Boot, Initialization, and Login: Section 1 Transcript

Introduction

Welcome to the Boot, Initialization, and Login Module. The complete boot process includes all the steps from turning the power on to sitting at a login prompt. While we will discuss each portion of the boot process in this module, we will place additional focus on the post-kernel boot process, also known as user space initialization, because this is where a user has significant control over what processes will run.

User space initialization can be accomplished via one of many system initialization methods. Many UNIX variants, such as RHEL or CentOS 5 (and earlier), Debian 7 (and earlier), and Solaris 9 (and earlier), use the System V (often shortened to SysV) initialization method. Modern operating systems use newer system initialization methods, such as the Service Management Facility (or SMF, which is used in Solaris 10 and newer systems), Upstart, and systemd.

We will begin by discussing the startup actions that occur on systems that use the SysV-based system initialization method, since it is widely deployed in UNIX and UNIX-like systems, and forms a solid basis for understanding other system initialization methods.

Additionally, most system initialization methods are in some way backwards compatible with SysV. Specifically, we will focus on how the initialization process starts the system by virtue of the /etc/inittab configuration file and the startup scripts. We will also compare and contrast the different approaches to starting services at boot time that newer system initialization methods employ.

Throughout this module, you'll be presented with opportunities to assess and apply what you've learned.

At the end of this module, you will be able to:

- Describe the boot process from power-on to the login prompt,
- Determine what processes start at boot-time,
- Trace the initialization method through the post-kernel boot process (also known as user space initialization),
- Use tools to examine or modify boot configuration, and
- Analyze boot configuration files.

Prerequisites

As you are traversing through this module, you are expected to use Internet search engines and system `man` pages to discover and learn about a variety of commands. Some of the commands you need to research and become proficient with are: `chkconfig`, `init`, `invoke-rc.d`, `runlevel`, `service`, `svcadm`, `svc.startd`, `svcs`, `systemctl`, `initctl`, `update-rc.d`, and `who`.

Before continuing on, take the time to research the commands displayed - especially any that may be unfamiliar to you. You may print a copy of the commands found in the Resources for this module and/or record data you learn during your research.

Click Next when you are ready to continue.

Bypass Exam Introduction

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

If you do not wish to take the Bypass Exam, you can use the Content Menu to proceed to the next section of the module.

Boot, Initialization, and Login: Section 2 Transcript

Boot Process Introduction 1/6

Knowledge of the boot process is essential to being able to determine how and under what circumstances a process may be started. It is also key to discovering an application's persistence, or how a process starts automatically after a reboot. With the exception of advanced topics not covered in this course, once the system is booted up, there is very little we can do to influence the Basic Input and Output System (BIOS), the newer Unified Extensible Firmware Interface (UEFI), the bootloader, also known as the bootstrap loader or bootstrap, or the running kernel. Let's begin our examination of the boot process with a high-level overview.

Boot Process Overview 2/6

The boot process consists of several stages: Power On and Firmware Initialization, Bootloader, Kernel Initialization, and User Space Initialization. Click each tab to learn about the boot processes displayed. When you have finished reviewing each concept, click the Next Slide button to continue with the module.

Power On and Firmware Initialization: When the system is powered on, the BIOS or UEFI code gets loaded by the processor, and performs a series of system checks known as the Power-On Self Test (POST). The BIOS or UEFI is also responsible for selecting the bootable device. Once the boot device is chosen, the BIOS or UEFI loads and executes the initial bootloader program from the Master Boot Record (MBR) or GUID Partition Table (GPT), respectively. This bootloader is responsible for loading and executing the next stage of the bootloader.

Bootloader: The most common bootloader used in UNIX-like systems is GNU (recursive for GNU's Not UNIX) GRand Unified Bootloader (GRUB). The bootloader typically presents boot options which include choosing a particular kernel to load and sending different options to the kernel. Once the kernel is chosen, the bootloader loads the kernel image into memory and initializes it. The bootloader is also responsible for mounting the initial RAM disk (initrd) image, which serves as a temporary root filesystem.

Kernel Initialization: The kernel is responsible for establishing memory management and checking the processor configuration. It then initializes the kernel threads that enable full system operation. This is accomplished when the bootloader loads the kernel image file along with the initial RAM disk (initrd) into memory. The kernel image is a bit-by-bit copy of how system memory should look at the beginning of kernel operation. The initrd is used by the kernel as a temporary root file system until the kernel is fully initialized and the persistent root file system is mounted. It contains a minimal set of directories and executables needed in order to mount the persistent root filesystem including the necessary drivers which allow the kernel to access the hard drive partitions, and other hardware. After kernel initialization, the kernel will then create the first user space process on the system. The execution of this process ends the initial boot sequence and begins the post-kernel boot process; which is responsible for getting the system into an operational state.

