Operating Systems (Client)

Lecture 3

The Linux Boot Process

Dr. Kevin Farrell

Lecture Overview

- Bootstrapping
- Automatic and Manual Booting in Linux
- Booting PCs
- Boot Loaders: LILO and GRUB
- Booting Single-User Mode
- Start-up Scripts
- Rebooting and Shutting Down
- References

Bootstrapping

Introduction

- Linux is a complex operating system
- Turning Linux systems on and off is more complicated than just flipping the power switch
- Both operations must be performed correctly if the system is to stay healthy.
- The bootstrapping process varies widely
- Warnings:
  - Assume some familiarity with the Filesystem, Drivers and the Kernel and Daemons.
  - the booting process is hardware dependent. The information that follows is generically true but may, in reality, differ for your system.

Bootstrapping 1

- Bootstrapping means “starting up a computer”
  - The normal facilities provided by the operating system are not available during the startup process, so the computer must “pull itself up by its own bootstraps”
- During bootstrapping:
  - the kernel is loaded into memory and begins to execute
  - A variety of initialisation tasks are performed, and
  - the system is then made available to users.
- Boot time is a period of special vulnerability
  - Errors in configuration files, missing or unreliable equipment, and damaged filesystems can all prevent a computer from coming up

Bootstrapping 3

- Several things must happen before a login prompt can appear:
  - Filesystems must be checked and mounted, and
  - system daemons started
- These procedures are managed by a series of shell scripts that are run in sequence by init
- The startup scripts often referred to as “rc files” because of the way they are named:
  - the “rc” stands for “runcom” or “run command,” for historical reasons
- The exact layout of the startup scripts and the manner in which they are executed vary among systems.

Bootstrapping 2

- Boot configuration is often one of the first tasks an administrator must perform on a new system
- Unfortunately, it is also one of the most difficult, and it requires some familiarity with many other aspects of Linux.
- When a computer is turned on, it executes boot code that is stored in ROM
- That code in turn attempts to figure out how to load and start your kernel
- The kernel probes the system's hardware and then spawns the system's init process, which is always PID 1

Automatic and Manual Booting in Linux

Automatic and Manual Booting

- Linux can boot in either automatic or manual mode
- Automatic mode: system performs complete boot procedure on its own (no external assistance)
- Manual mode: system follows automatic procedure up to a point; then turns control over to an operator before most initialisation scripts have been run
  - computer is then in “single-user mode.” Most system processes are not running; other users cannot log in.
- Important to understand automatic boot procedure and to know how to perform a manual boot
- Usually boot manually when some problem breaks automatic booting; for eg: a corrupted filesystem or an improperly configured network interface.

Steps in the Boot Process

- A typical bootstrapping process consists of six distinct phases:
  1. Loading and initialisation of the kernel
  2. Device detection and configuration
  3. Creation of spontaneous system processes
  4. Operator intervention (manual boot only)
  5. Execution of system start-up scripts
  6. Multi-user operation
- Administrators have little control over most of these steps. But, can change bootstrap config. by editing the system start-up scripts.

Kernel Initialisation (1)

- The Linux kernel is itself a program
- First bootstrapping task is to get this program into memory so that it can be executed
- The pathname of the kernel is usually **/vmlinuz** or **/boot/vmlinuz**
- Linux implements a two-stage loading process
  - Stage1: the system ROM loads a small boot program into memory from disk
  - Stage 2: This program then arranges for the kernel to be loaded

Kernel Initialisation (2)

- Kernel performs memory tests to determine available RAM
- Some of the kernel's internal data structures are statically sized, so the kernel sets aside a fixed amount of real memory for itself when it starts, which is reserved for the kernel
- Kernel prints message on console that reports total physical memory and amount available to user processes.

Hardware Configuration (1)

- One of the kernel's first chores is to examine the machine's environment to see what hardware is present
- When you construct a kernel for your system, you tell it what hardware devices it should expect to find
- When the kernel begins to execute, it tries to locate and initialise each device that you have told it about
- The kernel prints out a line of cryptic info. about each devices it finds

Hardware Configuration (2)

- The device info. provided at kernel config. time is often minimal
- In that case, the kernel tries to determine the other info. it needs by probing the bus for devices and asking the appropriate drivers for info.
- Drivers for devices that are missing or that do not respond to a probe will be disabled
- If a device is later connected to the system, it may also be possible to load or enable a driver for it on the fly

System Processes (1)

