Operating Systems

Chapter 1: Introduction

Chapter 1: Introduction

1.1 What Operating Systems Do
1.2 Computer-System Organization
1.3 Computer-System Architecture
1.4 Operating-System Operations
1.5 Resource Management
1.6 Security and Protection
1.7 Virtualization
1.8 Distributed Systems
1.9 Kernel Data Structures
1.10 Computing Environments
1.11 Free/Libre and Open-Source Operating Systems

Chapter Objectives

Operating systems are everywhere, from cars and home appliances that include “Internet of Things” devices, to smart phones, personal computers, enterprise computers, supercomputers, and cloud computing environments.

In this chapter, we look into a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system.

Dominant Operating Systems of today

- The dominant desktop operating system is Microsoft Windows
- macOS by Apple Inc. is in second place
- Varieties of Linux are collectively in third place
- In the mobile (smartphone and tablet combined) sector, Google's Android on smartphones is dominant followed by Apple's iOS
- Linux distributions are dominant in the server and supercomputing sectors. Since November 2017, all of the world's fastest 500 supercomputers run Linux-based operating systems
- Other specialized classes of operating systems, such as embedded and real-time systems, exist for many applications.

What is an Operating System?

- An **operating system** (**OS**) is system software that manages computer's hardware.
- OS provides a basis for application programs and acts as an intermediary between the user of a computer and the computer hardware.
- A fundamental responsibility of an operating system is to allocate the resources such as the CPU, memory, I/O devices, as well as storage. to programs.
- The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner

Role of OS in the overall Computer System

A computer system can be divided roughly into four components:

1. **User**

   People, machines, other computers

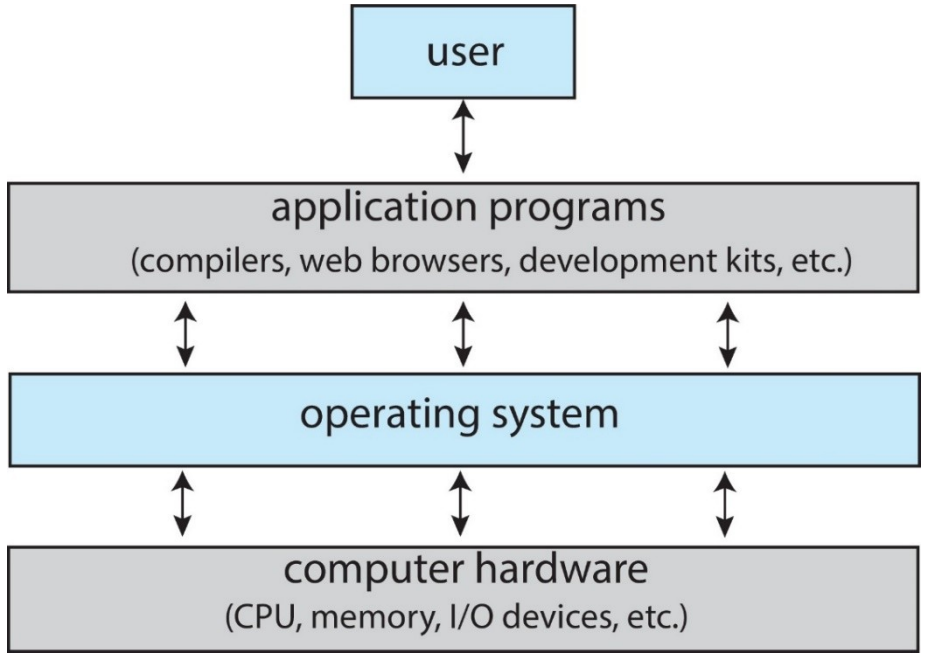
2. **Application programs** – define the ways in which the system resources are used to solve the user's computing problems

   Such as Word Processors, Compilers, Web browsers, Database systems, Video games

3. **Operating system -** controls the hardware, and coordinates its use for various application programs for the various users

4. **Hardware** – provides basic computing resources for the system

   CPU, memory, I/O devices



Abstract View of Components of Computer

OS design goals

Some operating systems are designed to be convenient, others to be efficient, and others to be some combination of the two:

OS from the User's Viewpoint

The user's view of the computer varies according to the interface being used:

- **Personal Computers** - single-user, no resource sharing
  - OS is designed mostly for **ease of use, security,** and **good performance**
- **Mainframe** multi-user, resource sharing
  - Operating system is a **resource allocator** and **control program,** making efficient use of hardware and managing execution of user programs
- **Mobile devices** like smartphones and tables are resource poor, optimized for usability and battery life
  - User interfaces generally feature touch screens, voice recognition
- **Embedded computers** Some computers have little or no user interface, such as embedded systems in home devices in devices and automobiles
  - operating systems and applications are designed to run primarily without user intervention

OS from the System's viewpoint

From the computer's point of view, the operating system is the program most intimately involved with the hardware.

1. In this context, we can view an OS as a **resource allocator**

   A computer system has many **resources** that may be required to solve a problem: **CPU time, memory space, file-storage space, I/O devices,** and so on

2. OS **controls** various **I/O devices** and user programs

3. OS **manages** the **execution** of the user programs to **prevent errors** and improper use of the computer

Defining Operating Systems

No universally accepted definition

- Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems

- Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers.
- Apple's iOS and Google's Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

Defining Operating Systems (cont.)

The **one program running at all times on the computer**—usually called the **kernel**.

Along with the kernel, there are two other types of programs:

1. **system programs**, which are associated with the operating system but are **not necessarily part of the kernel,** and

2. **application programs**, which include all programs not associated with the operation of the system.

The operating system includes:

In summary, for our purposes, the operating system includes:

1. The always running **kernel**,

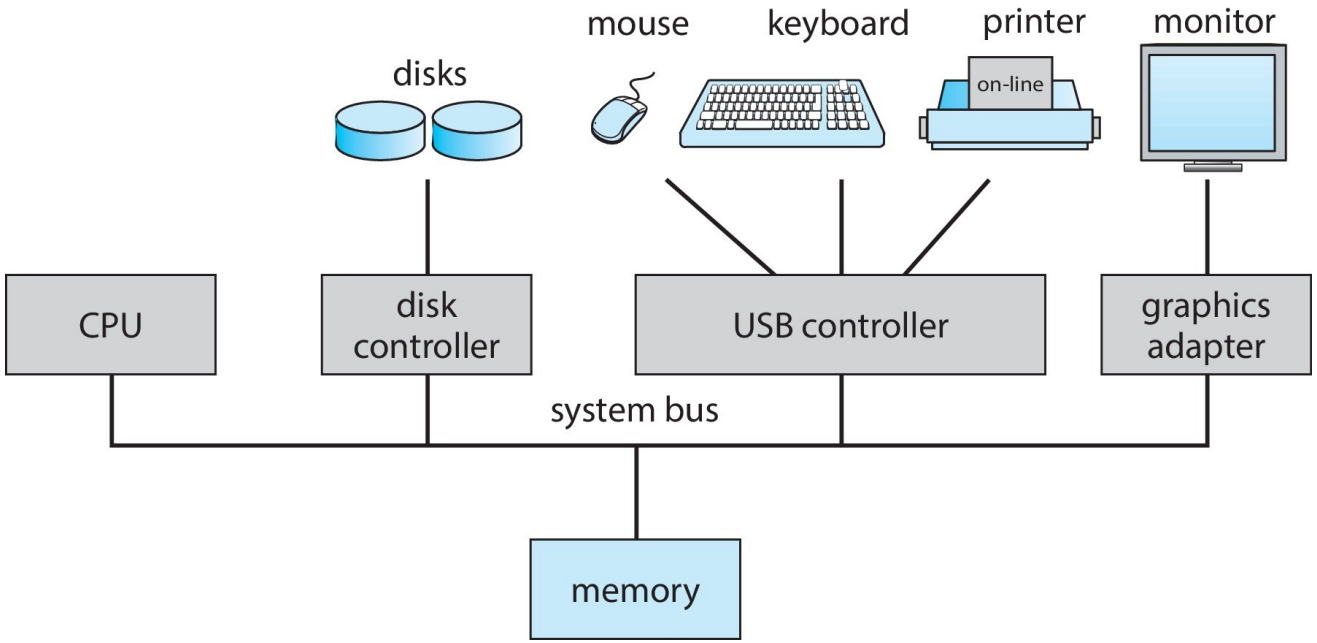
2. Middleware frameworks that ease application development and provide features, and

3. System programs that aid in managing the system while it is running.

1.2 Computer System Organization

Computer System Organization

- A general-purpose computer system consists of one or more **CPU**s and a number of device controllers connected through a common **bus** that provides access between components and shared memory
  - Typically, OS have a device driver for each device controller
  - The CPU and device controllers can execute in parallel, competing for memory cycles



Interrupts

Interrupts are a key part of how OS and hardware interact.

- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Interrupts alert the CPU to events that require attention.
- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to interrupt service routine. On completion of the interrupt service routine, the CPU resumes the interrupted computation.
- The **interrupt vector** contains the addresses of all the service routines. Only a predefined number of interrupts is possible.
- The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt.

Interrupts (cont.)

- Most CPUs have two interrupt request lines.:
  - Nonmaskable interrupts – for events like unrecoverable memory errors
  - Maskable interrupts – can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
    - maskable interrupt is used by device controllers to request service
- Interrupts are used throughout modern operating systems to handle asynchronous events
- Device controllers and hardware faults raise interrupts.
- To enable the most urgent work to be done first, modern computers use a system of interrupt priorities.

Storage

- All forms of **memory** provide an **array of bytes**.
- Each byte has its own address.

Storage Definitions and Notation

The basic unit of computer storage is the **bit** . A bit can contain one of two values, 0 and 1.
All other storage in a computer is based on collections of bits.
Given enough bits, it is amazing how many things a computer can represent:
numbers, letters, images, movies, sounds, documents, and programs, to name a few.
A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage.
A word is made up of one or more bytes. For example, a computer that has 64-bit registers
And 64-bit memory addressing typically has 64-bit (8-byte) words.
A computer executes many operations in its native word size rather than a byte at a time.

A **kilobyte** , or KB , is 1,024 bytes; a **megabyte** , or **MB** , is $1{,}024^2$ bytes; a **gigabyte** , or GB , is $1{,}024^3$ bytes; a **terabyte** , or **TB** , is $1{,}024^4$ bytes; and a **petabyte** , or **PB** , is $1{,}024^5$ bytes.
Computer manufacturers round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.
Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

Memory

The **CPU** can load instructions only from **memory**

- So any programs must first be loaded into memory to run.
- General-purpose computers run most of their programs from rewritable memory, called **main memory** (also called random-access memory, or RAM).
- Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM).**

Instruction–execution cycle

As executed on a system with a **von Neumann architecture**,

1. first **fetch**es an instruction **from memory** and stores that instruction in the **instruction register**.
2. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register.
3. After the instruction on the operands has been executed, the result may be **store**d back **in memory**.

Main Memory

Ideally, we want the programs and data to reside in main memory permanently, but:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Secondary storage

Large **nonvolatile** storage capacity.

Most programs (system and application) are stored in secondary storage until they are loaded into memory.

Many programs then use secondary storage as both the source and the destination of their processing

1. **Non-volatile memory (NVM)** devices– faster than hard disks. **electrical** **semiconductor**-based electronic circuits
   - non-volatile memory chips (Flash memory Storage) – EEPROM, SSD, etc.
   - Becoming more popular as capacity and performance increases, price drops

2. **Hard Disk Drives** (**HDDs**) – rigid metal or glass platters covered with **magnetic** recording material
   - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
   - The **disk controller** determines the logical interaction between the device and the computer

Tertiary storage

- CD-ROM, blue-ray - optical
- magnetic tapes, and so on

nonvolatile

Slow and large.

