Systems Software

Week 1: Introduction, Development Tools and Scripting Techniques

Notes By: Jonathan McCarthy

Overview

- Revision – Unix and Linux
- Architecture and components of a Unix system
- Programming in a Unix environment
- Shell Scripting / C Programming
- Software Tools
- Programming Philosophies
- Standards
- Design Principles

UNIX®
An Open Group Standard

What is Unix?

- UNIX is an operating system (OS) that was developed by AT&T in 1969 by Ken Thompson and Dennis Ritchie.
- The UNIX OS is made up of different programs that provide different features and functionality to keep the system running stable and to allow the users to perform specific tasks etc…
- Unix systems are multi-user and multi-tasking and can be used for servers and for users personal PC's

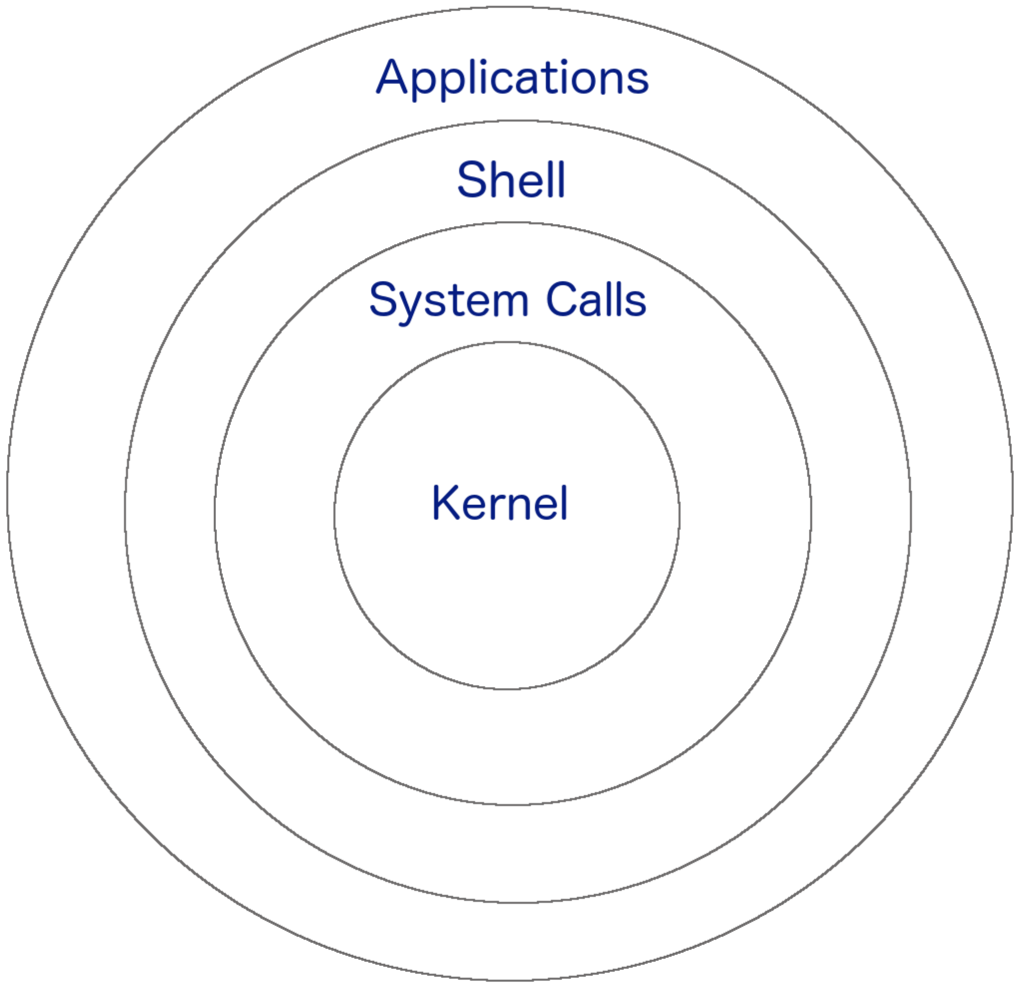
Unix / Linux Examples

- Linux Examples:
  - Debian GNU/Linux
  - Ubuntu
  - Mandriva Linux
  - Red Hat Enterprise Linux
  - Gentoo
  - Fedora
  - SUSE
  - Linux Enterprise
  - openSUSE
  - Linux Mint
  - Slackware Linux