User Space Initialization: Once the kernel is initialized, the user space initialization process begins. This stage is also called the post-kernel boot process. All other events prior to this point can be referred to as the initial boot sequence. There are several different system initialization methods that can be used for user space initialization. However, the process for all methods is similar from a high level. First, the initial user space process started by the kernel is responsible for setting up important operating system functions, such as providing console interfaces and setting up default signal handlers for processes. This process is also used for starting up all application services and daemons. The initial process accomplishes user space initialization by reading various configuration files and then executing actions in accordance with those configurations. This includes basic system configurations as well as starting any processes and services required for a stable, functional, and operational system. Once user space initialization is complete, a login shell becomes available.

Power-on and Firmware Initialization

Now that you know the fundamentals of the UNIX boot process, let's examine the process in more detail.

At power-on, the computer loads the firmware interface (BIOS or UEFI), which is traditionally stored on a Read-Only Memory (ROM) chip, though flash memory is more common today. The firmware interface first runs through hardware checks, known as POST (Power On Self-Test). Assuming no hardware errors are detected, the firmware interface locates available boot sectors on any bootable devices. When it finds the boot record on a storage device, the bootloader code is then loaded onto memory. Control is then passed to the code that was loaded from the boot record.

The boot loader contained within the boot record points to a location on the boot device (defined by a physical address) where the boot loader code is loaded and executed. The code loaded from the boot record is usually called the first stage of the bootloader. The code that is loaded by the Stage 1 bootloader varies, depending on which bootloader is used.

The Bootloader

Before being capable of loading the kernel into Random-Access Memory (RAM) and running it, the bootloader must be used. Bootloaders can often be configured to have multiple operating system choices, such as Windows and Linux. Additionally, they can be configured to boot from several different kernel versions on the same system. The most commonly deployed bootloader for UNIX-like systems is the GRand Unified Bootloader (GRUB). Even newer versions of Solaris, including Solaris 10, use GRUB. Most multi-boot scenarios use GRUB; however, there are numerous types of bootloaders that exist. The primary function of a bootloader is to get the kernel loaded into memory and running.

The Multi-stage Approach to Bootloading

Modern systems usually use a "chained" or "multi-stage" approach to bootloading; generally a 2- or 3-stage approach. As mentioned earlier, the bootloader code within the boot record is just smart enough to load a larger bootloader into memory. Subsequent stages allow the system to increase the complexity of code and applications which can be run.

For example, in GRUB, the Stage 1 bootloader executes a small program that is primarily responsible for loading the Stage 1.5 bootloader. A Stage 1.5 bootloader usually provides filesystem support, which allows the bootloader to read from the boot device by pathname, instead of by physical address. Stage 1.5 then loads the Stage 2 bootloader, which contains the bulk of the bootloader code, to include that which allows users to select a kernel and kernel boot options.

The primary function of Stage 2 is to locate and load the kernel into RAM and hand off control to the kernel. Once the kernel image is loaded into memory, control is then handed over to the kernel to begin user space initialization.

Older versions of Solaris prior to Solaris 10 1/06 release use a different system, but the steps are familiar. Like GRUB, Solaris installs a Stage 1 bootloader in the boot record called `mboot`, short for master boot. The `mboot` program loads and runs a program called `pboot` (primary boot), which subsequently invokes the `bootblk` program. Similar to Stage 2 GRUB, the `bootblk` program provides the user with the opportunity to select kernel and boot options.

It is important to note that because the bootloader and its configuration files live on a machine's filesystem, they can be changed either accidentally or maliciously.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Boot, Initialization, and Login: Section 3 Transcript

Kernel Initialization

Once the bootloader loads the appropriate kernel image into memory, it passes execution to the kernel code. The sequence of events thereafter can vary between different kernel versions. Generically speaking, the kernel initialization process starts with the kernel loader initiating a couple of preparatory steps before loading the kernel proper -- some consider this a bootloader phase. During this phase, the kernel loader establishes memory management, interrogates the Central Processing Unit (CPU), then passes execution to the kernel code, which finishes the kernel initialization procedure.

The location of the kernel image varies depending on the vendor. Some typical locations are `/kernel` or `/platform` on Solaris systems, and `/boot` on Linux systems. The kernel image files on Linux systems are usually identifiable by the file name beginning with `vmlinuz`. These images are in a self-extracting, compressed format.

Once the kernel extracts itself, it then loads and mounts the initial root filesystem. This temporary filesystem is known as either the initrd (initial RAM disk) or initramfs (initial RAM filesystem).

The initrd image is a RAM-based block device, called a ramdev, which contains an image of a filesystem. In order to use this filesystem, the kernel must have a driver loaded that is compatible with whatever filesystem type was used to format the initrd device.

The initramfs is a compressed archive file and is uncompressed then mounted directly into the kernel as a tmpfs filesystem. As such, it does not need a driver, it is always on-kernel.

The initrd is the old way of implementing the initial RAM disk whereas initramfs is the newer method. However, both initrd and initramfs are often both still referred to as initrd. Regardless of which is used, a temp filesystem is required in order to load the kernel modules and drivers needed to make the persistent filesystems available. It contains a minimal set of directories and executables needed in order to mount the root filesystem. The temp filesystem image is typically located within the `/boot` directory alongside the kernel image.

