Operating system

Chapter 1

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

Thus, some operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two.

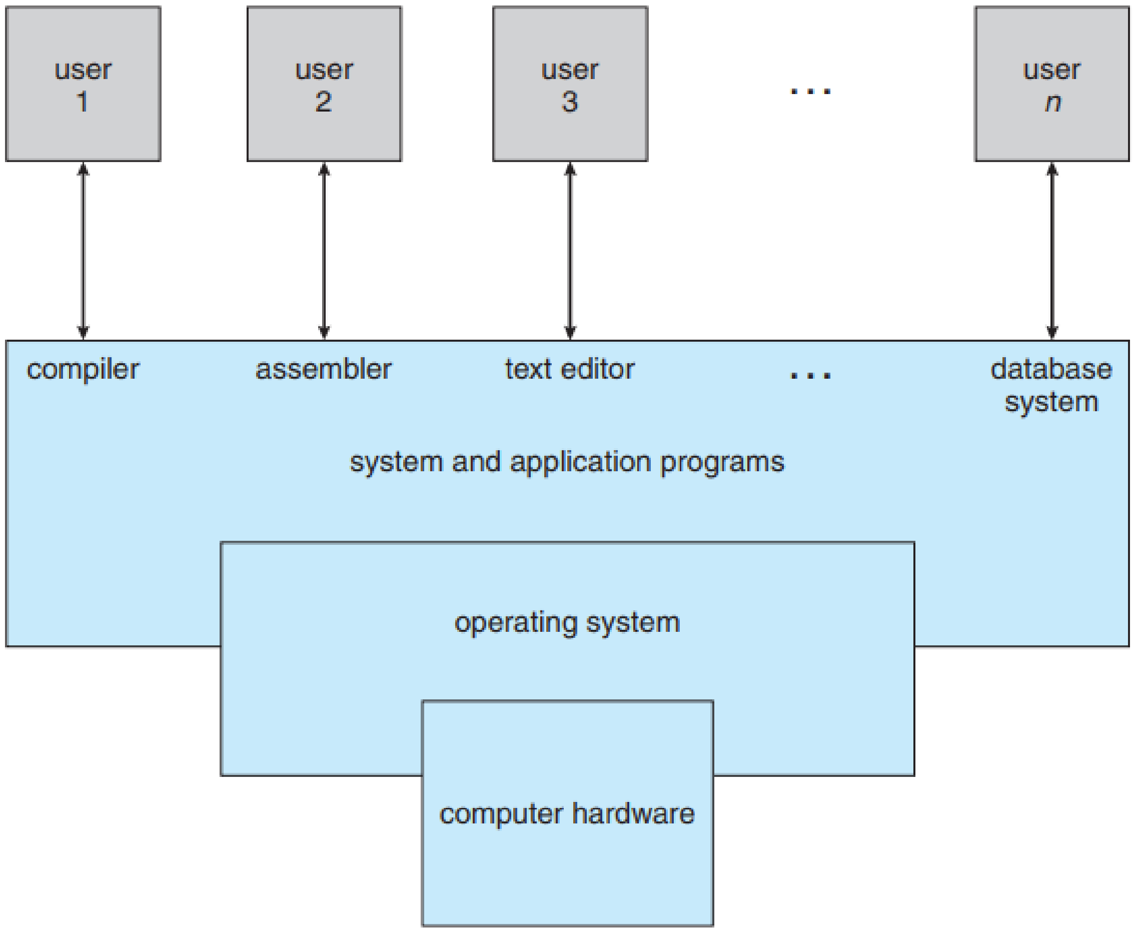


Figure 1.1 Abstract view of the components of a computer system.

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and the ***users.***

Computer system can be divided into four components:

- **Hardware – provides basic computing resources**
  - CPU, memory, I/O devices
- **Operating system**
  - Controls and coordinates use of hardware among various applications and users
- **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
  - Word processors, compilers, web browsers, database systems, video games
- **Users**
  - People, machines, other computers

The **operating system** controls the hardware and coordinates its use among the various application programs for the various users.

An **operating system** it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

To understand more fully the operating system's role, we explore operating systems from two viewpoints:

- **The user**
- **The system.**

The user

- The user's view of the computer varies according to the interface being used
- **Single user computers** (e.g., PC, workstations). Such systems are designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. the operating system is designed mostly for **ease of use** and **good performance.**
- **Multi user computers** (e.g., mainframes, computing servers). These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization -- to assure that all available CPU time, memory, and I/O are used efficiently and that no individual users takes more than their air share

- **Handheld computers** (e.g., smartphones and tablets). The user interface for mobile computers generally features a **touch screen.** The systems are resource poor, optimized for usability and battery life.
- **Embedded computers** (e.g., computers in home devices and automobiles) The user interface may have numeric keypads and may turn indicator lights on or off to show status. The operating systems are designed primarily to run without user intervention.

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. There are two different views:

- The operating system is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- The operating systems is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

Defining Operating System

Moore’s Law predicted that the number of transistors on an integrated circuit would double every
eighteen months, and that prediction has held true.

- The **fundamental goal of computer systems** is to execute user programs and to make solving user problems easier.
- Since bare hardware alone is not particularly easy to use, application programs are developed.
  - These programs require certain common operations, such as those controlling the I/O devices.
  - The common functions of controlling and allocating resources are brought together into one piece of software: the **operating system**.
- A more common definition, and the one that we usually follow, is that the operating system is 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:**
  - **System programs**, which are associated with the operating system but are not necessarily part of the kernel.

- l **Application programs**, which include all programs not associated with the operation of the system.

n 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.

n For example, each of the two most prominent mobile operating systems -- Apple's iOS and Google's Android -- feature a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

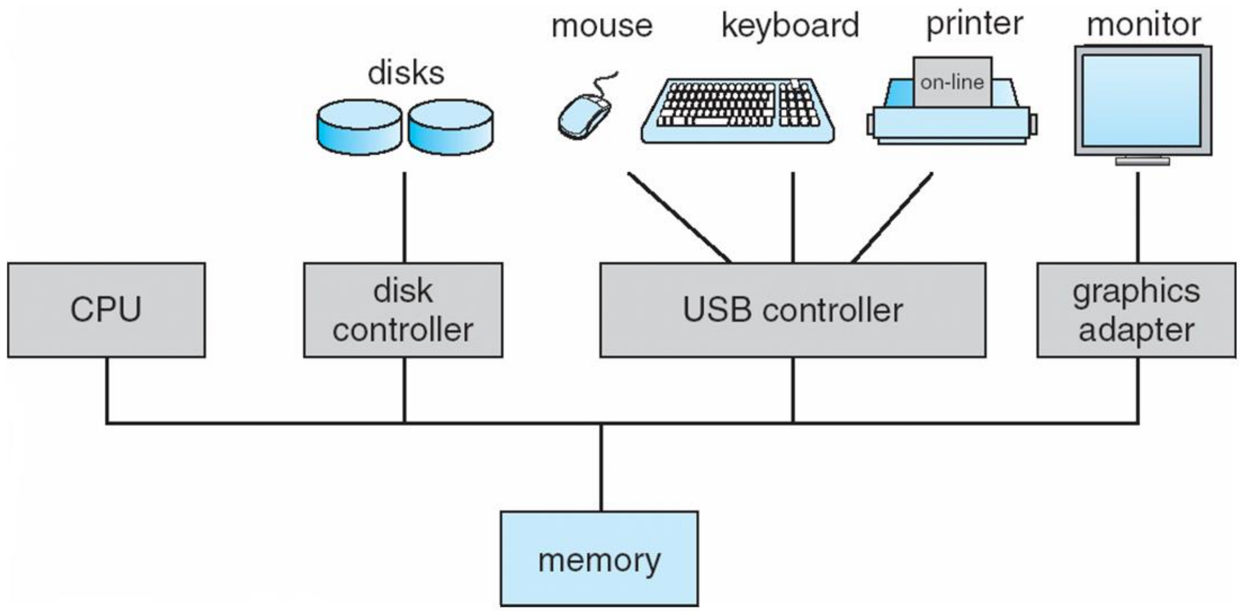
Computer System Operation

n Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). Each device controller has a local buffer.

n CPU moves data from/to main memory to/from local buffers.

n The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

Modern computer system



Computer Startup

n **Bootstrap program** is loaded at power-up or reboot

l Typically stored in **ROM or EPROM**, generally known as **firmware**

l Initializes all aspects of system

  - Loads operating system kernel and starts execution

- 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 processes**, or **system daemons** that run the entire time the kernel is running.
- On UNIX, the first system process is **init** 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.
- The occurrence of an event is usually signaled by an **interrupt.**

Interrupts

- There are two types of interrupts**:**
  - **Hardware** -- a device may trigger an interrupt by sending a signal to the CPU, usually by way of the system bus.
  - **Software** -- a program may trigger an interrupt by executing a special operation called a **system call or monitor call.**
- A software-generated interrupt (sometimes called **trap** or **exception**) is caused either by an error (e.g., divide by zero) or a user request (e.g., an I/O request).
- An operating system is **interrupt driven.**
- An interrupt-service routine is a collection of routines (modules), each of which is responsible for handling one particular interrupt (e.g., from a printer, from a disk)
- The transfer is generally through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction.

Storage structure

The CPU can load instructions only from memory, so any programs to run must be stored there.

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)**.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.