Makeup of a Unix System

- A Unix system has three main components:
  - Kernel
  - Shell
  - Programs

Unix Architecture



The Kernel

- The kernel can be thought of the software that is used to interact with the systems hardware (handling files, disks, networking etc)
- The kernel is an integral part of the OS and is always running.
- As part of the install process, the kernel is specifically built to suit the machine.
- If any hardware components are changed in a given machine, the kernel will have to be re-built.

The Kernel

- The interface to the kernel is a layer of software called **system calls**.
- Programs use system calls to execute operations.
- Libraries are common functionalities that are available for use and sit on top of the system calls. These can be used by programs to perform specific tasks. The program could use the library of access the system call interface directly.

Kernel Responsibilities

- System Memory Management
- Software Program Management
- Hardware Management
- Filesystem Management

The Shell

- The shell is an **interface between the user and the kernel.**
- The shell is a command line interpreter (CLI)
- A user can enter commands via the CLI, the program will run and the program can interact with the kernel by using system calls to perform tasks.
- There are a number of different shells that can be used: Bash / Korn / C Shell etc.

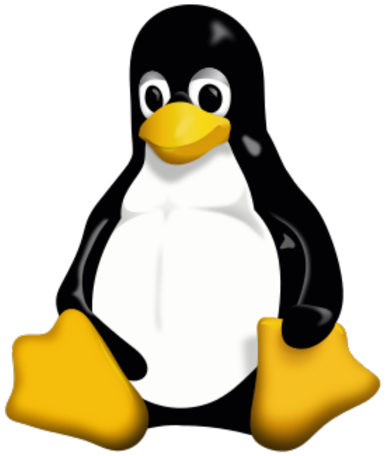
Commands

- There are a large number of commands that can be used to perform specific tasks in a UNIX environment.
- Eg: mv, cp, ls, grep, pwd etc….
- A number of different options can be used with the commands to perform more specific tasks.
- A number of commands can be grouped together to create a shell script to perform a larger task.

Programs

- When the specific functionality needed is greater than the basics of shell scripts, a program can solve this problem.
- A program can be written in a language to solve the problem.
- The program will have more control over the system and can access the OS API's and make system calls to perform very low level specific tasks. This can offer better performance, efficiency and maintainability
- **Question: Why would you prefer a C program over a Shell Script?**

What is Linux?



- Linux is an open source
- Free to use and distribute
- Servers and Client machines
- Linux is the kernel used by the GNU operating system.
- Linux is a unix-like operating system kernel and open source.
  - UNIX is proprietary.

GNU Project and the Free Software Foundation

- In 1983 Richard Stallman created the GNU project (short for "GNU's Not Unix").
- The main goal was to create a free software Unix-like system
- The Linux kernel was created be Linus Torvalds in 1991 and released as as free software under the GNU General Public License.
- A lot of the GNU packages that were developed for Linux have made their way into some Unix systems.
- Linux systems have gained prominence as the operating system of choice for servers for business