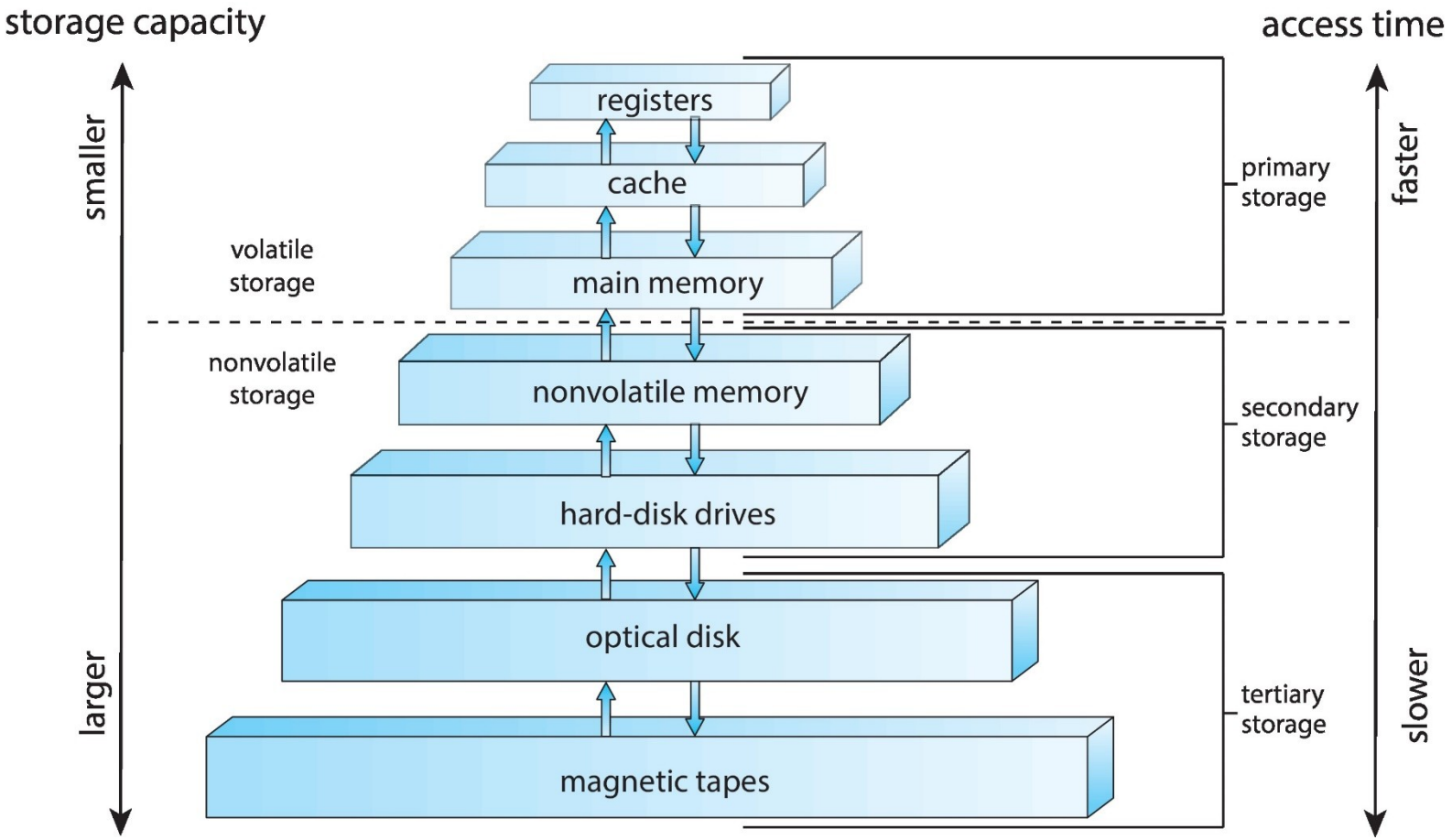
For special purposes like storing backup copies

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility

- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy



1.3 Computer System Architecture

Computer-System Architecture

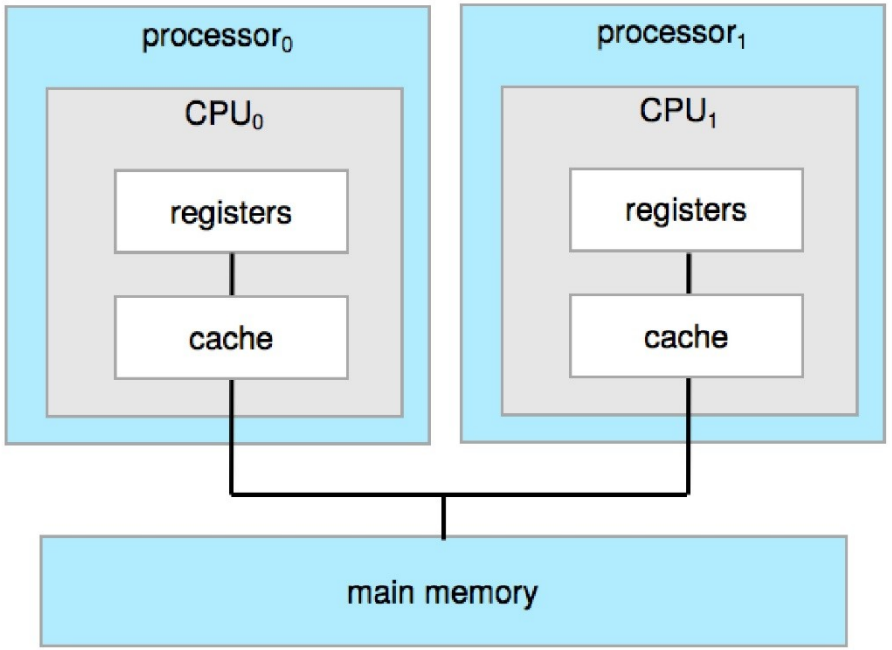
Modern **Multiprocessor systems**,

Systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

- from mobile devices to servers, have two (or more) processors, each with a single-core CPU
- also known as **parallel systems**, **tightly-coupled systems**
- advantages include increased throughput, economy of scale, increased reliability – graceful degradation or fault tolerance

Two types:

1. **Asymmetric Multiprocessing** – each processor is assigned a specie task.
2. **SMP - Symmetric Multiprocessing** – each peer processor performs all tasks, including OS functions and user processes.
   - Most common.

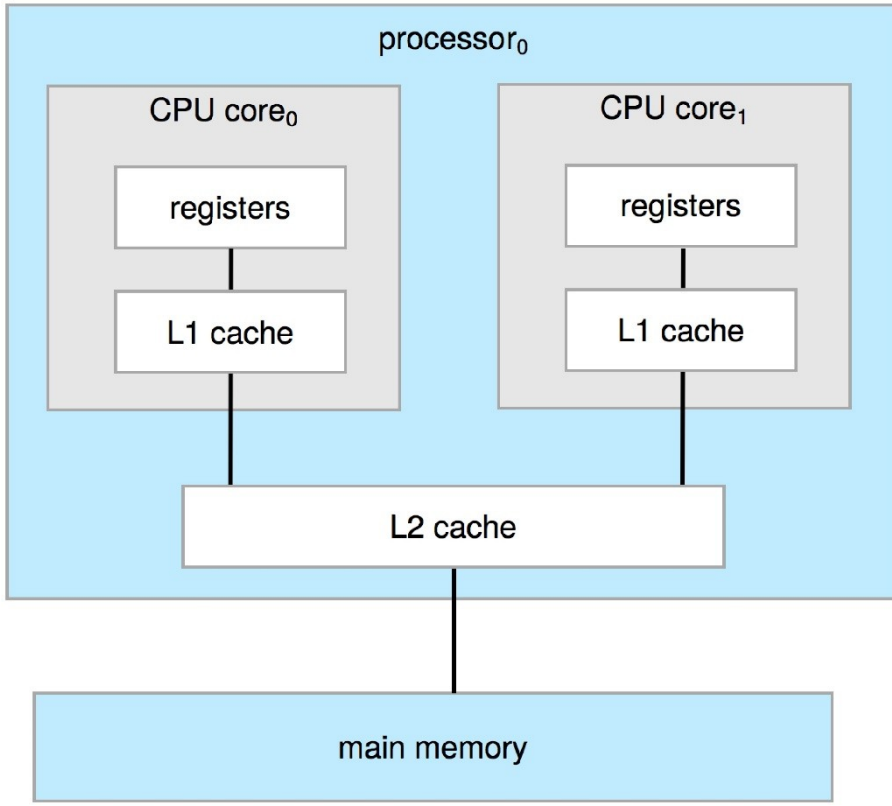


Symmetric Multiprocessing Architecture

Multicore Systems

Multicore

- Multiple computing cores reside on a single chip
- More efficient than Multi-chips with single core
- On-chip communication faster than between-chip communication
- Use significantly less power - important for mobile devices as well as laptops



Level 2 cache in the Duel-Core design is local to the chip, is smaller, and is shared by processing cores

A Dual-Core Design with two cores on the same processor chip

Multicore SMP Systems

- Windows
- macOS
- Linux
- Android
- iOS

Virtually all modern OS

1.4 Operating System Operations

Computer Startup

To start running, a computer needs an initial program, or **bootstrap** program

- Typically, it is stored within the computer hardware in **read-only memory (ROM)** or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware** (ie., storage that is nonvolatile, and cannot be written to frequently)

- Jnitializes all aspects of the system, from CPU registers to device controllers to memory contents.

- Bootstrap program locates **operating-system kernel** **and** **loads into memory**

Computer Startup (cont.)

Once the **kernel** is loaded and executing, it can start providing services to the system and its users.

Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become **system daemons**, which run the entire time the kernel is running.

On Linux, the first system program is "systemd," and it starts many other daemons.

Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

Interrupt-driven OS

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen.

Events are almost always **signal**ed by the occurrence of an interrupt.

A **trap** (or an **exception**) is a software-generated interrupt caused by:

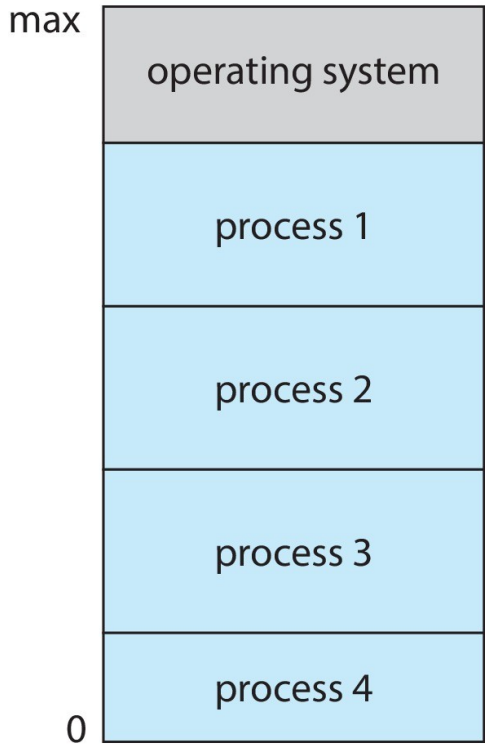
1. An error (for example, division by zero or invalid memory access)
2. A specific request from a user program that an operating-system service be performed by executing a special operation called a **system call**.

Multiprogramming

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times.

- **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute.
- The operating system keeps several processes in memory simultaneously.
- The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete.
- In a non-multiprogrammed system, the CPU would sit idle.
- In a multiprogrammed system, the operating system simply switches to, and executes, another process.
- When ***that*** process needs to wait, the CPU switches to ***another*** process, and so on.
- Eventually, the first process finishes waiting and gets the CPU back.
- As long as at least one process needs to execute, the CPU is never idle.

Memory Layout for Multiprogrammed System



Multitasking

Multitasking is a logical extension of multiprogramming.

- In multitasking systems, the CPU executes **multiple processes by switching among them**, but the switches occur frequently, providing the user with a fast **response time**.

- Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

- I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen.

- Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Dual-mode and Multi-mode Operation

Dual-mode operation allows OS to protect itself and other system components

- **User Mode** and **Kernel Mode**
  - System **boot time** hardware starts in **kernel mode**
  - OS **is then** loaded and starts user applications in user mode

- Increasingly CPUs support **Multi-mode** operations
  - i.e. **virtual machine manager** (**VMM**) **mode** for guest **VMs** – more power than user processes, but fewer than kernel
  - Intel processors have 4 modes , 0 (kernel), 1 &2 (OS services), 3 (user)

Dual-mode

Two separate ***modes*** of operation:

1. **user mode** and
2. **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**)

A bit, called the **mode bit**, is added to the **hardware** of the computer to indicate the current mode: kernel (0) or user (1).

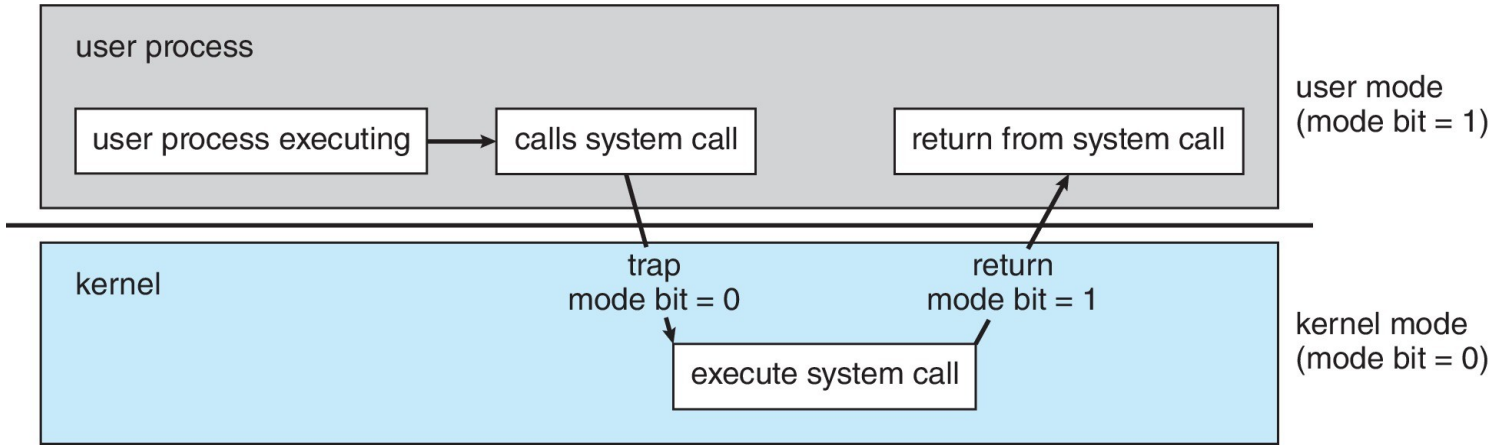
Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).

