Introduction

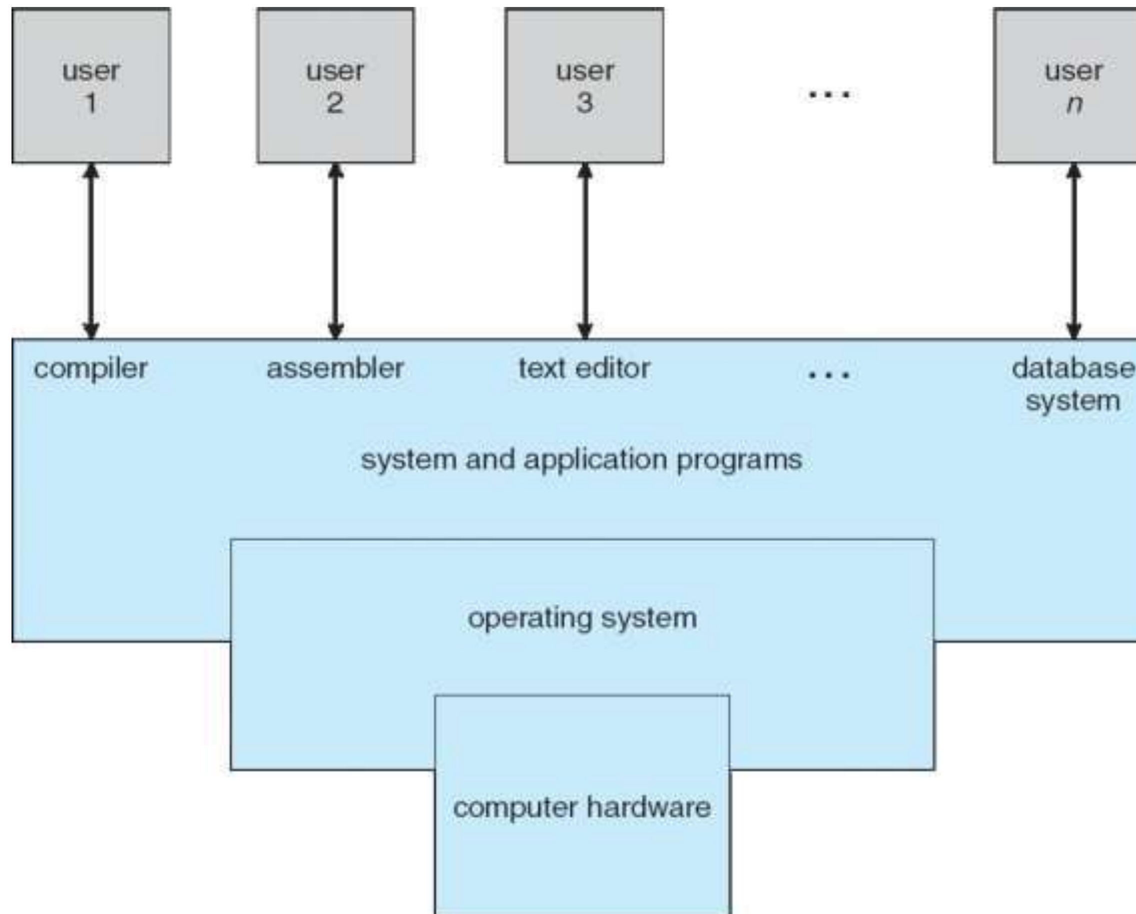
Computer System Structure

- 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

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

Four Components of a Computer System



Operating Systems Goals

- Users want **convenience**, **ease of use** and **good performance**
- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer

Functions of an Operating System

- Process Management
- Memory Management
- Storage Management
- Device management
- Protection and Security

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data

Process Management Activities

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

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

Memory Management

- To execute a program all (or part) of the instructions and data 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 de-allocating memory space as needed

Storage Management

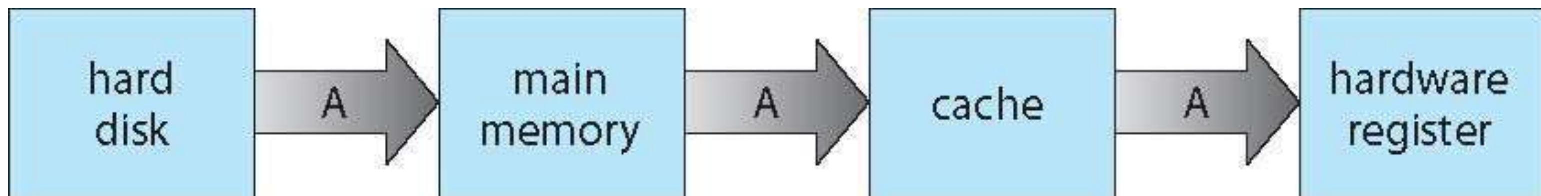
- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
- **File-System management**
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - 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

Storage Management

- **Mass-Storage Management**
  - Usually disks used to store data that does not fit in main memory or data that must be kept for a "long" period of time
  - Entire speed of computer operation hinges on disk subsystem and its algorithms
  - OS activities
    - Free-space management
    - Storage allocation
    - Disk scheduling

Storage Management

- **Caching**
  - Cache Management
  - Cache Coherency
    - In multiprocessor environment all CPUs have the most recent value in their cache
    - A copy of A may exist in several caches. Any update on A should reflect in all other caches.



Device Management (I/O Subsystem)

- One purpose of OS 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

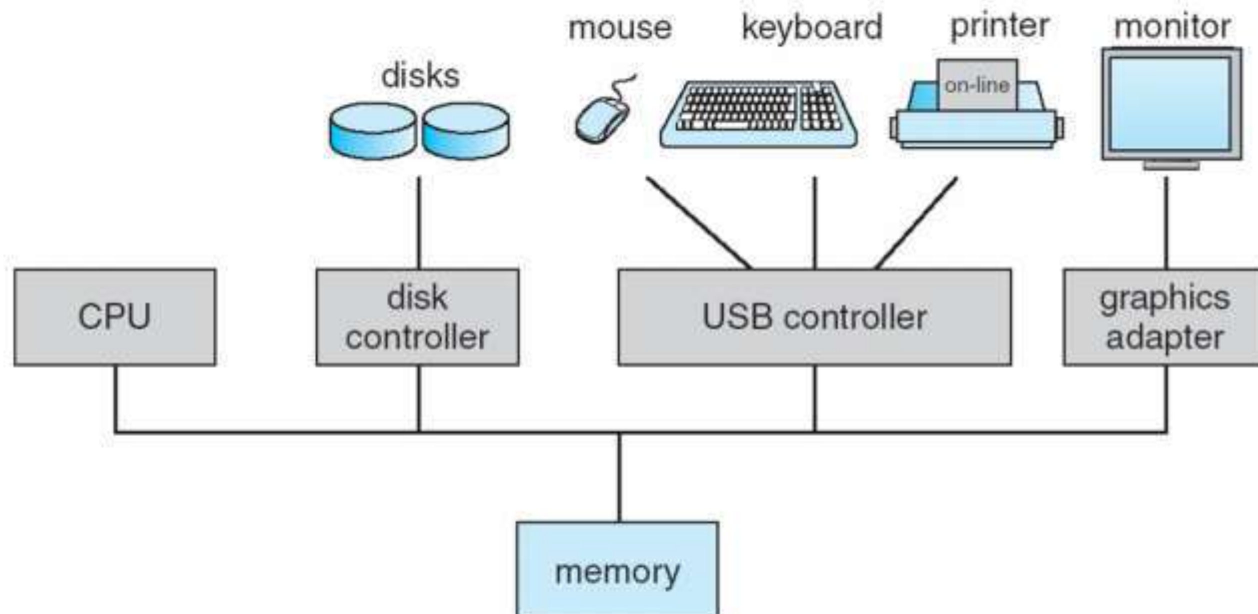
- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first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

Operating System Definition (Cont.)