- **Main memory** – the only large storage media that the CPU can access directly
  - **Random access**
  - Typically **volatile**
- **Secondary storage** – extension of main memory that provides large **nonvolatile** storage capacity
  - Hard disks – 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
  - **Solid-state disks** – faster than hard disks, nonvolatile
    - Various technologies
    - Becoming more popular
- Tertiary storage

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

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.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

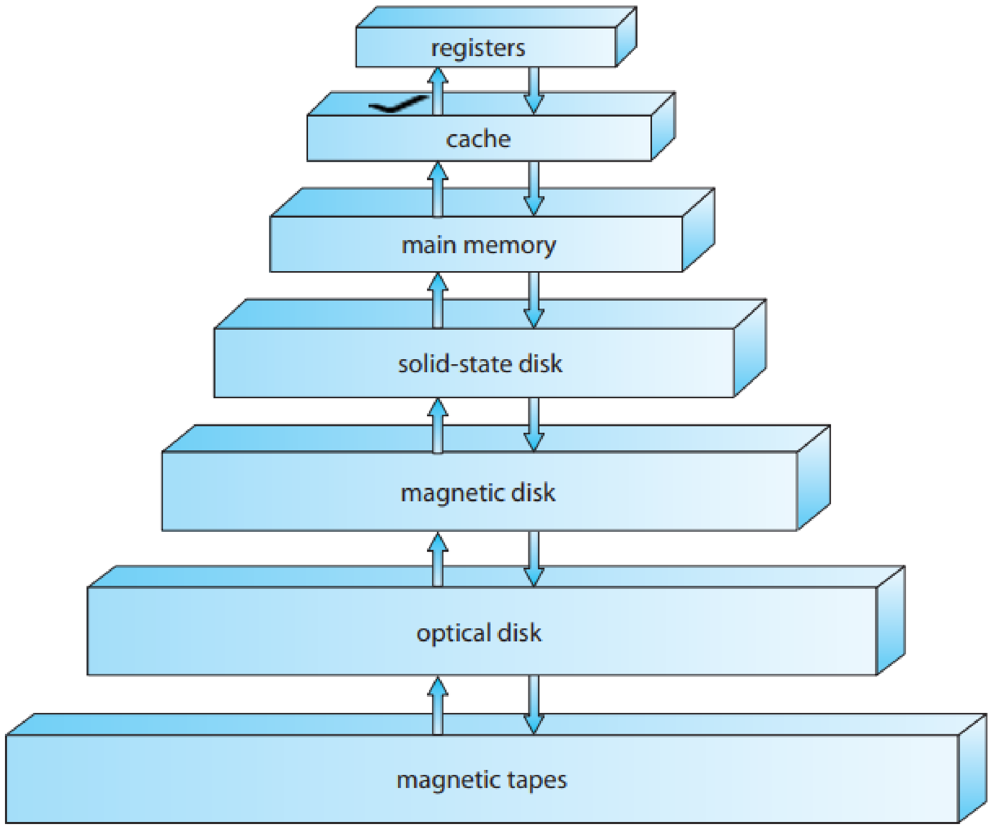


Figure 1.4 Storage-device hierarchy.

- 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.
- A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage.
- A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes.
  - 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
  - a **petabyte**, or **PB**, is $1{,}024^5$ bytes
  - exabyte, zettabyte, yottabyte

- Storage systems organized in hierarchy
    - Speed
    - Cost
    - Volatility
- **Caching** – copying information from “slow” storage 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

I/O structure

- A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device. More than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (**SCSI) controller.
- A device controller maintains some **local buffer storage and a set of special-purpose registers.**
- The **device controller** is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

Direct Memory Access Structure

- Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O.
- To solve this problem, **direct memory access** (DMA) is used.
    - After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.
    - Only one interrupt is generated per block, to tell the device driver that the operation has completed. While the device controller s performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture.

Computer System Architecture

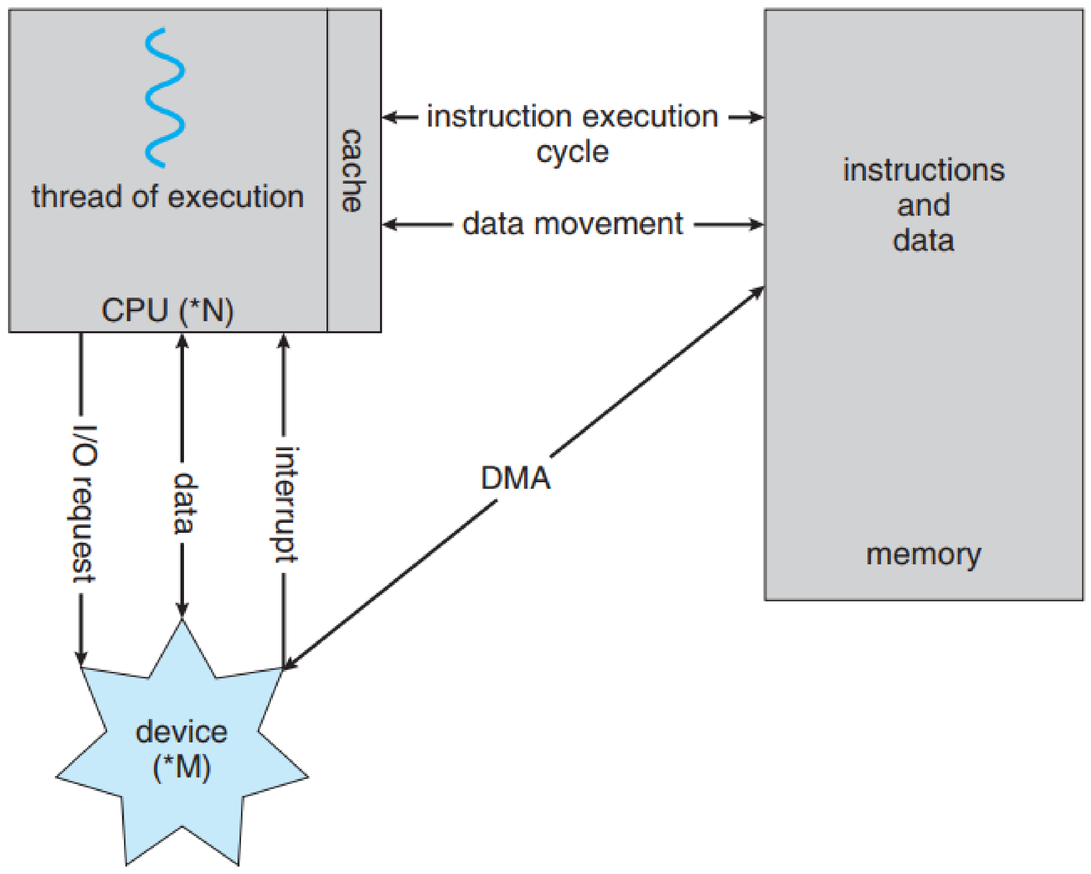


Figure 1.5 How a modern computer system works.

- **Single general-purpose processor**
  - On a **single processor system**, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
  - Most systems have special-purpose processors as well
  - If there is only one general-purpose CPU, then the system is a single-processor system.
- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:

- **Increased throughput**

  By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with *N* processors is not *N*, however; rather, it is less than *N*.

- **Economy of scale**

  Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.

- **Increased reliability** – graceful-degradation/fault-tolerance\

The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**.

Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation.

Two types of multiprocessing:

- **Symmetric Multiprocessing** – each processor performs all tasks

  Each processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory.

  Many processes can run simultaneously—*N* processes can run if there are *N* CPUs—without causing performance to deteriorate significantly.

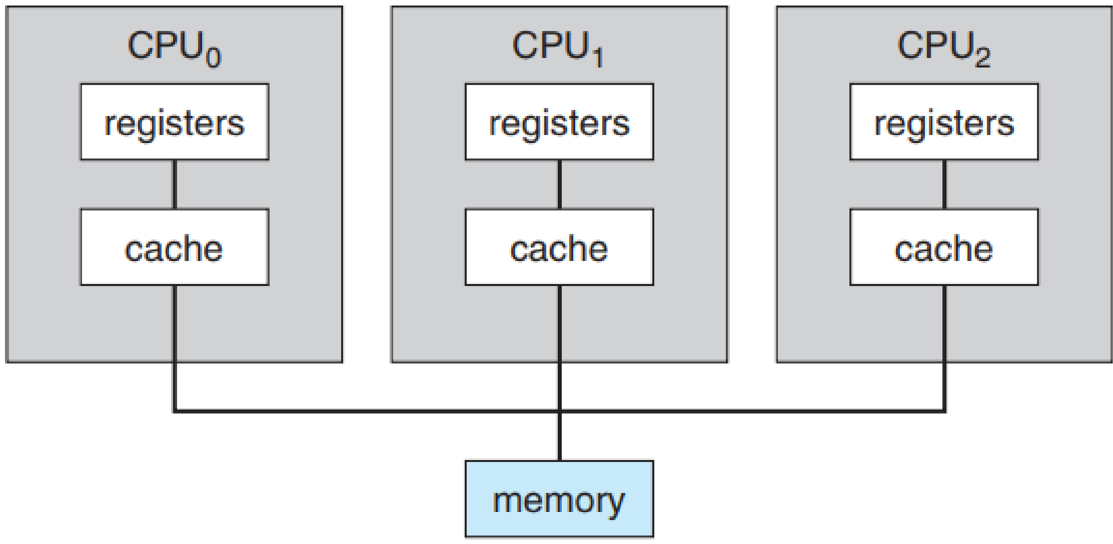


Figure 1.6 Symmetric multiprocessing architecture.

- **Asymmetric Multiprocessing** – each processor is assigned a specific task.

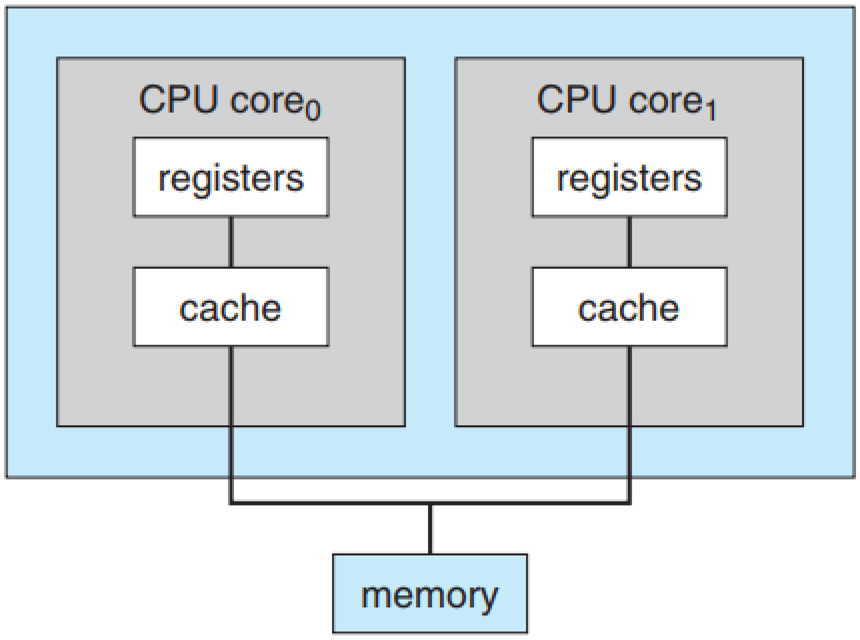
Multiprocessing can cause a system to change its memory access model from **uniform memory access(UMA) to non-uniform memory access (NUMA).**

UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time.

With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management.