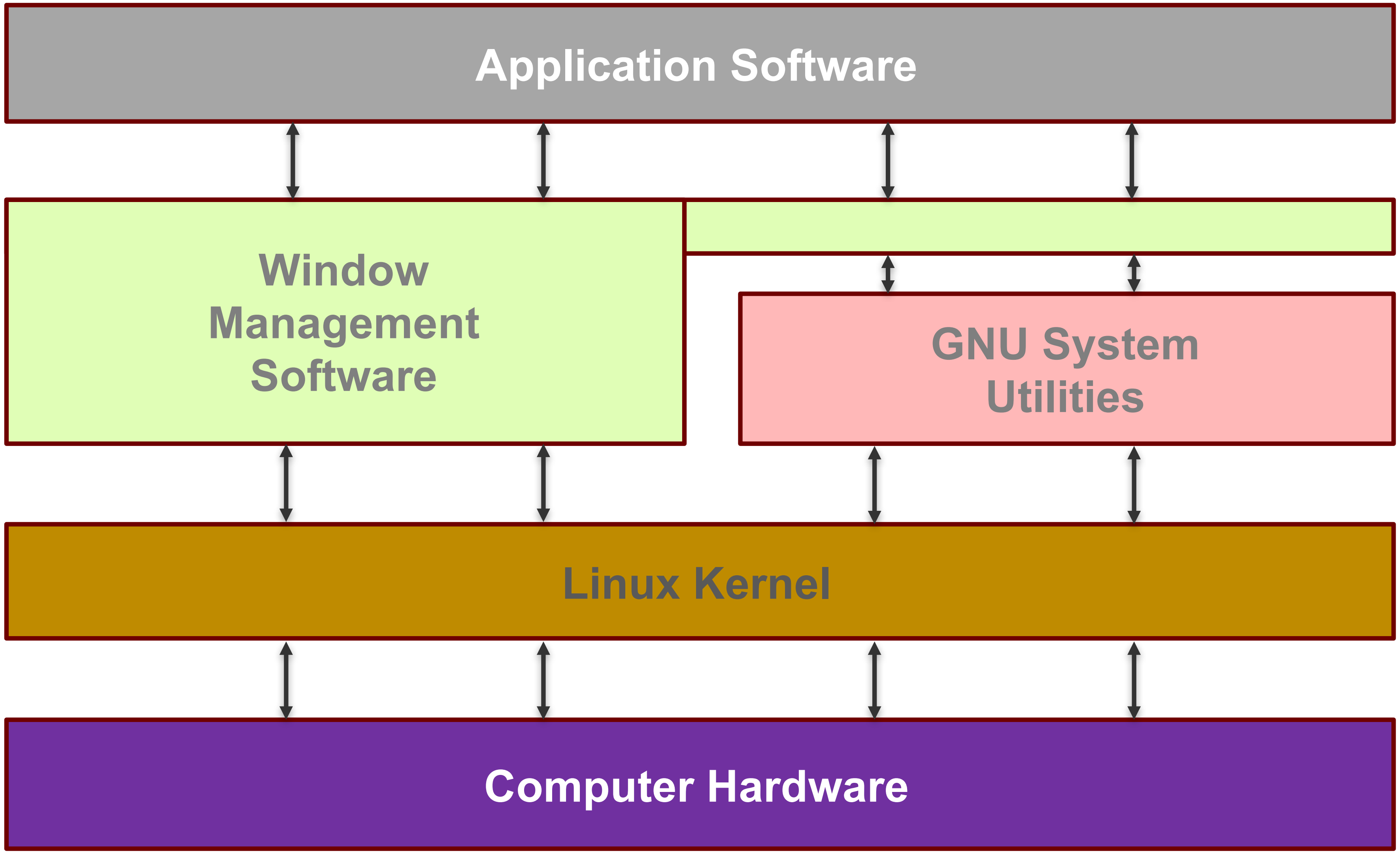
GNU Utilities

- An operating system needs utility programs to perform basic operations and tasks as part of its general operation.
- The GNU organisation developed a complete set of UNIX utilities that is free for use in a Linux environment.
- The main packages of the Core GNU Utilities are:
  - File handling
  - Manipulating Text
  - Managing Processes

What is the difference between Unix and Linux

- Linux distros are free to obtain and use. Unix is not.
- Linux is unix-like operating system released as free and open source. Linux is created by Linus Torvalds. Linux distributions uses Linux kernel to build the whole operating system.
- Linux is not derived from Unix source code, but its interfaces are intentionally like Unix (POSIX Standards).
- Linux can be thought of as a clone of UNIX.
- UNIX systems come as a complete OS with associated tools etc, all from the one vendor.