Thus, whenever the operating system gains control of the computer, it is in kernel mode.

The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program

Transition from User to Kernel Mode

- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions designated as **privileged**, only executable in kernel mode
  - System call changes mode to kernel, return from call resets it to user



1.5 Resource Management

Process Management

- A **process** is a **program in execution**. It is a **unit of work** within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files, etc.
  - Initialization data
- **Single-threaded** process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- **Multi-threaded** process has one program counter per thread

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

To execute a program all (or part) of the **instructions** must be in **memory**

- All (or part) of the **data** that is needed by the program must be in **memory**
- Memory management determines what is in memory and when.
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

File-system Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by **device** (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

- File-System management
  - **Files** usually organized into directories
  - **Access control** on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

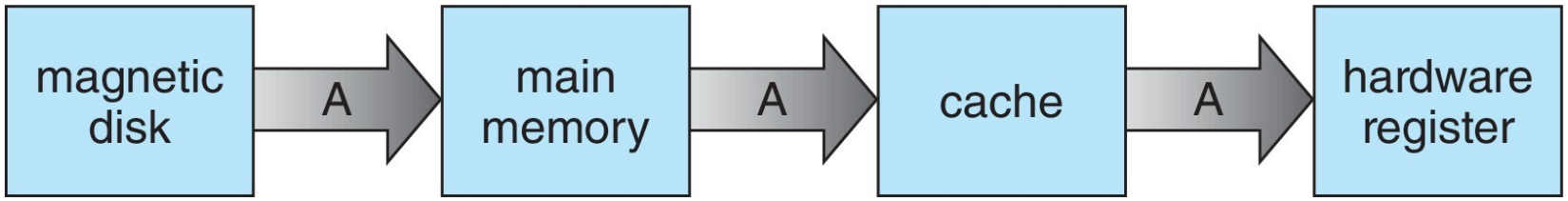
Mass-Storage Management

- Usually **disks** used to **store** data that does not fit in main memory or data that must be kept for a “long” **period of time**
- Proper management is of central importance
- Entire **speed of computer operation** hinges on disk subsystem and its algorithms
- OS activities
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications

Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Migration of data “A” from Disk to Register



- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy
- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist

Protection and Security

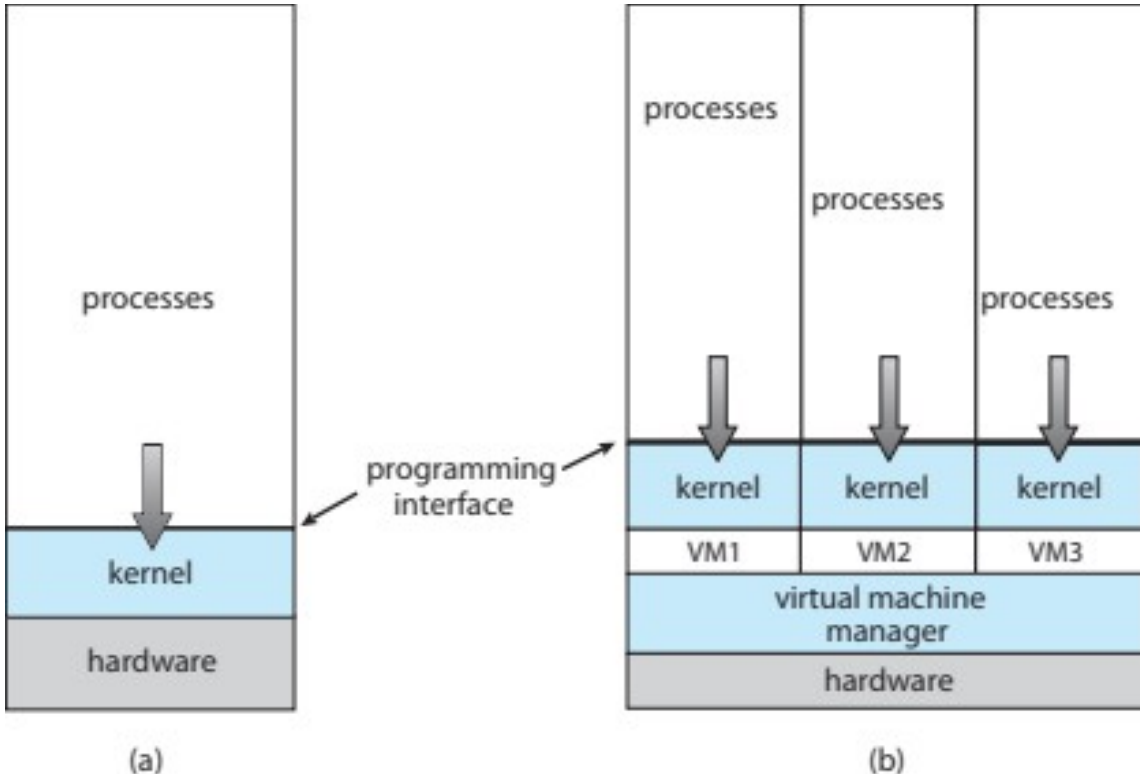
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** (extra permissions given by OS) allows user to change to effective ID with more rights

Virtualization

- **Hardware virtualization** or *platform virtualization* refers to the **creation of a virtual machine that acts like a real computer with an operating system**. Software executed on these virtual machines is separated from the underlying hardware resources. For example, a computer that is running Arch Linux may host a virtual machine that looks like a computer with the Microsoft Windows operating system; Windows-based software can be run on the virtual machine.
- **Emulation** involves **simulating computer hardware in software**, typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU.
  - That same concept can be extended to allow an entire operating system written for one platform to run on another.

Computing Environments - Virtualization

- **Virtualization** allows operating systems to **run as applications within other operating systems**.
- VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications.
- Windows was the **host** operating system, and the VMware application was the **virtual machine manager (VMM).** The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.



Distributed Systems

A **distributed system** is a **collection of physically separate**, possibly heterogeneous computer systems that are **network**ed to provide users with access to the various resources that the system maintains.

- Access to a **shared resource increases computation speed, functionality, data availability, and reliability**.
- **TCP/IP** most common network protocol, and it provides the fundamental architecture of the **Internet**
- Networks are characterized based on the distances between their nodes:
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)
- **Network Operating System** provides features between systems across network
    - Communication scheme allows systems to exchange messages
    - Illusion of a single system

Free (free/libre) and Open-Source Operating Systems

Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary**

- Counter to the **copy protection** and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has "copyleft" **GNU Public License (GPL)**
  - Copyleft - software work may be used, modified, and distributed freely on condition that anything derived from it is bound by the same condition
  - Free software and open-source software are two different ideas championed by different groups of people
    - http://gnu.org/philosophy/open-source-misses-the-point.html/
- Examples include **GNU/Linux** and **BSD UNIX** (including **Darwin**, the core kernel of **macOS**), and many more
- Free Virtualbox VMM tool (http://www.virtualbox.org)
  - Use to run guest operating systems for exploration

The Study of Operating Systems

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BUSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from https://curlie.org/Computers/Software/Operating_Systems/Open_Source/

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (http://www.vmware.com) providesa free "player" for Windows on which hundreds of free "virtual appliances" can run. Virtualbox (http://www.virtualbox.com) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

Introduction to Linux

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released in 1991, by Linus Torvalds.

Today, Linux systems are used in every domain, from embedded systems to supercomputers. 90% of all cloud infrastructure is powered by Linux, including supercomputers and cloud providers. 74% of smartphones in the world are Linux-based.

I. FILE HANDLING UTILITIES

cat

cat command concatenates files and print it on the standard output.

SYNTAX:

The Syntax is

cat [OPTIONS] [FILE]...

OPTIONS:

-A	Show all.
-b	Omits line numbers for blank space in the output.
-e	A $ character will be printed at the end of each line prior to a new line.
-E	Displays a $ (dollar sign) at the end of each line.
-n	Line numbers for all the output lines.
-s	If the output has multiple empty lines it replaces it with one empty line.
-T	Displays the tab characters in the output.
-v	Non-printing characters (with the exception of tabs, new-lines and form- feeds) are printed visibly.

EXAMPLE:

1. To Create a new file:
   cat > file1.txt
   This command creates a new file named file1.txt. Start typing the file content onto the display screen. After typing, press <ctrl>d to end and save the file.
2. To Append data into the file:
   cat >> file1.txt
   To append data into the same file, use append operator >> to write into the file, else the file will be overwritten (i.e., all of its contents will be erased).
3. To display a file:
   cat file1.txt
   This command displays the data in the file.
4. To concatenate several files and display

1. cat file1.txt file2.txt

   The above cat command will concatenate the two files (file1.txt and file2.txt) and it will display the output in the screen. Sometimes the output may not fit the monitor screen. In such situation you can print those files to a new file or display the file using less command.

   cat file1.txt file2.txt | less
2. To concatenate several files and to transfer the output to another file.

   cat file1.txt file2.txt > file3.txt

   In the above example the output is redirected to new file file3.txt. The cat command will create new file file3.txt and store the concatenated output into file3.txt.

rm

rm Linux command is used to remove/delete the file from the directory

Caution: Always use *rm -i* not *rm*

SYNTAX:

The Syntax is

rm [options..] [file | directory]

OPTIONS:

-f	Remove all files in a directory without prompting the user.
-i	Interactive. With this option, rm prompts for confirmation before removing any files.
-r (or) -R	Recursively remove directories and subdirectories in the argument list. The directory will be emptied of files and removed. The user is normally prompted for removal of any write-protected files which the directory contains.

EXAMPLE:

1. To Remove / Delete a file:

   rm file1.txt

   Here rm command will remove/delete the file file1.txt.
2. To delete a directory tree:

   rm -ir tmp

   This rm command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.
3. To remove multiple files

   rm file1.txt file2.txt

   rm command removes file1.txt and file2.txt files at the same time.

cd

cd command is used to change the directory.

Syntax is:

cd [directory | ~ | ./ | ../ | -]

OPTIONS:

-L	Use the physical directory structure.
-P	Forces symbolic links.

EXAMPLE:

1. cd sub
   This command will take you to the sub-directory called sub from its parent directory.
2. cd **..**
   This will change to the parent-directory from the current working directory/sub-directory.

   Note: Double dots **..** refers to parent directory. Single dot **.** refers to current directory
3. cd ~
   This command will move to the user's home directory which is "/home/username".

cp

cp command copy files from one location to another

If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

SYNTAX:

The Syntax is

cp [OPTIONS]... SOURCE DEST
cp [OPTIONS]... SOURCE... DIRECTORY
cp [OPTIONS]... --target-directory=DIRECTORY SOURCE...

OPTIONS:

-a	same as -dpR.
-v	verbose
-b	like --backup but does not accept an argument.
-f	if an existing destination file cannot be opened, remove it and try again.
-p	same as –preserve mode, ownership, timestamps.

EXAMPLE:

1. Copy a file:
   cp file1 file2
   The above cp command copies the content of file1.php to file2.php.
2. To backup the copied file:
   cp -b file1.php file2.php
   Backup of file1.php will be created with '~' symbol as file2.php~.
3. Copy folder and subfolders:
   cp -R srcdir destdir
   cp -R command is used for recursive copy of all files and directories in source directory
4. cp -Rv dev bak

ls

ls command lists the files and directories in current working directory

SYNTAX:

The Syntax is

ls [OPTIONS]... [FILE]

OPTIONS:

-l	Lists all the files, directories and their mode, Number of links, owner of the file, file size, Modified date and time and filename.
-t	Lists in order of last modification time.
-a	Lists all entries including hidden files.
-d	Lists directory files instead of contents.
-p	Puts slash at the end of each directories.
-u	List in order of last access time.
-i	Display inode information.
-ltr	List files order by date.
-lSr	List files order by file size.

EXAMPLE:

1. Display root directory contents:
   ls /
   lists the contents of root directory.
2. Display hidden files and directories:
   ls -a
   lists all entries including hidden files and directories.
3. Display inode information:
   ls -i
   7373073 book.gif
   7373074 clock.gif
   The above command displays filename with inode value.

ln

ln command is used to create link to a file (or) directory.

It helps to provide soft link (shortcut) for desired files.

Inode will be different for source and destination.

SYNTAX:

The Syntax is

ln [options] existingfile(or directory)name newfile(or directory)name

OPTIONS:

-f	Link files without questioning the user, even if the mode of target forbids writing. This is the default if the standard input is not a terminal.
-n	Does not overwrite existing files.
-s	Used to create soft links.

EXAMPLE:

1. ln -s file1.txt file2.txt
   Creates a symbolic link to 'file1.txt' with the name of 'file2.txt'. Here inode for 'file1.txt' and 'file2.txt' will be different.
2. ln -s nimi nimi1
   Creates a symbolic link to 'nimi' with the name of 'nimi1'.

mkdir

mkdir command is used to create one or more directories.

SYNTAX:

The Syntax is

mkdir [options] directories

OPTIONS:

-m	Set the access mode for the new directories.
-p	Create intervening parent directories (path) if they don't
-v	Print help message for each directory created.

EXAMPLE:

1. Create directory:
   mkdir test
   The above command is used to create the directory 'test'.
2. Create directory and set permissions:
   mkdir -m 666 test
   The above command is used to create the directory 'test' and set the read and write permission.

rmdir

rmdir command is used to delete/remove a directory and its subdirectories.

SYNTAX:

The Syntax is

rmdir [options..] Directory

OPTIONS:

-p	Allow users to remove the directory dirname and its parent directories that become empty.

EXAMPLE:

1. To delete/remove a directory
   rmdir tmp
   rmdir command will remove/delete the directory tmp if the directory is empty.
2. To delete a directory tree:
   rm -ir tmp
   This command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.

mv

mv command is short for move. It is used to move/rename file

mv command is different from cp command as it completely removes the file from the source and moves to the directory specified, where cp command just copies the content from one file to another.

SYNTAX:

The Syntax is

mv [-f] [-i] oldname newname

OPTIONS:

-f	This will not prompt before overwriting (equivalent to --reply=yes). mv -f will move the file(s) without prompting even if it is writing over an existing target.
-i	Prompts before overwriting another file.

EXAMPLE:

1. To Rename / Move a file:
   mv file1.txt file2.txt
   This command renames file1.txt as file2.txt
2. To move a directory
   mv hscripts tmp
   In the above line mv command moves all the files, directories and sub-directories from hscripts folder/directory to tmp directory if the tmp directory already exists. If there is no tmp directory it renames the hscripts directory as tmp directory.
3. To Move multiple files/More files into another directory
   mv file1.txt tmp/file2.txt newdir
   This command moves the files file1.txt from the current directory and file2.txt from the tmp folder/directory to newdir.

diff

diff command is used to find differences between two files.

SYNTAX:

The Syntax is

diff [options..] from-file to-file

OPTIONS:

-a	Treat all files as text and compare them line-by-line.
-b	Ignore changes in amount of white space.
-c	Use the context output format.
-e	Make output that is a valid ed script.
-H	Use heuristics to speed handling of large files that have numerous scattered small changes.
-i	Ignore changes in case; consider upper- and lower-case letters equivalent.
-n	Prints in RCS-format, like -f except that each command specifies the number of lines affected.
-q	Output RCS-format diffs; like -f except that each command specifies the number of lines affected.
-r	When comparing directories, recursively compare any subdirectories found.
-s	Report when two files are the same.
-w	Ignore white space when comparing lines.
-y	Use the side by side output format.

EXAMPLE:

1. Compare files ignoring white space: diff -w file1.txt file2.txt
   This command will compare the file file1.txt with file2.txt ignoring white/blank space and it will produce output.

2. Compare the files side by side, ignoring white space: diff -by file1.txt file2.txt
   This command will compare the files ignoring white/blank space, It is easier to differentiate the files.

3. Compare the files ignoring case. diff -iy file1.txt file2.txt
   This command will compare the files ignoring case(upper-case and lower-case).

wc

Short for word count, wc displays a count of lines, words, and characters in a file.

Syntax

wc [*-c* | *-m* | *-C*] [*-l*] [*-w*] [*file* ...]

-c	Count bytes.
-m	Count characters.
-C	Same as -m.
-l	Count lines.

Examples

wc myfile.txt - Displays information about the file myfile.txt.

comm

Select or reject lines common to two files.

Syntax

comm [-1] [-2] [-3] file1 file2

-1	Suppress the output column of lines unique to file1.
-2	Suppress the output column of lines unique to file2.
-3	Suppress the output column of lines duplicated in file1 and file2.
file1	Name of the first file to compare.

Examples

comm myfile1.txt myfile2.txt

The above example would compare the two files myfile1.txt and myfile2.txt.

II. SECURITY BY FILE PERMISSIONS

This section will cover the following commands:

- chmod - modify file access rights
- su - temporarily become the superuser
- chown - change file ownership
- chgrp - change a file's group ownership

File permissions

Linux uses the same permissions scheme as UNIX. Each file and directory on your system is assigned access rights for the owner of the file, the members of a group of related users, and everybody else. Rights can be assigned to read a file, to write a file, and to execute a file (i.e., run the file as a program).

To see the permission settings for a file, we can use the `ls` command as follows:

```
$ ls -l filename
```

Let's try another example. We will look at the`bash` program which is located in the `/bin` directory:

```
$ ls -l /bin/bash
```

```
-rwxr-xr-x 1 root root 1183448 Feb 1  2023 /bin/bash
```

Here we can see:

- The file "/bin/bash" is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

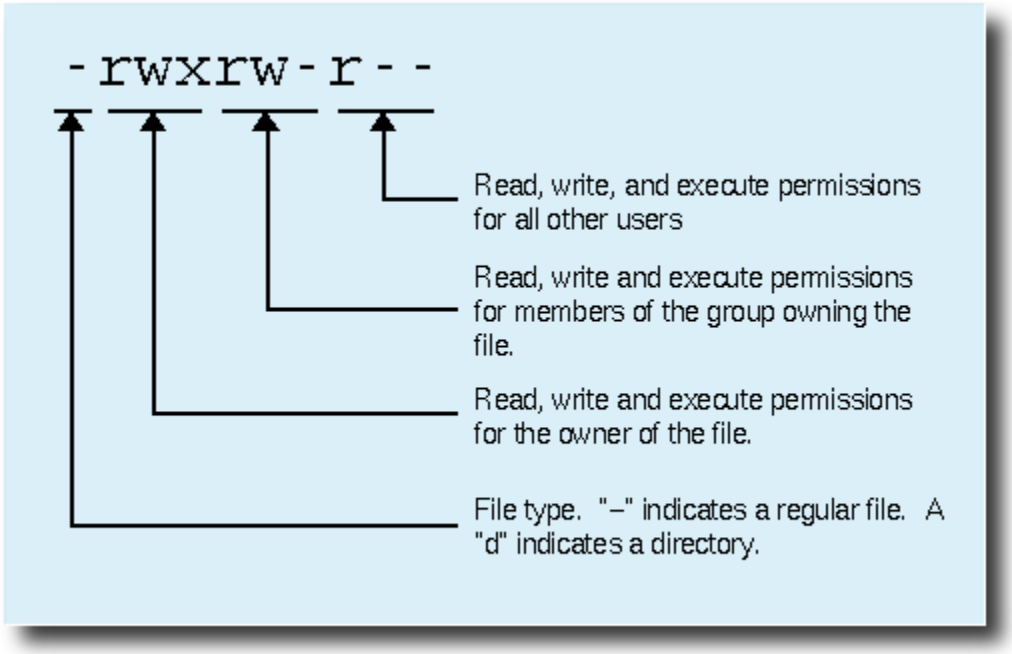
File Permission

#	File Permission
0	none
1	execute only
2	write only
3	write and execute
4	read only
5	read and execute
6	read and write
7	set all permissions

OPTIONS:

-c	Displays names of only those files whose permissions are being changed

In the diagram below, we see how the first portion of the listing is interpreted. It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.



chmod

The `chmod` command is used to change the permissions of a file or directory. To use it, you specify the desired permission settings and the file or files that you wish to modify. There are more ways to specify the permissions.

It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works:

rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7
rw- = 110 in binary = 6
r-x = 101 in binary = 5
r-- = 100 in binary = 4

Now, if you represent each of the three sets of permissions (owner, group, and other) as a single digit, you have a pretty convenient way of expressing the possible permissions settings. For example, if we wanted to set `some_file` to have read and write permission for the owner, but wanted to keep the file private from others, we would:

```
$ chmod 600 some_file
```

Here is a table of numbers that covers all the common settings. The ones beginning with "7" are used with programs (since they enable execution) and the rest are for other kinds of files.

Value	*Meaning*
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	*(rwxr-xr-x)* The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	*(rwx------)* The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	*(rw-rw-rw-)* All users may read and write the file.
644	*(rw-r--r--)* The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	*(rw-------)* The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Directory permissions

The `chmod` command can also be used to control the access permissions for directories. In most ways, the permissions scheme for directories works the same way as they do with files. However, the execution permission is used in a different way. It provides control for access to file listing and other things. Here are some useful settings for directories:

Value	*Meaning*
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.
755	*(rwxr-xr-x)* The directory owner has full access. All others may list the directory, but cannot create files nor delete them. This setting is common for directories that you wish to share with other users.
700	*(rwx------)* The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must be kept private from others.

-c Change the permission for each file.

su

Becoming the superuser for a short while

It is often useful to become the superuser to perform important system administration tasks, but as you have been warned (and not just by me!), you should not stay logged on as the superuser. In most distributions, there is a program that can give you temporary access to the superuser's privileges. This program is called `su`(short for substitute user) and can be used in those cases when you need to be the superuser for a small number of tasks. To become the superuser, simply type the `su` command. You will be prompted for the superuser's password:

```
$ su Password:
[root@Linuxbox me]#
```

After executing the `su`command, you have a new shell session as the superuser. To exit the superuser session, type `exit` and you will return to your previous session.

sudo

In some distributions, most notably Ubuntu, an alternate method is used. Rather than using `su`, these systems employ the `sudo` command instead. With `sudo`, one or more users are granted superuser privileges on an as needed basis. To execute a command as the superuser, the desired command is simply preceeded with the`sudo` command. After the command is entered, the user is prompted for the user's password rather than the superuser's:

```
$ sudo some_command Password:
```

III. PROCESS UTILITIES

ps

ps command is used to report the process status. ps is the short name for Process Status.

SYNTAX:

The Syntax is ps
[options]

OPTIONS:

-a	List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal..
-A or e	List information for all processes.
-d	List information about all processes except session leaders.
-e	List information about every process now running.

-f Generates a full listing.
-j Print session ID and process group ID.
-l Generate a long listing.

EXAMPLE:

1. ps

Output:

```
PID TTY          TIME CMD
2540 pts/1    00:00:00 bash
2621 pts/1    00:00:00 ps
```

In the above example, typing ps alone would list the current running processes.

kill

kill command is used to kill the process.

SYNTAX:

The Syntax is

kill [-s] [-l] %pid

OPTIONS:

-s	Specify the signal to send. The signal may be given as a signal name or number.
-l	Write all values of signal supported by the implementation, if no operand is given.
-pid	Process id or job id.
-9	Force to kill a process.

EXAMPLE:

Step by Step process:

- Open a process music player.
  xmms
  press ctrl+z to stop the process.
- To know group id or job id of the background task.
  jobs -l
- It will list the background jobs with its job id as,
- xmms 3956
  kmail 3467
- To kill a job or process.
  kill 3956
  kill command kills or terminates the background process xmms.

at

Schedules a command to be ran at a particular time

Syntax

at executes commands at a specified time.

atq lists the user's pending jobs, unless the user is the superuser; in that case, everybody's jobs are listed. The format of the output lines (one for each job) is: Job number, date, hour, job class.

atrm deletes jobs, identified by their job number.

batch executes commands when system load levels permit; in other words, when the load average drops below 1.5, or the value specified in the invocation of atrun.

at [-c | -k | -s] [-f filename] [-q queuename] [-m] -t time [date] [-l] [-r]

-c C shell. csh(1) is used to execute the at-job.

-k Korn shell. ksh(1) is used to execute the at-job.

-s Bourne shell. sh(1) is used to execute the at-job.

-f filename Specifies the file that contains the command to run.

-m Sends mail once the command has been run.

-t time Specifies at what time you want the command to be ran. Format hh:mm. am / pm indication can also follow the time otherwise a 24-hour clock is used. A timezone name of GMT, UCT or ZULU (case insensitive) can follow to specify that the time is in Coordinated Universal Time. Other timezones can be specified using the TZ environment variable. The below quick times can also be entered:

midnight - Indicates the time 12:00 am (00:00).
noon - Indicates the time 12:00 pm.
now - Indicates the current day and time. Invoking at - now will submit submit an at-job for potentially immediate execution.

date Specifies the date you wish it to be ran on. Format month, date, year. The following quick days can also be entered:

today - Indicates the current day.
tomorrow - Indicates the day following the current day.

-l Lists the commands that have been set to run.

-r Cancels the command that you have set in the past.

Examples

at -m 01:35 < atjob

Run the commands listed in the 'atjob' file at 1:35AM, in addition to all output that is generated from job mail to the user running the task.

III. FILTERS

more

more command is used to display text in the terminal screen.

SYNTAX:

The Syntax is

more [options] filename

OPTIONS:

-c	Clear screen before displaying.
-e	Exit immediately after writing the last line of the last file in the argument list.
-n	Specify how many lines are printed in the screen for a given file.
+n	Starts up the file from the given number.

EXAMPLE:

1. more -c index.php
   Clears the screen before printing the file .
2. more -3 index.php
   Prints first three lines of the given file. Press **Enter** to display the file line by line.

head

head command is used to display the first ten lines of a file, and also specifies how many lines to display.

SYNTAX:

The Syntax is

head [options] filename

OPTIONS:

-n	To specify how many lines you want to display.
-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.

EXAMPLE:

1. head index.php
   This command prints the first 10 lines of 'index.php'.
2. head -5 index.php

The head command displays the first 5 lines of 'index.php'.

3. head -c 5 index.php

The above command displays the first 5 characters of 'index.php'.

tail

tail command is used to display the last or bottom part of the file. By default it displays last 10 lines of a file.

SYNTAX:

The Syntax is

tail [options] filename

OPTIONS:

-l	To specify the units of lines.
-b	To specify the units of blocks.
-n	To specify how many lines you want to display.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.
-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.

EXAMPLE:

1. tail index.php

It displays the last 10 lines of 'index.php'.

2. tail -2 index.php

It displays the last 2 lines of 'index.php'.

3. tail -n 5 index.php

It displays the last 5 lines of 'index.php'.

4. tail -c 5 index.php

It displays the last 5 characters of 'index.php'.

cut

cut command is used to cut out selected fields of each line of a file. The cut command uses delimiters to determine where to split fields.

SYNTAX:

The Syntax is cut

[options]

OPTIONS:

-c	Specifies character positions.
-b	Specifies byte positions.
-d flags	Specifies the delimiters and fields.

EXAMPLE:

1. cut -c1-3 text.txt

   Output:

   Thi

   Cut the first three letters from the above line.

2. cut -d, -f1,2 text.txt

   Output:

   This is, an example program

   The above command is used to split the fields using delimiter and cut the first two fields.

paste

paste command is used to paste the content from one file to another file. It is also used to set column format for each line.

SYNTAX:

The Syntax is paste [options]

OPTIONS:

-s	Paste one file at a time instead of in parallel.
-d	Reuse characters from LIST instead of TABs

EXAMPLE:

1. paste test.txt>test1.txt

   Paste the content from 'test.txt' file to 'test1.txt' file.

2. ls | paste - - - -

   List all files and directories in four columns for each line.

sort

sort command is used to sort the lines in a text file.

SYNTAX:

The Syntax is sort [options] filename

OPTIONS:

-r	Sorts in reverse order.
-u	If line is duplicated display only once.
-o filename	Sends sorted output to a file.

EXAMPLE:

1. sort test.txt

   Sorts the 'test.txt'file and prints result in the screen.

2. sort -r test.txt

   Sorts the 'test.txt' file in reverse order and prints result in the screen.

uniq

Report or filter out repeated lines in a file.

Syntax

uniq [-c | -d | -u] [-f fields] [-s char] [-n] [+m] [input_file [output_file]]

-c	Precede each output line with a count of the number of times the line occurred in the input.
-d	Suppress the writing of lines that are not repeated in the input.
-u	Suppress the writing of lines that are repeated in the input.
-f fields	Ignore the first fields fields on each input line when doing comparisons, where fields is a positive decimal integer. A field is the maximal string matched by the basic regular expression: [[:blank:]]*[^[:blank:]]* If fields specifies more fields than appear on an input line, a null string will be used for comparison.
-s char	Ignore the first chars characters when doing comparisons, where chars is a positive decimal integer. If specified in conjunction with the -f option, the first chars characters after the first fields fields will be ignored. If chars specifies more characters than remain on an input line, a null string will be used for comparison.
-n	Equivalent to -f fields with fields set to n.
+m	Equivalent to -s chars with chars set to m.
input_file	A path name of the input file. If input_file is not specified, or if the input_file is -, the standard input will be used.
output_file	A path name of the output file. If output_file is not specified, the standard output will be used. The results are unspecified if the file named by output_file is the file named by input_file.

Examples

uniq myfile1.txt > myfile2.txt - Removes duplicate lines in the first file1.txt and outputs the results to the second file.

IV. General Commands

date

date command prints the date and time.

SYNTAX:

The Syntax is

date [options] [+format] [date]

OPTIONS:

-a	Slowly adjust the time by sss.fff seconds (fff represents fractions of a second). This adjustment can be positive or negative.Only system admin/ super user can adjust the time.
-s date-string	Sets the time and date to the value specfied in the datestring. The datestr may contain the month names, timezones, 'am', 'pm', etc.
-u	Display (or set) the date in Greenwich Mean Time (GMT-universal time).

Format:

%a	Abbreviated weekday(Tue).
%A	Full weekday(Tuesday).
%b	Abbreviated month name(Jan).
%B	Full month name(January).
%c	Country-specific date and time format..
%D	Date in the format %m/%d/%y.
%j	Julian day of year (001-366).
%n	Insert a new line.
%p	String to indicate a.m. or p.m.
%T	Time in the format %H:%M:%S.
%t	Tab space.
%V	Week number in year (01-52); start week on Monday.

EXAMPLE:

1. date command
   date
   The above command will print
2. To use tab space:
   date +"Date is %D %t Time is %T"
   The above command will remove space and print as To know the week number of the year, date -V
3. To set the date,
   date -s "1/28/2023 11:37:23"

echo

echo command prints the given input string to standard output.

SYNTAX:

The Syntax is

echo [options..] [string]

OPTIONS:

-n	do not output the trailing newline
-e	enable interpretation of the backslash-escaped characters listed below
-E	disable interpretation of those sequences in STRINGs

Without -E, the following sequences are recognized and interpolated:

\NNN	the character whose ASCII code is NNN (octal)
\a	alert (BEL)
\\	backslash
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

EXAMPLE:

1. echo command
   echo "CS 4440 OS"
   The above command will print as CS 4440 OS.
2. To use backspace:
   echo -e "CS \4440 \OS"
   The above command will remove space and print as CS4440OS.
3. To use tab space in echo command
   echo -e "CS\t4440\tOS"
   The above command will print as CS 4440 OS

passwd

passwd command is used to change your password.

SYNTAX:

The Syntax is

passwd [options]

OPTIONS:

-a	Show password attributes for all entries.
-l	Locks password entry for name.
-d	Deletes password for name. The login name will not be prompted for password.
-f	Force the user to change password at the next login by expiring the password for name.

EXAMPLE:

1. passwd

   Entering just passwd would allow you to change the password. After entering passwd you will receive the following three prompts:

   Current Password:

   New Password:

   Confirm New Password:

   Each of these prompts must be entered correctly for the password to be successfully changed.

pwd

pwd - Print Working Directory. pwd command prints the full filename of the current working directory.

SYNTAX:

The Syntax is

pwd [options]

OPTIONS:

-P	The pathname printed will not contain symbolic links.
-L	The pathname printed may contain symbolic links.

EXAMPLE:

1. Displays the current working directory.
   pwd
   If you are working in home directory then, pwd command displays the current working directory as /home.

cal

cal command is used to display the calendar.

SYNTAX:

The Syntax is

cal [options] [month] [year]

OPTIONS:

-1	Displays single month as output.
-3	Displays prev/current/next month output.
-s	Displays sunday as the first day of the week.
-m	Displays Monday as the first day of the week.
-j	Displays Julian dates (days one-based, numbered from January 1).
-y	Displays a calendar for the current year.

EXAMPLE:

1. cal

cal command displays the current month calendar.

2. cal -3 5 2023

Chapter 2: Operating-System Structures

Design Goals

An operating system provides the environment within which programs are executed.

It is important that the goals of the system be well defined before the design begins.

1. One view focuses on the services that the system provides;
2. Another, on the interface that it makes available to users and programmers;
3. A third, on its components and their interconnections.

2.1 Operating-System Services

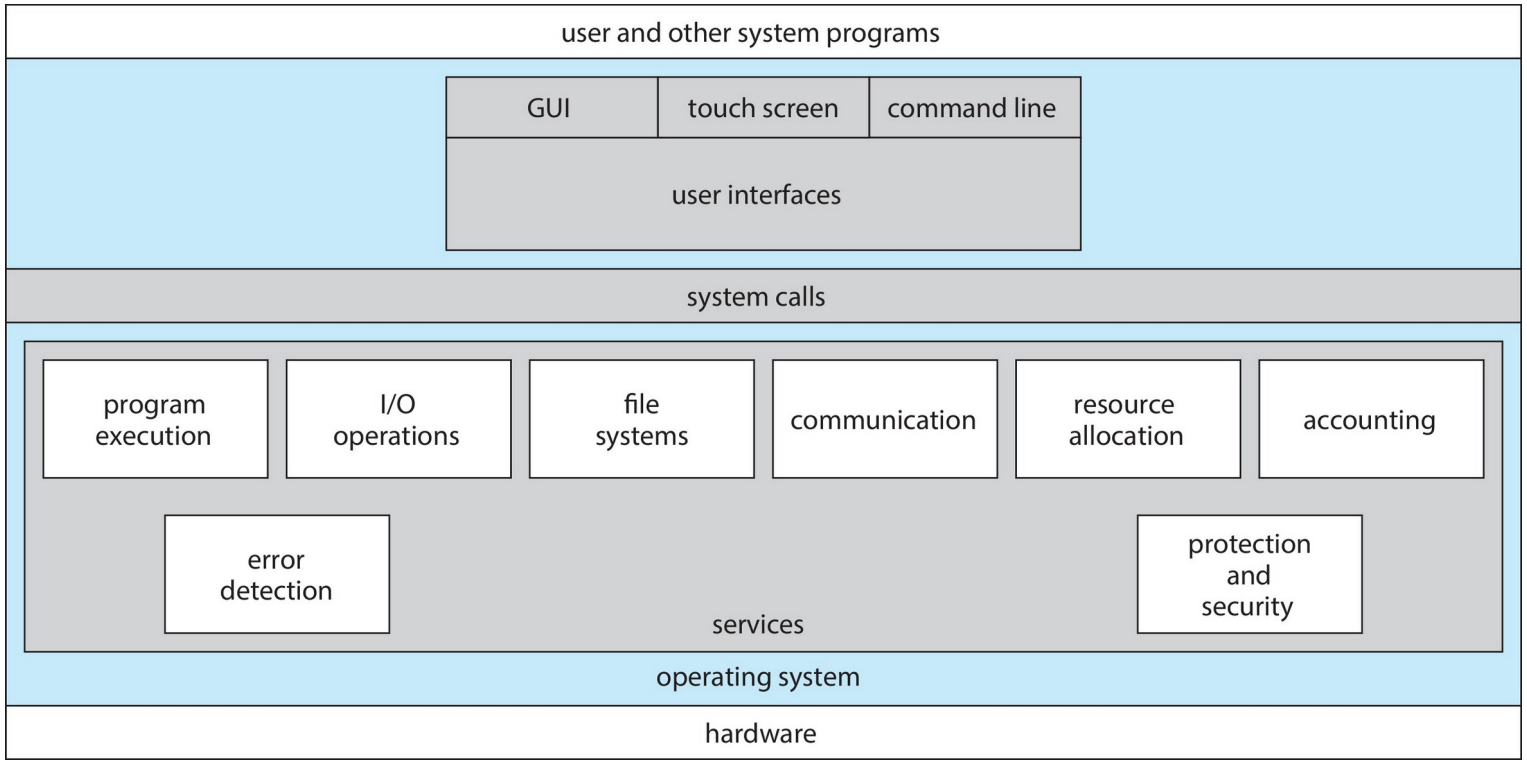
Operating System Services

The specific services provided differ from one operating system to another, but we can identify common classes.

These services make the programming task easier for the programmer

1. One set of operating-system services provides functions that are helpful to the user
2. Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

A View of Operating System Services



1. Operating System Services- for user ease

One set of operating-system services provides functions that are helpful to the user:

1. **User interface** - Almost all operating systems have a user interface (**UI**).
   - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen, Batch**
2. **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
3. **I/O operations** - A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

Operating System Services - for user (cont.)

4. **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management. Many operating systems provide a variety of file systems.
5. **Communications** – Processes may exchange information, on the same computer or between computers over a network
   - Communications may be via shared memory or through message passing (packets moved by the OS)
6. **Error detection** – OS needs to be constantly aware of possible errors
   - May occur in the CPU and memory hardware, in I/O devices, in user program
   - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
   - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

2. Operating System Services – for system efficiency

Another set of OS functions exists for ensuring the **efficient operation** of the **system** itself via **resource sharing:**

1. **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
   - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
2. **Logging** - To keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.
3. **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
   - **Protection** involves ensuring that all access to system resources is controlled
   - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

2.2 User and Operating-System Interface

User Interfaces

There are several ways for users to interface with the operating system.

Three fundamental approaches:

One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system.

The other two allow users to interface with the operating system via a graphical user interface, or GUI.

Command-line interface - CLI

CLI or **command interpreter** allows users to directly enter commands to be performed by the operating system

- On systems with multiple command interpreters to choose from, the interpreters are known as **shells**.
  - UNIX and Linux systems provide several shells, including the C shell, Bourne-Again shell (bash), and Korn shell. Third-party shells and free user-written shells are also available.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on.