Multicore Systems

- Most CPU design now includes multiple computing cores on a single chip. Such multiprocessor systems are termed **multicore**.
- **Multicore systems can be more efficient than multiple chips with single cores because:**
  - On-chip communication is faster than between-chip communication.
  - One chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for laptops as well as mobile devices.
- Note -- **while multicore systems are multiprocessor systems, not all multiprocessor systems are multicore.**



1.7 A dual-core design with two cores placed on the same chip.

Aside from architectural considerations, such as cache, memory, and bus contention, these multicore CPUs appear to the operating system as *N* standard processors.

Blade servers are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis.

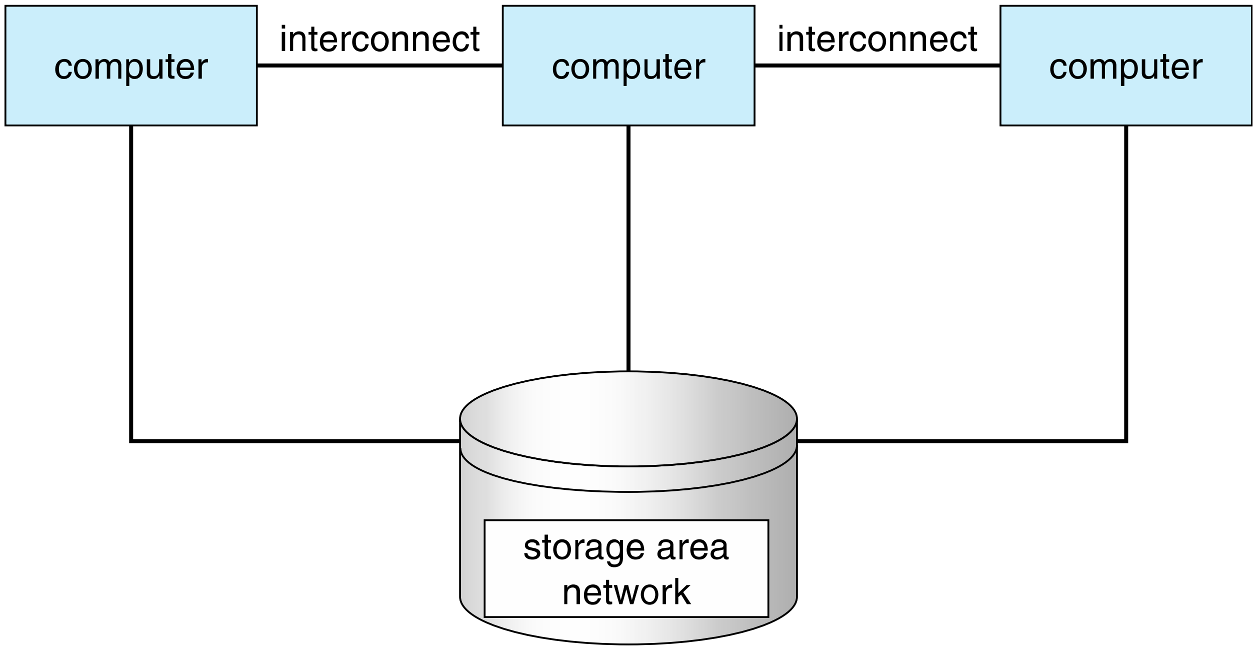
The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

Clustered systems

Like multiprocessor systems, but multiple systems working together.

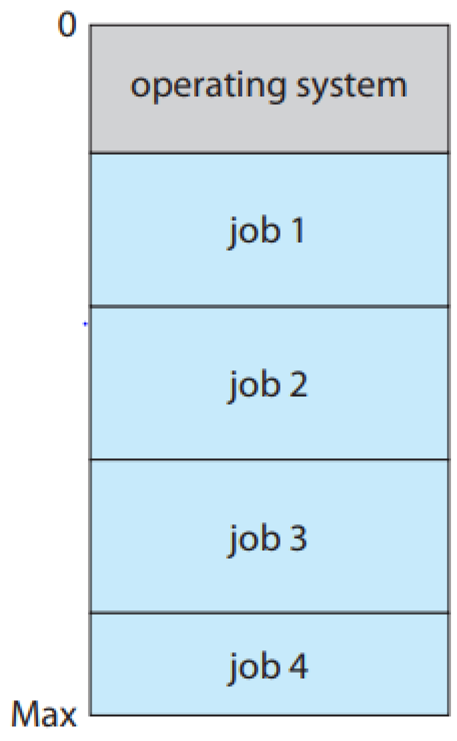
- Usually sharing storage via a **storage-area network (SAN)**
- Provides a **high-availability** service which survives failures
  - **Asymmetric clustering** has one machine in hot-standby mode
  - **Symmetric clustering** has multiple nodes running applications, monitoring each other
- Some clusters are for **high-performance computing (HPC)**
  - Applications must be written to use **parallelization**

- Some have **distributed lock manager** (**DLM**) to avoid conflicting operations
- The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into
  separate components that run in parallel on individual computers in the cluster.
- Parallel clusters allow multiple hosts to access the same data on shared storage.



General Structure of clustered system

Operating system structure



.9 Memory layout for a multiprogramming system.

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

Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously. Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**.
This pool consists of all processes residing on disk awaiting allocation of mainmemory.

n **Single user** cannot keep CPU and I/O devices busy at all times

n **Multiprogramming** organizes jobs (code and data) so CPU always has one to execute

n A subset of total jobs in system is kept in memory

n **Batch systems:**

    - One job selected and run via **job scheduling**
    - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing systems:**
    - Logical extension of batch systems -- CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

A program loaded into memory and executing is called a **process**.

- Timesharing is also referred to as **multitasking.**
- **Response time** should be < 1 second
- Each user has at least one program executing in memory. Such a program is referred to as a **process**
- If several processes are ready to run at the same time, we need to have **CPU scheduling.**
- If processes do not fit in memory, **swapping** moves them in and out to run

Virtual memory allows execution of processes not completely in memory

If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**,

If several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**,

The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**.

Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory

<u>Operating System Operations</u>

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

An interrupt service routine is provided to deal with the interrupt.

If a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system
itself.

Dual mode and multimode operations

To ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code.

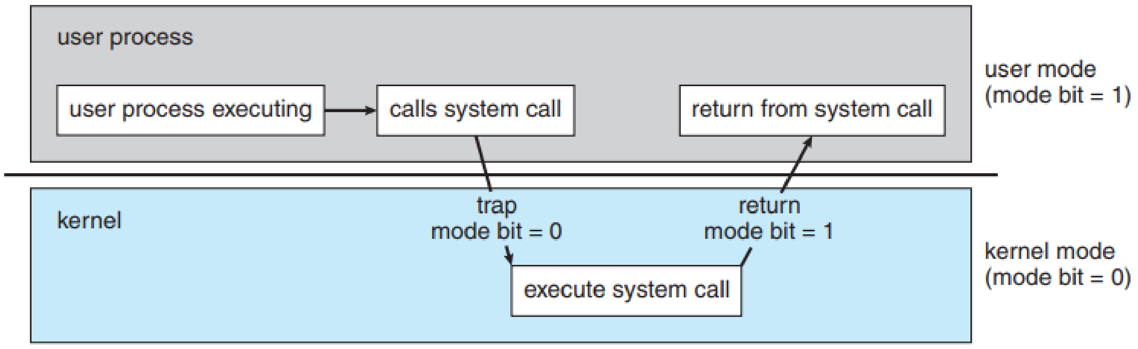


Figure 1.10 Transition from user to kernel mode.

Two separate ***modes*** of operation: **user mode** and **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).

When a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. 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 dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.

We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.

The hardware allows privileged instructions to be executed only in kernel mode.

If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute
the instruction but rather treats it as illegal and traps it to the operating system

CPUs that support virtualization (Section 16.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)**—and the virtualization management software—is in control of the system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode.

Timer

To prevent process to be in infinite loop (process hogging resources), a **timer** is used, which is a hardware device.

- n Timer is a counter that is decremented by the physical clock.
- n Timer is set to interrupt the computer after some time period
- n Operating system sets the counter (privileged instruction)
- n When counter reaches the value zero, and interrupt is generated.
- n The OS sets up the value of the counter before scheduling a process to regain control or terminate program that exceeds allotted time

Process Management

- n **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*.**
- n Process needs resources to accomplish its task
  - l CPU, memory, I/O, files, etc.
  - l Initialization data
- n Process termination requires reclaim of any reusable resources
- n **A thread** is a basic unit of CPU utilization within a process.
  - l Single-threaded process. Instructions are executed sequentially, one at a time, until completion
  - l Process has one **program counter** specifying location of next instruction to execute
- n **Multi-threaded** process has one program counter per thread
- n Typically, a system has many processes, some user, some operating system running concurrently on one or more CPUs
  - l Concurrency by multiplexing the CPUs among the threads

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

- n 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

Storage Management

- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Files are stored in a number of different storage medium.
    - Disk
    - Flash Memory
    - Tape
- Each medium is controlled by device drivers (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

Secondary 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
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications

Cache Management

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache are smaller (size-wise) than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use and data must be in main memory before being moved to secondary storage for safekeeping.

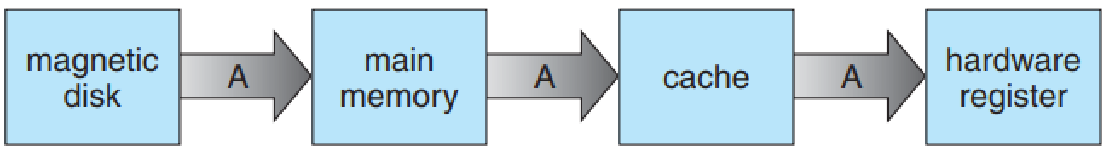


Figure 1.12 Migration of integer A from disk to register.

We must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**

I/O Subsystem

- One purpose of an operating system is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
    - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
    - General device-driver interface
    - Drivers for specific hardware devices

Protection and Security

- **Protection** – A mechanism for controlling access of processes (or users) to resources defined by the OS
- **Security** – A 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

- l User ID is associated with all files and processes of that user to determine access control
- l Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
- l **Privilege escalation** allows user to change to effective ID with more rights

Virtualization

- n Allows operating systems to run applications within other OSes
  - l Vast and growing industry
- n **Emulation** used when the source CPU type is different from the target type (i.e., PowerPC to Intel x86)
  - l Generally slowest method
  - l When computer language not compiled to native code – **Interpretation**
- n **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - l Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
  - l **VMM** (virtual machine Manager) provides virtualization services

Kernel Data Structures

Lists, Stacks, and Queues

An **array** is a simple data structure in which each element can be accessed directly.

Lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence.

Implementing this structure is a **linked list**, in which items are linked to one
another. Linked lists are of several types:
• In a ***singly linked list,*** each item points to its successor, as illustrated in
• In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor, as illustrated in • In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null.

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items.

A **stack** is a sequentially ordered data structure that uses the last in, first out **(LIFO)** principle for adding and removing items, meaning that the last item placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as ***push*** and ***pop***, respectively. An
operating system often uses a stack when invoking function calls.

A **queue**, in contrast, is a sequentially ordered data structure that uses the
first in, first out **(FIFO)** principle: items are removed from a queue in the order
in which they were inserted.

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the ***left child*** and the ***right child***. A **binary search tree** additionally requires an ordering between the parent's two children in which $left\ child <= right\ child$.

A **hash function** takes data as its input, performs a numeric operation on this data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size n can require up to $O(n)$ comparisons in the worst case, using a hash function for retrieving data from table can be as good as $O(1)$ in the worst case, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

n **Bitmap** – string of n binary digits representing the status of n items

n Linux data structures defined in

include files <linux/list.h>, <linux/kfifo.h>, <linux/rbtree.h>

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

n **Portals** provide web access to internal systems

n **Network computers** (**thin clients**) are like Web terminals

n Mobile computers interconnect via **wireless networks**

n Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks

- Collection of separate, possibly heterogeneous, systems networked together
  - **Network** is a communications path, **TCP/IP** most common
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)
- **Network Operating System** provides features to allow sharing of data between systems across a network.
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system
- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server system** provides an interface to client to request services (i.e., database)
  - **File-server system** provides interface for clients to store and retrieve files

Peer to peer

  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via ***discovery protocol***

Cloud computing

- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay

- l **Private cloud** – run by a company for the company's own use
- l **Hybrid cloud** – includes both public and private cloud components
- l Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
- l Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)

Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

- n Real-time embedded systems most prevalent form of computers
  - l Vary considerable, special purpose, limited purpose OS, **real-time OS**
  - l Use expanding

Chapter 2

Operating system Structures.

The goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

Operating System Services

An **operating system** provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs

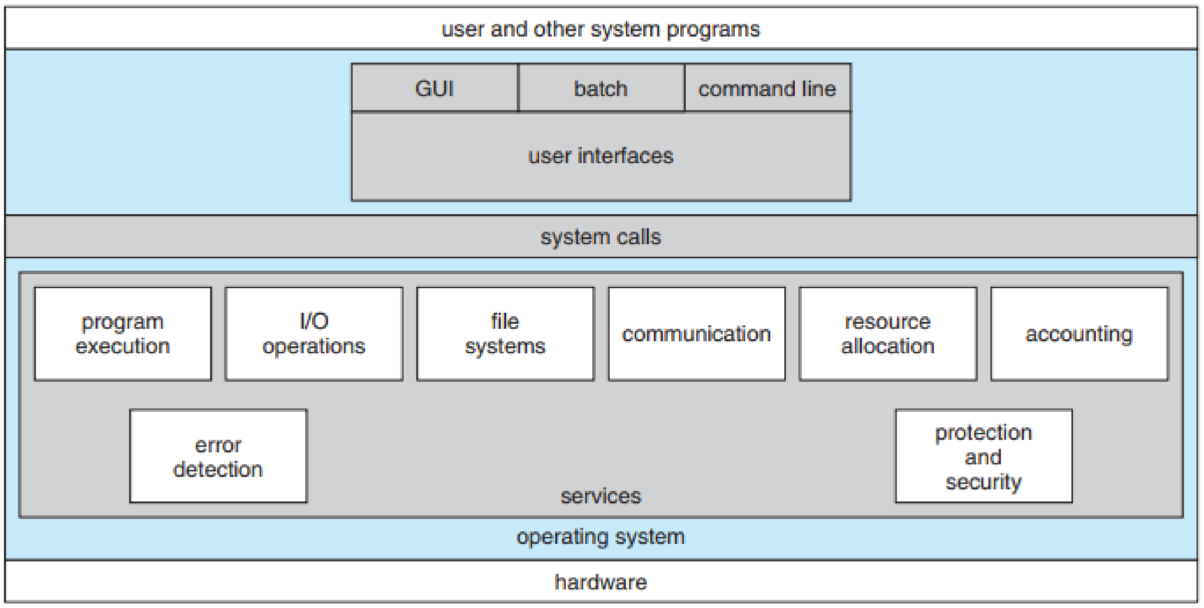


Figure 2.1 A view of operating system services.

OS services helpful to the user

- **User interface** - Almost all operating systems have a **user interface** (**UI**). This interface can take several forms:
  - **Command-Line (CLI)** -- uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options).
  - **Graphics User Interface (GUI)** -- the interface is a window system with a pointing device to direct I\O, choose from menus, and make selections and a keyboard to enter text..
  - **Batch Interface** -- commands and directives to control those commands are entered into files, and those files are executed

Some systems provide two or all three of these variations.

- **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)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
- **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

- **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)
- **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

OS Services for Ensuring Efficient Operation

- **Resource allocation -** When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
    - Many types of resources -   CPU cycles, main memory, file storage, I/O devices.
- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **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

Command Interpreters (CLI)

CLI 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.**
- The main function of the command interpreter is to get and execute the next user-specified command.

Graphical User Interfaces (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 is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

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

System administrators who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call
- 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
- Three most common APIs are:
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **systemcall interface** that serves as the link to system calls made available by the operating system.

System calls parameter passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

To ensure the integrity of the data being shared, operating systems often provide system calls allowing
a process to **lock** shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include acquire lock() and release lock().

To start a new process, the shell executes a fork() system call. Then, the selected program is l memory via an exec() system call, and the program is executed.

Types of System Calls

- System calls can be grouped roughly into six major categories:
  - Process control,
  - File manipulation,
  - Device manipulation,
  - Information maintenance,
  - Communications,
  - Protection.

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

System Calls – File Management

- Create file
- Delete file
- Open and Close file
- Read, Write, Reposition
- Get and Set file attributes

System Calls – Device Management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

System Calls -- Information Maintenance

- get time or date,
- set time or date

- get system data,
- set system data
- get and set process, file, or device attributes

System Calls – Communications

- create, delete communication connection
- if **message passing model:**
    - send, receive message
        - To **host name** or **process name**
        - From **client** to **server**
- If **shared-memory model:**
    - create and gain access to memory regions
- transfer status information
- attach and detach remote devices

System Calls – Protection

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

System Programs

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

- System programs provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls. Others are considerably more complex.
- They can be divided into:
    - File manipulation
    - Status information sometimes stored in a File modification

- Programming language support
    - Program loading and execution
    - Communications
    - Background services
    - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls

- **File management**
    - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
    - Some programs ask the system for information - date, time, amount of available memory, disk space, number of users
    - Others programs 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 information

- **File modification**
    - Text editors to create and modify files
    - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **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

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

Operating system design and implementation

- Two groups in terms of defining 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
- Specifying and designing an OS is highly creative task of **software engineering**

Mechanisms and Polices

- Important principle to separate

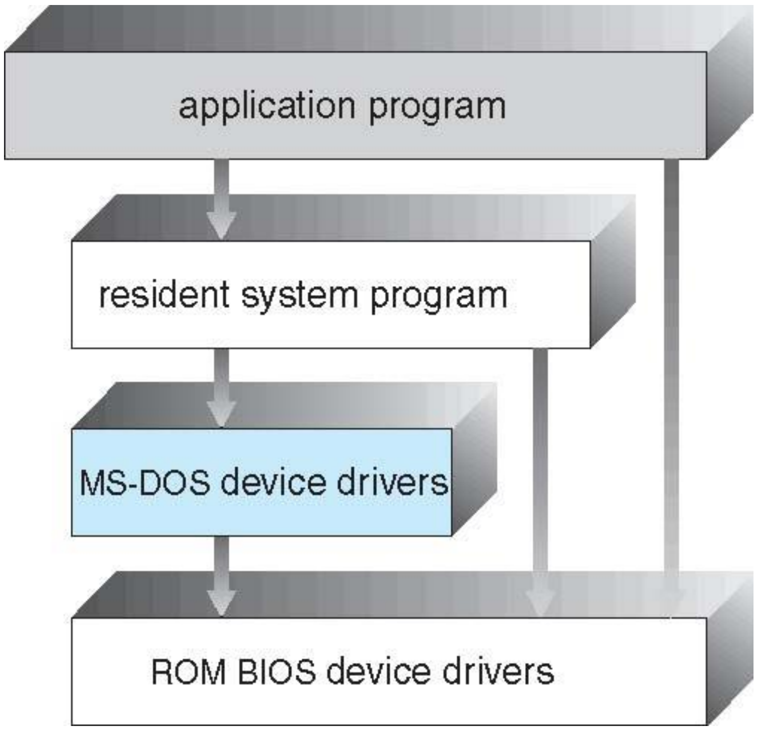
Policy: ***What*** will be done?
Mechanism: ***How*** to do it?