Kernel Threads

The main function of the kernel is to provide a transparent interface that allows processes access to physical resources on a computer. This is done via a system call layer that processes can use to request system resources through the kernel. However, the kernel also periodically performs housekeeping operations that do not require any request from usermode processes. All of these functions are done through kernel threads.

A kernel thread is somewhat similar to a process except that it resides entirely in kernel memory. A good example of a kernel thread is the scheduler `(sched)` kernel thread in Solaris 10, which is also known as the swapper. The purpose of the swapper is to share the processor among all

processes.

A single CPU core can only process one thing at a time. However, several processes may need to use the CPU simultaneously. Because of this, the swapper manages which processes get CPU time. To do so, the swapper allocates each process a small quantum of time to execute; this period of time is called a slice. The swapper then uses an algorithm based on process priority to determine what process to run next, which may sometimes lead to swapping out lower priority processes for higher priority processes.

At boot time, the Solaris kernel initialization code creates the swapper and assigns it a Process Identifier (PID) of zero (0). It then creates the `init` process and assigns it a PID of 1 and a Parent Process Identifier (PPID) of zero (0), as if it was created from the swapper. All other subsequent kernel threads are created from the swapper and consequently receive a PPID that matches the PID of the swapper. Some commonly seen kernel threads on Solaris include `pageout` and `fsflush`, which you will notice have PPIDs of 0.

In many Linux systems, the swapper doesn't exist as a kernel thread, however the kernel initialization code creates the `init` process with a PID of 1 and a PPID of 0. The kernel also creates the kernel thread daemon (commonly seen in the process list as `kthread` on older Linux kernels or `kthreadd` on kernels above 2.6.32). The kernel thread daemon is used to create all other kernel threads. This is why all other kernel threads appear as children of the kernel thread daemon. Nonetheless, in both examples, the init process is used as a point of origin for the creation of all processes in the system using the *fork()* and *exec()* system calls.

Section Completed

Boot, Initialization, and Login: Section 4 Transcript

User Space Initialization: Introduction

The post-kernel boot process begins once the initial boot sequence is complete and the kernel has been initialized. The post-kernel boot process, or user space initialization, brings the system into an operational state with all required services and processes running.

There are several different methods used to accomplish user space initialization. These methods are commonly referred to as init systems, system initialization methods, initialization systems, or startup methods. System initialization methods are sometimes be referred to as "init systems"; however, in this context, the word init is short for initialization. An "init system" is not to be confused with the process named init. This is an important distinction because, although most init systems use a process named init for the first user space process, some system initialization methods do not. Either way, they can still be referred to as "init systems".

Despite being referred to as system initialization methods, their role in the boot process only comes into play after kernel initialization. Their primary function is to startup all required user space processes. The method used depends on the operating system and distribution. The most commonly used methods are: SysV, Upstart, systemd, and SMF.

There are several different methods you can use to determine which system initialization method a system is using. However, some methods don't always work on all distributions due to differences in implementation. Additionally, since SysV compatibility is built into almost all methods, it can be confusing to identify which is being used.

Let's discuss the differences between each method and how to tell them apart. One of the quickest and more reliable ways of determining which method may be used is to simply look at the man page for init. The init command is present on SysV, Upstart, systemd, as well as SMF. However, it is provided to the system via different packages that are related to the system initialization method being used so the description of init in the man page will differ depending on which is being used. Let's look at some examples.

User Space Initialization: Examples

Click each tab to learn about the man pages for each system initialization method. Note that each method is different and gives a good indication of what is being used. The SysV and SMF man pages for `init` look very similar. However, if you are on a Solaris 10+ system, you can assume it is using SMF. So that distinction should be clear.

When you have finished reviewing each method, click the Next Slide button to continue with the module.

SysV: Introduction

SysV is one of the oldest system initialization methods among modern UNIX-like systems. SysV uses a serial initialization method which starts processes and services one after the other. Although newer methods have been developed to replace SysV, many systems still utilize it. Most of those that do not use SysV, generally have some form of backwards compatibility built in.

Before we begin, it's worth noting that the file locations mentioned within this section are based on Red Hat based implementation of SysV. Other distributions still function in a very similar way but all file locations may or may not be the same. Refer to the Resources for major differences in implementation between different distributions.

On SysV systems, `init` is the first user space process with a PID of 1. As the first user space process, `init` becomes the ancestor of all other processes that ultimately run. In other words, since all processes are spawned from `init`, or one of its descendants, all other processes have a parent. Therefore, when chaining processes, they can always be chained back to `init`.

The `/etc/rc.d` directory contains the startup scripts used by `init`, including those for user space initialization, local customization, and for the user space services that the system provides. Some are basically operating system services that run in user space, such as the iptables firewall and the Common UNIX Printing System (CUPS) printer server. Others are application services that the machine is providing, such as web server and mail server services. Let's review the `/etc/rc.d` directory and the files and scripts within it.

SysV: /etc/rc.d directory: (Part One)

The first script of interest is the `/etc/rc.d/rc.sysinit` script. This script contains configuration items to be run before other services and typically includes setting the time zone and hostname, mounting filesystems, and other such tasks.

Next, notice the `/etc/rc.d/rc.local` script, which configures items to be run after all services are up. These scripts are typically used for system specific configurations, such as Virtual Private Network (VPN) tunnels or custom scripts run by system administrators.

The `/etc/rc.d/init.d` subdirectory contains the actual startup scripts used to control services. Each script in this directory must implement start and stop routines; however, many offer additional functionality, such as status and restart.

On most Linux systems, there is a single `/etc/rc.d/rc` (i.e. run command) script, which takes the desired runlevel as its sole argument. Other systems may have one script for each runlevel, usually called `rcX` (where X equals the runlevel). In either case, this script is responsible for examining the contents of the `/etc/rc.d/rcX.d` directory. The `rc` script is responsible for starting and stopping services according to the configuration in the `rcX.d` directory.

The `/etc/rc.d/rcX.d` (or `rc0.d` through `rc6.d`) directories define which services should be running in the associated runlevel. Let's examine one of these directories in more detail.

SysV: /etc/rc.d directory: (Part Two)

The contents of the `rc0.d` through `rc6.d` directories are usually symbolic links to scripts in the

`/etc/rc.d/init.d` directory. The naming convention of these files is important! The first character is a `K` (for kill) or an `S` (for start). These characters tell `rc` whether or not the service should be running in that runlevel. Services are only started, or killed, if their status needs to change. So if a service isn't already running, the kill script will not be executed. Similarly, if a service is already running, the start script will not be executed.

Keep in mind that since there are no services running when a system first starts, kill scripts generally will not run at boot time. Also, it's important to note that when moving from one runlevel to another, start scripts will not run until all necessary kill scripts have been executed. For each kill script, `rc` invokes the script with a `stop` argument; start scripts are invoked with a `start` argument.

Additionally, notice the second and third characters of each script. These represent numerical priority; required actions with a lower priority are run first. Any entries with the same action and priority are executed in alphabetical order.

SysV: /etc/inittab

Let's explore some lines of the `/etc/inittab` configuration file used by SysV systems. While it may not be obvious at first glance, each line of the `/etc/inittab` file consists of four fields - identification (ID), runlevel, action, and the program+options fields.

Click each icon displayed to learn more about each specific field. When you have finished exploring, click the Next Slide button to continue with the module.

Identification (ID) Field: The ID field is comprised of one to four alphanumeric characters. Some ID values have special meaning, but generally they are arbitrarily chosen. Each ID is used to uniquely identify the line within the internal processing of `init`'s process list. These should be unique. When a duplicate ID is used, system behavior can be very unpredictable. For example, you may find that you are unable to change runlevels successfully during system operation. In this example, `id` stands for initdefault and `si` stands for system initialization.

Runlevel Field: The `init` process determines which set of scripts to run based upon a runlevel. A runlevel is a single digit (or the single letter S on some systems) that is associated with a specific operating state on a UNIX OS. Whenever `init` runs, it is either given a runlevel as an argument, or it reads a default runlevel from the `/etc/inittab` file. The init process compares this runlevel to the runlevel field of each line of `/etc/inittab`, and performs the action in that line if there is a match. The runlevel field may contain more than one runlevel, and if the runlevel field is left blank, `init` will perform the associated action in all runlevels.

Action Field: The action field consists of a keyword that tells `init` how it should interpret the information on the line and may indicate the way in which `init` should execute the associated program. There are many different action values that can be used.

Program Field: The program field contains the path and name of the executable that should be run, along with all the command line options that should be used.

SysV: /etc/inittab Lines: (Part One)

The `init` process reads the `/etc/inittab` file which determines which runlevel to initialize the system into. Depending on the runlevel, a series of scripts are executed in order to complete the initialization process. The first script executed is the `rc.sysinit` script, which completes several system initialization tasks.

Let's examine some `/etc/inittab` lines in more detail. Notice that in the first line, the ID field is id, the runlevel is set to the default value of 3, and the special action keyword is `initdefault`, which tells `init` the runlevel it should change to when booting. In this example, the system will boot into the initial runlevel three (3). The program part of the line is empty since there is no action for `init` to take. So this line does not start any processes.

Now let's examine the second line. Notice that the runlevel field is empty. This is because the special action for this line is `sysinit`, which indicates that this line should be executed during the system's initial boot sequence, regardless of runlevel settings. The `sysinit` lines will be read and executed before all other lines, and `init` will wait for the program to finish execution before moving to the next line. Keep in mind that `sysinit` lines only execute during initial boot; if the user manually changes runlevels with the `init` command, the `sysinit` lines will not run again. The program field in this line contains the full path of the executable plus command line arguments. In this case, `init` will execute the command `/etc/rc.d/rc.sysinit` at system boot.

Seems pretty simple, right? Let's explore some more `/etc/inittab` lines to become familiar with some other common special action keywords.

SysV: /etc/inittab Lines: (Part Two)

Notice that the next seven lines look very similar in nature. Again, the runlevels of each determine when a particular line gets processed and the program executed. In this case, in the third line, entering runlevel two (2) will cause the line to execute. The fourth and sixth lines will execute when entering runlevel three (3) and runlevel five (5), respectively. These lines use the action keyword `wait`, which indicates that `init` should wait for the corresponding program to finish execution before continuing to the next line. Additionally, the program field is filled in with the path of the executable plus the respective command line argument. In this case, `init` will execute the command `/etc/rc.d/rc` with the desired runlevel as an argument, an operation we will examine shortly.

Further down in the file, notice that there are multiple numbers listed in the runlevel field. This means that the command will execute when entering any of the runlevels listed in this field, in this case runlevels two (2), three (3), four (4), and five (5). Notice that the special action keyword is respawn, which tells `init` that it should continually monitor the process once it is executed and respawn a new copy of the program should it ever terminate. In this particular line, `init` will execute the command `/sbin/mingetty` to the terminal tty1. These lines start up terminals for virtual consoles on the system used for login.

While we have not covered all the action keywords that `init` can use, `initdefault`, `sysinit`, `wait`, and `respawn` are the ones seen most often.

SysV: Runlevels

Once `rc.sysinit` has been executed, init runs `/etc/rc.d/rc`, passing it one of 7 different runlevels depending on what was configured in the `/etc/inittab` file. As previously discussed, runlevels drive program execution after the system initialization scripts, which are common to all runlevels, are executed at boot. Some lines in `/etc/inittab` get processed regardless of the runlevel. Other lines get processed only in specific runlevels. Each UNIX-like operating system, such as Linux, has its own way of doing things. Displayed are the traditional runlevels utilized by most init-based systems. The runlevels displayed show how they are implemented by UNIX variants. The exact meaning of each runlevel differs among different distros.

When the `/etc/rc.d/rc` script is executed for the configured runlevel on boot, all of the scripts which start with the letter `S` in the corresponding `/etc/rc.d/rcX.d` directory are executed with the parameter `start`. If the system is changed from one runlevel to another after boot then all the scripts that start with the letter `K` are executed with the argument `stop` before executing the start scripts. Following the `S` or `K` in each script name is also a number, (i.e. `S55sshd`). This number indicates the sequential order in which to stop or start the scripts for that runlevel. The `S` and `K` scripts exist to support changing from one runlevel to another on the fly after the system has already been initialized.

SysV: Configuring Runlevel at Boot Time

It's important to note that while an operating system typically boots into the default runlevel, a user with access to the console can configure the specific runlevel at boot time by modifying the kernel parameters in the bootloader. In addition, a super-user may switch a running system to a new runlevel using the `init` command. The new `init` process must re-read its configuration file, terminate certain processes, and launch the appropriate programs for the new runlevel.

This being said, the new process does not replace the first `init` with a PID of 1. It terminates when the new runlevel is established. To view the current runlevel, simply use the command `who -r`. Note that on some Linux versions, the command `runlevel` can be used to view the current runlevel.

SysV: Management Tools

System V primarily uses the `service` and `chkconfig` commands for management. You can learn more about these commands by visiting following the man pages:

```
man 8 service
man 8 chkconfig
```

Why SysV needed to be replaced

We have now covered the SysV initialization system in great depth. SysV has been the standard across many UNIX variants dating back to its release in 1983. However, the need to replace it with something better has been discussed within the UNIX community for quite some time. Since this discussion began, several alternatives have been developed, including some that have become the SysV replacement for various distributions. So why was there a need to replace SysV?

As discussed, the SysV initialization process is serial, meaning that one task starts after the last task has successfully started. The issue with this is that sometimes it can result in a longer than usual boot time if an error occurs with even a single task. Additionally, the SysV `rc script` model was not a very straightforward and efficient design and it lacked some standardization. Essentially, it was just a bunch of shell scripts which could be written however an application developer chose.

The majority of the UNIX community desired a simpler user space initialization process which could handle parallel processing of tasks, had a more standardized design, and an improved system to handle service dependencies. Several different system initialization methods were developed in an attempt to respond to these concerns. Of those, Upstart, SMF, and systemd were among the most successful. Although the three are wildly different, one commonality among them all is their ability to perform parallel processing. In the next sections, we will cover these three parallel system initialization methods in more depth.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Boot, Initialization, and Login: Section 5 Transcript

Upstart

As mentioned, prior to Upstart, most UNIX systems used a startup mechanism which started services in sequence. Upstart was developed as a replacement for SysV which can run initialization steps in parallel or simultaneously.

