Kernel Basics and Configuration Files: Section 1 Transcript

Introduction

1/3

Welcome to the Kernel Basics and Configuration Files module. The emergence of personal computers has led to the current computing landscape we know today. However, the very first computer wasn't equipped with an operating system. Each program was self-contained. In other words, each system had to execute all the operations and individually handle input and outputs. Furthermore, only one program could run until completion before another one could be loaded into the machine. A more usable solution was needed.

Eventually, system kernels that could execute multiple programs concurrently and perform all input and output operations on their behalf were developed. Due to the design approach used during its development, UNIX became a front runner of all the operating systems created at this point in time. During this module, we will learn what makes UNIX so successful by discussing a few of the mechanisms implemented in the UNIX kernel and its descendants. We will also discuss some architectural details of the UNIX kernel while setting our emphasis on the very important concept of system calls. Finally, we will use some tools made available to trace system calls.

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:

- Differentiate between UNIX and UNIX-like Operating Systems,
- Define the kernel,
- Differentiate between kernel space and user space,
- Interpret `strace` and `truss` utility outputs to determine system calls used by programs,
- Describe UNIX configuration mechanisms, including system and kernel configuration files, and
- Explain how to load and unload kernel modules.

Prerequisites

2/3

As you are traversing through this module, you are also expected to use Internet search engines and UNIX `man` pages to discover and learn about a variety of UNIX system calls and commands. Some of the system calls that you will need to research and become proficient with are: `chmod()`, `chown()`, `close()`, `execve()`, `free()`, `malloc()`, `open()`, `read()`, `recv()`, `send()`, `socket()`, and `write()`. In addition, some of the commands that you will need to research and become proficient with are: `chmod`, `chown`, `dmesg`, `insmod`, `lsmod`, `make`, `modinfo`, `modprobe`, `pmap`, `strace`, `sysctl`, `truss`, and `uname`.

Before continuing on, take the time to research the UNIX system calls and commands displayed - especially any that may be unfamiliar to you. You may print a copy of the system calls and commands (see Resources on the toolbar) and/or use the "Notes" option on the toolbar to record data you learn during your research.

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.

Kernel Basics and Configuration Files: Section 2 Transcript

Evolution of UNIX

Introduction: The rich history of UNIX (shortened from Uniplex Information and Computing System - UNICS) makes it difficult to define a term that breathes life into so many vital aspects of everyday living - telephone calls, the Internet, eCommerce, computer aided design, manufacturing control systems, and so much more. The best way to gain an appreciation of this complexity is to briefly examine the evolution of UNIX. Click another button on the timeline to continue learning about the evolution of UNIX. If you have finished reviewing all sections of the timeline, click the Next Slide button in the navigation bar to continue with the module.

Early UNIX Versions: UNIX evolved from a series of AT&T/Bell Labs Operating Systems developed in the late 1960's and 1970's. The original UNICS and the Seventh-Edition were notable milestones in this evolution. Why? Simply put, due to an antitrust lawsuit which precluded AT&T from entering the computer industry, AT&T began to freely distribute its early editions of UNIX with the source code, which was beneficial for many reasons, including customization and academic pursuits. Click another button on the timeline to continue learning about the evolution of UNIX. If you have finished reviewing all sections of the timeline, click the Next Slide button in the navigation bar to continue with the module.

Post-Commercialization: Once the restrictions against AT&T were removed in 1983 in the landmark decree that broke up Bell Telephone Systems, AT&T began selling the UNIX System V, commonly referred to as SYSV. Meanwhile, researchers at the University of California, Berkeley developed the Berkeley Standard Distribution (BSD) version of UNIX, based on the original AT&T source code. Unfortunately, SYSV and BSD were not compatible. In other words, users could not run code written for one on the other. This created confusion and discord in the market place. Click another button on the timeline to continue learning about the evolution of UNIX. If you have finished reviewing all sections of the timeline, click the Next Slide button in the navigation bar to continue with the module.

POSIX: Realizing this market discord was untenable, an Institute of Electrical and Electronics Engineers (IEEE) working group created a compromised set of standards to uniformly define the product - the Portable Operating System Interface (POSIX). From an operational perspective, POSIX, and its successor the "Single UNIX Specification," created a great deal of overlap in the basic command line utilities across UNIX variants. The most widely deployed UNIX operating systems using the POSIX standards, or Single UNIX Specifications, are the SYSV descendants and include: Solaris (developed by Sun Microsystems and now owned by Oracle), AIX (owned by IBM), HP-UX (owned by Hewlett Packard). However, even Apple's MAC OS X uses the POSIX standards - though at its core Apple's OS is based on BSD. Other operating systems that mostly comply with the POSIX standards, with the differences generally being minor implementation details, include: Linux (developed by Linus Torvalds) and BSD variants such as FreeBSD, NetBSD, and OpenBSD. Click another button on the timeline to continue learning about the evolution of UNIX. If you have finished reviewing all sections of the timeline, click the Next Slide button in the navigation bar to continue with the module.

UNIX Defined: Since POSIX and the "Single UNIX Specification" are instrumental in allowing a variety of UNIX systems to work in harmony, they also play an important role in answering our question "What is UNIX?" Simply put, for the purpose of this course, UNIX is any OS that generally conforms to the POSIX standards. This being said, it is important to point out that many variants are not truly UNIX systems, but are simply operating systems that behave like UNIX systems, thus are commonly referred to as "UNIX-like" systems. Click another button on the timeline to continue learning about the evolution of UNIX. If you have finished reviewing all sections of the timeline, click the Next Slide button in the navigation bar to continue with the module.

Trademark Requirements

2/6

It is also important to note the term UNIX is trademarked by The Open Group. To use the UNIX trademark, along with ensuring a product meets the "Single UNIX Specification" (the successor of POSIX) - a collection of specification documents that are part of the X/Open Common Application Environment (CAE) - vendors must pay a licensing fee to brand their products. Those variants that use POSIX, or the Single UNIX Specification, but are not licensed by The Open Group, are generally considered to be systems that behave like UNIX, but are not true UNIX. Again, these variants are commonly referred to as UNIX-like systems.

During this course we will primarily focus on the Solaris variant of UNIX, but we will discuss other UNIX-like systems, such as Linux as well. Let's begin by defining a kernel.

The UNIX Kernel

3/6

A UNIX **kernel** is a set of critical programs that provide an environment in which UNIX **processes**, or running programs identified by a unique Process Identifier (PID), execute and interact with system resources such as storage, memory, and Input/Output (I/O) devices. In a nutshell, the kernel acts as the heart of the OS and provides the core functionality of a UNIX operating system.

Kernel Space vs. User Space

4/6

Since the kernel has many responsibilities and it is important for the kernel to be protected from rogue processes, such as malware and those unintentionally created by a user, an operating system is usually divided into two distinct regions - kernel space and user space.

Kernel space is the area of virtual memory where the kernel code runs and has full access to system resources. To keep a machine secure, only allow the most-trusted, well-tested code to run in the kernel space. Additionally, since the kernel typically runs some aspects of memory that cannot be accessed directly by the processes of the normal user, system administrators generally place access rights on the kernel space. This stops users from unwittingly crashing a machine or affecting memory and other resources owned by other programs or the OS kernel.

User space, on the other hand, is the area of virtual memory where user processes run. Since user space programs cannot access system resources directly, typically these programs make requests of the OS through **system calls** (program requests that provide an interface between a process and an operating system). This essentially makes user space a form of computer security as it **sandboxes**, or restricts, user programs.

Now that you are familiar with the basics of a UNIX kernel, let's examine in more detail various functions of the kernel.

Kernel Functions

5/6

Click each tab to learn about the kernel functions displayed. When you have finished reviewing each concept, click the Next Section button to continue with the module.

Manages Resources:
The kernel manages how resources are shared amongst processes. For example, the kernel scheduler allocates CPU cycles to different processes, memory management allocates RAM to processes, and the file subsystem manages concurrent device access. Click another tab to continue learning about kernel functions. If you have finished reviewing kernel functions, click the Next Section button in the navigation bar to continue with the module.

Mediates Processes:
The kernel also mediates all process interaction with system resources as well as the interaction between processes. Mediation includes authorization checks (users may see messages stating that a process doesn't have access to certain system resources, such as a file), inter-process communications (for example, data can be exchanged between two processes via a pipe), and other methods to keep processes safely in user space. Click another tab to continue learning about kernel functions. If you have finished reviewing kernel functions, click the Next Section button in the navigation bar to continue with the module.

Provides Abstraction Layer:
In addition to its other functions, the kernel provides an abstraction layer for access to system resources. This abstraction layer minimizes the need for processes to know hardware specifics. For each device in the system, the kernel has a device driver, or code, that understands how to interact with that device at an electrical level. Instead of user processes having this level of interaction with devices, the kernel only allows access to system resources through system calls. Click another tab to continue learning about kernel functions. If you have finished reviewing kernel functions, click the Next Section button in the navigation bar to continue with the module.

Section Completed

6/6

Kernel Basics and Configuration Files: Section 3 Transcript

System Calls 1/5

The UNIX kernel essentially provides a set of functions that user applications use to request services to the kernel. These functions are called system calls. System calls allow processes to perform file operations, such as *open()*, *read()*, *write()*, and *close()*; process manipulation operations to include creating or destroying processes, such as *fork()*, *execve()*, and *kill()*; memory operations, such as allocating or freeing memory using *malloc()* and *free()*; and even sending data via the network using *socket()*, *recv()*, and *send()*; not to mention so many other functions.

The most interesting feature of system calls is that they allow the kernel to abstract access to system resources at the highest level. Processes can therefore access various devices regardless of their specificity while the low-level details on how to access devices is left to the kernel and the device drivers. In other words, since UNIX provides system calls such as *read()* and *write()*, which can abstract data transfer at a very high-level, many UNIX utilities can work equally well to read and write from and to any storage or transfer device, such as a printer, a USB memory stick, and even system memory. Essentially this level of abstraction is what enables the UNIX "Everything is a File" philosophy.

Additionally, UNIX systems provide various utilities that allow users to verify what system calls are used during the execution of an application.

Utilities: strace and truss 2/5

Utilities are programs that can be used for maintenance, configuration, and debugging of a UNIX System. The `strace` program is a Linux utility that allows the user to determine exactly what system calls are executed in extreme detail. Solaris ships with a tool with similar functionalities called `truss`. The `strace` or `truss` utilities display their result as if the command was running directly on the current command line. Let's review the Linux `strace` utility to learn what data utilities provide regarding system calls.

The Linux strace Utility 3/5

Consider the following usage of the Linux `strace` utility: `strace -o output.txt echo Test for Echo`

The command `strace` takes the option `-o output.txt`, which tells `strace` to dump its trace output to a file named `output.txt`. The execution of the command, `echo Test for Echo`, is the command we wish to trace. Notice that the `echo` command actually runs, and we see its output on screen.

The Linux strace Utility

We can also examine the `strace` output in `output.txt`. To do this we would use the `less` command with the `-N` flag so we can display line numbers. Notice that there are a variety of system calls made by the executable. Let's discuss some of them in detail. Click each line number displayed to learn more about each system call. When you have finished exploring, click the Next Slide button in the navigation bar to continue with the module.

Line 1: Notice that the first line shows the *execve()* system call, which loaded and executed the `echo` command. Also, notice that the program is actually `/bin/echo`. We will discuss the reason for this in a later module, but essentially the *execve()* system call loads and runs the `/bin/echo` executable.

Line 3: The *mmap()* system call requests a segment of memory for the process to use. Other system calls related to memory management are *malloc()*, *free()*, *munmap()*, and *mprotect()*.

Line 4: The *access()* system call checks if the process has permission to access the file; the `R_OK` flag in this line shows the process is checking if it has permission to read the file.

Line 5: The *open()* system call attempts to open the `/etc/ld.so.cache` file in read-only mode. Notice the return value of this call is three, which is a file descriptor. File descriptors are unique numbers that the UNIX system associates to open files. These numbers are used to perform various operations on opened files, such as reading, writing, or closing. We will learn more about file descriptors when we discuss shells in the next module.

Line 6: The *fstat()* system call looks for status information about the file, such as the access rights of the file, the owner of the file, the size of the file and more. This is done using the file descriptor.

Line 8: The *close()* system call tells the kernel the process is done using the resource. The *close()* system call takes for argument the file descriptor of the opened file. In this case, the `/etc/ld.so.cache` file that was opened with file descriptor three on line #5 is now closed.

Line 9: On this line we notice that there is another *open()* system call. Notice the file descriptor three is reused. The same file descriptor cannot be used multiple times on concurrently opened files; however, the OS can reuse a file descriptor once the original file is closed.

Line 10: The *read()* system call reads data from the file. The return value (832) is the number of bytes read.

Line 32: Skipping down to line thirty-two, we find the *write()* system call, which is used to write the output to the terminal. Notice that *write()* uses the file descriptor one, which refers to the Standard Output (STDOUT) for the process. Again, we will discuss this more when we discuss shells in the next module.

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.

Kernel Basics and Configuration Files: Section 4 Transcript

System Configuration Files 1/5

As you have probably begun to realize, a UNIX operating system is a complicated piece of software. With this in mind, you might expect a large number of configuration options, or **tunable parameters**. Configuration options are mostly made available to the system through a collection of configuration files. These configuration files determine what initial applications are launched when the system boots, maintain the list of users in the system, and decide who is authorized for access. Other configuration files are used to define network configuration options, such as the machine name on the network, initial Internet Protocol (IP) addresses, and even the IP address of the default Domain Name System (DNS) servers. The list goes on and on.

System Configuration Files: Storage 2/5

UNIX systems typically store most configuration files somewhere within the `/etc` directory. These files simply need to be modified, usually by a system administrator, in order to change the default behavior of the system. This being said, other configuration files are added whenever a new application is installed on the system, while others are written from scratch by system administrators and users. Generally, configuration files can be changed inline through the command line. Additionally, unlike Windows, UNIX configuration files are usually in a human-readable format, generally ASCII text, which means that no special tools, such as the Windows Registry Editor (Regedit), are required to view or modify them.

System Configuration Files: UNIX vs. Windows 3/5

The set of configuration options read from configuration files by a UNIX system at startup, or boot time, is called the **start-up configuration**. Whenever the system is already running, the configuration loaded in memory that can be changed via the command line is called the **running configuration**. Windows and UNIX systems treat start-up and running configurations differently.

In a Windows system, the Windows Registry reflects the current running configuration. Then, during a clean shutdown, the Windows Registry is written to disk. Upon reboot the Windows Registry is restored "as-is." Therefore, making a change to a Windows Registry affects both the running configuration <u>and</u> the start-up configuration. However, in UNIX, changes made to the running configuration will not affect the start-up configuration, or vice-versa.

In other words, in a UNIX system making a change will affect one <u>or</u> the other as the files are not intertwined. This separation allows UNIX systems to differentiate between the status of the operating system, or service currently running, and the status of the operating system, or service, as it will be whenever the system is rebooted. A notable exception to this is the modern Linux Network Manager which will read changes to network configuration files and alter the running configuration accordingly.

Since operating system configuration files are required throughout system initialization, including the boot process; kernel loading; and service provisioning, it is also important to have a working knowledge of the boot, kernel, and service configuration files. For the purpose of this module, we will discuss kernel configuration files and service configuration files. Boot configuration files will be covered in a later module. Let's begin by discussing kernel configuration files.

Kernel Configuration Files

4/5

While we are not going to discuss kernel configuration in the compilation sense - that is creating custom kernels - it is important to discuss **tunable parameters** (specific values that change kernel behavior) that are configured at **runtime**, or during the execution of a computer program. Why? Simply put, when kernel configuration files are changed, system performance is immediately altered - either improved or worsened. Therefore, to avoid irreparable damage to a system, it is critical to understand the kernel parameters that can be altered during runtime.

There are a variety of kernel parameters that can be altered during runtime, particularly network and memory parameters. The start-up configuration (across many UNIX variants, including Linux, Solaris, and BSD) for these values is in the `/etc/sysctl.conf` file. One way to change the running configuration files of currently running processes is by directly interfacing with the `/proc` directory. This can also be accomplished using the `sysctl` command. Let's conduct an exercise to test how kernel parameters can be altered during runtime.

Exercise Introduction

5/5

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.

Kernel Basics and Configuration Files: Section 5 Transcript

Loadable Kernel Modules

In addition to affecting the kernel behavior by changing the start-up configuration or the running configuration, it is also possible to configure the kernel with **loadable kernel modules**. Without getting into the technicalities of kernel design, it is important to understand that certain kernel functions, particularly device drivers, are not part of the core kernel. This is for both security and performance reasons. Instead the kernel code for certain functions resides in a loadable kernel module, which can be loaded either at start-time or during runtime, as the code it implements is needed. The `modprobe` and `insmod` commands are used to load or unload these kernel modules either during, or after, start-up.

The specific configuration file used to determine which modules are loaded at boot time varies depending on the UNIX version. Some Linux systems use the file `/etc/modules`, while others may use `/etc/modules.conf` or `/etc/sysconfig/modules` to configure the kernel. BSD systems generally use `/boot/loader.conf` for kernel configuration.

As mentioned earlier, kernel configuration files are only one type of configuration file. Let's now take a few moments to examine service configuration files.

Service Configuration Files

In UNIX, a **service** or **daemon** is an application, or program, running in userspace that provides specific functionality. **Service configuration files** are read by their respective services whenever they are started. On most UNIX systems, services are usually started at boot time by the init process; however, services can also be started, stopped, and restarted manually by a user or by the system administrator. In these instances, services still read their configuration files. Some services accept signals that will force them to reload their configuration files. For example, Apache web server accepts the HUP signal to reload its configuration file located in `/etc/httpd.conf`.

Service configuration files provide the default parameters to allow the application to function properly. As an example, let's consider the Secure Shell (SSH) daemon, which allows users to remotely login to a UNIX system. Some of the parameters included in its configuration files are:

- Port number and IP address to use to receive connections,
- Maximum number of connections to allow simultaneously,
- Grace period for login,
- Number of allowed retries after an incorrect password is entered,
- Method to use to authenticate users,
- Filenames and locations for log files, and
- Maximum size of each log file before it starts to be overwritten from the beginning.

Kernel Basics and Configuration Files: Section 6 Transcript

Summary

You have completed the Kernel Basics and Configuration Files module.

You should now be able to:

- Differentiate between UNIX and UNIX-like Operating Systems,
- Define the kernel,
- Differentiate between kernel space and user space,
- Interpret `strace` and `truss` utility outputs to determine system calls used by programs,
- Describe UNIX configuration mechanisms, including system and kernel configuration files, and
- Explain how to load and unload kernel modules.

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.