- No universally accepted definition
- "The one program running at all times on the computer" is the **kernel**.
- Everything else is either
  - a system program (ships with the operating system) , or
  - an application program.

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer Startup

- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt

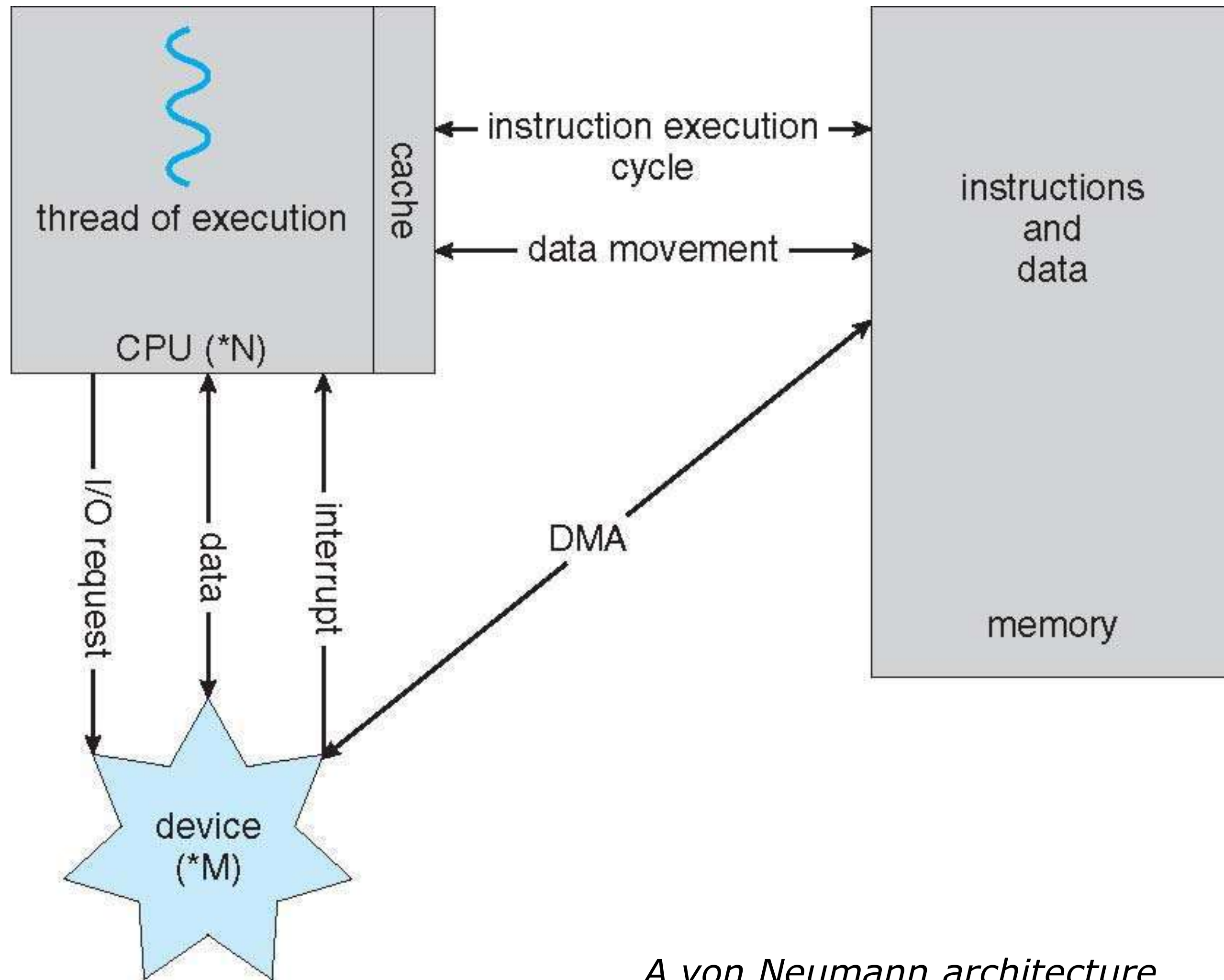
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
  - **polling**
  - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

How a Modern Computer Works



A von Neumann architecture

Computer System Architecture

Single-Processor Systems

- On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.

- Almost all single processor systems have other special-purpose processors like device-specific processors, such as disk, keyboard, and graphics controllers.

- All of these special-purpose processors run a limited instruction set and do not run user processes.

- Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status.

Single-Processor Systems

- In some other systems special-purpose processors are low-level components built into the hardware.
- The operating system cannot communicate with these processors; they do their jobs autonomously.

Multiprocessor Systems

- Two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

1. Increased throughput

- By increasing the number of processors, more work done in less time.
- The speed-up ratio with *N processors* is not *N, however; rather, it is less than N.*
- *When multiple processors* cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly.
- Contention for shared resources also, lowers the expected gain from additional processors.

Multiprocessor Systems

2. **Economy of scale**

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

3. **Increased reliability**

If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

Multiprocessor Systems

- The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation.**
- **Fault tolerance** is the property that enables a system to continue operating properly in the event of the failure of (or one or more **faults** within) some of its components.

Multiprocessor Systems

- The multiple-processor systems in use today are of two types.

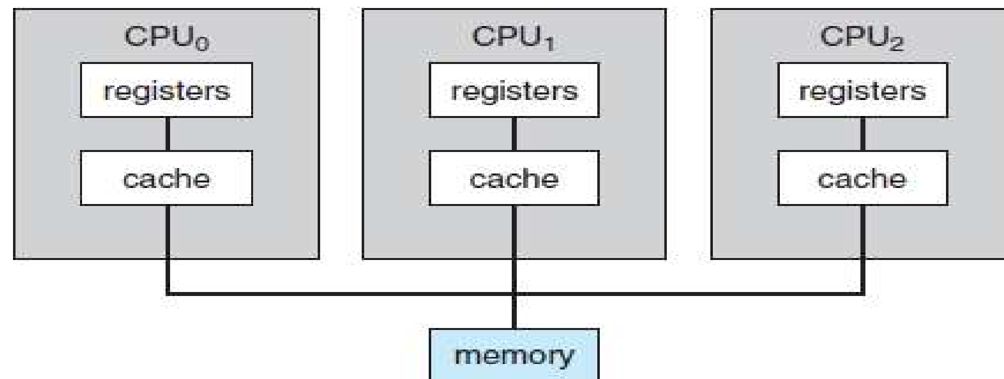
Asymmetric multiprocessing

- **Each processor is assigned** a specific task.
- A ***boss processor controls the system; the other processors either*** look to the boss for instruction or have predefined tasks.
- This scheme defines a boss–worker relationship.
- The boss processor schedules and allocates work to the worker processors.

Multiprocessor Systems

Symmetric multiprocessing (SMP)

- Each processor performs all tasks within the operating system.
- All processors are peers; no boss–worker relationship exists between pr



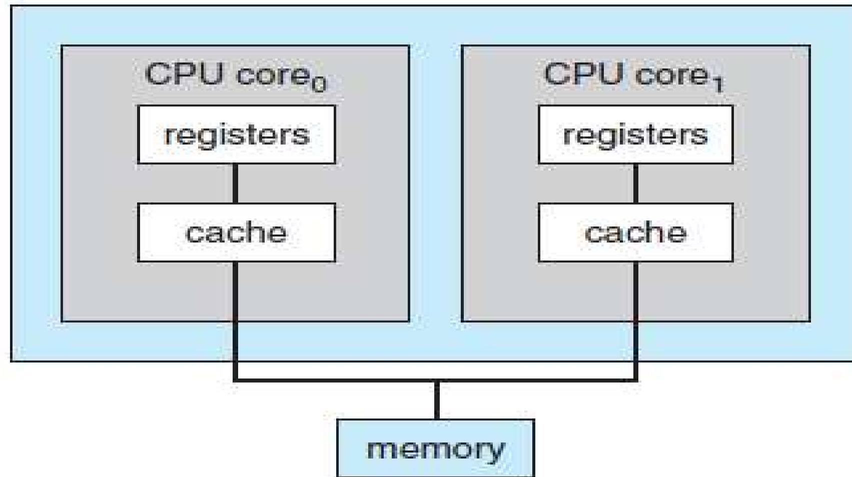
Symmetric multiprocessing architecture.