Command interpreter implementation

These commands can be implemented in two general ways:

1. Command interpreter itself contains the code to execute the command.
   - Code sets up the parameters and makes the appropriate system call
   - The number of commands that can be given determines the size of the command interpreter since each command requires its own implementing code

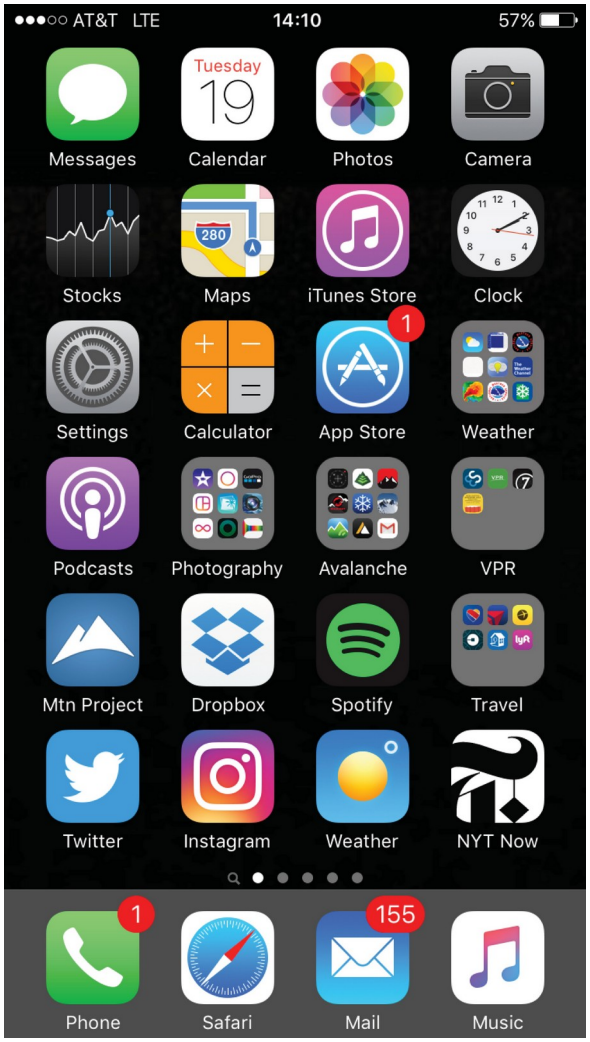
2. An alternative approach—used by UNIX, among other operating systems —implements most commands through system programs. Merely uses the command to identify a file to be loaded into memory and executed.
   - Example: UNIX command to delete a file `rm file.txt` would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`.
   - Programmers can add new commands to the system easily by creating new files with the proper program logic.

Graphical User Interface- GUI

- User-friendly **desktop** metaphor **interface**
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows - GUI with **CLI** "command" shell
  - Apple macOS - "***Aqua***" GUI interface with **UNIX kernel** underneath and **shells** available
  - Unix and Linux have **CLI** with optional GUI interfaces (**CDE, KDE** *K(ool) Desktop Environment*, ***GNOME*** desktop by the GNU project

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference.

- **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface.
  - It is more efficient, giving faster access to the activities they need to perform.
  - On some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable.
  - Further, command-line interfaces make repetitive tasks easier, in part because they have their own programmability.
    - For example, if a frequent task requires a set of steps, those steps can be recorded into a file, and that file can be run just like a program.
    - The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

2.3 System Calls

System Calls

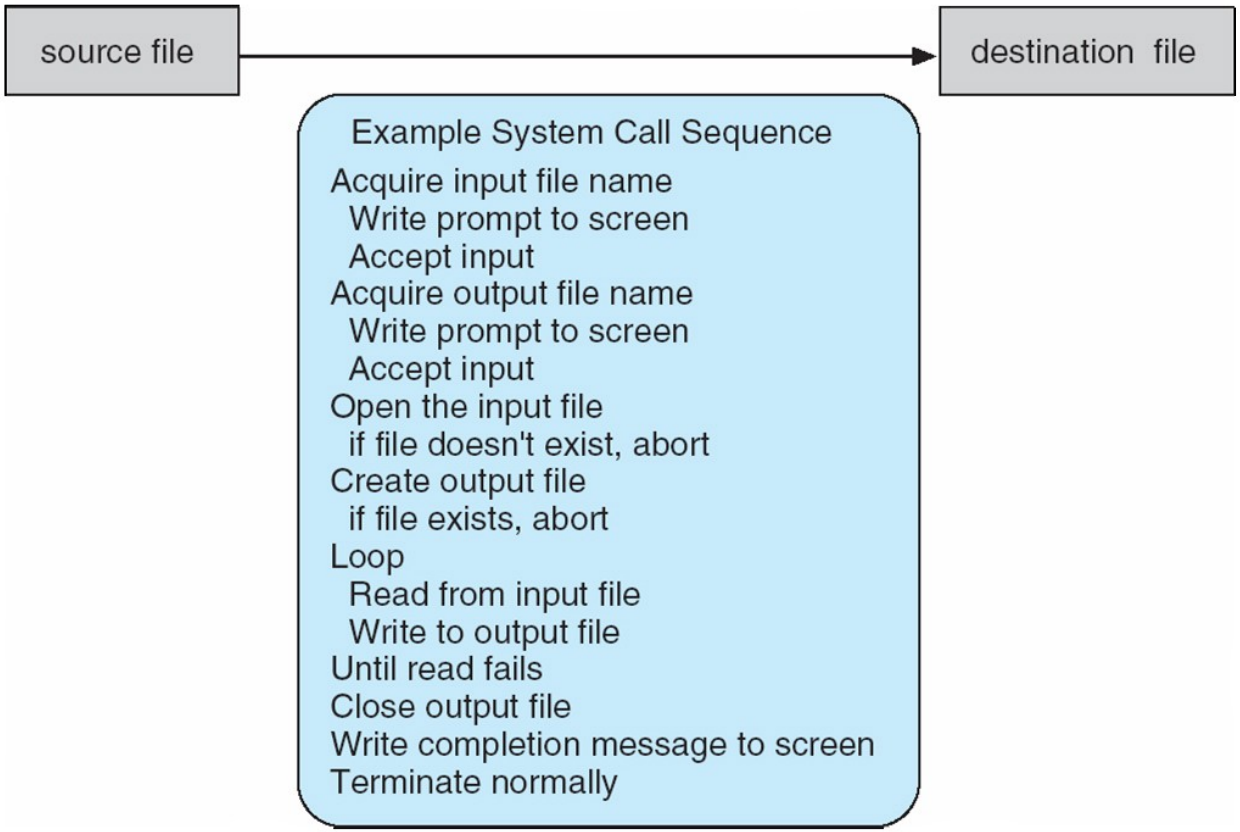
System calls provide an **interface** to the services provided by the **OS**

- Generally available as functions written in C and C++
- Certain low-level tasks may have to be written using assembly-language instructions.
  - Example: tasks where hardware must be accessed directly
    - Even simple programs may make heavy use of the operating system.
    - Frequently, systems execute thousands of system calls per second.

    Most programmers never see this level of detail.
    - Each operating system has its own name for each system call.

Example of System Calls

- System call sequence to copy the contents of one file to another file



Application Programming Interface

System Calls are mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use.

- The **API specifies a set of functions** that are **available to an application programmer**, including the parameters that are passed to each function and the return values the programmer can expect.

- Functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
- An API can specify the interface between an application and the operating system.

- A programmer accesses an API via a library of code provided by the operating system.
  - In UNIX and Linux for programs written in the C language, the library is called **libc**

System Calls (cont.)

Two modes of OS:

1. **Kernel mode:** Privileged and powerful mode used by the operating system kernel

2. **User mode:** Where most user applications run

- **System calls** work silently in the background, **interfacing with the kernel** to get work done.

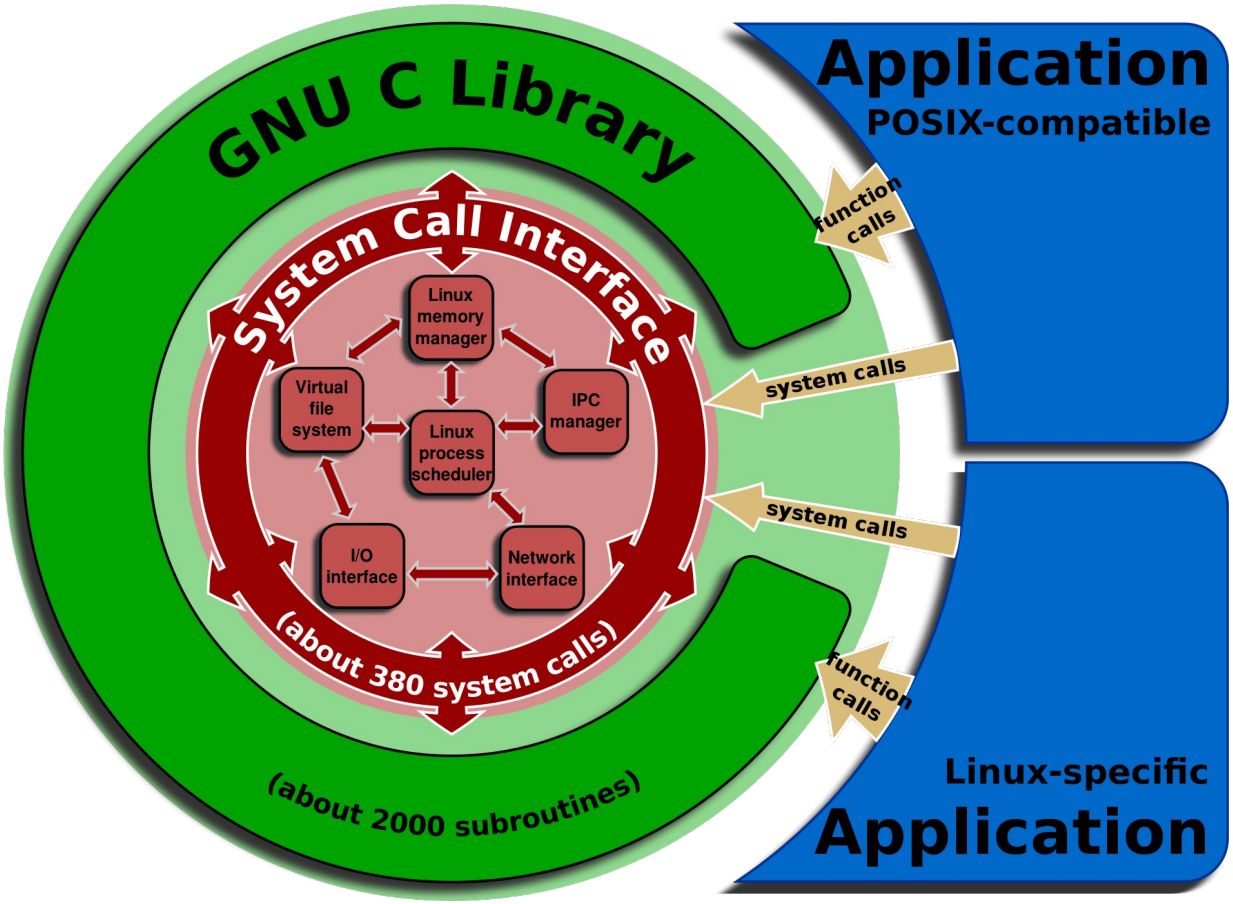
- System calls are very similar to function calls, which means they accept and work on arguments and return values.
  The only difference is that **system calls enter a kernel**, while function calls do not.

- Switching from user space to kernel space is done using trap (exception). Most of this is hidden away from the user by using system libraries (**glibc** on Linux systems).

- Even though system calls are generic in nature, the mechanics of issuing a system call are very much machine-dependent.

Linux API

The Linux API is the kernel–user space API, which allows programs in user space to access **system resources** and **services of the Linux kernel**. It is composed out of the System Call Interface of the Linux kernel and the subroutines in the GNU C Library (glibc).



Most Common APIs

Three of the **most common APIs** are:

1. **Win32 API** for **Windows,**

2. **POSIX API** for **POSIX-based systems (including virtually all versions of UNIX, Linux, and macOS**), and

   Portable Operating System Interface is a family of standards for compatibility between operating systems

3. **Java API** for the **Java virtual machine (JVM)**

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Reasons for API

Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

1. Program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API. (Architectural differences often make this more difficult
2. System calls can often be more detailed and difficult to work with than the API available to an application programmer.

