Chapter -2 Fundamental Concepts

2.1 The Core Operating System: The Kernel

- The term operating system is commonly used with two different meanings:
  - To denote the entire package consisting of the central software managing a computer's resources and all the accompanying standard software tools, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
  - More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

- The term **kernel** is often used as a synonym for the second meaning,
- Note it is possible to run programs on a computer without a kernel.

- **Tasks performed by the kernel:-**
  - **Process scheduling:**

    Like other UNIX systems, Linux is a preemptive multitasking operating system, Multitasking means that multiple processes can run simultaneously reside in memory and each may receive use of the CPU(s).

  - **Memory management:**

    Because the (RAM) remains a limited resource (Non-renewable recourses ex: 4G for address bus 32) and correspondingly the size if the SW is grown, Linux employs virtual memory management.

    main advantages of VM:
    - Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel.
    - Only part of a process needs to be kept in memory allowing more processes to be held in RAM simultaneously.

- **Provision of a file system:**
  The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.
- **Creation and termination of processes**
  The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.
- **Access to devices**
  The kernel provides programs with an interface that standardizes and simplifies access to devices.
- **Networking:**
  The kernel transmits and receives network messages (packets) on behalf of user processes.

- **Kernel mode and user mode**
  - Modern processor architectures typically allow the CPU to operate in at least two different modes: user mode and kernel mode.
  - When running in user mode, the CPU can access only memory that user space; attempts to access memory in kernel space result in a hardware exception.
  - When running in kernel mode, the CPU can access both user and kernel memory space.
  - Certain operations can be performed only while the processor is operating in kernel mode. Examples executing the halt instruction to stop the system, accessing the memory-management hardware, and initiating device I/O operations.
- **Process versus kernel views of the system.**
  For a process, many things happen asynchronously:
  An executing process doesn't know when it will next time out, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. doesn't know where it is located in RAM or, in general, whether a particular part of its memory space is currently resident

in memory or held in the swap area (a reserved area of disk space used to supplement the computer's RAM).

process operates in isolation; it can't directly communicate with another, and can't itself create a new process or even end its own existence.

By contrast, a running system has one kernel that knows and controls everything.

2.2 The Shell

- A shell is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands. Such a program is sometimes known as a command interpreter.
- The term login shell is used to denote the process that is created to run a shell when the user first logs in.

  NOTE: no login shell it's the shell that we open in wen we want to execute some commands
- A number of important shells have appeared over time:
  - **Bourne shell (sh):**
    - written by Steve Bourne. The Bourne shell contains many of the features familiar in all shells: I/O redirection, pipelines, filename generation (globbing), variables, manipulation of environment variables, background command and execution.
    - All later UNIX implementations include the Bourne shell in addition to any other shells they might provide.
  - **C shell (csh):**
    - This shell was written by Bill Joy at the University of California at Berkeley
    - The C shell was not backward compatible with the Bourne shell.
    - Although the standard interactive shell on BSD was the C shell.
  - **Korn shell (ksh):**
    - This shell was written by David Korn at AT&T Bell Laboratories. While maintaining backward compatibility with the Bourne shell, it also incorporated interactive features similar to those provided by the C shell.

- **Bourne again shell (bash):**
  - This shell is the GNU project's reimplementation of the Bourne shell
  - It supplies interactive features similar to those available in the C and Korn shells
  - On Linux, the Bourne shell, sh, is typically provided by bash emulating sh as closely as possible just type command sh to be able to emulatie the Bourne shell.

- **shell scripts**, which are text files containing shell commands. For this purpose, the shells typically associated with programming languages: variables, loop and conditional statements, I/O commands, and functions.

2.3 Users and Groups

- **Users**
  - Every user of the system has a unique login name (username) and a corresponding:
    numeric user ID (UID). For each user, these are defined in /etc/passwd, which includes the following additional information:
    Group ID: group ID of the first of the groups of which the user is a member.
    Home directory: the initial directory into which the user is placed after logging in.
    Login shell: the name of the program to be executed to interpret user commands.

```
john:x:1000:1000:John Doe:/home/john:/bin/bash
```

    The password record may also include the user's password, in encrypted form. However, for security reasons, the password is often stored in the separate etc/shadow.