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

Multiprocessor Systems

- Multiprocessing adds CPUs to increase computing power.
- Multiprocessor systems are termed as **multicore when** multiple computing **cores** on a single chip.
- They 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.

Multiprocessor Systems



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

Clustered Systems

- A type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs.
- Clustered systems differ from the multiprocessor systems in that they are composed of two or more individual systems or nodes joined together.
- **Each node may be a single processor system or a multicore** system.
- Clustered computers share storage and are closely linked via a local-area network LAN or a faster interconnect, such as InfiniBand.
- Clustering is usually used to provide **high-availability service**.
  - **High Availability:** Service will continue if one or more systems in cluster fails.

Clustered Systems

- Service will continue even if one or more systems in the cluster fail.
- A layer of cluster software runs on the cluster nodes.
- Each node can monitor one or more of the others (over the LAN).
- If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine.
- The users and clients of the applications see only a brief interruption of service.

Clustered Systems

- Clustering can be structured asymmetrically or symmetrically.

Asymmetric clustering

- One machine is in **hot-standby mode** while the other is running the applications.
- The hot-standby host machine does nothing but monitor the active server.
- If that server fails, the hot-standby host becomes the active server.

Symmetric clustering

- Two or more hosts are running applications and are monitoring each other.
- This structure is more efficient, as it uses all of the

Clustered Systems

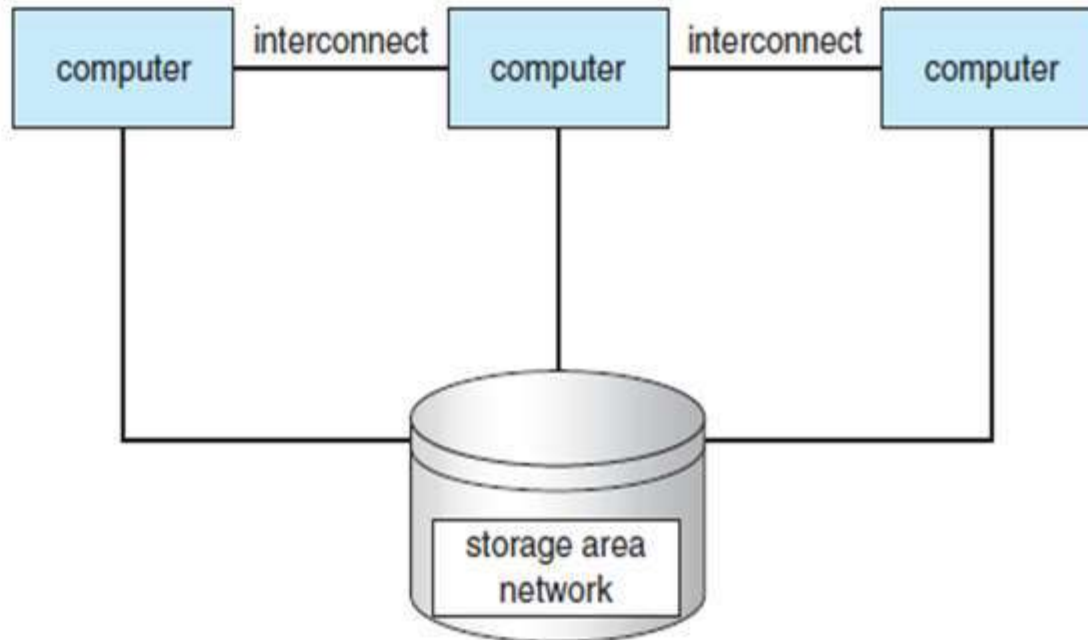
- Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments.

Parallelization

  - Divides a program into separate components that run in parallel on individual computers in the cluster.
  - These applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

- Parallel clusters allow multiple hosts to access the same data on shared storage
- Access control and locking of shared storage by **Distributed Lock Manager (DLM)**

Clustered Systems

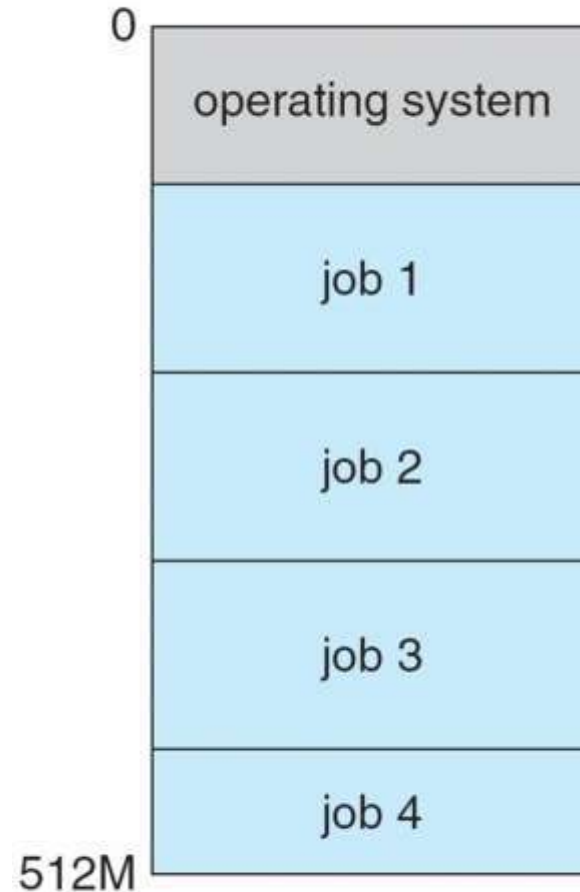


General structure of a clustered system.

Operating System Structure

- **Multiprogramming** (**Batch system**) needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - When it has to wait (for I/O for example), OS switches to another job

Memory Layout for Multiprogrammed System



Operating System Structure

- **Timesharing** (**multitasking**) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**

Both Timesharing and Mutiprogramming

- **Job pool : In secondary memory**
- **Job scheduling :** several process ready for memory but not enough space
- If several jobs ready to run at the same time ⇨ **CPU scheduling**
- If processes don't fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory
  - Physical memory
  - Logical memory

