**Chapter 2: Fundamental Concepts**

2.1 The Kernel

The **operating system** is the entire package (central software + tools, such as the GUI)

The **kernel** is the portion of the operating system that manages and allocates computer resources, such as CPU, RAM, and devices. The kernel performs *process scheduling* (CPU access for applications), *memory management* (RAM access for applications), *provides a file system* (disk access), *provides access to devices* (keyboard, mice, USB ports, hard drives, etc.), *networking* (internet, LAN), and *provides an* ***application programming interface (APU)***, i.e. a set of system calls that applications can use.

2.2 The Shell

A **shell** is a special type of program that reads user commands and executes programs in response. This is also sometimes known as a *command interpreter*. A **login shell** is the process that creates a shell when the user first logs in. Some common shells include the Bourne shell (sh), the C shell (csh), the Korn shell (ksh), and the Bourne again shell <3 (bash). Shells can also run shell scripts, text files containing shell commands.

2.3 Users and Groups

**Users** have a login name (like gormanme) and a corresponding numeric user ID (UID) and additional information in the password file, **/etc/passwd**.

**Groups** are used to control access to system resources (for admins), with information such as group name, group id (GID) and list of users in the group stored in **/etc/group**.

**Superuser** is a special user that can bypass all system permission checks, and has a special login name: “root.” The superuser account is used for administration (not for login!!)

2.4 Directories, links, and files

The kernel uses a single **hierarchical directory** system, where all drives and resources are under the **root directory**, “/” (slash). Compare to windows, where each drive has its own mount letter (C: drive, D: drive, etc.), most drives are typically mounted in /media/\*.

**Files** include “normal files,” but also pipes, symbolic links, and directories. **Directories** are special files that have a table of filenames + references (kind of like a tree / linked list). **Symbolic links** are specially marked files containing the name of another file (kind of like a pointer).

Filenames are up to *255 characters long*, and cannot contain slashes and null characters. **Pathnames** are strings consisting of an initial lash followed by a series of filenames (possibly multiple directories). **Absolute** pathnames are with respect to the root directory. **Relative** pathnames are with respect to the **current working directory** of the process.

Each file as an associated user ID and group ID, which together comprise their permissions with read(r)/write(w)/execute(x) permissions for the user, group, and everyone else.

2.5 File I/O Model

UNIX hasthe concept of **universality of I/O (input/output).** This means the same system calls are used for all types of files, including devices.

**File descriptors** are non-negative (zero or greater) integers assigned to open files by the kernel.

2.6 Programs

**Programs** exist in two forms: **source code** (human readable) and **binary machine-language instructions** (computer-readable).

A **filter** is a program that reads from stdin, does stuff, and then writes the transformed stuff to stdout. Examples include cat, grep, and awk.

**Command line arguments** are words supplied on the command line when a program is run.

2.7 Processes

A **process** is an instance of an executing program, and are loaded, allocated, and set up by the kernel. A process can be divided into *segments*: text (the instructions), data (variables used), the heap (area for extra memory) and the stack (a piece of memory used for local variables).

Processes have unique integer **process identifiers** (PID), and the PID of their parents (PPID). They also have associated user and group IDs for tracking permissions. A **privileged process** is one that has an effective user ID (determines permissions) of the superuser.

**Capabilities** are privileges given to the superuser, and can be granted to other processes (so they can do some superuser actions but not others).

The **init** process is started by the kernel at boot is the parent (or grandparents or great…) of all processes. Init’s main job is to create and monitor everything needed for the system to run.

A **daemon** is a special process that differs from normal ones in that it is long lived (from boot to shutdown) and it runs in the background.

An **environment list** is a set of **environment variables** that are in the processes’ memory. Environment lists are inherited from the parent process (whatever called this process).

2.8 Memory Mappings

mmap() creates a **memory mapping**, where a file (including devices) is “mapped” or basically mirrored to virtual memory (the kernel’s memory, which is RAM + some hard drive). This allows operations to be done faster than just reading/writing the file directly.

2.9 Static and Shared Libraries

An **object library** is a file containing compiled code for a set of functions that can be used for an application, kind of like stdlib.h or stdio.h. Libraries can either be **static** (just for this one application) or **shared** (stored by the system, the application just tells the kernel it needs that library).