OS Basics:: The Linux System



Main Components in a Linux System

- The Linux Kernel
- The GNU Utilities
- Graphical Desktop Environment
- Application Software

Standards

- The IEEE was the main body responsible for the standardization of Unix.
- The IEEE Unix standards is know as **POSIX**, this was the first Unix standard.
- The POSIX standards has detailed definitions regarding shells and command sets and interfaces for non-C programming languages.
- **The Open Group** (a group of Unix vendors), is now responsible for the standardization of Unix.

UNIX and the C Compiler

- The UNIX command for compiling a C program is gcc
- gcc is a compiler from GNU for Linux.
- A program needs to be compiled ant this process creates a binary code file that can be executed. The binary file may not work on another machine, if the program is moved to another machine it should be re-compiled.

Manual Pages

- Linux distros include manual pages for the following:
  - (1) User commands
  - (2) System calls
  - (3) Standard library functions
  - (8) System/Admin commands
- The numbers are related to the page sections.
- Eg: man ls
- The manual is a very good resource and should be used to get more information and detail for commands etc….

Info

- The Info documentation contains detailed information on the Linux system and some of its core programs.

Unix Development Philosophy I

- The **UNIX philosophy** is documented by **Doug McIlroy** in the The Bell System Technical Journal from 1978:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Unix Development Philosophy II

- The philosophy documented by **Doug McIlroy was** later summarized by **Peter H. Salus** in ***A Quarter-Century of Unix (1994)***.
- The Unix philosophy (**by Doug McIlroy**):
  1. Write programs that do one thing and do it well.
  2. Write programs to work together.
  3. Write programs to handle text streams, because that is a universal interface.

Unix Development Philosophy III

- Rob Pike (one of the great masters of C programming) detailed a different set of rules in Notes on C Programming:
- Rule 1. You can’t tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don’t try to second guess and put in a speed hack until you’ve proven that’s where the bottleneck is.
- Rule 2. Measure. Don’t tune for speed until you’ve measured, and even then don’t unless one part of the code overwhelms the rest.
- Rule 3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don’t get fancy. (Even if n does get big, use Rule 2 first.)

Source: The Art of Unix Programming by Eric Steven Raymond 2003

Unix Development Philosophy III

- Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
- Rule 5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.
- Rule 6. There is no Rule 6.

Source: The Art of Unix Programming by Eric Steven Raymond 2003

What is a design principle?

- Software design principles are best practices to follow when developing an application. The principles are detailing how to properly develop an application in OOP.
- Following the principles should help in avoiding a bad design of the application.

- What are the implications of a bad design?

Design Patterns

- A design pattern offers a solution to software design problems that occur frequently in real world application development.
- The origin of design patterns is attributed to Christopher Alexander, who worked as an architect in the field of building construction and introduced the concept of design patterns as a common vocabulary for design discussions.
- Alexander described a pattern as a problem that frequently occurs which has an associated core solution to this problem.

Design Patterns

- This solution can be reused every time this problem occurs, and may deviate slightly depending on the problem being solved.
- Design patterns are recurring solutions to software design problems you find again and again in real-world application development.

General Design Principles

- DRY – Don’t Repeat Yourself
- KISS – Keep It Simple Stupid
- YAGNI – You Ain’t Gonna Need It
- Tell, Don’t Ask
- SoC – Separation of Concerns

DRY – Don’t Repeat Yourself

- The DRY design principle places an emphasis on not duplicating data or behaviour in the code base.
- When developing large\complex systems it can be easy to duplicate data \ code \ logic to get something working.
- The problem with this is that when changes need to be made to the application, it can be difficult to identify what to update. Maintainability and extendibility will suffer.
- The principle requires that data or behaviour should occur once in the system and offer it through a reusable unit.