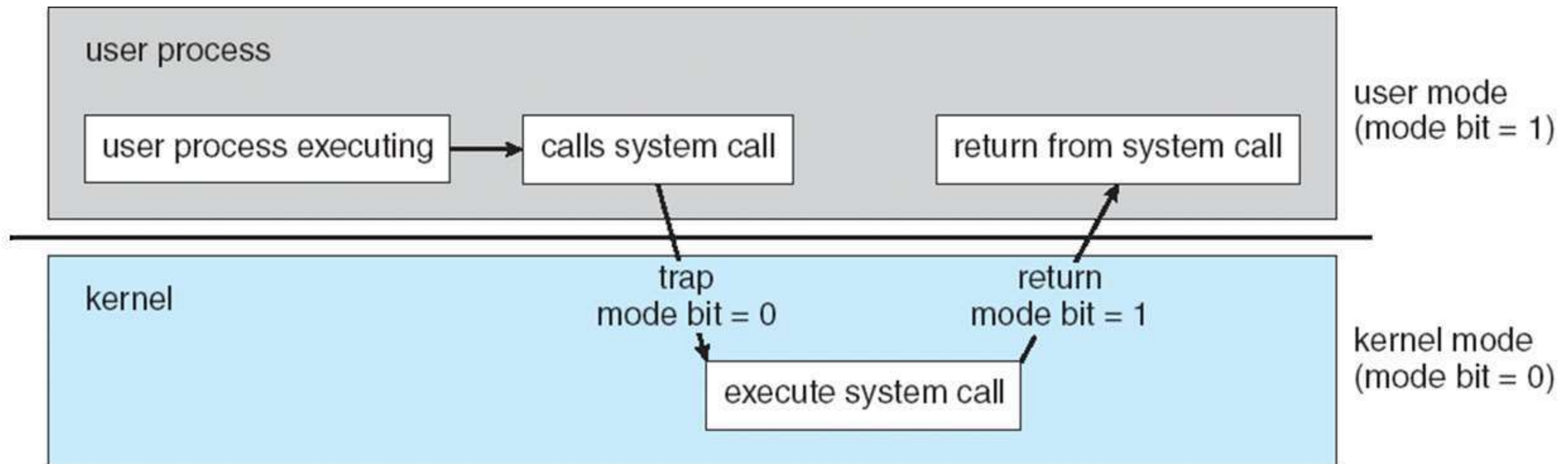
Operating-System Operations

- **Interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap):**
    - Software error (e.g., division by zero)
    - Request for operating system service
    - Other process problems include infinite loop, processes modifying each other or the operating system

Operating-System Operations (cont.)

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
  - i.e. **virtual machine manager** (**VMM**) mode for guest **VMs**

Transition from User to Kernel Mode

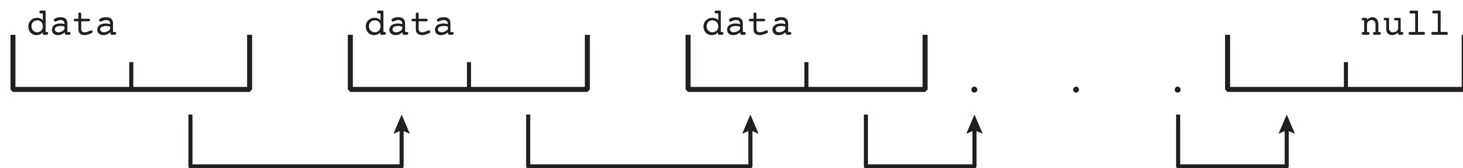


Transition from User to Kernel Mode

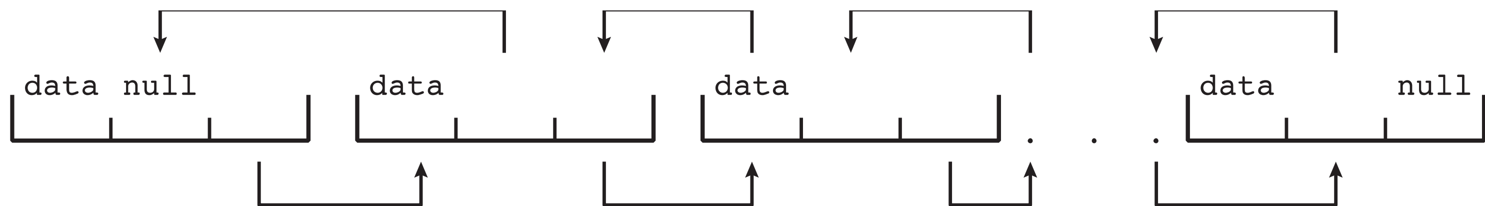
- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Kernel Data Structures

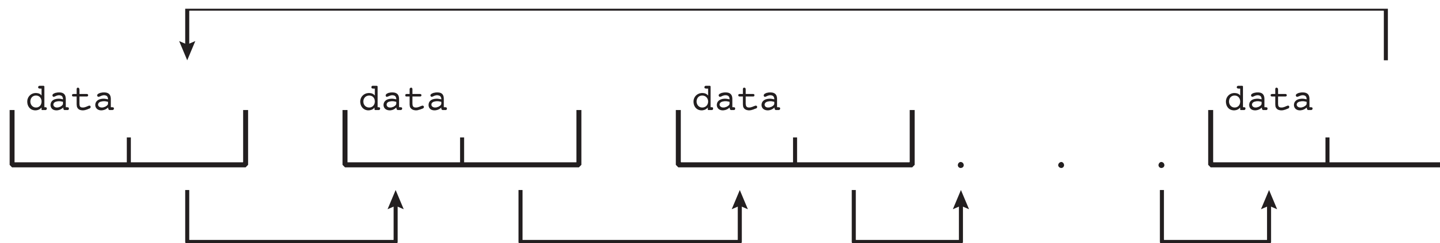
- **List : Linked list**
  - ***Singly linked list***



  - ***Doubly linked list***



  - ***Circular linked list***

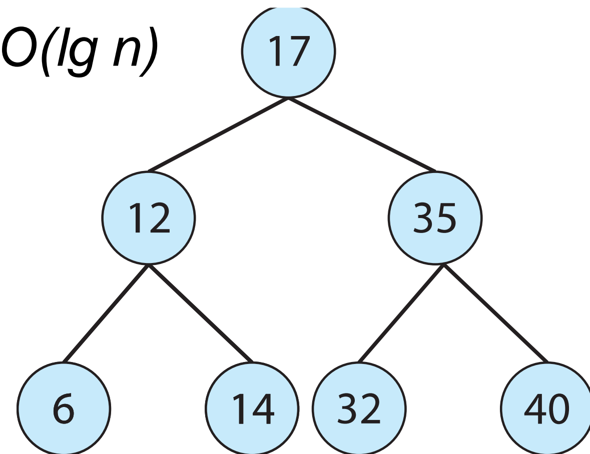


Kernel Data Structures

- **Stack**
  - Uses LIFO
  - *push* and *pop* Operations
- **Queue**
  - FIFO

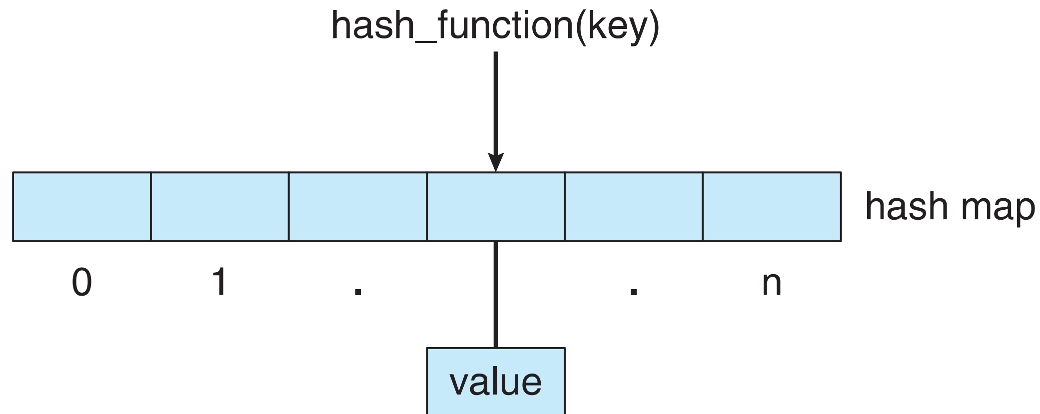
Kernel Data Structures

- **Trees**
  - **General Tree**
  - **Binary Tree**
  - **Binary search tree**
    - left <= right
    - Search performance is *O(n)*
    - **Balanced binary search tree** is *O(lg n)*