- Mechanisms determine how to do something, policies decide what will be done

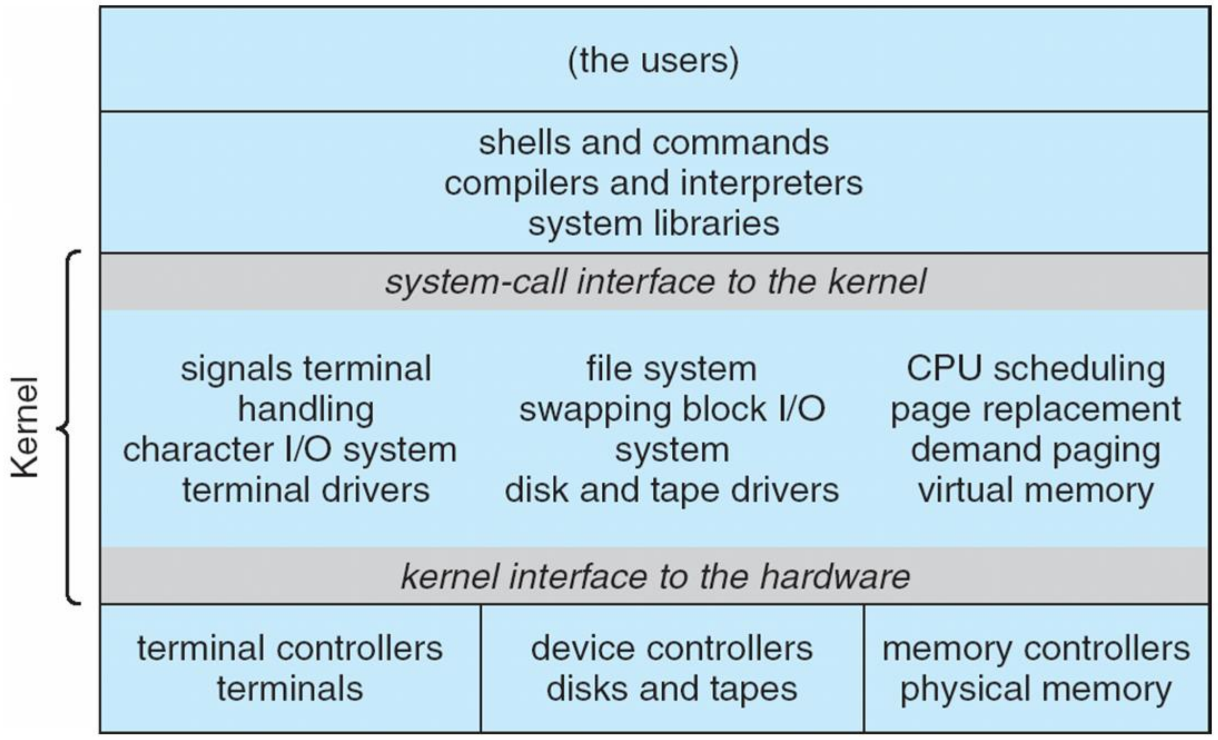
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- **Emulation** can allow an OS to run on non-native hardware

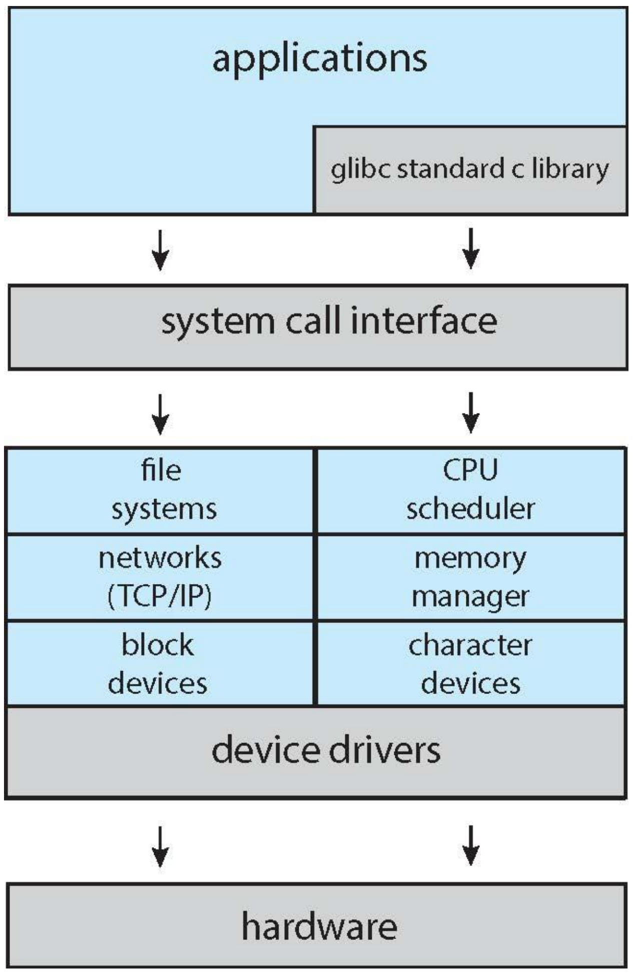
Operating system structure

- Various ways to structure an operating system:
  - Monolithic structure(A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system.)
    - Simple structure – MS-DOS
    -
    - More complex – UNIX
    - More complex – Linux
  - Layered – An abstraction
  - Microkernel - Mach



UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers,





The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.

Microkernels

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through **message passing,**

- Moves as much from the kernel into user space
- **Mach** 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

Modularity

- The monolithic approach results in a situation where changes to one part of the system can have wide-ranging effects to other parts.
- The advantage of this modular approach is that changes in one component only affect that component, and no others, allowing system implementers more freedom when changing the inner workings of the system and in creating modular operating systems.

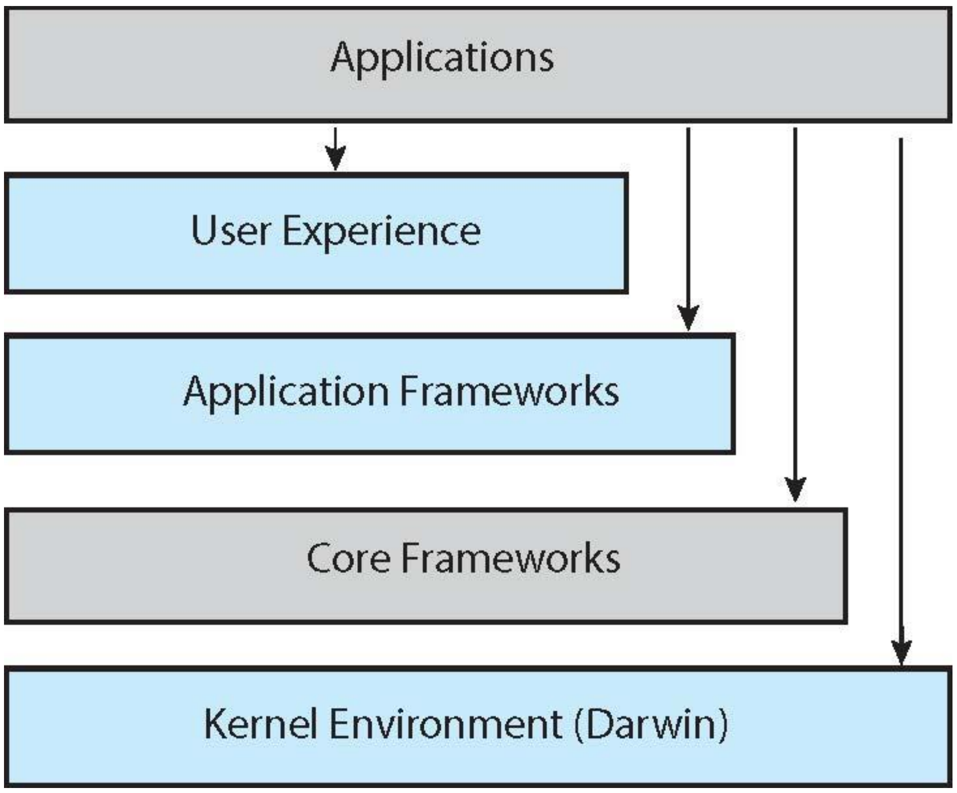
Layered Approach

- A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

Modules

- Many modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc

Architecture of Mac OS X and iOS



Macox

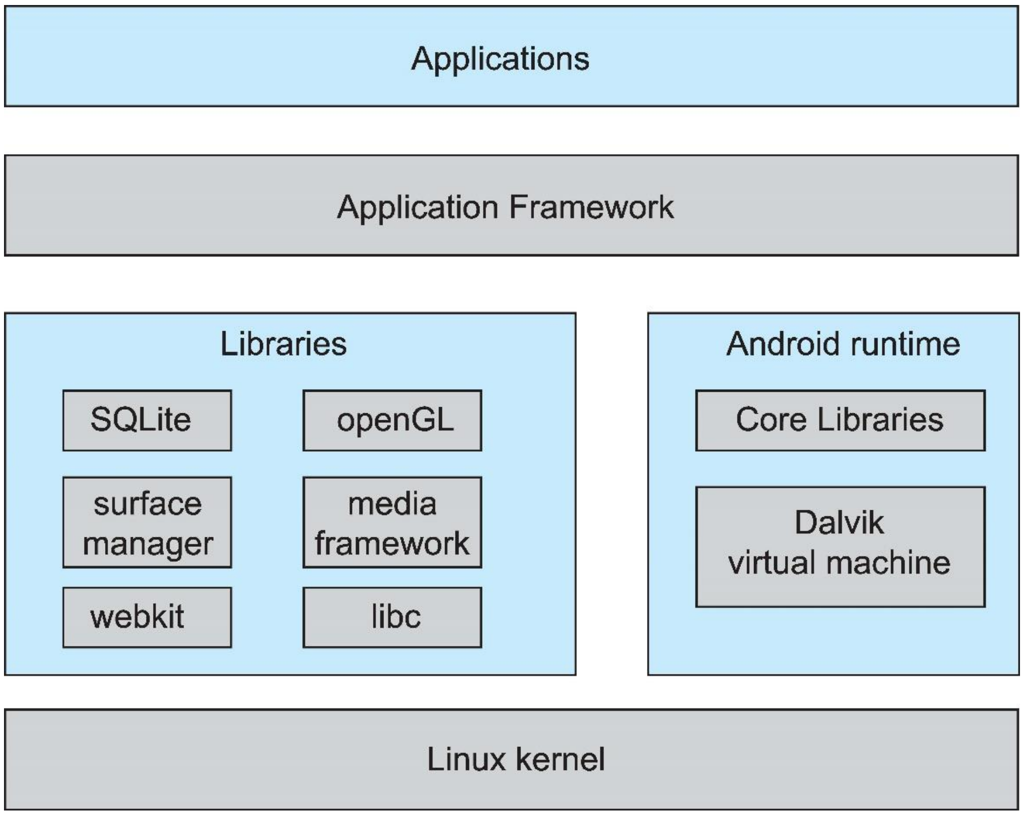
he Mach
microkernel and the BSD UNIX kernel.