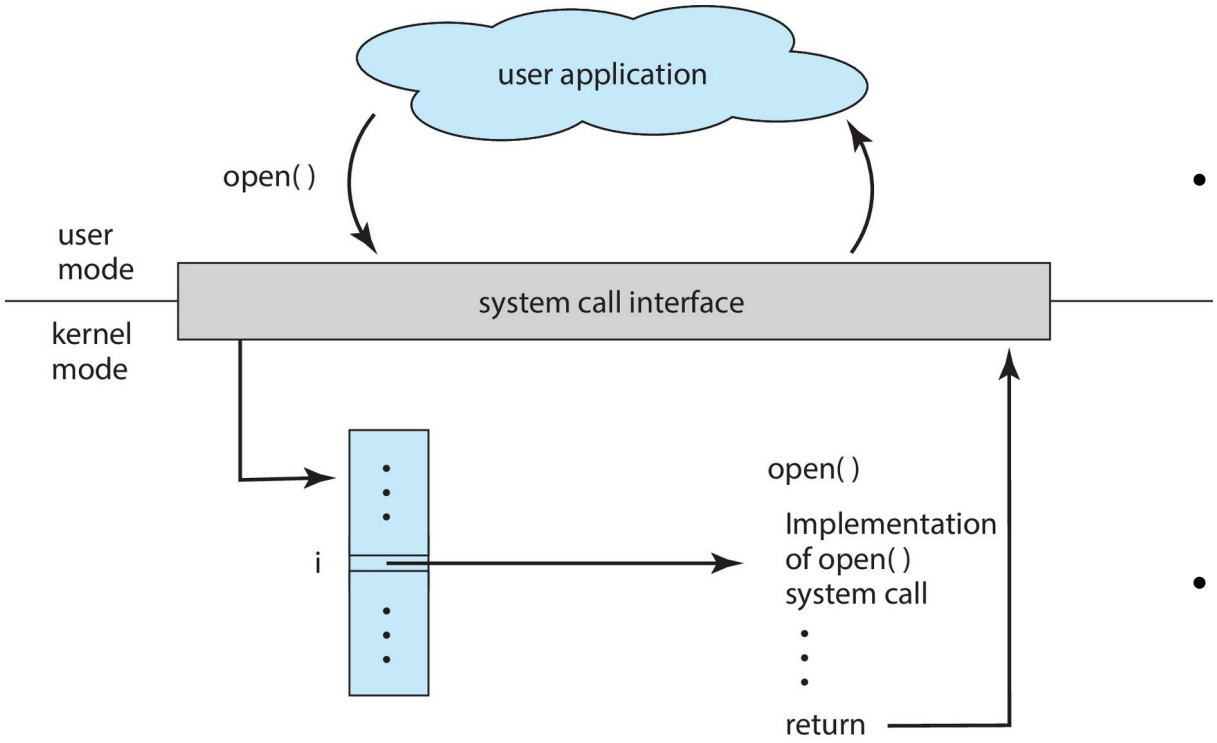
- The caller need know nothing about how the system call is implemented
- Most **details of OS interface hidden from programmer by API**

Correlation between a function in the API and its associated system call within the kernel:

Many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

API – System Call – OS Relationship

The handling of a user application invoking the `open()` system call



- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the OS
- Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system-call interface then invokes the intended system call in the **operating-system kernel** and returns the status of the system call.

2.3.3 Types of System Calls

Types of System Calls

- **Process control**
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

Types of System Calls (cont.)

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- **Information maintenance**
  - get **time or date**, set time or date
  - get **system data**, set system data
  - get and set process, file, or device **attributes**
- **Communications**
  - create, delete **communication connection**
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain **access to memory** regions
  - transfer status information
  - attach and detach **remote devices**

Types of System Calls (Cont.)

- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

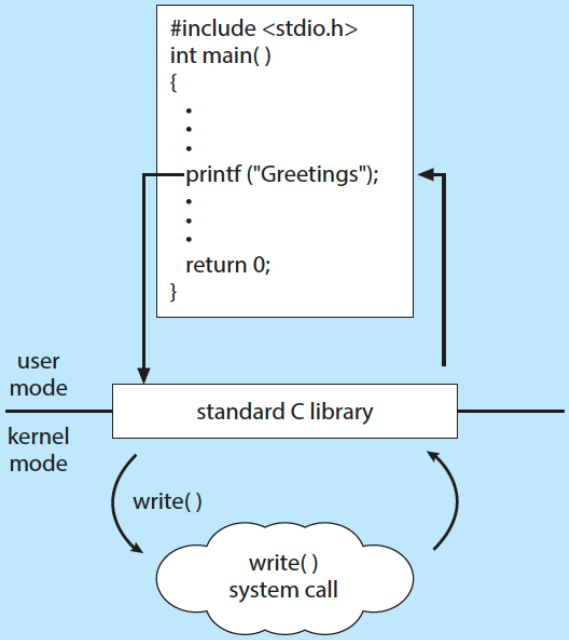
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

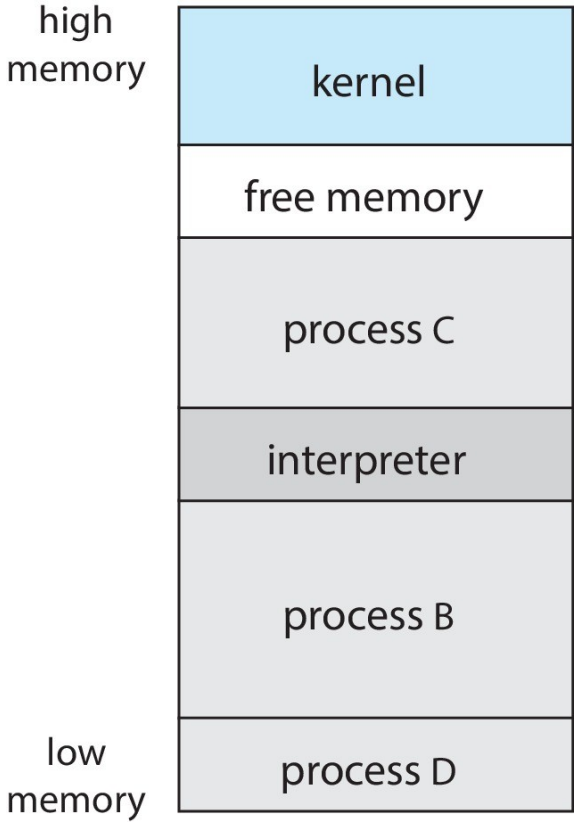
THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



Example: FreeBSD - Multitasking

- Unix variant – from Berkeley UNIX
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes
  - fork() system call to create process
  - exec() to load program into process
  - Shell waits for process to terminate
  - or runs the process "in the background" and continues with other user commands. Here,
    - cannot receive input from keybord
    - I/O done through files or a GUI
- Process exits with exit() command:
  - code = 0 – no error
  - code > 0 – error code



2.4 System Services

System Services

1. File management
2. Status information sometimes stored in a file
3. Programming language support
4. Program loading and execution
5. Communications
6. Background services
7. Application programs

System Services (System utilities)

System services, also known as system utilities, provide a convenient **environment for program development and execution**.

Some of them are simply user interfaces to system calls. Others are considerably more complex.

1. **File management** - **Create, delete, copy, rename, print, dump, list**, and generally **manipulate** files and directories

2. **Status information**
   - Some ask the system for info - **date, time, amount of available memory, disk space, number of users**
   - Others provide detailed performance, logging, and debugging information
   - Typically, these programs format and print the output to the terminal or other output devices
   - Some systems implement a **registry** - used to store and retrieve configuration (specific hardware and software details) information

System Services (Cont.)

3. **File modification**
   - Text editors to create and modify files
   - Special commands to search contents of files or perform transformations of the text
4. **Programming-language support** - Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, and Python)
5. **Program loading and execution-** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide Absolute loaders, Relocatable loaders, Linkage editors, Overlay-loaders, and debugging systems
6. **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
   - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Services (Cont.)

6. **Background Services**
   - Launch at boot time
     - Some for system startup, then terminate
     - Others continue to run till shutdown
   - Constantly running system-program processes are known as **services**, **subsystems**, or daemons.
   - Provide facilities like disk checking, process scheduling, error logging, printing
   - OS that run important activities in user context rather than in kernel context may use daemons to run these activities.
7. **Application programs**
   - Don't pertain to system
   - Run by users
   - Not typically considered part of OS

Application programs

Along with system programs, most operating systems are supplied with programs that are **useful in solving common problems** or performing **common operations**.

Application programs include

- web browsers,
- word processors and text formatters,
- spreadsheets,
- database systems,
- compilers,
- plotting and statistical-analysis packages,
- Games
- …

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls.

2.5 Linkers and Loaders

Program → Process

Programs reside on disk as a binary executable. (e.g. prog.exe or a.out). To run on CPU, must be brought into memory and placed in the context of a **process**

Steps:

1. **Compiler**

   Source files (.c, .cpp, etc.) **compiled** into **object files** designed to be loaded into any physical memory location, a format known as **relocatable object file (.obj)**

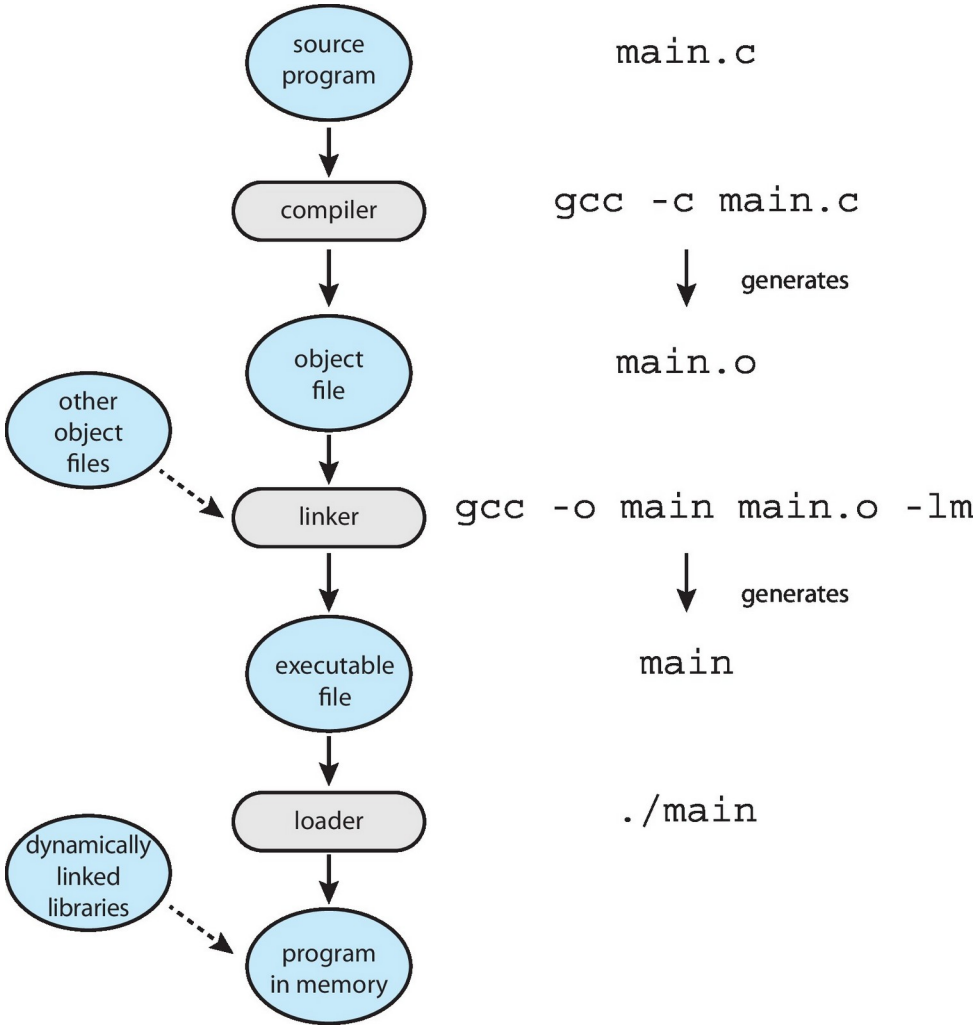
2. **Linker**

   1. **Linker** combines object (.obj) files into single binary **executable file (.exe)**
      - Linking also includes other object files or libraries

3. **Loader**

   - To run on a CPU core, the binary executable (.exe) file is brought into memory by **loader**, and be placed in the context of a **process**
   - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses

The Role of the Linker and Loader



gcc -c compiles source files without linking

Converts into binary file .o object code

gcc -l links with link shared libraries
-lm includes math library
gcc -o generates an executable output program

Example Program Execution

1. Enter the program name on the command line. for example, ./main

2. The shell first **creates** a **new process** to run the program using the fork() system call.

3. The shell then invokes the loader with the exec() system call, passing exec() the name of the executable file.

4. The loader then **loads** the specified program **into memory** using the ***address space*** of the newly created process.

When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.

DLL, ELF

Not all libraries are linked into the executable file and loaded into memory

- Rather, **dynamically linked libraries** (in **Windows**, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
  - The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally **linked** and is **loaded** if it is required during program **run time**.
- Object, executable files have standard formats, so operating system knows how to load and start them.
  - Format includes compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program
    - In UNIX and **Linux** the standard format is **ELF** (Executable and Linkable Format)

Why Applications are Operating System Specific

Apps compiled on one system usually not executable on other operating systems

- Each operating system provides its own **unique system calls**
- Own file formats, etc.

Apps can run on **multiple operating system** in one of 3 ways:

1. Written in interpreted language like Python, Ruby, and **interpreter available** for multiple operating systems
2. App written in language that **includes a VM** containing the running app
   - like Java which includes loader, byte-code verifier, etc.
3. **Use standard language or API,** compile separately on each operating system to run on each
   - Like POSIX API on variants of UNIX like OS

- The application can be written in an interpreted language (such as Python

or Ruby) that has an interpreter available for multiple operating systems.

- The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications.

Application Binary Interface (ABI)

APIs specify certain functions at the application level. **ABI** (Application Binary Interface) is the architecture equivalent of API

An ABI is used to define how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization, and so on.

If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI.

Cross-platform compatibility

These differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system
on a specific CPU type (such as Intel x86 or ARMv8), the application will fail to run.

- **RTE (run-time environment)**—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.

Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

2.7 Operating-System Design and Implementation

OS Design Goals

- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System goals**
  - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Implementation

- Much variation
  - Early Operating Systems in assembly language
  - Now many are written in C, C++, with small amounts of the system written in assembly language.
- In fact, more than one higher level language is often used.
  - The lowest levels of the kernel might be written in assembly language and C.
  - Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages.
- More high-level the language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

2.8 Operating-System Structure

Operating System Structure

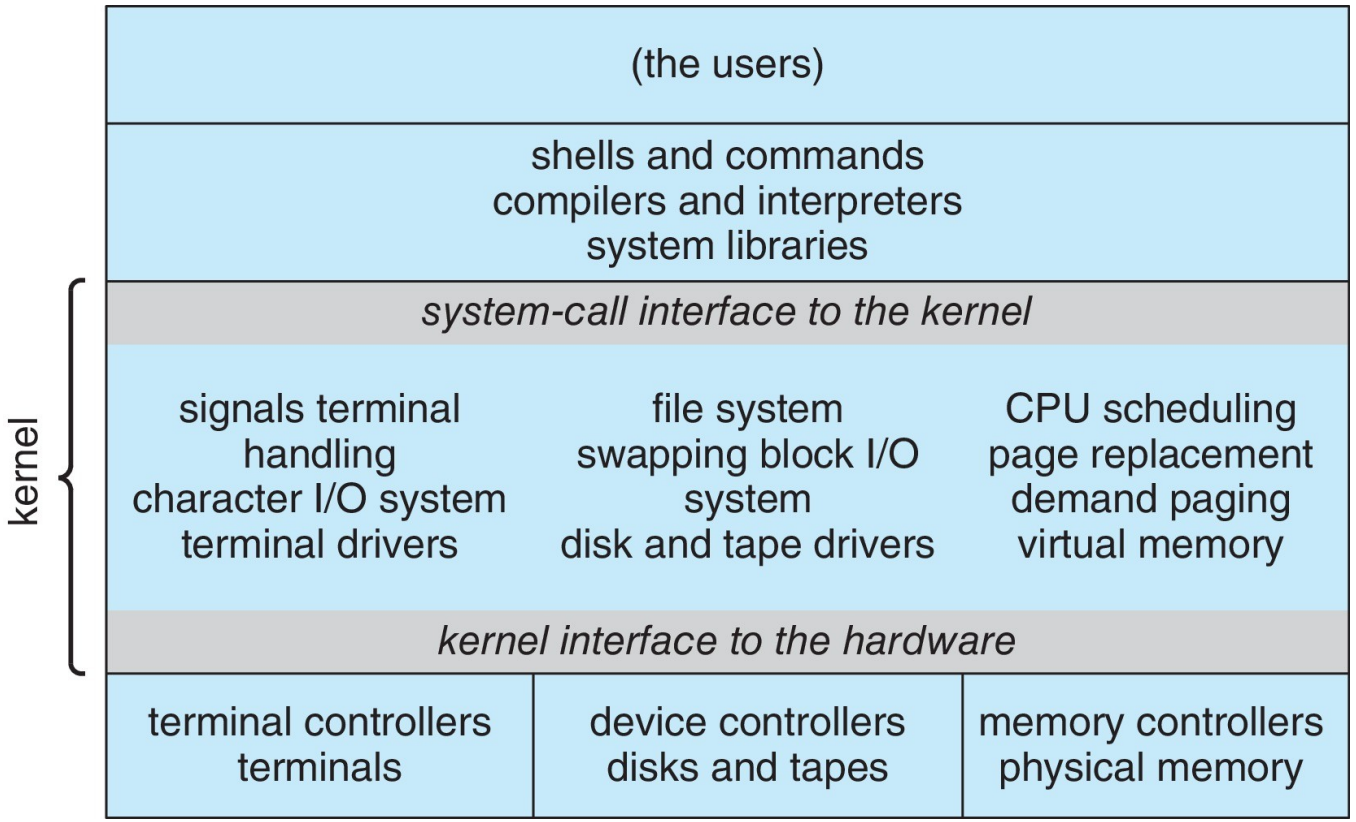
- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- A common approach is to partition the task into small components, or modules, rather than have one single system.
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

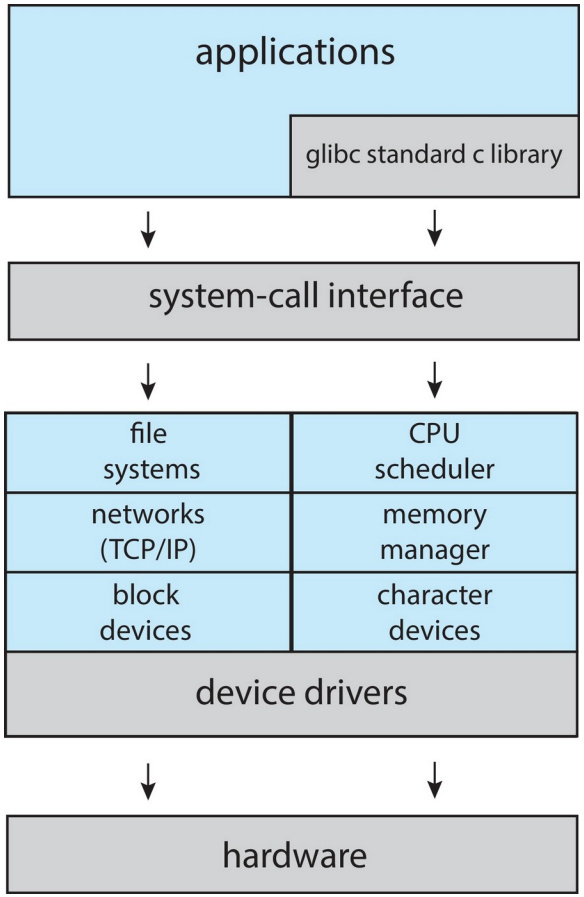
Traditional UNIX System Structure

Beyond simple but not fully layered



Linux System Structure

Monolithic plus modular design

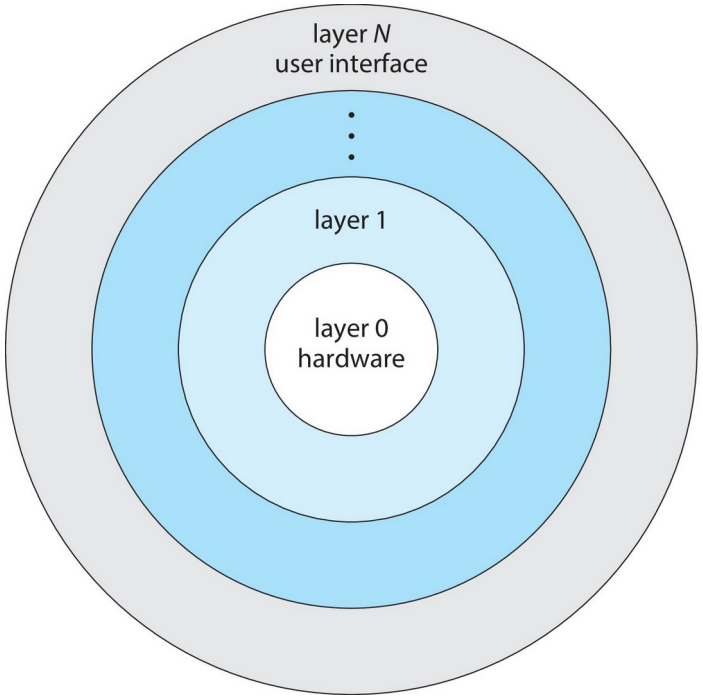


The Linux operating system is based on UNIX and is structured similarly.

Applications typically use the glibc standard C library when communicating with the system call interface to the kernel.
The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but it does have a modular design that allows the kernel to be modified during run time.

Layered Approach

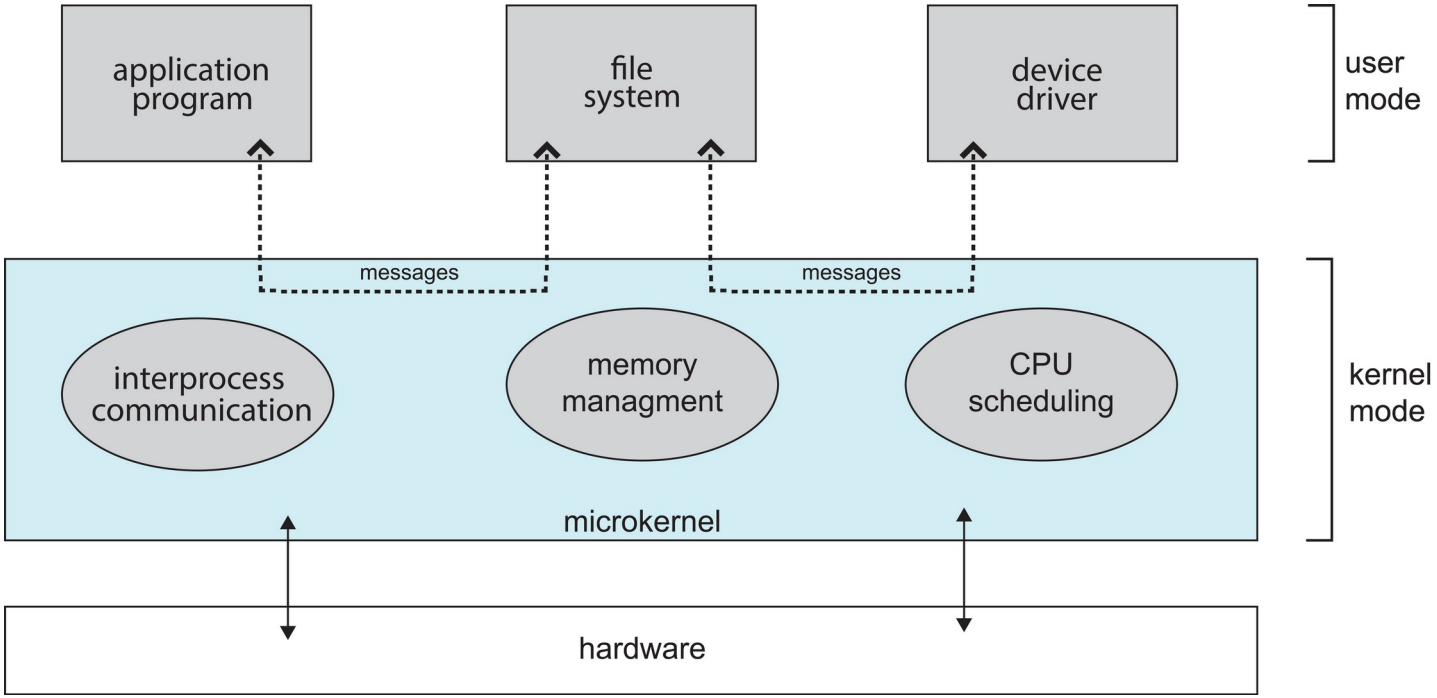
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

GNU nano text editor

It is a popular text editor among Linux users.

Opening and Creating Files

To open an existing file or to create a new file, type `nano` followed by the file name:

```
nano filename
```

This opens a new editor window, and you can start editing the file.

- At the bottom of the window, there is a list of the most basic command shortcuts to use with the nano editor.

  All commands are prefixed with either `^` or `M` character. The caret symbol (`^`) represents the `Ctrl` key. For example, the `^J` commands mean to press the `Ctrl` and `J` keys at the same time. The letter `M` represents the `Alt` key.

  You can get a list of all commands by typing `Ctrl+g`

- If you want to open a file with the cursor on a specific line and character use the following syntax:

  ```
  nano +line_number,character_number filename
  ```

  If you omit the `character_number` the cursor will be positioned on the first character.

  For example to open the 3rd line 5th character in file named foo:

  ```
  nano +3,5 foo
  ```

- To move the cursor to a specific line and character number, use the `Ctrl+_` command. (i.e, `Ctrl+ underscore.)` The menu on the bottom of the screen will change. Enter the number(s) in the “Enter line number, column number:” field and hit `Enter`.

Searching and replacing

To search for a text, press `Ctrl+w`, type in the search term, and press `Enter`. The cursor will move to the first match. To move to the next match, press `Alt+w`.

If you want to search and replace, press `Ctrl+\`. Enter the search term and the text to be replaced with. The editor will move to the first match and ask you whether to replace it. After hitting `Y` or `N` it will move to the next match. Pressing `A` will replace all matches.

Copying, cutting, and pasting

To select text, move the cursor to the beginning of the text and press `Alt+a`. This will set a selection mark. Move the cursor to the end of the text you want to select using the arrow keys. The selected text will be highlighted. If you want to cancel the selection press `Alt+6`

Copy the selected text to the clipboard using the `Alt+6` command. `Ctrl+k` will cut the selected text.

If you want to cut whole lines, simply move the cursor to the line and press `Ctrl+k`. You can cut multiple lines by hitting `Ctrl+k` several times.

To paste the text move the cursor to where you want to put the text and press `Ctrl+u`

Saving and Exiting

To save the changes you've made to the file, press `Ctrl+o`. If the file doesn't already exist, it will be created once you save it.

To exit nano press `Ctrl+x`. If there are unsaved changes, you'll be asked whether you want to save the changes.

For more information about Gnu Nano visit the official nano documentation page.

Reference:

https://linuxize.com/post/how-to-use-nano-text-editor/