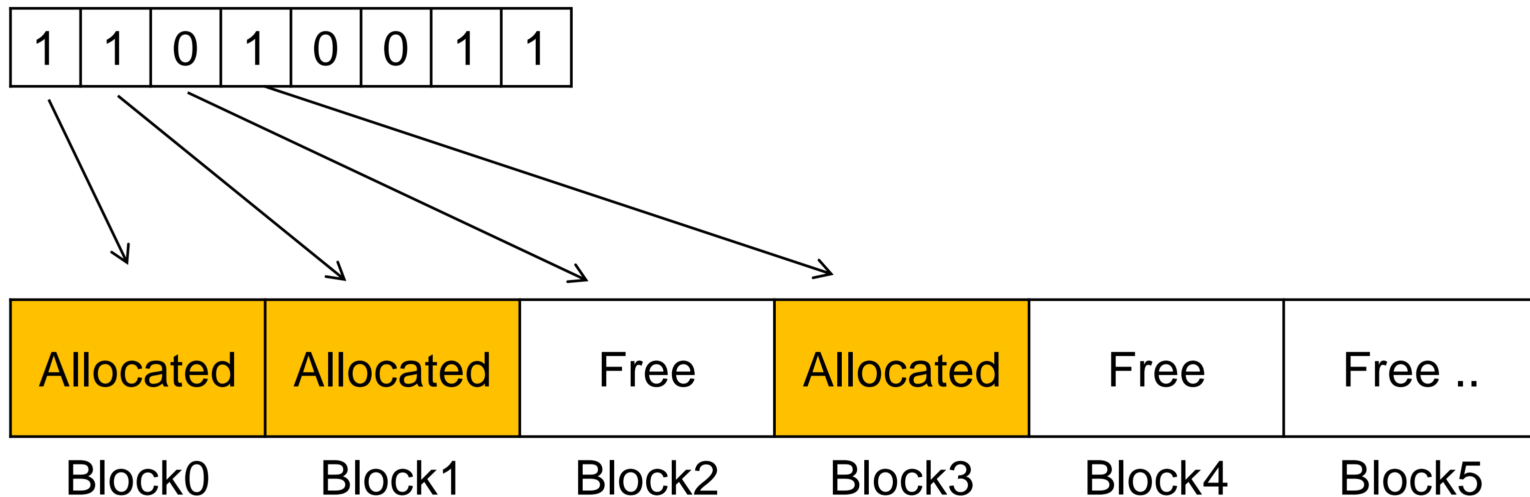
Kernel Data Structures

- **Hash function** can create a **hash map**



Kernel Data Structures

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



- Linux data structures defined in

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

Computing Environments

- Traditional Computing
- Mobile Computing
- Distributed Systems
- Client-Server Computing
- Peer-to-Peer Computing
- Virtualization
- Cloud Computing
- Real-Time Embedded Systems

Traditional Computing

- Stand-alone general purpose machines
- Most systems interconnect with others (i.e., Internet)
- **Portals** provide web access to internal systems
- **Network computers** (**thin clients**) are like Web terminals
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks
- All started from Batch systems, then interactive systems and then to time – sharing systems.

Mobile Computing

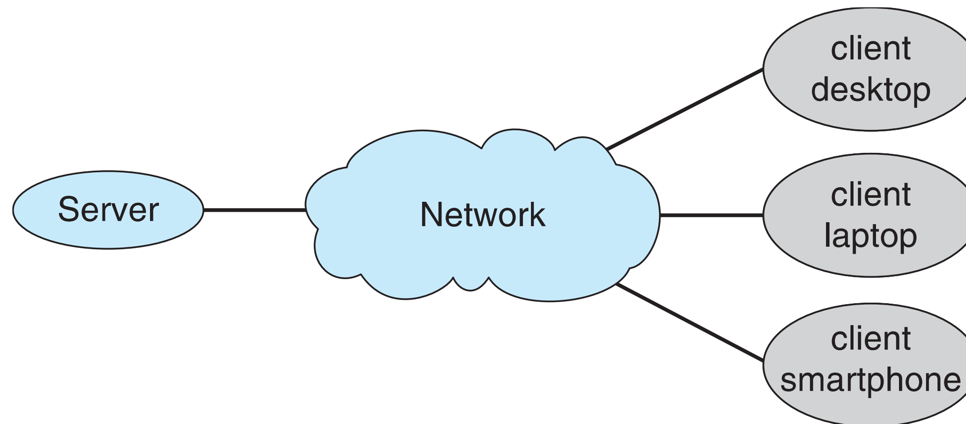
- Handheld smartphones, tablets, etc
- What is the functional difference between them and a "traditional" laptop?
- Extra feature – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**

Distributed Systems

- Distributed computing
  - Collection of separate, possibly heterogeneous, systems networked together
    - **Network** is a communications path, **TCP/IP** most common
      - **Local Area Network** (**LAN**)
      - **Metropolitan Area Network** (**MAN**)
      - **Wide Area Network** (**WAN**)
      - **Personal Area Network** (**PAN**)
  - **Network Operating System** provides features between systems across network
    - Communication scheme allows systems to exchange messages
    - Illusion of a single system

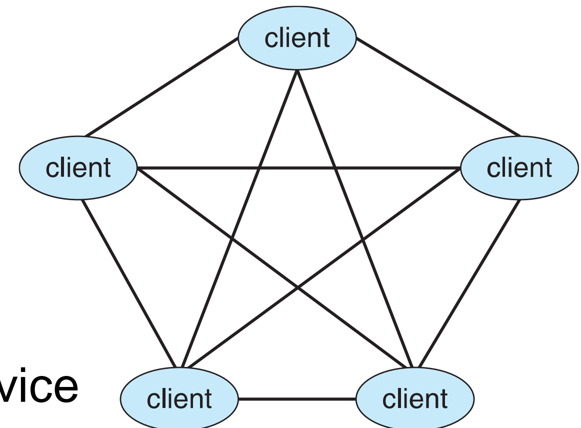
Client-Server Computing

- 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 Computing

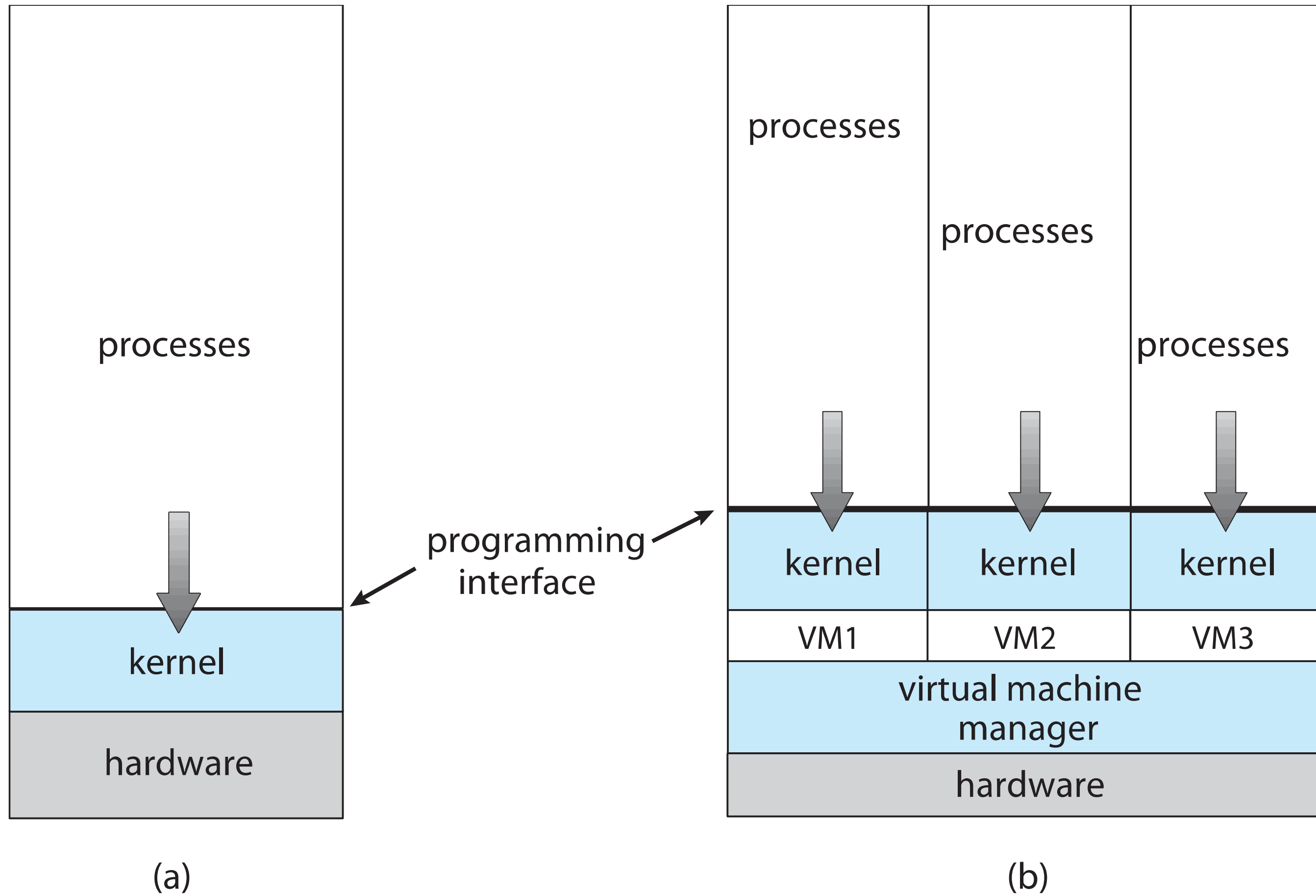
- Another model of distributed system
- P2P does not distinguish clients and servers
  - Instead all nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via ***discovery protocol***
  - Examples include Napster and Gnutella, **Voice over IP** (**VoIP**) such as Skype



Virtualization

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

Virtualization



Virtualization

- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - Developing apps for multiple OSes without having multiple systems
  - QA testing applications without having multiple systems
  - Executing and managing compute environments within data centers

Cloud Computing

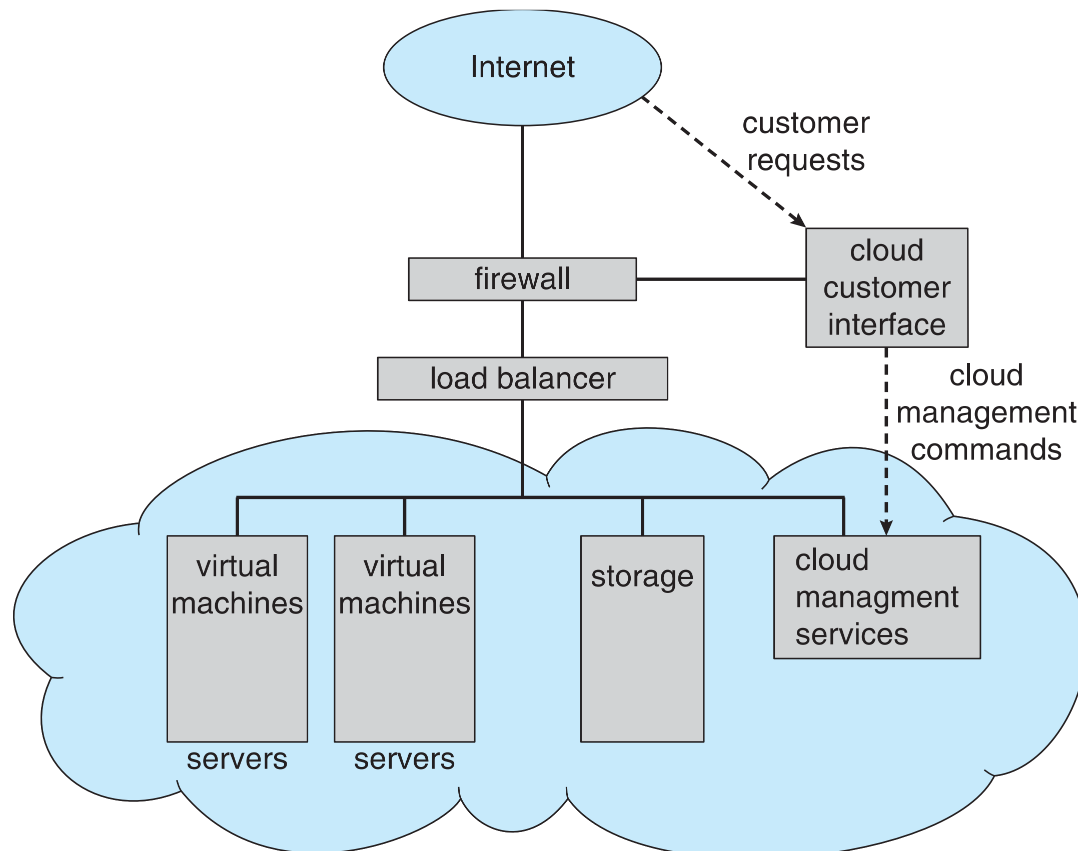
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet,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

Cloud Computing

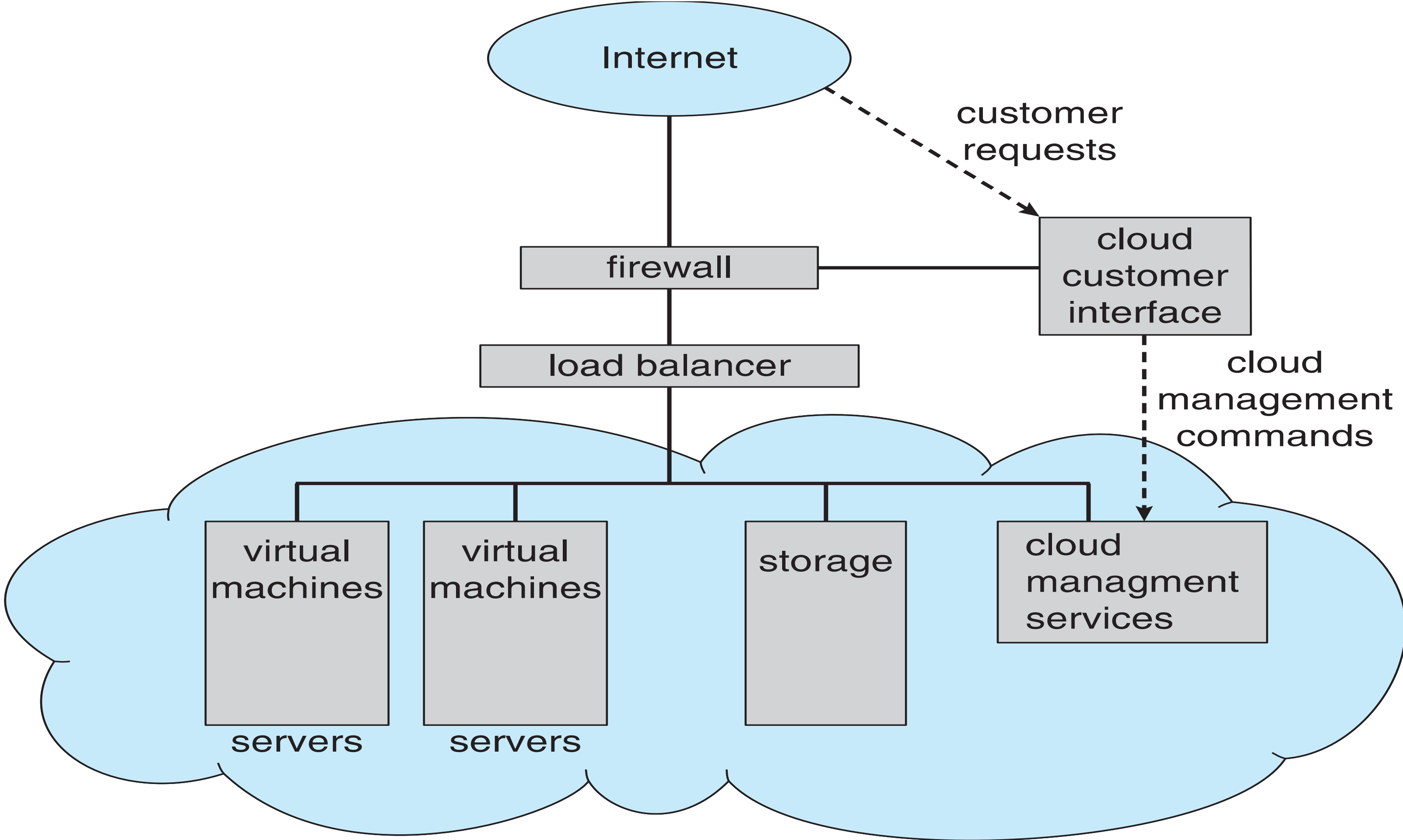
- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
  - 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)