- **Groups**
  - For administrative purposes—in particular, for controlling access to files and other system resources—it is useful to organize users into groups.
  - In early UNIX implementations, a user could be a member of only one group. BSD allowed a user to simultaneously belong to multiple groups, an idea that was taken up by other UNIX implementations.
  - Each group is identified by a single line in the system group file, /etc/group, which includes the following information:

    Group name: the (unique) name of the group.

    Group ID (GID): the numeric ID associated with this group.

    User list: a comma-separated list of login names of users who are members of this group.

```
wheel:x:10:root,john
developers:x:1001:john,mary,alex
staff:x:50:mary,sam
```

- **Superuser:**
  - One user, known as the superuser, has special privileges within the system. The superuser account has user ID 0, and has the login name root.
  - On typical UNIX systems, the superuser bypasses all permission checks in the system.
  - The superuser can access any file in the system, regardless of the permissions on that file, and can send signals to any user process in the system.

2.4 Single Directory Hierarchy, Directories, Links, and Files

- The kernel maintains a single hierarchical directory structure to organize all files in the system At the base of this hierarchy is the root directory, named / (slash).

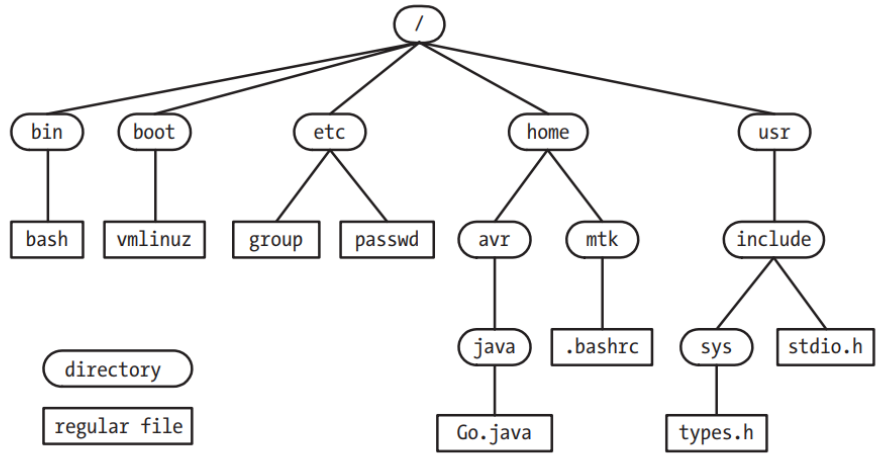

Figure 2-1: Subset of the Linux single directory hierarchy

- **File types**

  These file types includes regular, devices, pipes, sockets, directories, and symbolic links.

- **Directories and links**
  - A directory is a special file whose contents take the form of a table of filenames coupled with references to the corresponding files(I node)
  - files may have multiple links, and thus multiple names, in the same or in different directories. (Hard Links)
  - Every directory contains at least two entries: . (dot), which is a link to the directory itself, and .. (dot-dot), which is a link to its parent directory

- **Symbolic links**
  - Like a normal link, a symbolic link provides an alternative name for a file.
  - But whereas a normal link is a filename-plus-pointer entry in a directory list, a symbolic link is a specially marked file containing the name of another file(file referred to by a pointer)

- When accessing the symbolic links, the kernel automatically dereferences the original file
- If a symbolic link refers to a file that doesn't exist it's said to be dangling link
- The kernel imposes limits on the number of dereferences to handle the possibility of circular chains of symbolic links.

- **Filenames**
  - On most Linux file systems, filenames can be up to 255 characters.
  - Filenames may contain any characters except (/) and (\0)
  - However, it is advisable to employ only letters and digits, This 65-character set, [-._a-zA-Z0-9], is referred to in SUSv3 as the portable filename character set

- **Pathnames**
  - A pathname is a string consisting of an optional initial slash (/) followed by a series of filenames separated by slashes.
  - The pathname consists of directory part of a pathname, while the name following the final slash is sometimes referred to as the file or base part
  - **An absolute pathname** begins with a slash (/) and specifies the location of a file with respect to the root directory
  - **A relative pathname** specifies the location of a file relative to a process's current working directory

- **Current working directory**
  - Each process has a current working directory
  - A process inherits its current working directory from its parent process. A login shell has its initial current working directory set to the location named in the home directory.