- Once basic initialisation is complete, the kernel creates several “spontaneous” processes in user space; (spontaneous because they are not created through the normal system fork mechanism).
- The number and nature of spontaneous processes may vary
- Under Linux, there is no visible PID 0. **init** (always PID 1) is accompanied by several memory and kernel handler processes, typically, **kflushd**, **kupdate**, **kpiod**, and **kswapd**

System Processes (2)

- But, only **init** is really a full-fledged user process. The others are actually portions of the kernel that have been dressed up to look like processes for scheduling or architectural reasons.
- Once spontaneous processes created, kernel's role in bootstrapping is complete
- However, none of the processes that handle basic operations (such as accepting logins) have been created, nor have most of the Linux "**daemons**" been started
- All of these tasks are taken care of (indirectly, in some cases) by init.

Operator Intervention: Manual Boot (1)

- If the system is to be brought up in single-user mode, a command-line flag passed in by the kernel notifies init of this as it starts up
- init eventually turns control over to `sulogin`, a special version of `login` that prompts for the root password. If you enter the right password, the system will spawn a root shell. You can type <Control-D> instead of a password to bypass single-user mode and continue to multi-user mode.

Operator Intervention: Manual Boot (2)

- From the single-user shell, you can execute commands in much the same way as when logged in on a fully booted system.
- Daemons don't normally run in single-user mode, so commands that depend on server processes (e.g., mail) won't work correctly.
- The **fsck** command is normally run during an automatic boot to check and repair filesystems. When you bring the system up in single-user mode, you may need to run **fsck** by hand.
- When the single-user shell exits, the system will attempt to continue booting into multi-user mode.

Execution of Start-up Scripts

- By the time the system is ready to run its start-up scripts, it is recognisably Linux
- Even though it doesn't quite look like a fully booted system yet, there are no more "magic" steps in the boot process
- The start-up scripts are just normal shell scripts, and they're selected and run by init according to an algorithm that is relatively comprehensible.
- The care, feeding, and taxonomy of start-up scripts merits a major discussion of its own.

Multi-User Operation (1)

- After the initialisation scripts have run, the system is fully operational, except that no one can log in.
- For logins to be accepted on a particular terminal (including the console), a **getty** process must be listening on it.
- init spawns these getty processes directly, completing the boot process

Multi-User Operation (2)

- init is also responsible for spawning graphical login systems such as xdm or gdm if the system is set up to use them.
- Important to remember that init continues to perform an important role even after bootstrapping is complete
- init has one single-user “run level” and several multi-user “run levels”. Each of these determine which of the system’s resources are enabled.

Booting PCs

Booting PCs vs Proprietary Hardware (1)

- When a machine boots, it begins by executing code stored in ROMs.
- On a machine designed explicitly for UNIX or another proprietary OS, the code is typically firmware that knows how to use the connected devices, talk to the network on a basic level, and understand disk-based filesystems.
- Such omniscient firmware is very convenient for system admins. For eg, you can just type in the filename of a new kernel, and the firmware will know how to locate and read that file.

Booting PCs vs Proprietary Hardware (2)

- On PCs, initial boot code is called a BIOS, and is extremely simplistic compared to the firmware of a proprietary machine.
- In fact, a PC has several levels of BIOS: one for the machine itself, one for the video card and one for the SCSI card if system has one.
- The BIOS knows about some of the devices on the motherboard, typically the IDE controller (and disks), keyboard, serial ports, and parallel ports. SCSI cards are usually only aware of the devices that are connected to them.

The PC Boot Process (1)

- The BIOS will let you select which devices you want to boot from.
- The BIOS is generally limited to booting from the first IDE CD-ROM drive or the first IDE hard disk. Some BIOSes acknowledge the existence of SCSI cards.
- The PC will then try to load the first 512 bytes of the disk (first sector of the disk). This 512-byte segment is known as the **Master Boot Record** or MBR

The PC Boot Process

- The MBR contains a program that tells the PC from which disk partition to load a secondary boot program (the “boot loader”).
- The default MBR is a simple program that tells the PC to get its boot loader from the first partition on the disk.
- Linux provides a more sophisticated MBR to deal with multiple OSes and kernels.
- Once the MBR has chosen the partition to boot from, it tries to load the boot loader specific to that partition. The boot loader is then responsible for loading the kernel.