Cloud Computing

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications



Cloud Computing



Real-Time Embedded Systems

- Real-time embedded systems most prevalent form of computers
- These devices are found everywhere, from car engines manufacturing robots, microwave ovens etc.
- Usually, they have little or no user interface,
  - spend their time monitoring and managing hardware devices.

Real Time Embedded Systems

Types

1. General-purpose computers, running standard operating systems—such as Linux — with special-purpose applications to implement the functionality.
2. Hardware devices with a special-purpose embedded operating system providing just the functionality desired.
3. Hardware devices with application specific integrated circuits (**ASICs)** that perform their tasks without an operating system.

Real Time Embedded Systems

- Embedded systems almost always run **real-time operating systems.**
- A real-time system has well-defined, fixed time constraints
- Processing *must be done within the defined constraints, or the system will fail.*
- Eg:-
  - Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real time systems.
  - Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

Interprocess Communication

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by other processes, including sharing data

Interprocess Communication

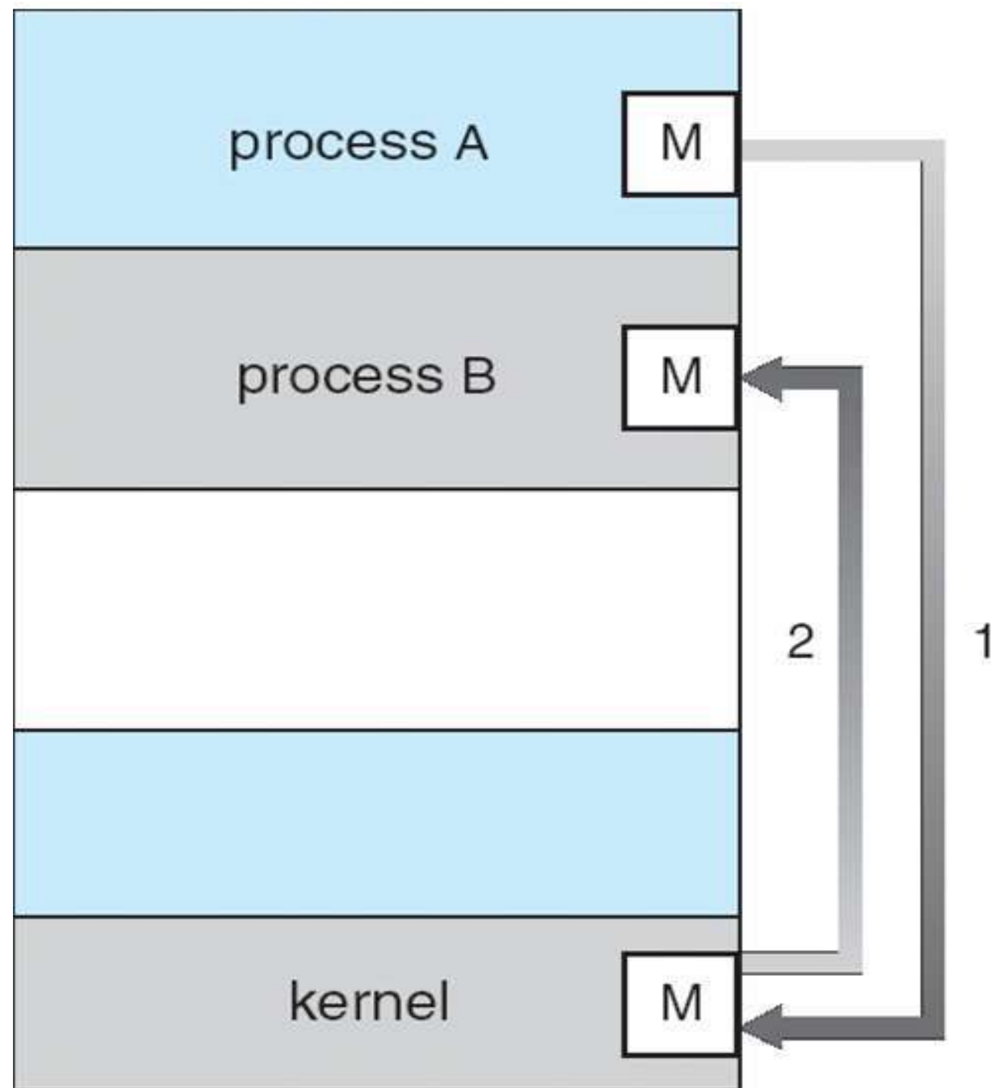
Cooperating Processes

- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication

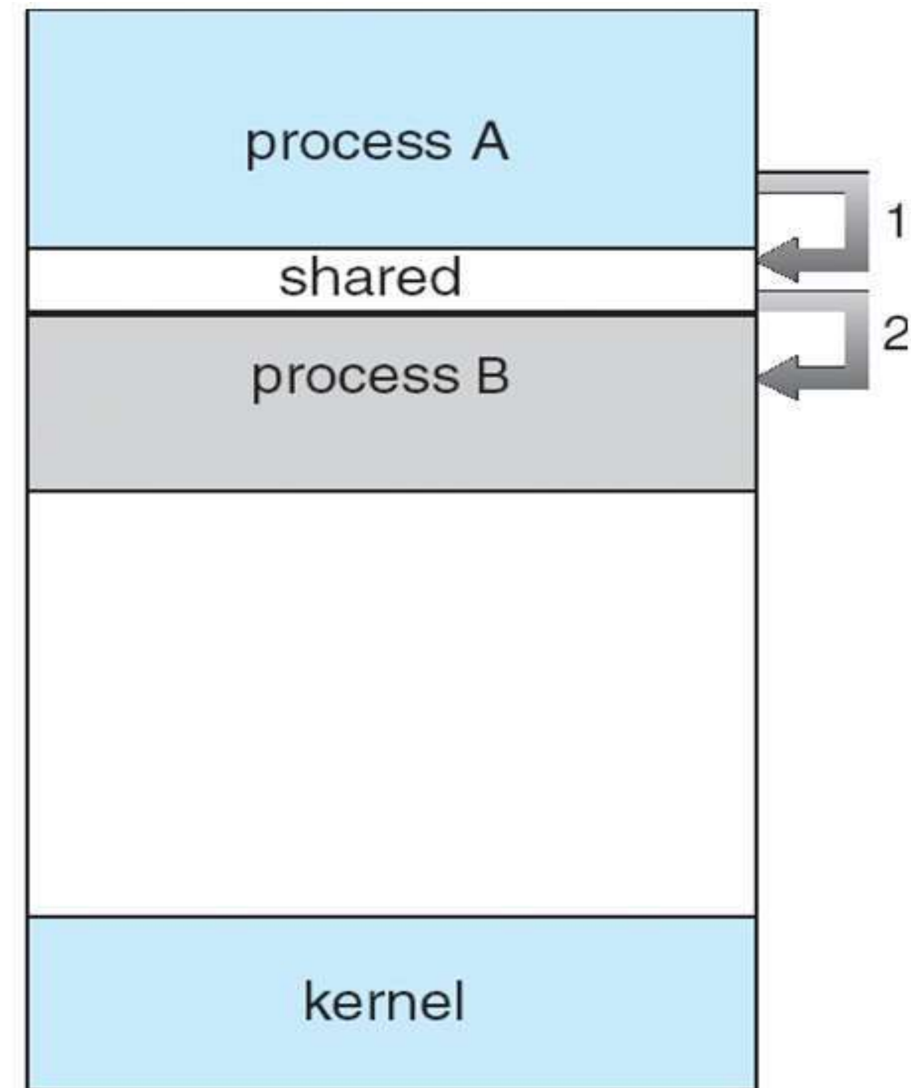
- Cooperating processes need **interprocess communication** (IPC)
- Two models of IPC
  - **Shared memory**
    - A region of memory that is shared by cooperating processes is established.
  - **Message passing**
    - communication takes place by means of messages exchanged between the cooperating processes

Communications Models



(a)

Message Passing



(b)

Shared Memory

Interprocess Communication

- Shared memory
  - For sharing large amount of data
  - Is faster because system call is needed only to establish shared-memory region
- Message passing
  - Useful for exchanging smaller amounts of data
  - Easier to implement
  - Time-consuming because every message need system call

Shared –Memory Systems

- Shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

[Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction.]

Shared –Memory Systems

- Process can then exchange information by reading and writing data
- The form of the data and the location are determined by these processes and are not under the operating system's control

Shared – Memory Systems

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- Solution to producer-consumer problem – Shared memory
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Producer

```
while (true) {
  /* Produce an item in next_produced*/

    while (( (in + 1) % BUFFER_SIZE) == out)
        ;                          /* do nothing -- no free buffers
*/

    buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
    }
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
            ;       // do nothing -- nothing to consume