Android

- n Developed by Open Handset Alliance (mostly Google)
  - l Open Source
- n Similar stack to iOS
- n Based on Linux kernel but modified
  - l Provides process, memory, device-driver management
  - l Adds power management
- n Runtime environment includes core set of libraries and Dalvik virtual machine
  - l Apps developed in Java plus Android API

- Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM

n Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc



Operating system debugging

n **Debugging** is finding and fixing errors, or **bugs**

n OS generate **log files** containing error information

n Failure of an application can generate **core dump** file capturing memory of the process

n Operating system failure can generate **crash dump** file containing kernel memory

n Beyond crashes, performance tuning can optimize system performance

l Sometimes using ***trace listings*** of activities, recorded for analysis

l **Profiling** is periodic sampling of instruction pointer to look for statistical trends

- Kernighan’s Law: Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

System boot

- **When power initialized on system, execution starts at a fixed memory location**
    - **Firmware ROM used to hold initial boot code**
- **Operating system must be made available to hardware so hardware can start it**
    - **Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it**
    - **Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk**
- **Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options**
- **Kernel loads and system is then running**

Chapter 3

- An operating system executes a variety of programs:
    - Batch system – **jobs**
    - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
    - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
    - **Stack** containing temporary data
        - Function parameters, return addresses, local variables
    - **Data section** containing global variables
    - **Heap** containing memory dynamically allocated during run time

Processes

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

Operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive. I

A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**.

Process concept

n **Process** – a program in execution; process execution must progress in sequential fashion

n Program is ***passive*** entity stored on disk (**executable file**), process is ***active***

  l Program becomes process when executable file loaded into memory

n Execution of program started via GUI mouse clicks, command line entry of its name, etc

n One program can be several processes

  l Consider multiple users executing the same program

Process structure

n A process is more than the program code, which is sometimes known as the **text** section.

n It also includes the current activity:

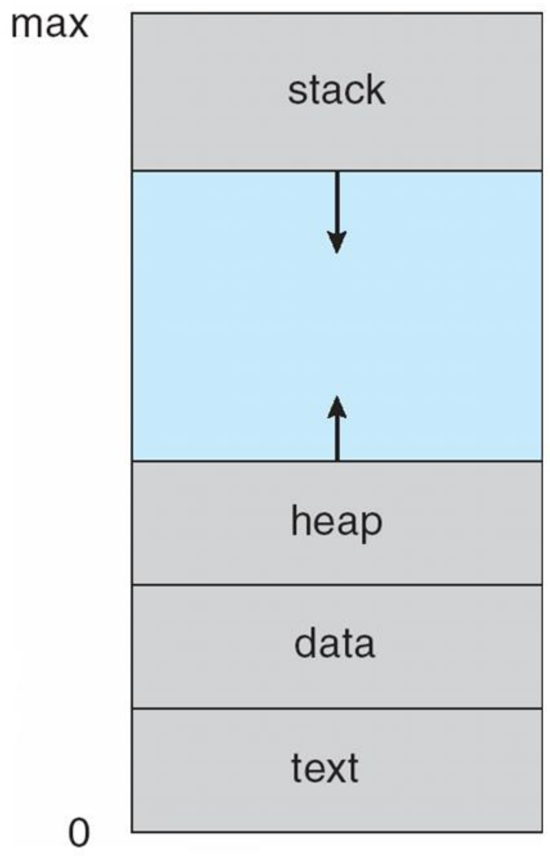
  l The value of the **program counter**

  l The contents of the **processor's registers**.

n It also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables)

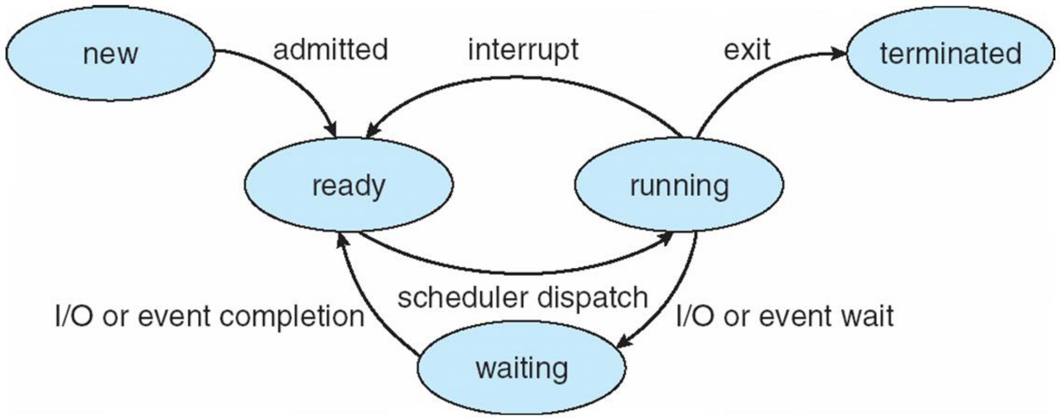
n It also includes the **data section**, which contains global variables.

n It may also include a **heap**, which is memory that is dynamically allocated during process run time.



Process states

- n As a process executes, it changes **state**
  - l **new**: The process is being created
  - l **running**: Instructions are being executed
  - l **waiting**: The process is waiting for some event to occur
  - l **ready**: The process is waiting to be assigned to a processor
  - l **terminated**: The process has finished execution
- n Diagram of Process State

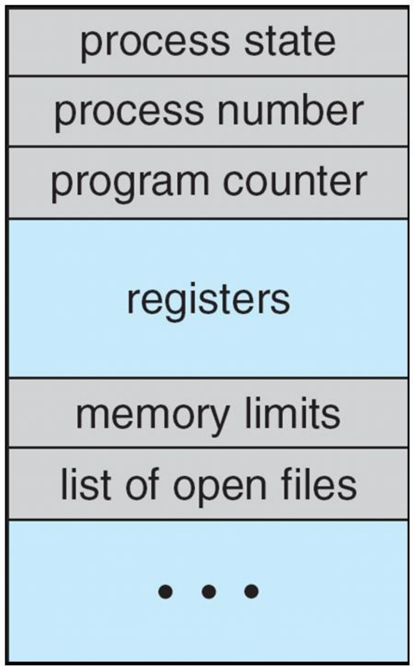


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- n Process state – running, waiting, etc
- n Program counter – location of instruction to next execute
- n CPU registers – contents of all process-centric registers
- n CPU scheduling information- priorities, scheduling queue pointers
- n Memory-management information – memory allocated to the process
- n Accounting information – CPU used, clock time elapsed since start, time limits
- n I/O status information – I/O devices allocated to process, list of open files



Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution.

- n So far, process has a single thread of execution
- n Consider having multiple program counters per process
  - l Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- n Need storage for thread details, multiple program counters in PCB
- n Covered in the next chapter
- n A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:
  • The process could issue an I/O request and then be placed in an I/O queue.
  • The process could create a new child process and wait for the child's termination.
  • The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Process Scheduling

n Maximize CPU use

  l Quickly switch processes onto CPU for time sharing

n **Process scheduler** selects among available processes for next execution on CPU

n Maintains **scheduling queues** of processes

  l **Job queue** – set of all processes in the system

  l **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

  l **Device queues** – set of processes waiting for an I/O device

  l Processes migrate among the various queues

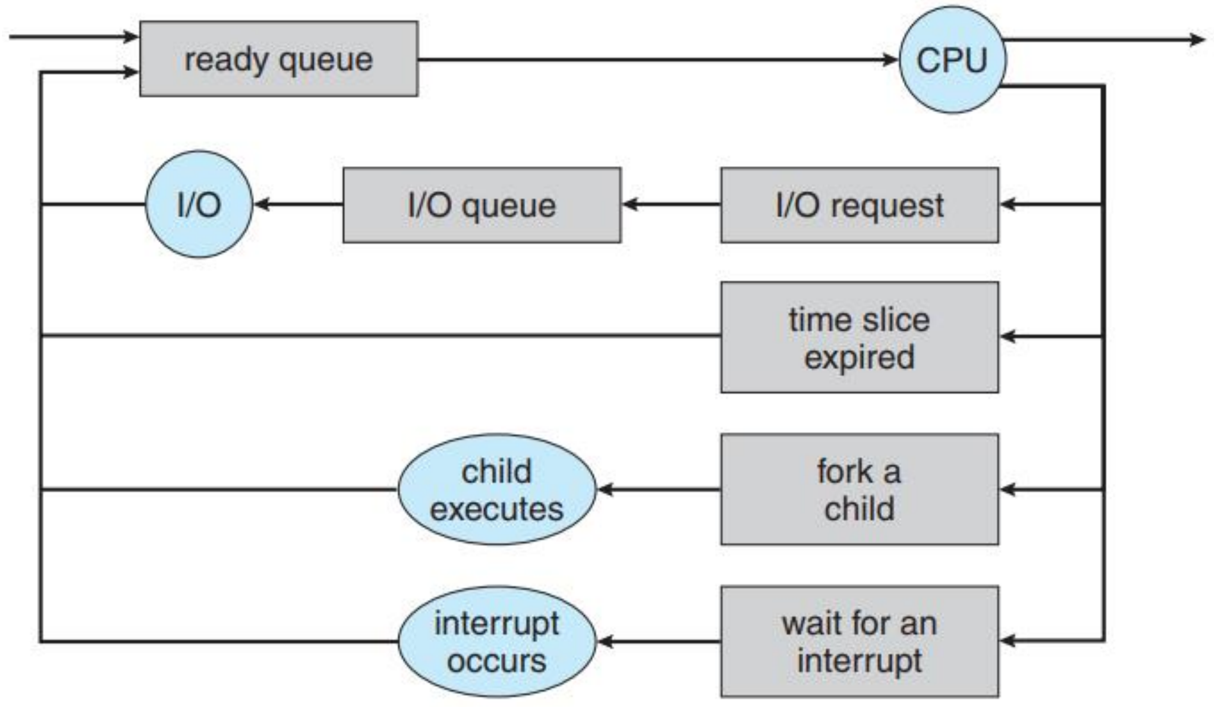


Figure 3.6 Queueing-diagram representation of process scheduling.