Before we begin, like with SysV, it's worth noting that the file locations mentioned within this section are based on Red Hat based implementation of Upstart. Other distributions still function in a very similar way but all file locations may or may not be the same. Please refer to the resources for major differences in implementation between different distributions.

A system using Upstart, at first glance, can appear very similar to a SysV system due to their commonalities and Upstart's backward compatibility with SysV. However, there are major differences that set them apart.

On Upstart systems, like SysV systems, after the kernel initialization is complete, the `/sbin/init` process is started with a PID of 1.

The Upstart version of init will then read a series of files from `/etc/init` and its subdirectories. Each file within, whose name typically ends with `.conf`, defines a service for the system. The format of the Upstart scripts is also more complicated than a simple SysV `rc` shell script. The `.conf` files specify different triggers which will cause the service to be started or stopped as well as what executable to run in order to start the service.

One of the `.conf` scripts actually runs the `/etc/rc.d/rc` script with the desired runlevel. It then executes all of the `/etc/rc.d/rcX.d` scripts supported by the traditional SysV initialization method. This exists for backwards compatibility and supports services for which no Upstart scripts exist.

Upstart: Management Tools

Upstart utilizes a variety of commands depending on the distribution. Generally, the commands used to manage an Upstart system are `initctl` and `update-rc.d`. To learn more about these commands, refer to the following man pages:

man 8 initctl

man 8 update-rc.d

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting

your own Internet research.

Boot, Initialization, and Login: Section 6 Transcript

systemd: Introduction (Part One)

The systemd initialization method is one of the newest methods used by most modern Linux distributions.

The development of systemd, just as Upstart, aimed to address the issues with the SysV initialization method. Although Upstart's approach to system initialization solves many problems related to dependencies, as well as starting processes in parallel, it does so with a "greedy event-based" model. Basically, it tries to start all services and processes as soon as possible based on events. Upstart rushes to start as many processes as it can, as quickly as it can even though some services may not be necessary. Upstart will start all enabled services at boot even if they are not being utilized. Arguably, starting all available services and daemons at boot is wasteful.

Like Upstart, the issues with dependencies and starting services in parallel are solved with systemd. However, systemd only starts a process if at least one other process requires it. This approach means that services that are rarely used will not be started until a request is made for it, at which time it will start on demand. Upstart had no real method of determining what services were frequently used and which were used less often. For this reason, most systems that used Upstart in the past, including Canonical, the original developers of Upstart, have made the switch to systemd.

Newer releases of Debian, Red Hat, and their forks and variants have moved away from SysV and Upstart and now use systemd.

Now let's discuss how systemd works.

systemd: Introduction (Part Two)

Where SysV initialization is comprised entirely of `rc` bash scripts and Upstart is a combination of configuration files and scripts, systemd initialization is accomplished entirely by configuration files. Still, systemd supports legacy compatibility with SysV `rc` scripts in the typical `/etc/rc.d/rcX.d` directories if necessary.

Like Upstart, systemd was designed to run initialization steps in parallel versus serial. Its design allows for incredibly fast boot times and implements a robust solution for solving many dependency issues among systemd units.

After kernel initialization, the first user space process is `/usr/lib/systemd/systemd` with a PID of 1. On some distributions, however, the first user space process appears as `/sbin/init`, which is actually a symbolic link to `/usr/lib/systemd/systemd`.

The systemd process reads a series of configuration files, referred to as units, from the directories `/etc/systemd/system` and `/usr/lib/systemd/system`. There are several different types of units such as `service`, `socket` and `target`. The type of unit can be determined by the file

suffix, for example: `.service`, `.socket`, or `.target`. A service unit is similar to a typical SysV or Upstart service. Service units are system daemons that run in the background and provide services such as the Apache `httpd.service` or OpenSSH `sshd.service`.

systemd: Targets

Traditional runlevels, as implemented by SysV, are replaced by targets in systemd. There is no direct comparison of runlevels to targets as far as system initialization is concerned. Each target has its own directory in `/etc/systemd/system` and each of those directories contain a configuration file used to start each desired service, which differ from the familiar startup scripts in SysV.

systemd uses an internal dependency hierarchy and transitions through a number of targets during system initialization before reaching its end state. For example, to boot into the `graphical.target` as the default target, `systemd` will traverse several other targets, including, but not limited to: `local-fs-pre.target`, `local-fs.target`, `sysinit.target`, `basic.target`, and `multi-user.target` prior to reaching the `graphical.target`. Think of each target as a checkpoint along the way to the final destination -- the default target. This includes loading all units related to each individual target prior to reaching and executing units related to the default target.

Although there are significant differences in the system initialization process between SysV and systemd, the end state is similar enough. At a high level, traditional runlevels can be compared to targets as follows.

In systemd, the `/etc/inittab file` is not used to set the default target. Instead, a symbolic link named `/etc/systemd/system/default.target` points to the desired target in `/usr/lib/systemd/system/*.target` (where `*.target` is equal to one of the targets displayed), effectively setting the default target to boot into.