Boot Loaders: LILO and GRUB

Boot Loaders: LILO (1)

- Traditional Linux boot loader; it is very stable and well documented. (LILO = LInux LOader)
- LILO comes with almost all Linux distributions
- LILO can be installed either into the MBR of the disk or into the boot record of the Linux root partition.
- LILO is configured and installed with the `lilo` command.

Boot Loaders: LILO (2)

- LILO bases the installed configuration on the contents of the /etc/lilo.conf file
- To change your boot configuration, you simply update /etc/lilo.conf and re-run `lilo`
- LILO must be reconfigured every time the boot process changes—in particular, every time you want to add a new boot partition and every time you have a new kernel to boot.

Boot Loaders: GRUB (1)

- GRUB is a new complete replacement for LILO; it is both more flexible and more complex.
- It's a common aftermarket option that can be added to any Linux system; it also receives official support from Red Hat.
- Popular among users who run a variety of OSes (such as Windows, OpenBSD, FreeBSD, etc.) on the same machine or who are actively working on kernel development.

Boot Loaders: GRUB (2)

- GRUB is also useful for folks who change their system config. frequently. No need to re-install each time boot config. is changed since GRUB reads its config. file on the fly at boot time, eliminating an extra step.
- Install GRUB on your boot drive by running `grub-install`. This command takes the name of the device from which you'll be booting as an argument.
- Unfortunately, GRUB has its own way of naming physical disk devices which is very different from the standard Linux convention.

Multi-Booting on PCs (1)

- Since many OSes run on PCs, it is common practice to set up a machine to be able to boot several different OSes
- To make this work, you need to configure a boot loader to recognise all of the different operating systems on your disks.
- Every disk partition can have its own second-stage boot loader. However, there is only one MBR.

Multi-Booting on PCs (2)

- When setting up a multi-boot configuration, you must decide which boot loader is going to be the “master.”
- Your choice will often be dictated by the vagaries of the OSes involved. LILO and GRUB are the best options for a system that has a Linux partition. The exceptions to this rule are Windows NT and 2000, which *sometimes* need to have their own boot loader in the MBR.

Booting Single-User Mode

Booting Single-User Mode (1)

- Usually enter Linux's single-user mode through LILO
- How you reach the LILO prompt varies based on the exact hardware of your system.
- At the LILO prompt, enter the label of the configuration you want to boot (as specified in lilo.conf) followed by-s or single.
- For eg: default config. shipped with Red Hat is called "linux", so to boot that config. into single-user mode, you'd use:
  - `LILO:` linux single

Booting Single-User Mode (2)

- LILO accepts a variety of other command-line options:

Option	Meaning
root=/dev/foo	Tells the kernel to use /dev/foo as the root device
single	Boots to single-user mode
init=/sbin/init	Tells the kernel to use /sbin/init as its init program
ether=0,0,eth 1	Makes the kernel probe for a 2nd Ethernet card
init=/bin/bash	Starts only the bash shell; useful for emergency recovery

Start-up Scripts

Start-Up Scripts (1)

- After you exit from single-user mode (or, in the automated boot sequence, at the point at which the single-user shell would have run), init executes the system start-up scripts
- These are ordinary shell scripts that are interpreted by `bash`
- Their exact location, content, and organisation varies considerably depending on your Linux distribution

Start-Up Scripts (2)

- Some tasks that are often performed in the start-up scripts are:
  - Setting the name of the computer
  - Setting the time zone
  - Checking the disks with fsck (only in automatic mode)
  - Mounting the system's disks
  - Removing old files from the /tmp directory
  - Configuring network interfaces
  - Starting up daemons and network services

Start-Up Scripts (3)

- Most start-up scripts are quite verbose and print out a description of everything they are doing
- This can be helpful if the system hangs midway through booting or if you are trying to locate an error in one of the scripts.
- In the past, it was common practice to modify start-up scripts to make them do the right thing for a particular environment

Start-Up Scripts (4)

- Nowadays, the scripts supplied with your system should be general enough to handle most any configuration
- Instead of putting the details of your local configuration in the code of the scripts, you put them in a separate configuration file (or set of files) that the scripts consult
- The config. files are normally just mini `bash` scripts that the start-up scripts include to define the values of certain shell variables.

Init and Run Levels (1)

- Traditionally, init defines 7 **run levels**, each of which represents a particular complement of services that the system should be running:
  - Level 0 is the level in which the system is completely shut down
  - Level 1 or S represents single-user mode
  - Levels 2 through 5 are multi-user levels
  - Level 6 is a “reboot” level

Init and Run Levels (2)

- Levels 0 and 6 are special in that the system can't actually remain in them; it shuts down or reboots as a side effect of entering them.
- On most systems, the normal multi-user run level is 2, 3 or 5
- Run level 5 is often used for X Windows login processes such as xdm, giving a multi-user run level with a graphical user login
- Run level 4 is rarely used
- Run levels 1 and S are defined differently on each system.

Init and Run Levels (3)

- Single-user mode traditionally init level 1. It brought down all multi-user and remote login processes and made sure the system was running a minimal complement of software.
- However, since single-user mode provides root access to the system, administrators wanted the system to prompt for the root password whenever it was booted into single-user mode.
- The **S** run level was created to address this need: it spawns a process that prompts for the root password.

Init and Run Levels (4)

- On Linux, the **S** level serves only this purpose; it's not a destination in itself.
- There are more run levels ***defined*** than are strictly necessary or useful. The usual explanation for this is that a phone switch had 7 run levels, so it was thought that a UNIX system should have at least that many.
- Linux actually supports up to 10 run levels, but levels 7 through 9 are undefined.

Init and Run Levels (5)

- The **/etc/inittab** file tells init what to do at each run level. Its format varies from system to system, but the basic idea is that inittab defines commands that are to be run (or kept running) when the system enters each level.
- As the machine boots, init ratchets its way up from run level 0 to the default run level set in /etc/inittab. To accomplish the transition between each pair of adjacent run levels, init runs the actions spelled out for that transition in /etc/inittab. The same progression is made in reverse order when the machine is shut down.

Init and Run Levels (6)

- Unfortunately, the semantics of the inittab file are somewhat rudimentary. To map the facilities of the inittab file into something a bit more flexible, Linux systems implement an additional layer of abstraction in the form of a "change run levels" script (usually /etc/init.d/rc) that's called from inittab.
- This script in turn executes *other* scripts from a run-level-dependent directory to bring the system to its new state.

Init and Run Levels (7)

- It's usually not necessary for system administrators to deal directly with /etc/inittab because the script-based interface is adequate for almost any application
- We will tacitly ignore the inittab file and the other glue that attaches init to the execution of start-up scripts. Just keep in mind that when we say that init runs such-and-such a script, the connection may not be quite so direct.

Init and Run Levels (7)

- The master copies of the start-up scripts reside in the **/etc/init.d** directory. Each script is responsible for one daemon or one particular aspect of the system.
- The scripts understand the arguments **start** and **stop** to mean that the service they deal with should be initialised or halted. Most also understand **restart**, which is typically the same as a **stop** followed by a **start**.
- As a system admin., you can manually start and stop individual services by running their associated init.d scripts by hand.

Redhat Start-Up Scripts (1)

- Start-up scripts are one of the things which distinguish Linux distributions from each other. Red Hat's are a mess!
- At each run level, init invokes the script **/etc/rc.d/rc** with the new run level as an argument
- **/etc/rc.d/rc** usually runs in "normal" mode, in which it just does its thing, but it can also run in "confirmation" mode, where it asks before it runs each start-up script

Redhat Start-Up Scripts (2)

- Use the chkconfig command to manage the links in the run-level directories.
- Red Hat also has an **rc.local** script much like that found on BSD systems. **rc.local** is the last script run as part of the start-up process. It's best to avoid adding your own customisations to rc.local because this file gets overwritten by the install scripts

Redhat Boot Process Config. Files

- Most configuration of Red Hat's boot process should be achieved through manipulation of the config. files in **/etc/sysconfig**. The table summarises the function of the items in this directory:

Files and subdirectories of Red Hat's /etc/sysconfig directory

File/Dir	Function or contents
apmd	Lists arguments for the Advanced Power Management Daemon
clock	Specifies the type of clock that the system has (almost always UTC)[a]
console	A mysterious directory that is always empty
hwconf	Contains all of the system's hardware info. Used by Kudzu.
i18n	Contains the system's local settings (date formats, languages, etc.)
init	Configures the way messages from the startup scripts are displayed
keyboard	Sets keyboard type (use "us" for the standard 101-key U.S. keyboard)
mouse	Sets the mouse type. Used by X Windows and **gpm**.
network	Sets global network options (hostname, gateway, forwarding, etc.)
network-scripts	A directory that contains accessory scripts and network config files
pcmcia	Tells whether to start PCMCIA daemons and specifies options
sendmail	Sets options for **sendmail**

a. If you multiboot your PC, all bets are off as to how the clock's time zone should be set.