Schedulers

n **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates a CPU

  l Sometimes the only scheduler in a system

- l Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)

n **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue

- l Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
- l The long-term scheduler controls the **degree of multiprogramming**

n Processes can be described as either:

- l **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- l **CPU-bound process** – spends more time doing computations; few very long CPU bursts

n Long-term scheduler strives for good ***process mix***

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler**

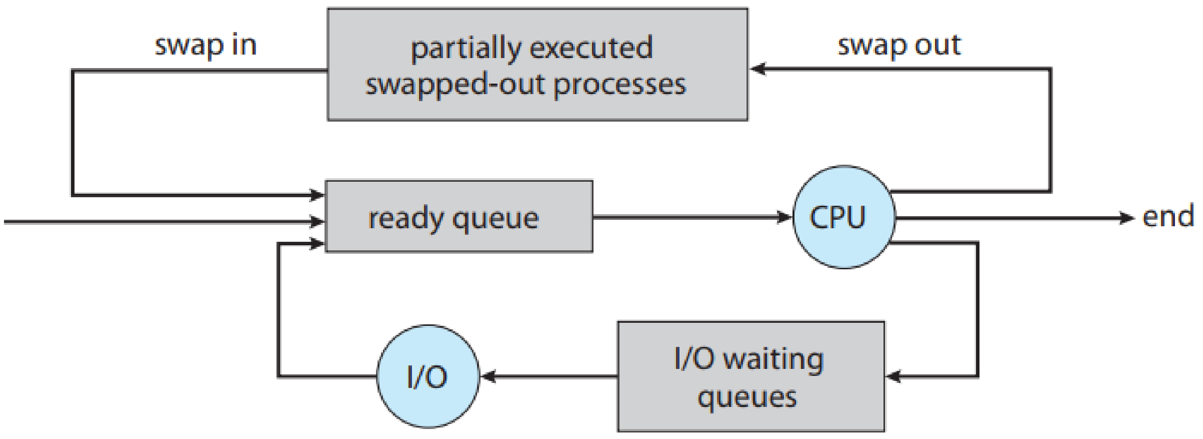


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Multitasking in Operating System

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Starting with iOS 4, it provides for a
  - Single **foreground** process – controlled via user interface
  - Multiple **background** processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

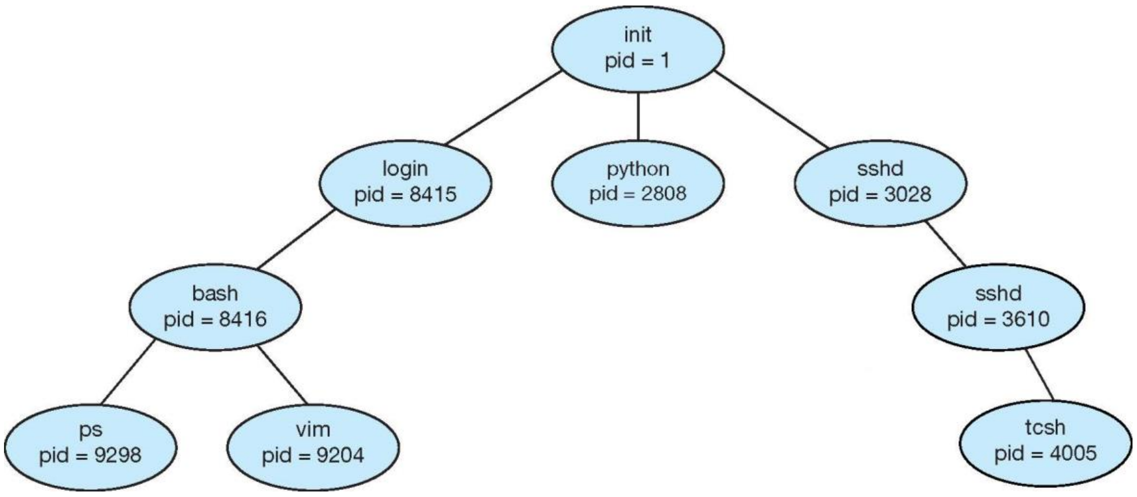
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
- **System must provide mechanisms for:**

- process creation,
- process termination,
- and so on as detailed next

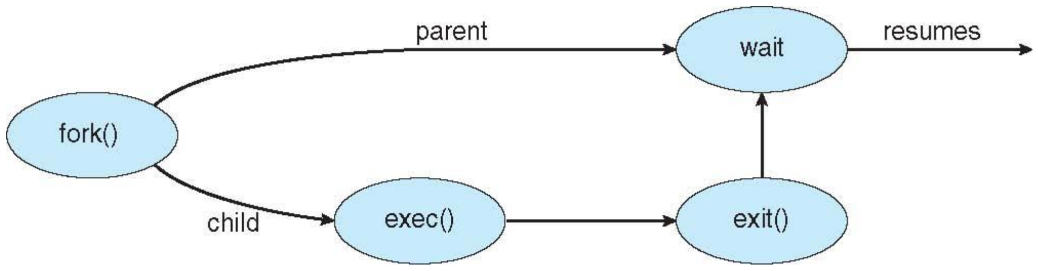
Process Creation

- A **process** may create other processes.
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, a process is identified and managed via a **process identifier** (**pid**)
- A Tree of Processes in UNIX



- Resource sharing among parents and children options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - A child is a duplicate of the parent address space.
  - A child loads a program into the address space.

- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()**replaces the process' memory space with a new program



Process Termination

- A process terminates when it finishes executing its final statement and it asks the operating system to delete it by using the **exit()** system call.
  - At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call.
  - All the resources of the process are deallocated by the operating system.
- A parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow a child process to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process

pid = wait(&status);

- If no parent waiting (did not invoke **wait()**) process is **zombie**

- If parent terminated without invoking **wait**, process is **orphan**

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - Cooperating processes can affect or be affected by other processes, including sharing data
  - Independent processes cannot affect other processes
- Reasons for having cooperating processes:
  - Information sharing

    several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - Computation speedup (multiple processes running in parallel)
  - Modularity

    We may want to construct the system in a modular fashion,
    dividing the system functions into separate processes or threads,
  - Convenience
- Cooperating processes need **interposes communication** (**IPC**)
- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory** and **message passing**.

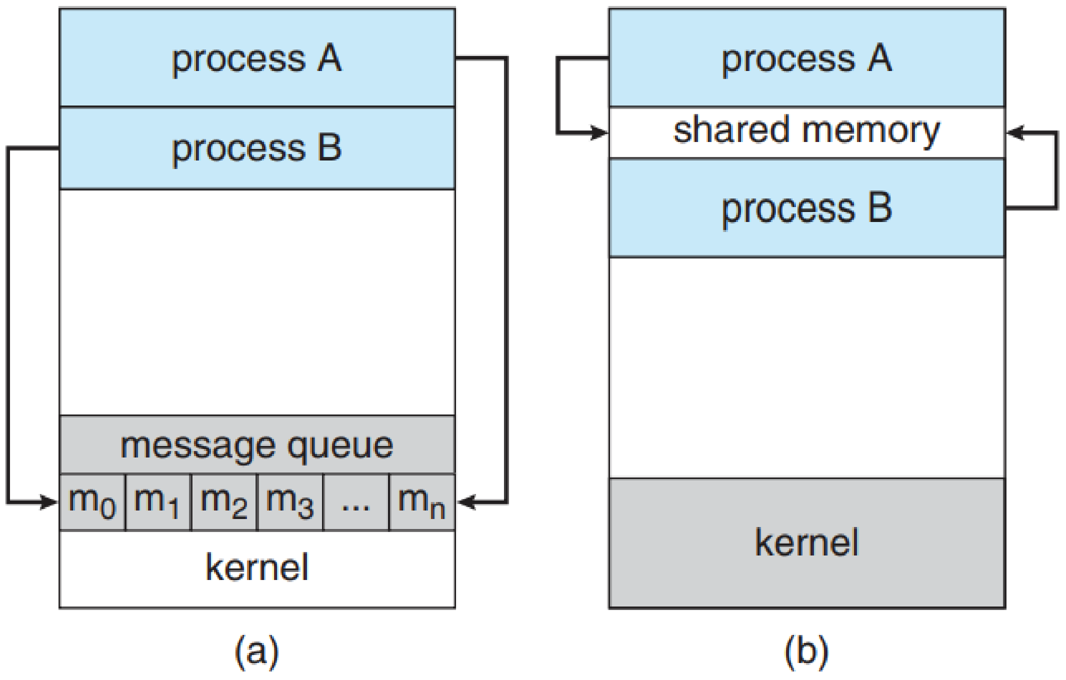


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

SHARED MEMORY

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

Synchronization

- Cooperating processes that access shared data need to synchronize their actions to ensure data consistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration of the problem – The producer-Consumer problem**
  - Producer process produces information that is consumed by a Consumer process.
  - The information is passed from the Producer to the Consumer via a buffer.
  - Two types of buffers can be used:
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that a fixed buffer size

Message Passing Systems

- Mechanism for processes to communicate and to synchronize their actions
  - Without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

Here are several methods for logically implementing a link and the send()/receive() operations:

Physical:

- Shared memory
- Hardware bus
- Network

- **Logical:**
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link

  - The link may be unidirectional, but is usually bi-directional

This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

Indirect communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - delete a mailbox
- Primitives are defined as:
  - **send**(*A*, *message*) – send a message to mailbox A
  - **receive**(*A*, *message*) – receive a message from mailbox A
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes

- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Indirect communication Issues

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Blocking and Non-blocking schemes

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive --** the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways:

1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)

2. Bounded capacity – finite length of *n* messages Sender must wait if link full

3. Unbounded capacity – infinite length Sender never waits

Example of IPC Systems

- There are four different IPC systems.
  - POSIX API for shared memory
  - Mach operating system, which uses message passing
  - Windows IPC, which uses shared memory as a mechanism for providing certain types of message passing.
  - Pipes, one of the earliest IPC mechanisms on UNIX systems.

Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer **msg_send(), msg_receive(), msg_rpc()**
  - Mailboxes needed for commuication, created via **port_allocate()**

Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle (an abstract reference to a resource) to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates a private **communication port** and returns the handle to the client.

Pipes

- Acts as a conduit allowing two processes to communicate
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.

  Producer writes to one end (the **write-end** of the pipe)Consumer reads from the other end (the **read-end** of the pipe)

- **Named pipes** – can be accessed without a parent-child relationship.

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) is used to refer to system on which process is running. That is, when a computer refers to address 127.0.0.1, it is referring to itself.

Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **MulticastSocket class** – data can be sent to multiple recipients

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation

- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters (marshalling involves packaging the parameters into a form that can be transmitted over a network).
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures.
- Must be dealt with concerns differences in data representation on the client and server machines.
- Consider the representation of 32-bit integers.
  - **Big-endian.** Store the most significant byte first
  - **Little-endian.** Store the least significant byte first.

Big-endian is the most common format in data networking . It is also referred to as **network byte order**.

- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Chapter 4

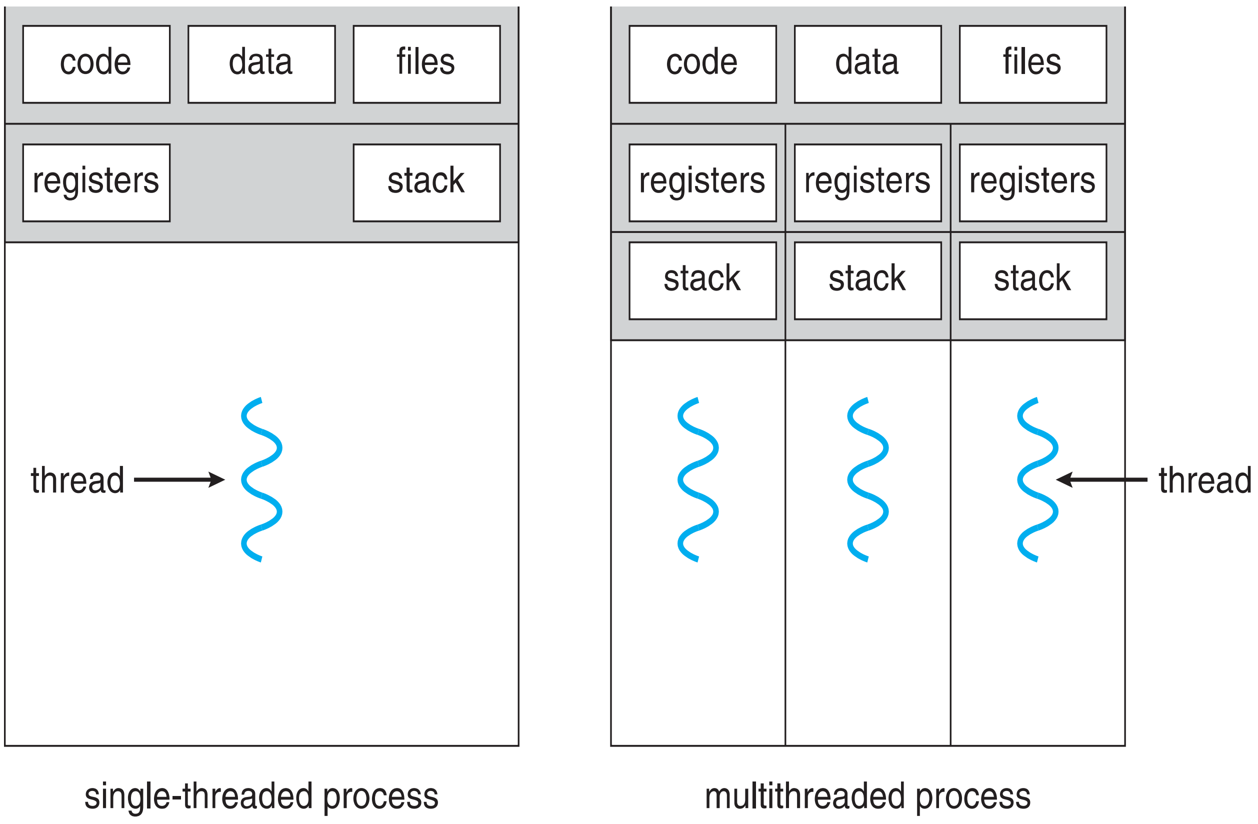
Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

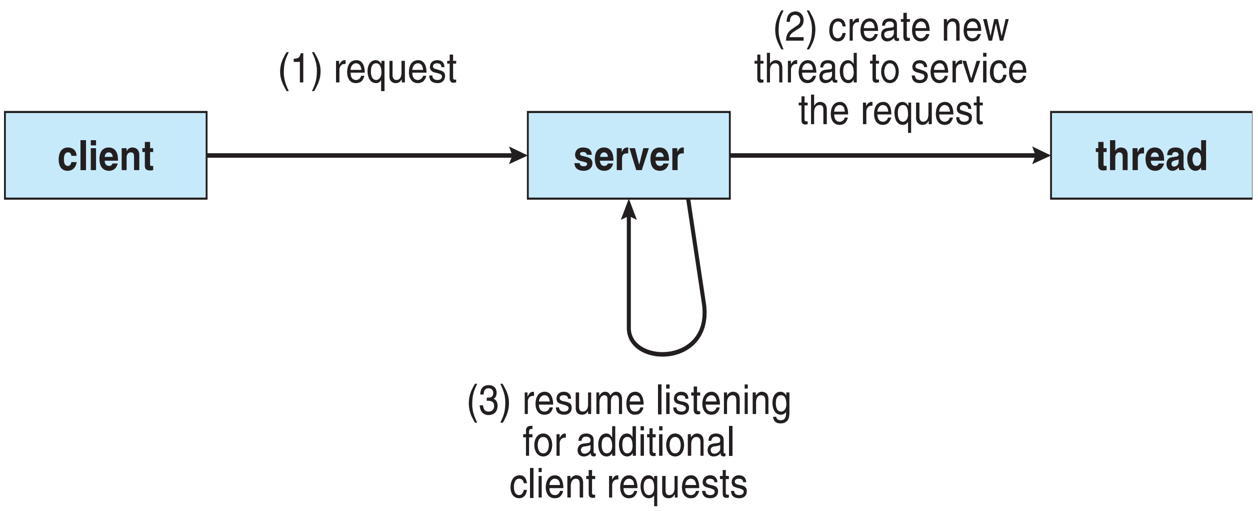
It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or ***heavyweight***) process
has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
    - Update display
    - Fetch data
    - Spell checking
    - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single and multithread



Multithreaded Server Architecture


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multi-CPU systems**. Multiple CPUs are placed in the computer to provide more computing performance.
- **Multicore systems**. Multiple computing cores are placed on a single processing chip where each core appears as a separate CPU to the operating system

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

- There is a fine but clear distinction between concurrency and parallelism..
- A concurrent system supports more than one task by allowing all the tasks to make progress.
- In contrast, a system is parallel if it can perform more than one task simultaneously.

- Types of parallelism
    - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading
    - CPUs have cores as well as ***hardware threads***
    - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Multicore Programming

- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
    - **Dividing activities**
    - **Balance**
    - **Data splitting**
    - **Data dependency**

  - **Testing and debugging**

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Type of Parallelism

- **Data parallelism**. The focus is on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
- **Task parallelism.** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

User and Kernel Threads

- Support for threads may be provided at two different levels:
  - **User threads** - are supported above the kernel and are managed without kernel support, primarily by user-level threads library.
  - **Kernel threads** - are supported by and managed directly by the operating system.
- Virtually all contemporary systems support kernel threads:
  - Windows, Linux, and Mac OS X

Relationship between user and Kernel threads

- Three common ways of establishing relationship between user and kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

One-to-One Model

- Each user-level thread maps to a single kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux

Many-to-One Model

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

Two-level Model

- Similar to many-to-many, except that it allows a user thread to be **bound** to kernel thread
- Examples
    - IRIX
    - HP-UX
    - Tru64 UNIX
    - Solaris 8 and earlier

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
    - Library entirely in user space
    - Kernel-level library supported by the OS
- Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Two techniques for creating threads in a Java program.

- One approach is to create a new class that is derived from the Thread class and to override its run() method.
- An alternative is to define a class that implements the Runnable} interface, as shown below

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool
    - Separating task to be performed from mechanics of creating task allows different strategies for running task
        - That is, tasks could be scheduled to run periodically
- Windows API supports thread pools:

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

Grand Central Dispatch

- Two types of dispatch queues:
    - serial – blocks removed in FIFO order, queue is per process, called **main queue**
        - Programmers can create additional serial queues within program
    - concurrent – removed in FIFO order but several may be removed at a time
        - Three system wide queues with priorities low, default, high

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling

    - Synchronous and asynchronous
- Thread cancellation of target thread
    - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations
- Does **fork()**duplicate only the calling thread or all threads?
    - Some UNIX systems have two versions of fork
- **exec()** usually works as normal – replace the running process including all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
    1. Signal is generated by particular event
    2. Signal is delivered to a process
    3. Signal is handled by one of two signal handlers:
        1. default
        2. user-defined
- Every signal has **default handler** that the kernel runs when handling signal
    1. **User-defined signal handler** can override default
    2. For single-threaded, signal delivered to process
- Where should a signal be delivered for multi-threaded?
    1. Deliver the signal to the threat to which the signal applies
    2. Deliver the signal to every thread in the process
    3. Deliver the signal to certain threads in the process
    4. Assign a specific threat to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- The thread to be canceled is referred to as **target thread**

- Cancelation of a target thread may be handled using two general approaches:
    - **Asynchronous cancellation** terminates the target thread immediately
    - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- The difficulty with cancellation occurs in situations where:
    - Resources have been allocated to a canceled thread.
    - A thread is canceled while in the midst of updating data it is sharing with other threads.

Pthread Cancellation

- Phread cancellation is initiated using the function:

pthread_cancel()

The identifier of the target Pthread is passed as a parameter to the function

- The default cancelation type is the deferred cancelation
    - Cancellation only occurs when thread reaches **cancellation point**
    - One way for establishing a cancellation point is to invoke the **pthread_testcancel()** function.
    - If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
    - Local variables visible only during single function invocation
    - TLS visible across function invocations
- Similar to **static** data
    - TLS is unique to each thread
- This data structure is known as a – **lightweight process** (**LWP**)

- Appears to be a virtual processor on which process can schedule user thread to run
- Each LWP is attached to kernel thread.

■ Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library

■ This communication allows an application to maintain the correct number kernel threads

Chapter 5

Process Synchronization

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.

Concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

A situation like this, where several processes access and manipulate the same data concurrently and the
outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

The Critical-Section Problem

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*-1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table,
writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their
critical sections at the same time.

The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.

The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times
that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**.

A **preemptive kernel** allows a process to be preempted while it is running in kernel mode.

A **preemptive kernel** may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.

A **preemptive kernel** is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

A **nonpreemptive** kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

A **nonpreemptive** kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Peterson's solution.

A classic software-based solution to the critical-section problem known as **Peterson's solution**.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1. For convenience, when presenting *Pi*, we use *Pj* to denote the other process; that is, j equals 1 - i.
Peterson's solution requires the two processes to share two data items:
int turn;
boolean flag[2];
The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.

Synchronization Hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.

All these solutions are based on the premise of **locking** —that is, protecting critical regions through
the use of locks.

Hardware features can make any programming task easier and improve system efficiency.
The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Mutex Locks

The simplest of these tools is the **mutex lock**. (In fact, the term ***mutex*** is short for ***mut***ual ***ex***clusion.) We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock,

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process
"spins" while waiting for the lock to become available.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
The wait() operation was originally termed P (from the Dutch ***proberen,*** "to test"); signal() was originally called V (from ***verhogen,*** "to increment"). The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // busy wait
S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
S++;
}
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly.