CST8177 – Linux II

Linux Boot Process

Reference information from the text,

http://www.linuxdoc.org
and several other web sites

Linux Boot Process

Topics covered in this slide-set

- Basic definition of the boot process
  - Description of boot loader(s)
    - P.O.S.T., BIOS & MBR
    - LILO vs GRUB
  - Typical Boot Loaders
    - Using LILO
    - Using GRUB
  - The Linux boot process in detail
  - The different ways of booting Linux
  - Boot disks and other critical recovery mechanisms

Linux Boot Process

From the standpoint of any O/S, the very first part of the operating system boot process is what's commonly called the Boot Strap process.

- the O/S loads itself and all required components and drivers into memory to allow processes to communicate with the kernel.
- Once this is done, other applications, drivers and necessary tools for the function of the O/S can be loaded.

However, that's an incomplete view of the boot process. Several steps, at the hardware level, have to successfully before any of the "Boot Strap" process(es) can even begin.

Linux Boot Process

The steps that are required prior to the boot process include (see previous courses):

- P.O.S.T.
  - Power On Self-Test
  - Core hardware verification and testing
  - CMOS verifies its stored settings
- CMOS examines the list of physical devices for the first boot device available with a boot-sector enabled on it (called MBR or **M**aster **B**oot **R**ecord).

  This address contains code to load the O/S kernel or a boot loader to load the O/Ses available to it.

  This loading of the initial kernel O/S code from the physical address given by the CMOS is called the Boot Strap Process.

Boot Loader

- Linux boot loader
  - Resides in the MBR or in the Linux boot (**`/boot`**) directory (or partition) on the system
  - It's the first code to run when the MBR is accessed on the boot drive by BIOS after the P.O.S.T. process is completed
  - boots Linux and/or any other boot partition bootstrap-capable O/S available, so long as they are listed in the boot loader’s configuration file
  - can also allow users to pass various parameters to the kernel before loading the kernel.
- Boot loader is typically installed during installation or by booting to a Linux partition (using a bootable device), configuring the boot loader, and installing the boot loader manually.

Linux Boot Loader

- N.B. The boot loader doesn’t have to support a filesystem to access a file on it if it has a list of sectors for the file and is prepared to handle the BIOS interrupts. Thus, it can access files without having to mount them.
- A boot loader is not required in order to run Linux if it is the only O/S on the system or by using alternative boot methods (floppy, CD or DVD, boot EPROM, USB device, network, etc) or using the **`loadlin`** Linux loader.

  A boot loader can certainly make it easier to manage multi-boot capabilities with other operating systems and/or multiple versions of a kernel on the same system.

Linux Boot Loader

- Linux boot loaders in use today include **LILO** and **GRUB**. If no boot loader is used, then **loadlin** acts as the boot sector loader.
- LILO: LInux LOader
  - UNIX text-based boot loaders adapted for Linux
  - very structured config file **/etc/lilo.conf**
  - limited security features
- GRUB: GRand Unified Boot-loader shell
  - modern replacement for LILO
  - more like a special-purpose shell than an boot loader application
  - text or GUI based interface and config
  - structured config files in **/boot/grub**
  - very good security features

Using LILO

- LILO can load any O/S, so long as it allows for chain-loading the O/S from a specific boot sector on the storage medium (which is true of most O/S)
- Access to LILO and to modify **`/etc/lilo.conf`** is reserved to root for obvious reasons
- LILO can be installed:
  - by booting to a Linux partition
  - modifying the **`/etc/lilo.conf`** file to match multiple boot settings
  - running **`lilo`** to install it in the MBR
  - the **`lilo -q`** command can be used to query LILO about what images are currently defined and which one is the default (marked *)

Using LILO

- LILO files:
  - configuration file for LILO options and settings

    /etc/lilo.conf
  - part of the source code for LILO

    /boot/boot.b and **/boot/chain.b**
  - binary executable

    /sbin/lilo
  - Documentation

    /usr/share/doc/lilo-<version>

Using LILO

- Configuration file: **`/etc/lilo.conf`**
  - general section
    - for LILO and general kernel options
    - general features and security options
  - stanza (each boot partition definition)
    - allows for multi-boot by defining a stanza for each boot partition
    - each stanza can contain specific options and information on the particular boot partition
    - Each boot partition requires its own stanza to define its requirements and functions
  - See grubby(8) and LILO HowTo's for more information on the structure and options available for the configuration file and stanzas

Using GRUB

- GRUB is one of the more powerful boot loaders available today.
- It contains some striking similarities with LILO in its basic capabilities, but far surpasses it in higher security, flexibility and overall capabilities.
- GRUB can load more O/Ses than LILO. It can also natively access most known filesystems without the need to actually mount the filesystem in question - which is a massive advantage.
- Access to GRUB and modifying its configuration files is reserved to root for obvious reasons.

Using GRUB

- The main difference between GRUB and LILO:
  - the menu interface (GUI or text available)
  - the command line shell built into GRUB
- The GRUB shell can be accessed by the administrator locally or remotely, and be used to change parameters or stanzas temporarily (i.e. for this one boot only) or permanently within the configuration information.
- These simple features, and the capabilities built into them, allows for a much higher flexibility of design and purpose while still allowing an easy navigation mechanism between boot partitions for users.

Using GRUB

- GRUB :
  - recognizes multiple executable formats
  - supports non-multiboot kernels
  - supports chain-loading other boot-loaders
  - allows loading multiple kernel modules
  - allows loading a human-readable text config file
  - GUI/Graphical interface, but can run in text mode.
  - no preset limit on the number of menu options
  - allows for dynamically modifying of configuration and menu entries from the GRUB command line interface before even selecting any of the boot images

. . .

Using GRUB

- GRUB :
  - support multiple filesystems types transparently
  - supports automatic decompression of GZIP files
  - access data on any installed device
  - allows for loading an arbitrary O/S any way you like without recording the physical position of the kernel on the disk
  - independent of drive geometry translations
  - detect all installed RAM
  - supports network booting
  - supports remote terminals

And many more...

Using GRUB

GRUB sample config (**/boot/grub/grub.conf**) with highlighted **stanza**:

```
# grub.conf generated by anaconda
default=0
###timeout-0
timeout=10
###hiddenmenu
splashimage=(hd,0)/grub/splash.xpm.gz
password --md5 $1$a43gs^yGSzfghefqSD.
title Fedora (2.6.32...)
  root (hd0.0)
  kernel /vmlinuz-2.6.32... ro
  root=/dev/hda7
  initrd /initrd-2.6.32...img
```

Using GRUB

- Due to the complexity, and the extensive set of commands and capabilities inherent to GRUB, we will not go into details about how it all works.

  GRUB has become the standard for boot loaders in the Linux/UNIX community - in other words, it is to your best advantage to learn about it !

  For more information on GRUB, its configuration and capabilities, visit:

  http://www.gnu.org/software/grub/manual/

Using GRUB

- GRUB error messages:
  - If an error occurs in part of the boot process under GRUB's control, an error message is generated, often with log information about it.
  - GRUB can even allow the user to regain some control of the system to attempt to debug the problem without having to reboot.
  - In most cases, however, the error information generated will need to be investigated and GRUB can then be restarted in interactive mode to give information to the boot process manually to fix or bypass the problem.
  - Because GRUB is more than just a simple boot loader, it provides more control and allows direct manipulation of the process.

Linux Boot Process

- BIOS loads from the MBR or **/boot** directory
  - The boot loader points to the appropriate boot device for loading MBR info, based on the boot image name selection at the boot loader prompt
  - Updates the boot sector information, if needed
- Once Linux is selected, the kernel image loads
  - The kernel image is mostly compressed, but an uncompressed portion of kernel image contains information on how to decompress the rest of the kernel into memory
  - The type of boot is dictated by entries in the map installer configuration file

    /boot/System.map-...

Linux Boot Process

- Load the Kernel image (continued)
  - Setup, decompression routines, and the compressed kernel are loaded from the boot device to the first megabyte of system RAM
  - Kernel takes over memory control and the boot process, decompressing the rest of itself into memory on its own (bootstrap process)
    - verifies kernel code content during decompression
    - Kernel is decompressed in protected mode
  - Kernel then starts memory management in linear (or flat) memory mode, releasing and cleaning out the memory used back into the free memory pool

Linux Boot Process

- Load the Kernel image (continued)
  - Kernel then starts actual boot process
    - completes read of **/boot/System.map** for the remainder of information about the system
    - verifies information gathered to ensure that everything is where it should be and working before accessing and using any of it
  - Kernel displays messages to the system console about the boot process while loading all the required modules and services
  - Kernel runtime messages are stored in **/var/log/messages**
    - The last relevant set of messages can also be accessed with **dmesg** command (dmesg(1)).

Linux Boot Process

- Load the Kernel image (continued)
  - Kernel contains several built-in parameters
    - boot device parameter table
    - device name to mount as root (/) filesystem once booting is complete
    - mode to mount root (/) filesystem as (typically read-only at this stage)
  - Kernel can also accept and process command line parameters from the user prior to start-up
    - allows for modifying defaults listed in configuration files or passing parameters only required for this particular boot session

Linux Boot Process

- Load the Kernel image (continued)
  - Kernel runs the low-level initialization of various hardware devices
    - this is only true of hardware required for the kernel to work properly and devices that have been compiled into the kernel (monolithic kernel) and not as modules
  - Kernel mounts the root (**/**) filesystem in r/o mode and verifies filesystem integrity
    - If everything is okay, remounts in r/w mode
    - If not, stops and has admin fix the problem(s)

Sample /etc/fstab

```
Prompt$ cat /etc/fstab
# /etc/fstab
# Created by anaconda on Tue May 25 12:47:22 2010
# See man pages fstab(5), findfs(8), mount(8) and/or
blkid(8) for more info
#
UUID=89224b3e-1955-4cc9-99ab-7406377b7553 /
          ext3    defaults        1 1
UUID=748f4f67-2442-484a-be51-2b3270888185 /home
          ext3    defaults        1 2
UUID=2a54d895-135f-4403-b1e7-8751c93d3ca2 swap
          swap    defaults        0 0
tmpfs         /dev/shm     tmpfs   defaults         0 0
devpts        /dev/pts     devpts  gid=5,mode=620   0 0
sysfs         /sys         sysfs   defaults         0 0
proc          /proc        proc    defaults         0 0
```

Linux Boot Process

- Load the Kernel image (continued)
  - Kernel then starts up the final internal processes required for it to function properly
  - Kernel releases control to the **`init`** daemon
    - System V **`init`** for RH Linux and Fedora; other distros vary
    - locates **`init`** daemon in **`/sbin`** directory and starts it
  - Kernel image remains in memory, with loaded configuration information, until system is rebooted or shutdown

Linux Boot Process

- **`init`** program
  - general purpose program that spawns new processes and restarts certain programs when they exit, based on the signals they are given.
  - **`init`** is responsible for:
    - communicating between the kernel and all other processes
    - managing all processes currently running and their parent / child relationship(s)
    - running a series of initialization programs and scripts when the system boots / starts-up
  - Kernel is hardwired to run the **`init`** process no matter what

Linux Boot Process

- init program (continued)
  - each process spawned by **`init`** seeks any required modules / processes
    - based on known module dependency within the process / module
    - spawns any required modules in the dependency listing
    - if spawned modules have dependencies, each module it is dependent on will be spawned as well
  - Determines the starting runlevel to use (usually 5, for GUI) from the file **`/etc/inittab`**
    - contains basic information about the runlevels for processes
    - more on runlevels soon

Linux Boot Process

- **`init`** program (continued)
  - Runs the script **`/etc/rc.d/rc.sysinit`** for necessary settings and processes
    - it will run other scripts and read config files for initializing different hardware capabilities and other subsystems.
  - Calls each **`/etc/rc`** S-scripts in numeric order:
    - Runs all scripts for the current runlevel from **`/etc/rc.d/rcX.d`**, where **`X`**=runlevel
    - Script names match **`^[SK][0-9]{2}.*`**
  - Runs **`/etc/rc.d/rc.local`** for local settings
  - once all modules are spawned, loaded, and executed properly, control is returned to the kernel for next stage

Linux Boot Process

- System startup files
  - stored in **/etc/rc.d** in Fedora and Red Hat
  - managed by script file called **rc** in **/etc/rc.d**
  - **rc** looks in **/etc/rc.d/rcX.d** directory for **Knnxxxx** and **Snnxxxx** script files to run:
    - **nn** - **00** to **99**: defines the run sequence
    - **xxxx** - process short name
    - **K** - indicates process to kill
    - **S** - indicates process to start
  - **Knnxxxx** and **Snnxxxx** are all symbolic links to the actual scripts in the **/etc/rc.d/init.d** directory. The **K** or **S** simply tells the **rc** script what parameter to pass to the script file in **/etc/rc.d/init.d** directory - **start** or **stop**.

Linux Boot Process

- System startup files - **rc** (continued)
  - The order in which each service is stopped or started is critical due to interdependencies between some of them
  - **init** runlevels can be edited manually or by using one of **ntsysv** (text-GUI terminal script), **chkconfig** (terminal text), or the **system-config-services** (GUI) tool to configure services to be started and stopped by runlevel.
  - Both **ntsysv** and **system-config-services** sort services in an appropriate order defined by the dependencies and the system and create required links.
  - You will have to be able to use **chkconfig**.

Linux Boot Process

- **`init`** starts the user login system
- After login, user startup scripts are executed
  - looks in home directory for **`.filename`** scripts, which are user configurable managed scripts for user settings and preferences
- The shell from **`/etc/passwd`** is then spawned with associated user-specific scripts and settings.
  - **`.bash_profile`** and other similar or shell related scripts are executed prior to the shell prompt being displayed
- Others **`.filename`** scripts, such as **`.Xdefaults`**, are run by specific processes when they are started up to determine the user's preferences

Booting Linux

- Several ways of booting Linux on your system
  - booting from a hard disk or USB device
    - compressed kernel image is on the Linux partition, which can be loaded either directly or through a boot loader, such a GRUB
    - kernel loads all required information from hard disk as it loads itself into memory
  - booting from the Linux CD-ROM or DVD
    - The disc contains a compressed kernel image
    - Contains basic configuration files in **`/boot`** and **`/etc`** directories to allow basic system
    - kernel may still point to hard disk (normal boot) or load everything it needs from the CD (Rescue Disk or Live CD/DVD)

Linux rescue disk

- Each Linux distro can create a rescue disk
  - generally not system specific
  - could be CD-ROM, DVD, or USB device
  - contains its own minimalist version of the kernel, key administration tools, and a shell
  - Gives an admin the tools required to fix a broken system without re-installing
  - allows for mounting filesystems on the system with or without entries in the **`/etc/fstab`** file
  - utilities and tools included varies from version to version, but by mounting undamaged existing partitions, installed applications can also be accessed and used

Linux Run Levels

- For the purpose of understanding how **`init`** works, the concept of the runlevel is used.
- They act as a method to define what processes are started / stopped, and what users are capable of doing.
- There are 10 runlevels available (0-9). Fedora currently uses 7 of them (0-6).
- Processes and services are associated with a single runlevel at any one time, but can exist in different runlevels concurrently.
- The system can only exist in one runlevel at a time. The runlevel to use at startup is determined by the **`rc`** script set from the information in the **`/etc/inittab`** file

Linux Run Levels

- Runlevels are being updated/replaced by UpStart, but are still valid (so far)
- Default Runlevel definition for most distros
  - Runlevel 0 - Halt
  - Runlevel 1 - Single user mode
  - Runlevel 2 - Multi-user mode, without NFS
  - Runlevel 3 - Full multi-user mode (no GUI)
  - Runlevel 4 - undefined
  - Runlevel 5 - Full multi-user mode (GUI)
  - Runlevel 6 - Reboot
  - Runlevel 7 - undefined
  - Runlevel 8 - undefined
  - Runlevel 9 - undefined

Linux Scripts

- Shell Scripts
  - ASCII text files containing shell commands, data manipulation and environment settings
  - Interpreted by a command shell (**bash**, **sh**, **csh** and **tcsh**, **ksh**, etc…)
  - Uses typical commands and utilities available on the system, along with shell built-ins
  - Each shell varies in its methods and built-in commands, but the basics remain very similar
  - Designed to perform a set of commands and settings repeatedly and quickly
  - Same commands used in shell scripts could easily be typed in from the command line, but script file acts like an intelligent batch file

Linux Scripts

- Special-purpose language scripts
  - Can also use scripting languages (PERL, Tcl/Tk, Python, etc…) if the interpreter is installed
  - Usually doesn’t need to be compiled, but depends on the language chosen
  - Interpreted by the language interpreter
  - Usually requires stronger / detailed programming skills and understanding than shell scripting
  - All scripts begin with **#!** (hash-bang) in column 1 of the first line, followed with the absolute path to the interpreter, plus options if needed