Linux under the Hood

Sander van Vugt

mail@sandervanvugt.nl

Pearson

Poll Questions - 1

- How would you rate your own Linux knowledge?
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5

Poll Questions - 2

- Which geographical area are you from
  - Australia / Pacific
  - Africa
  - Asia
  - Europe
  - North America
  - South America

Linux under the Hood

Agenda

Agenda

- How Linux is organized
- About C-code, scripts and compiled programs
- Understanding Linux Commands and how they work
- Understanding the kernel and hardware handling

About you

- Linux users / administrators / devops with intermediate experience in the Linux operating system

Linux under the Hood

About your Instructor

About me

- Sander van Vugt
- Linux instructor since 1996
- Regular presenter on Safaribooksonline Live training
- Author of several books and videos on Safaribooksonline
  - Red Hat Systems Administrator (RHCSA) with Virtual Machines
  - Linux Performance
  - Linux Under the Hood
- www.sandervanvugt.com
- mail@sandervanvugt.nl

Linux under the Hood

1. How Linux is Organized

Understanding the Stack

- The lower part is Kernel land
  - The computer uses hardware
  - To address the hardware, the Linux kernel is involved
  - It uses drivers (kernel modules) to address the hardware
- The upper part is User land
  - Users send instructions to hardware from a shell
  - Applications send instructions
  - Libraries are used by applications to provide common functionality
- System calls provide the glue between user land and kernel land

Zooming in on the Kernel

- The kernel deals with addressing different hardware components
  - CPU
  - RAM
  - Disk
  - Network
- These components can be monitored through the /proc filesystem
- Tuning is possible through /proc/sys
- Memory management is an important task of the Linux kernel
  - Each process gets its own reserved area in memory

Kernel Module Management

- The kernel itself is small and contains essential drivers only
- Additional drivers are provided dynamically
  - InitramFS is loaded while booting
  - systemd-udevd takes care of loading drivers automatically and on demand
- To access devices, device files in /dev are used
  - The kernel doesn't know about device file names
  - It uses device major and minor to address devices

Understanding Libraries

- Nearly all programs and processes are using libraries
- Libraries provide common functions that are shared between different commands
- Libc provides the key functions that can be used by all programs

Understanding the Shell

- The shell is the command interpreter that works with its own set of internal commands and that acts as user interface
- Variables play a key role in the working of the shell
- Shell scripts are common the Linux operating system

Understanding File Descriptors

- Linux is a file oriented operating system
- Everything is happening through file operations
  - I/O handling
  - Inter Process communication
- Every process keeps a table of file descriptors showing files currently in use
- Common file descriptors are
  - 0: STDIN
  - 1: STDOUT
  - 2: STDERR

Understanding file types and **lsof**

- Regular file
- Directory
- FIFO
- Character special file
- Block special file
- links
- and many more

Linux under the Hood

2. About C-code, Scripts and Compiled Programs

Understanding C programs

- Linux is written in C
- The source code of all Linux components is accessible as a readable file
- Source files written in C need to be compiled to a binary program
- Notice that an increasing amount of code on Linux nowadays is developed in Python

Understanding Git

- Linux code is written by the open source community
- The community needs a way to work together
- Git is the most common version control system which was created by Linux founder Linus Torvalds
- A Git folder contains the complete repository with a historic overview and incudes tracking functionality as well
- To work on software, developers can clone a Git folder from their local computer and work from there. Once finished, it can be merged from the users computer to the Git repository

Do you need to know C?

- If you're a developer: yes
- If you're an administrator or user: no
  - Most Linux components are very well documented
  - man pages, /usr/share/doc and much more
- For advanced troubleshooting or performance optimization it helps if you can read C

About Compilers and Makefiles

- C code needs to be compiled to get usable code
- Compiling a single file can be straightforward
- On complex programs, external files need to be included as well, and that might make the compiler line very long
- To make it easier, Makefiles can be used
- A Makefile is a recipe that defines exactly what needs to be done in the compilation process
- Use the **make** command to process instructions in a Makefile

Understanding Object Files

- Simple files can be compiled straightforward
- Complex programs are composed of many components and the compilation process consists of multiple steps
    - Source code is compiled into object files
    - An object file cannot run by itself, but contains machine specific code
    - These object files later are linked into libraries to compose an executable program
- The linker is a component that links everything. It exists as a separate command, but can be done directly from the C complier as well: **gcc -o myprog main.o aux.o**

Understanding Header Files

- Header files are additional C source files that contain type and library function declarations
- stdio.h is a common header file that is included to make these functions available
  - Use #include <stdio.h> in the source file to include the header
  - The compiler by default looks for headers in /usr/include
- Specific source code is often dependent of the header files
  - If header files cannot be found in the compilation process, parts of the code won't be compiled at all
- Before compiling everything, the C preprocessor (cpp) joins source code, object files and header files, after which the compiler takes care of the rest

Understanding Libraries

- Using libraries makes it easier to create programs with a common look and feel. It allows for the usage of standardized components so that the programmer can focus on what really matters in his binaries
- Two types of binaries are commonly used
  - Shared libraries: the program loads the library code when it is needed
  - Static libraries: during compilation, the linker copies the library code into the executable

About Library Versions

- On Linux, it is common that different versions of the same library co-exist
- This allows applications to use whatever version they need, without any risk of programs that don't work

Shared Library Location

- Shared libraries have the suffix.so and are stored in fixed paths
- The application itself doesn't need to know where to find them
- The ld.so helper program finds the library location and tells the application where they are found
    - Use ld.so.conf to include non-standard library paths (not recommended)
    - After modifying its contents, run **ldconfig -v**
- A much better option is to tell which libraries to use when compiling a program, so that later on it knows where they are

Linux under the Hood

3. Understanding Linux Commands and how they work

Opening a File for Read

- What needs to happen when a file is read, as in the **cat /etc/hosts** command?
  - The **cat** command must be read and loaded from disk in RAM
  - Related libraries must be found and loaded in RAM also
  - the /etc/hosts file must be located on disk
  - Permissions of the current user need to be checked on this file
  - If these are OK, the file contents can be copied to RAM
- All of these tasks are provided through system calls and library calls

Understanding System Calls

- Processes cannot access the kernel directly
- System calls are used as an interface for processes to the kernel
  - glibc provides a library interface to use system calls for programs
- Common tasks like opening, listening, reading and writing to files involve system calls
- The fork() and exec() system calls determine how a process starts
  - fork(): the kernel creates an almost identical copy of the current process and replaces that
  - exec(): the kernel starts a program, which replaces the current process

Getting Information about System Calls

- System calls are documented in man section 2
  - **man 2 intro**
  - **man s syscall**
  - **man 2 mmap**

Understanding Library Calls

- System calls are provided by the kernel to give access to restricted parts
- Library calls come from shared libraries and provide common funtionality
- There's a large number of library calls, virtually unlimited as new library calls can be provided easily through dynamic libraries
- See **man 3 intro**

User space versus Kernel space

- Hardware access is restricted to the kernel only
- The kernel provides system calls for users and processes to access hardware
- User space is memory that is allocated by the kernel for user processes and contains
  - Network configuration
  - Services, like web server
  - Applications
  - User interfaces
- Users are created to assign permissions and limits to entries on Linux
- Every process or task running on Linux has an owner

Using strace and ltrace

- **strace** shows system traces
- **ltrace** shows library traces

Understanding Signals

- Signals provide software interrupts; a method to tell a process that it has to do something
- Signals are strictly defined
  - See **man 7 signal**
- Some signals (sigkill, sigterm) cannot be ignored, others can be ignored
- Many signals are specific to a command

Linux under the Hood

4. Understanding Kernel Hardware Handling

Understanding Hardware Initialization

- The Linux kernel accesses hardware using drivers
- Drivers are available as kernel modules
- Device files in /dev provide a user interface to the devices
- Device files are mostly auto-generated
- Device files correspond to the device nodes, which are a major and minor that the kernel uses to identify devices
- Use **mknod** to manually generate

Monitoring Hardware Availability

- Different interfaces are available to monitor hardware
- /proc/devices
- dmidecode
- lspci
- lsusb
- dmesg
- lsscsi
- lscpu

Understanding **sysfs**

- /sys is used to provide information about devices and their attributes
- /sys/devices is dedicated to device information
- The **udevadm** command provides an interface to query device information from /sys
  - Use for instance **udevadm info --query=all --name=/dev/sda**

Understanding more about udev

- The Linux kernel initiates devices loading and next sends out uevents to the udevd user space daemon
- Udev catches the event and decides how to handle based on attributes that it has received in the event
- Udevd next reads its rules and takes action based on these rules
    - Default rules are in /lib/udev/rules.d
    - Custom rules are in /etc/udev/rules.d

5. Kernel Management

Fine-Tuning the Kernel

- Most Linux distributions come with a standard kernel that should not be modified
- Modifying kernel sources is provided through kernel modules
- In some environments, tuning the kernel is important to ensure the kernel is optimized to do exactly what it is supposed to be doing

Managing Kernel Modules

- **lsmod** will show current modules
- **modinfo** provides modules information
- **modprobe** is used to load kernel modules
- **modprobe.conf** and related files can be used to make parameters persistent

Optimizing the Kernel

- The /sys filesystem is used for hardware management
- the /proc filesystem is used for kernel tunables

Linux under the Hood

Next steps

Interested in more?

- Linux under the Hood
  - https://www.safaribooksonline.com/library/view/linux-under-the/9780134663500/
- Linux Performance Optimization
  - https://www.safaribooksonline.com/library/view/linux-performance-optimization/9780134985961/
- Linux Troubleshooting
  - https://www.safaribooksonline.com/videos/linux-troubleshooting-red/9780135207598?autoplay=false