Now let's take a look at systemd unit types.

systemd: Unit Files

In systemd, services are started during boot based on service unit configuration files. However, not all systemd units are service units. There are several types of systemd units that may be read during boot. There are two primary locations where these unit configuration files reside. The first, `/etc/systemd/system`, takes precedence over `/usr/lib/systemd/system`. That is to say, if two unit files with the same name exist in both locations, it will load the one in `/etc` versus `/usr/lib`. Most of the unit files in `/etc/systemd/system` are simply symbolic links to unit files within `/usr/lib/systemd/system`.

For a complete list of systemd unit files, see Resources.

systemd Units: Current State

There are several keywords that indicate the current state of a unit. The basic categories of statuses are `Load`, `Active`, and `Sub`. Within these categories, there are various states that a unit can have.

Click on the tabs displayed to learn more about each status in more detail and then click the Next Slide button to continue with the module.

`Load`: To get a list of all unit files and their current statuses you can use the `systemctl list-unit-files` or `systemctl list-units --all` command. The most common Load states of a unit are usually `enabled`, `disabled` or `static`. The `static` state means that the unit does not have an `[Install]` section in its configuration file. The `[Install]` section must be present in order for a unit to be `enabled`. Because of this, `static` units cannot be `enabled`.

However, that does not mean that it will not be started on boot. The `static` state usually means that the particular unit will perform a one-off action or may be used only as a dependency of another unit and is not typically run independently. It may also be triggered to start based on a condition or event. For instance, a `socket` unit may have an open network socket on the system and when a client attempts to connect to that socket, it will then start up the `static` unit to handle the request. This is effectively how the `systemd-journald` service functions.

`Active`: To see a list of all active units that `systemd` knows about, you can use the `list-units` argument with `systemctl`. The `list-units` argument displays units systemd has attempted to load into memory. Since `systemd` only reads units it thinks it needs, this list will not necessarily include all of the existing unit files on the system. To see a list of all available units within the `systemd` paths including those that `systemd` did not attempt to load, you must use the `list-unit-files` argument instead.

`Sub`: To see the sub activation states of `systemd` units, you can use the command `systemctl list-units`. Both the active and sub states combined is what describes a corresponding unit's activation state. For instance, if you were to run `systemctl status sshd` on a system who has sshd enabled and running, it is typical to see an activation state as shown.

systemd Units: Configuring

When a systemd unit is `enabled` to start on boot, the `systemctl enable <service>` command essentially creates a symbolic link located in `/etc/systemd/system` pointing to the appropriate configuration file in `/usr/lib/systemd/system`. Once the symlink has been created, it is considered `enabled`. The enabling of systemd units is orthogonal; meaning units can be `enabled` without being started and started without being `enabled`.

As discussed, unit files without an `[Install]` section are not meant to be `enabled`. Reasons for these types of units could be:

1. A unit may be statically `enabled` by being symlinked to another unit's `.wants/` or `.requires/` directory.
2. The purpose of the unit may be to act as a helper for some other unit which has a requirement dependency on it.
3. A unit may be started when needed via some form of trigger (socket, path, timer, D-Bus, udev, scripted systemctl call, etc.).

To add a custom service to a system, all you have to do is create a service unit file within

`/usr/lib/systemd/system` and give it, at a minimum, the following section:

```
/usr/lib/systemd/system/my.service:
    [Service]
    ExecStart=/path/to/file
```

The `ExecStart=` directive tells systemd what to execute when the service is started. To enable this unit to start automatically on boot, simply add the following section to the unit file:

```
/usr/lib/systemd/system/my.service:
    ...
    [Install]
    WantedBy=multi-user.target
```

Setting the `WantedBy=` directive to multi-user.target tells systemd to create a symbolic link to this unit file in the `/etc/systemd/system/multi-user.target.wants` directory. During system initialization, when it reaches the multi-user.target, it will start all services that have a symbolic link created within that directory.

Once these four lines are added to a unit file, you can manage it just as you would any other systemd service. Yes, it really is that easy! You can effectively use this method to start any process you want on boot. In order to make the service available, you must first run `systemctl daemon-reload` to make systemd aware of the new unit. Then you can start or enable that service using the `systemctl` command. To quickly determine what units have been modified on a system, you can run the `systemd-delta` command.

systemd Units: Dependencies

Units can also be configured to have dependencies. This means that the unit in question will not start unless other units it is dependent upon have already been started. For instance, you may not want a file sharing service started before networking has been started. To view a unit's dependencies, you can run the `systemctl list-dependencies` command on the unit in question. This will show you what units must be started if the specified unit is started (i.e. units that the unit in question is dependent upon). To see what units need to have the specified unit started in order for them to be started (i.e. units that are dependent upon the unit in question), simply add `--reverse` to the end of the command.