- *File ownership and permissions*
  - *Each file has an associated user ID and group ID that define the owner of the file and the group to which it belongs.*
  - *For the purpose of accessing a file, the system divides users into* ***three categories:*** *the* ***owner*** *of the file , users who are members of the group matching the file's group ID (* ***group****), and the rest of the world (****other****).*
  - *Three permission bits may be set for each of these categories:*
  - *read permission, write permission execute permission*
  - *and execute (sometimes called search) permission allows access to files within the directory.*

2.5 File I/O Model

- *In a UNIX like system everything is a file so One of the great features of the I/O model on UNIX systems is the concept of* universality of I/O.
- *This means that the same system calls (*open (), read(), write(), close(), *and so on) are used to perform I/O on all types of files, including device.*
- *The kernel essentially provides one file type: a sequential stream of bytes, which, in the case of disk files, disks, and tape devices, can be randomly accessed using the* lseek() *system call.*

- *File descriptors*
  - *the I/O system calls refer to open files using a file descriptor, a (usually small) nonnegative integer. A file descriptor is typically obtained by a call to open()*
  - *When a process opens a file, the kernel assigns a file descriptor to that specific file within the process*
  - *Normally, a process inherits three open file descriptors when it is started by the shell: [descriptor 0 is standard input, descriptor 1 is standard outputand descriptor 2 is standard error].*
  - *In an interactive shell or program, these three descriptors are normally connected to the terminal. In the stdio library, these descriptors correspond to the file streams stdin, stdout, and stderr.*

- ***The stdio library***
  - *To perform file I/O, C programs typically employ I/O functions contained in the standard C library. This set of functions, referred to as the stdio library, includes fopen(), fclose(), scanf(), printf(),and so on.*
  - *The stdio functions are layered on top of the I/O system calls (open(), close(), read(), write(), and so on).*

2.6 Programs

- *Programs normally exist in two forms. [source code, binary machine-language]*
  Source code: *text consisting of a series of statements written in programming language.*
  Binary machine-language: *instructions that the computer can understand*
  Note:
  Script: is a text file containing commands to be directly processed by a program such as a shell or other command interpreter
  To run the script:

```
source script_name.sh
```

- ***Filters***
  - *A filter is the name often applied to a program that reads its input from stdin, performs some transformation of that input, and writes the transformed data to stdout. Examples of filters include cat, grep, tr, sort, wc, sed, and awk.*

- ***Command-line arguments***
  - *In C, programs can access the command-line arguments, the words that are supplied on the command line when the program is run.*
  - *The main() function of the program is declared as follows:*

```
int main(int argc, char *argv[])
```

The argc variable contains the total number of command-line arguments, and the individual arguments are available as strings pointed to by members of the array argv. The first of these strings, argv [0], identifies the name of the program itself.

2.7 Processes

- a process is an instance of an executing program. When a program is executed, the kernel loads the code of the program into virtual memory, allocates space for program variables, and sets up kernel bookkeeping data structures to record various information (such as process ID, termination status, user IDs, and group IDs) about the process.

- **Process memory layout**

  A process is logically divided into the following parts, known as segments:

  1. **Text Segment (Code Segment):**
     - This segment contains the executable code of the program.
     - It is read-only to prevent the program from accidentally modifying its instructions.
     - Shared among all instances of the program to save memory.
  2. **Data Segment:**
     - Divided into initialized and uninitialized parts.
     - **Initialized Data Segment (Data Segment):** Contains global and static variables that are explicitly initialized by the programmer.
     - **Uninitialized Data Segment (BSS Segment, Block Started by Symbol):** Contains global and static variables that are not explicitly initialized by the programmer. By default, these are initialized to zero by the system.
  3. **Heap:**
     - The heap segment is used for dynamic memory allocation during program execution.
     - It grows upward towards higher memory addresses when memory is allocated (e.g., using `malloc` in C).
     - The heap size is not fixed and can be adjusted via system calls like `brk()` or `sbrk()`.