2.10 Interprocess Communication and Synchronization

Some processes need to communicate with each other, so Linux (and all UNIX based systems) provide tools for **interprocess communication** (IPC), including pipes (transfer data between processes), sockets (like pipes but can be on different hosts), file locking (prevent multiple processes from accessing at the same time), message queues (to send messages between processes, semaphores (used to sync the actions of processes, and shared memory (memory segments used by multiple processes).

2.11 Signals

**Signals** are essentially software interrupts: they inform processes that events or conditions have occurered. Some examples include the interrupt character (usually Control+C), when a child has terminated, a timer has elapsed, or the process tried to access memory it doesn’t have access to. Processes can ignore signals or handle them based on what the signal means.

2.12 Threads

**Threads** are essentially multiple related processes that share the same virtual memory, however they each have their own stack. This means they share global memory but not local memory. Threads are useful for applications that work better with multiple threads rather than actual multiple processes (such as draw and refresh threads for a map) or for parallel processing (such as on a dual core processor).

2.13 Process Groups and Shell Job Control

Each program ran by the shell gets a new process. All major shells (except sh) have **job control**, which allows for the execution of multiple commands at the same time. All of the processes in a pipeline (piped together) are in a *process group* or *job.*

2.14 Sessions, Controlling Terminals, and Controlling Processes

A **session** is a collection of process groups, and usually have a **controlling terminal**, such as the login session and the login shell.

2.15 Pseudoterminals

A **pseudoterminal** is a pair of connected, virtual devices known as the master and slave. The slave provides a terminal-like interface, and the master drives the slave. An example of a pseudoterminal is **ssh**, where the slave is the process that logs in and gets a terminal interface from the server/master.

2.16 Date and Time

**Real time** is measured from a fixed point (like the start of a process) or a standard point (like the calendar time).

**Process** time is the total amount of CPU time that a process has used.

2.17 Client-Server Architecture

A **client-server application** is an application broken down into the client (that requests a service from the server) and the server (which examines the client’s request and handles it, returning back a response).

2.18 Realtime

**Real-time applications** are those that need to respond quickly to input. Some Linux systems have support to ensure that these applications’ needs are met, however traditional UNIX is *not* realtime.

2.19 The /proc File System

The **/proc** directory is a virtual file system that provides access to kernel data structures in a form that looks like files and directories. The idea is that they can be easily parsed/read and modified by human users and scripts.

**Chapter 3: System Programming Concepts**

3.1 System Calls

A **system call** is a controlled entry point into the kernel, allowing processes to request the kernel to perform a task for them. Functionally they are used much like calling a C function.

3.2 Library Functions

A **library function** is one of the functions from a standard C library. Some use system calls, but many do not.

3.3 The Standard C Library; The GNU C Library (glibc)

The **GNU C Library** (glibc) is the most common Linux implementation of the standard C library. To view its version information, you can run the library as an executable! (just type in /lib/libc.so.\* in the command line on most distros). Glibc also provides \_\_GLIBC\_\_ and \_\_GLIBC\_MINOR\_\_ version constants.

3.4 Handling Errors from System Calls and Library Functions

System calls and library functions almost always return a status value indicating if they succeeded or failed. Checks should always be made to see if it succeeded, and errors should be handled if they occur. System calls and some library functions use *errno* (from the <errno.h> header), a global integer variable that identifies the particular error that occurred, although not all library functions do so.

3.5 Notes on the Example Programs in This Book

**Command line options** consist of a hyphen followed by a single letter, and an options argument ( -<letter> <argument> ). Note that options without an argument can be “stacked” (like “ls –la”).

3.6 Portability Issues

**Feature test macros** are a way to expose only the functions and constants of a library that meet a specific standard, like \_POSIX\_SOURCE, \_BSD\_SOURCE, or \_GNU\_SOURCE (glibc specific). This allows you to design code that is portable in accordance to a specific standard.

System data types (like process IDs) are defined through the C **typedef** feature. For example, Linux uses the line “typedef int pid\_t;” for process IDs. This means that when you define a process ID like “pid\_t process\_id”, it converts it into an integer type. There are many of these types for different types of systems data.

*Also consider differences in header files, macros, and structures between different implementations of UNIX, as they may have different contents, be in different orders, or may even not exist!*