Unlike SysV, there is no strict order in which all units are started on boot; however, there is a structure based on dependencies. The direction or starting point that systemd takes to determine what to start on boot is based on the file referenced by the `/etc/systemd/system/default.target` symbol. A quick way to determine this is to run: `systemctl get-default`.

The systemd initialization method handles dependencies and startup order by using several different directives in the unit configuration files. Some of these directives are displayed.

Additionally, if no `Wants=` directive is present but a `/etc/systemd/system/<name>.target.wants` directory exists for a given target, it will treat

each unit file within the `.wants` directory as if it were in a `Wants=` directive and start them accordingly.

systemd: Scenario

To understand what targets, services, and other units start up with the default target, it helps to work backwards just as systemd does in order to build a dependency tree. In this example, we are working with a RHEL 7 system using systemd. In this scenario, assume the `default.target` points to the `graphical.target`.

Click each tab in the order displayed to go through the scenario. When you have completed the scenario, click the Next Slide button to continue with the module.

systemd: Management Tools

Although complicated in its own ways, `systemd` got it right by creating a single command used to manage just about everything related to managing a `systemd` system. The options and arguments can be long, but the command supports a very robust tab completion system. This means entire options do not have to be typed out (similar to a Cisco IOS environment) as long as the characters that are typed are unique. You can also double tap tab at any point to quickly get a list of available options without having to consult the man page. The command is called `systemctl`.

When managing units with the `systemctl` command, if the type of unit is not specified, then it is assumed that the unit type is a service. For example, `systemctl status httpd` is the same as `systemctl status httpd.service`. To learn more about the `systemctl`, refer to the following man page:

```
man 1 systemctl
```

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Boot, Initialization, and Login: Section 7 Transcript

Service Management Facility (SMF)

In the mid-2000s, Solaris moved away from SysV initialization in favor of the Service Management Facility (SMF). Like Upstart and systemd, SMF services can be configured to start in parallel and resolve dependencies.

Dependency problems are avoided by SMF starting dummy services that act like the real services while the real services start up. This means that if `service X` opens a network socket, SMF will open the socket while `service X` starts up. Additionally, if service Y needs service X running as part of its startup checks, a dummy `service X` will make it look like service X is running even if it is not running yet. This prevents any delays in system initialization.

You will also notice that restarting of processes for SMF-based systems, such as Solaris 10, is accomplished using the `svc.startd` process rather than the `init` command used in SysV systems. Another difference is that SMF has milestones, which are more similar to systemd targets than runlevels. Milestones are generally thought of as checkpoints along the path a system takes from kernel initialization to the system being fully operational. Like Upstart and systemd, SMF supports legacy compatibility with SysV rc scripts in `/etc/rcX.d` directories.

When the system boots, the first process that gets started is `sched` with a PID of 0. It then executes `/sbin/init`, which has a PID of 1. All subsequent processes started during the system initialization process are children of `init`. First, the `init` process reads `/etc/inittab` (although it is not used in the same manner as it is on SysV and Upstart systems). The `/etc/inittab` file includes entries that it needs to provide to the SMF `svc.startd` daemon, which is responsible for starting all SMF services.

In addition to `svc.startd`, the `svc.configd` daemon is also started by `init`. This is the SMF configuration daemon and it is responsible for managing access to the SMF repository. The repository is a database on the system that is located at `/etc/svc/repository.db`. It maintains configuration information for all services, which will be needed by `svc.startd` during the system initialization. The configuration information contained within this database is not limited to service configurations. It also contains the default milestone the system is configured to boot into. The milestones and the runlevels to which they relate are as follows.

The appropriate services for the desired milestone are started in parallel until the system finishes system initialization. Once complete, the user is able to log in.

Some services are not managed by SMF directly. Rather, they are managed by `inetd` (Internet Service Daemon) which is similar in functionality to the newer `xinetd` daemon found on many Linux distributions. `inetd` is a delegated daemon for `inet` services (such as telnet) which helps manage `inet` specific properties associated with those services. It listens for requests from connecting clients on behalf of the services it manages. Client requests are then served by executing the appropriate process related to that service. These `inet` services are managed using

the `inetadm` command.

Based on the milestone that the system is configured to boot into, the appropriate services will be started in parallel until the system finishes system initialization. Once complete, the user will be able to log in.

SMF: Management Tools

SMF is probably the most unique of the four methods covered in this section. However, that's to be expected by traditional UNIX systems. There are several commands used to manage SMF but the two primary commands are `svcadm` and `svcs`. To learn more about these commands, refer to the following man pages:

```
man 1M svcadm
man 1 svcs
```

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Boot, Initialization, and Login: Section 8 Transcript

Summary

You have completed the Boot, Initialization, and Login module.

You should now be able to:

- Describe the boot process from power-on to the login prompt,
- Determine what processes start at boot time,
- Trace the initialization method through the post-kernel boot process (also known as user space initialization),
- Use tools to examine or modify boot configuration, and
- Analyze boot configuration files

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Section button to begin the Module Exam.