Rebooting and Shutting Down

Rebooting and Shutting Down (1)

- Linux filesystems buffer changes in memory, and write them back to disk only sporadically
- This scheme makes disk I/O faster, because recently accessed disk blocks are in memory, but it also makes the filesystem more susceptible to data loss when the system is rudely halted.
- Traditional UNIX and Linux machines were very touchy about how they were shut down

Rebooting and Shutting Down (2)

- Modern systems have become less sensitive, especially when you use a robust filesystem such as a journalled filesystem like **ext3fs**
- However, it is always a good idea to shut down the machine nicely when possible
- Improper shutdown can result in anything from subtle, insidious problems to a major catastrophe.

Rebooting and Shutting Down (3)

- On consumer-oriented operating systems, rebooting the operating system is an appropriate first course of treatment for almost any problem
- On a Linux system, it's better to think first and reboot second
- Linux problems tend to be subtler and more complex, so blindly rebooting is effective in only a smaller percentage of cases

Rebooting and Shutting Down (3)

- Linux systems also take a long time to boot, and multiple users may be inconvenienced.
- You may need to reboot when you add a new piece of hardware or when an existing piece of hardware becomes so “confused” that it cannot be reset
- If you modify a config. file that’s used only at boot time, you must reboot to make your changes take effect
- If the system is so wedged that you cannot log in to make a proper diagnosis of the problem, you obviously must reboot.

Rebooting and Shutting Down (4)

- Some things can be changed and fixed without rebooting, but whenever you modify a start-up script, you should reboot just to make sure that the system will come back up successfully.
- Unlike bootstrapping, which can be done in essentially only one way, there are a number of ways to shut down or reboot. They are:
  - Turning off the power
  - Using the **shutdown** command
  - Using the **halt** and **reboot** commands
  - Using **telinit** to change **init**'s run level
  - Using the **poweroff** command to tell the system to turn off the power

“shutdown”... for safety!

- Using **shutdown** is the safest, most considerate and most thorough way to initiate a halt, reboot or to return to single-user mode
- You can ask shutdown to wait a while before bringing down the system. During the waiting period, shutdown sends messages to logged-in users at progressively shorter intervals, warning them of the impending downtime
- Users cannot login when a shutdown is imminent, but they will see your message if you specified one.
- shutdown lets you specify whether the machine should halt (-h) or reboot (-r) after the shutdown is complete. You can also specify whether you want to forcibly fsck the disks after a reboot (-F) or not (-f).
- By default, Linux automatically skips the fsck checks whenever the filesystems were *properly* “umounted”.

“halt” ... for simplicity!

- The **halt** command performs the essential duties required to bring the system down
- It is called by **shutdown -h** but can also be used by itself
- **halt** logs the shutdown, kills non-essential processes, executes the **sync** system call (called by and equivalent to the **sync** command), waits for filesystem writes to complete, and then halts the kernel.
- **halt -n** prevents the sync call. It’s used by **fsck** after it repairs the root partition. If **fsck** did not use **-n**, the kernel might overwrite **fsck**’s repairs with old versions of the superblock that were cached in memory.

“reboot” ... quick and dirty restart!

- **reboot** is almost identical to halt, but it causes the machine to reboot instead of halting
- **reboot** is called by **shutdown -r**
- Like **halt**, it supports the **-n** flag

“telinit” ... for run level madness!

- You can use **telinit** to direct **init** to go to a specific run level. For example, to take the system to single-user mode, type:
  - `# telinit 1`
- When you use **telinit**, you do not get the nice warning messages or grace period that you get with shutdown, so most of the time you’ll probably want to avoid it
- **telinit** is most useful for testing changes to the **inittab** file

“poweroff” ... to say goodbye!

- The **poweroff** command is identical to **halt**, except that after Linux has been shut down, **poweroff** sends a request to the power management system (on systems that have one) to turn off the system’s main power
- This feature makes it easy to turn off machines remotely (for example, during an electrical storm).
- Unfortunately, there is no corresponding poweron command.
- The reason for this apparent oversight is left as an exercise for the reader!

References

- “Linux Administration Handbook”, by Nemeth, Snyder and Hein, (Prentice Hall, 2002).
- “Linux System Administration”, by Stanfield and Smith, (Sybex, Craig Hunt Linux Library, 2001)