KISS – Keep It Simple Stupid

- Try to keep the code simple. A system or process will ork best if it is kept simple.
- No use of overly complex code to get something working.
- By keeping the code solution simple and clean, this will help towards a maintainable and extensible system.

YAGNI – You Ain’t Gonna Need It

- Do not add additional code/functionality to a system, only enough to solve the given problem.
- Adding additional functionality to an application is also described as “***Gold Plating***”.
- The YAGNI principle ties into the TDD process, in that you just write enough code to get the test to pass.

SoC – Separation of Concerns

- This principle is closely related to the Single Responsibility Principle (SRP).
- The main purpose is to ensure that different concerns are not located in the same code base.
- The concern can be described as the behaviour of a class.
- From an OOP perspective SoC is trying to encapsulate data and behaviour to facilitate code reuse. Separate classes for different data and behaviour.
- SoC facilitates code reuse, maintainability, extendibility and testability.

S.O.L.I.D Design Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

- Rule: A class should have one and only reason to change.
- Ie. A class should have only one role or task.

Open-Closed Principle (OCP)

- Rule: Objects are open for extension, but closed for modification.
- OCP is trying to architect the program so that the functionality can be extended with minimum changes in the existing code. New functionality can be introduced through new classes, leaving the main codebase in place and not requiring modification.

Liskov Substitution Principle (LSP)

- Rule: Subclasses should be substitutable for their base classes.
- Based on Inheritance.
- LSP and OCP are closely related. A violation of the LSP rule will break the OCP rule.

Interface Segregation Principle (ISP)

- Rule: Instead of creating one large interface, split the methods of a contract into groups of responsibility.
- Don’t force a class to implement methods it will not use.
- Avoid fat interfaces.

Dependency Inversion Principle (DIP)

- Rule: An object shouldn't control the creation of its dependencies, it should just advertise what dependency it needs and let the caller provide it.

Summary of Eric Raymond's Rules

- **Rule of Modularity:** Write simple parts connected by clean interfaces.
- **Rule of Clarity:** Clarity is better than cleverness.
- **Rule of Composition:** Design programs to be connected to other programs.
- **Rule of Separation:** Separate policy from mechanism; separate interfaces from engines.
- **Rule of Simplicity:** Design for simplicity; add complexity only where you must.
- **Rule of Parsimony:** Write a big program only when it is clear by demonstration that nothing else will do.

Reference: The Art of Unix Programming - Eric Steven Raymond

Summary of Eric Raymond’s Rules

- **Rule of Transparency:** Design for visibility to make inspection and debugging easier.
- **Rule of Robustness:** Robustness is the child of transparency and simplicity.
- **Rule of Representation:** Fold knowledge into data so program logic can be stupid and robust.
- **Rule of Least Surprise:** In interface design, always do the least surprising thing.
- **Rule of Silence:** When a program has nothing surprising to say, it should say nothing.
- **Rule of Repair:** When you must fail, fail noisily and as soon as possible.

Reference: The Art of Unix Programming - Eric Steven Raymond

Summary of Eric Raymond’s Rules

- **Rule of Economy:** Programmer time is expensive; conserve it in preference to machine time.
- **Rule of Generation:** Avoid hand-hacking; write programs to write programs when you can.
- **Rule of Optimization:** Prototype before polishing. Get it working before you optimize it.
- **Rule of Diversity:** Distrust all claims for “one true way”.
- **Rule of Extensibility:** Design for the future, because it will be here sooner than you think.

Reference: The Art of Unix Programming - Eric Steven Raymond

To Do: Additional Reading

- Read Chapter 1 [Philosophy] from Eric Raymond’s Book - The Art of Unix Programming

References

- Unix Kernel: http://www.extropia.com/tutorials/unix/kernel.html
- Most Popular Distros: https://distrowatch.com/dwres.php?resource=major
- Book: Linux Command Line and Shell Scripting Bible (3rd Edition) – Wiley
- Book: The Art of Unix Programming - Eric Steven Raymond

Additional Reading

- The additional reading must be completed and is examinable in the terminal exam.

Questions