4. **Stack:**

- The stack segment is used for managing function calls and local variables.
- It supports the LIFO (Last In, First Out) mechanism to handle function calls, local variables, and return addresses.
- The stack grows downward towards lower memory addresses.
- Each thread in a process has its own stack.

- **Process creation and program execution**
  - A process can create a new process using the fork() system call. The process that calls fork() is referred to as the parent process, and the new process is referred to as the child process
  - The kernel creates the child process by making a duplicate of the parent process
  - The child *inherits* copies of the parent's data, stack, and heap segments. (The program text, which is placed in memory marked as read-only, is shared by the two processes.)
  - **execve()** system call to load and execute an entirely new program. An execve() call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program. The key point here is that the replacement of these segments occurs only within the calling (child) process's address space.
- **Process ID and parent process ID**

Each process has a unique integer process identifier **(PID)**. Each process also has a parent process identifier **(PPID)** attribute, which identifies the process that requested the kernel to create this process.

- **Process termination and termination status**

  A process can terminate in one of two ways:
  - By requesting its own termination using the _exit() system call or by being killed by the delivery of a signal then the process returns a termination status, to the parent process using the wait() system call

A termination status of **Zero** indicates that the process succeeded, and a **nonzero** status indicates that some error occurred

- **Process user and group identifiers (credentials)**
  - *Real user ID and real group ID:* These identify the user and group to which the process belongs. A new process inherits these IDs from its parent. A login shell gets its real user ID and real group ID from the corresponding fields in the system password file.
  - *Effective user ID and effective group ID:* These two IDs are used in determining the permissions that the process has when accessing protected resources such as files and. Typically, the process's effective IDs have the same values as the corresponding real IDs. Changing the effective IDs is a mechanism that allows a process to assume the privileges of another user or group, as described in a moment.
  - *Supplementary group IDs:* These IDs identify additional groups to which a process belongs. A new process inherits its supplementary group IDs from its parent.

- **Privileged processes**

  Traditionally, on UNIX systems, a privileged process is one whose effective user ID is 0 (**superuser**)
  - A Nonprivileged processes have a nonzero effective user ID and must abide by the permission rules enforced by the kernel.
  - A process may be privileged because it was created by another privileged process—for example, by a login shell started by root (superuser)
  - Another way a process may become privileged is via the set-user-ID mechanism, which allows an executable file to be run with the permissions of the file's owner rather than the permissions of the user who is running the executable

  To Do So:

```
chmod u+s programname
```

- **Capabilities**

  Linux divides the privileges into a set of distinct units called capabilities Each privileged operation is associated with a particular capability, and a process can perform an operation only if it has the corresponding capability. A traditional superuser process (effective user ID of 0) corresponds to a process with all capabilities enabled.

- **The init process**
  - When booting the system, the kernel creates a special process called init, the "parent of all processes," which is derived from the program file /sbin/init.
  - All processes on the system are created (using fork()) either by init or by one of its descendants.
  - The init process always has the process ID 1 and runs with superuser privileges.
  - The init process can't be killed (not even by the superuser), and it terminates only when the system is shut down.
  - The main task of init is to create and monitor a range of processes required by a running system.

- **Daemon processes**

  A daemon is a special-purpose process that is created and handled by the system in the same way as other processes, but which is distinguished by the following characteristics:
  - It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down.
  - It runs in the background, and has no controlling terminal from which it can read input or to which it can write output.

  Examples of daemon processes include syslogd, which records messages in the system log, and httpd, which serves web pages, also background tasks such backups, scheduled jobs or a periodic maintenance tasks.

- *Environment list:*
  - *which is a set of environment variables that are maintained within the user space memory of the process.*
  - *When a new process is created via fork(), it inherits a copy of its parent's environment.*
  - *When a process replaces the program that it is running using exec(), the new program either inherits the environment used by the old program*

  Example: *$ export MYVAR='Hello world'*

- *Resource limits*
  - *Each process consumes resources, such as open files, memory, and CPU time. Using the setrlimit() system call we can limits these recourses.*
  - *Soft Limit: limits the amount of the resource to the process limits (recommendation limit)*
  - *Hard Link: limits the amount of the resource to the process limits (Max limit)*
  - *When a new process is created with fork(), it inherits copies of its parent's resource limit settings*