be mounted within the same hierarchy. You can unmount a particular partition with the `umount` command, specifying either the mount point (such as /home) or the raw device (such as /dev/hda7).

For recovery purposes, the ability to control mount points lets you do forensic analysis on partitions -- using fsck or other tools -- without risk of further damage to a damaged filesystem. You may also custom mount a filesystem using various

options; the most important of these is mounting read-only using either of the synonyms `-r` or `-o ro`.

As a quick example, you might want to substitute one user directory location for another, either because of damage to one or simply to expand disk space or move to a faster disk. You might perform this switch using something like:

```
# umount /home # old /dev/hda7 home dir
# mount -t xfs /dev/sda1 /home # new SCSI disk using XFS
# mount -t ext3 /dev/sda2 /tmp # also put the /tmp on SCSI
```

Mounting at bootup with /etc/fstab

For recovery, system upgrades, and special purposes, it is useful to be able to mount and unmount filesystems at will. But for day-to-day operation, you generally want a pretty fixed set of mounts to happen at every system boot. You control the mounts that happen at bootup by putting configuration lines in the file /etc/fstab. A typical configuration might look something like this:

Listing 4. Sample configuration in /etc/fstab

```
/dev/hda7 / ext3 defaults 1 1
none /dev/pts devpts mode=0620 0 0
/dev/hda9 /home ext3 defaults 1 2
none /mnt/cdrom supermount
dev=/dev/hdc,fs=auto,ro,--,iocharset=iso8859-1,codepage=850,umask=0 0 0
none /mnt/floppy supermount
dev=/dev/fd0,fs=auto,--,iocharset=iso8859-1,sync,codepage=850,umask=0 0 0
none /proc proc defaults 0 0
/dev/hda8 swap swap defaults 0 0
```

Find more details in the next tutorial (on topic 203).

Resources

Learn

- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- "Boot loader showdown: Getting to know LILO and GRUB" (developerWorks, August 2005) discusses how boot loaders work and can help you decide which of the two most popular -- LILO and GRUB -- might work best for you.
- Find more resources for Linux developers in the developerWorks Linux zone.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

David Mertz, Ph.D.
David Mertz is Turing complete, but probably would not pass the Turing Test. For more on his life, see his personal Web page. He's been writing the developerWorks columns *Charming Python* and *XML Matters* since 2000. Check out his book *Text Processing in Python* .