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

  /* consume the item in next_consumed*/
    }
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

Interprocess Communication – Message Passing

- If P and Q wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Logical Implementation of Link

- Direct or indirect Communication (Naming)
- Synchronous or asynchronous communication
- 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 direct 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

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
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Properties of Indirect 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

Indirect Communication

- Consider a scenario of Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 execute 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

- Operations on OS mailbox
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**

- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   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

Interprocess Communication

using

Pipes

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Pipes

- Two Types of Pipes:-
  - 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.
  - Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

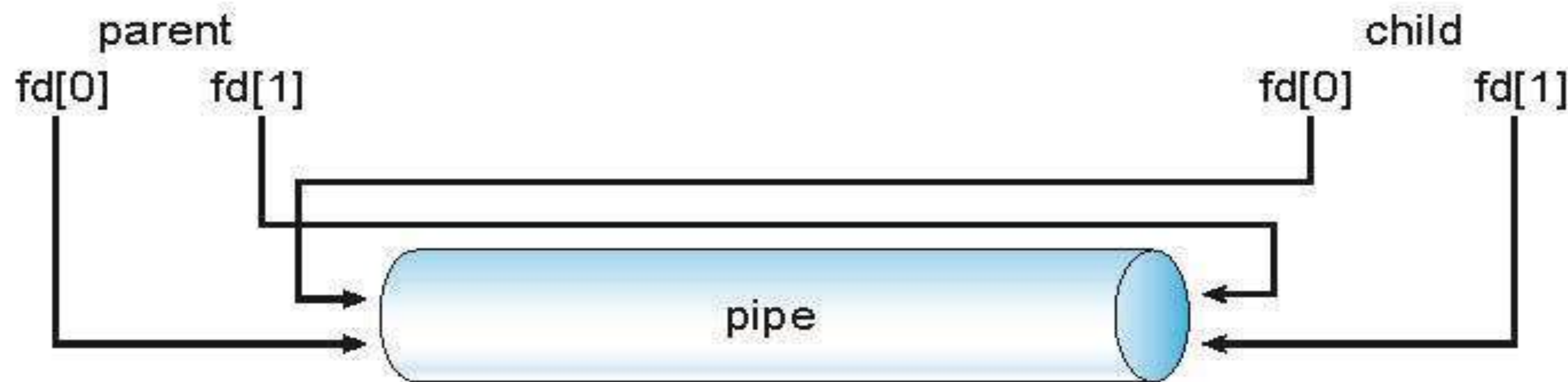
- Ordinary Pipes allow communication in standard producer-consumer style
- On UNIX systems, ordinary pipes are constructed using the function

  pipe(int fd[])

- Windows calls these **anonymous pipes**

Ordinary Pipes

- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



```
#include <sys/types.h>
#include <stdio.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
```

```
/* create the pipe */
if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
}
/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
}
```

```
if (pid > 0) {   /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
}
```

```
else {  /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      print("read %s", read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
      }
return 0;
}
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Named pipes continue to exist after communicating processes have finished
- Provided on both UNIX and Windows systems

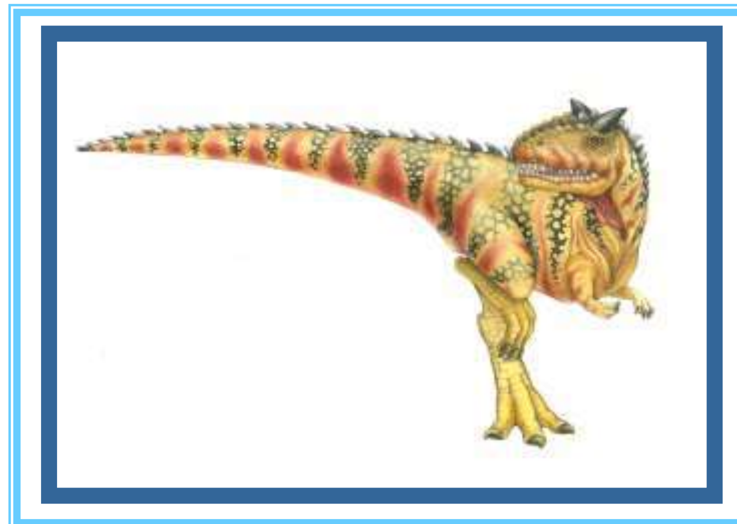
Named Pipe in UNIX

- Named pipes are referred to as **FIFOs** in UNIX systems
- A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(), and close() system calls.
- Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used.
- Additionally, the communicating processes must reside on the same machine.
- Only byte-oriented data may be transmitted across a UNIX FIFO

Named pipes on Windows systems

- Named pipes are created with the CreateNamedPipe() function
- A client can connect to a named pipe using ConnectNamedPipe().
- Communication can be accomplished using the ReadFile() and WriteFile() functions.
- Full-duplex communication is allowed
- The communicating processes may reside on either the same or different machines.
- Windows systems allow either byte- or message-oriented data.

Threads



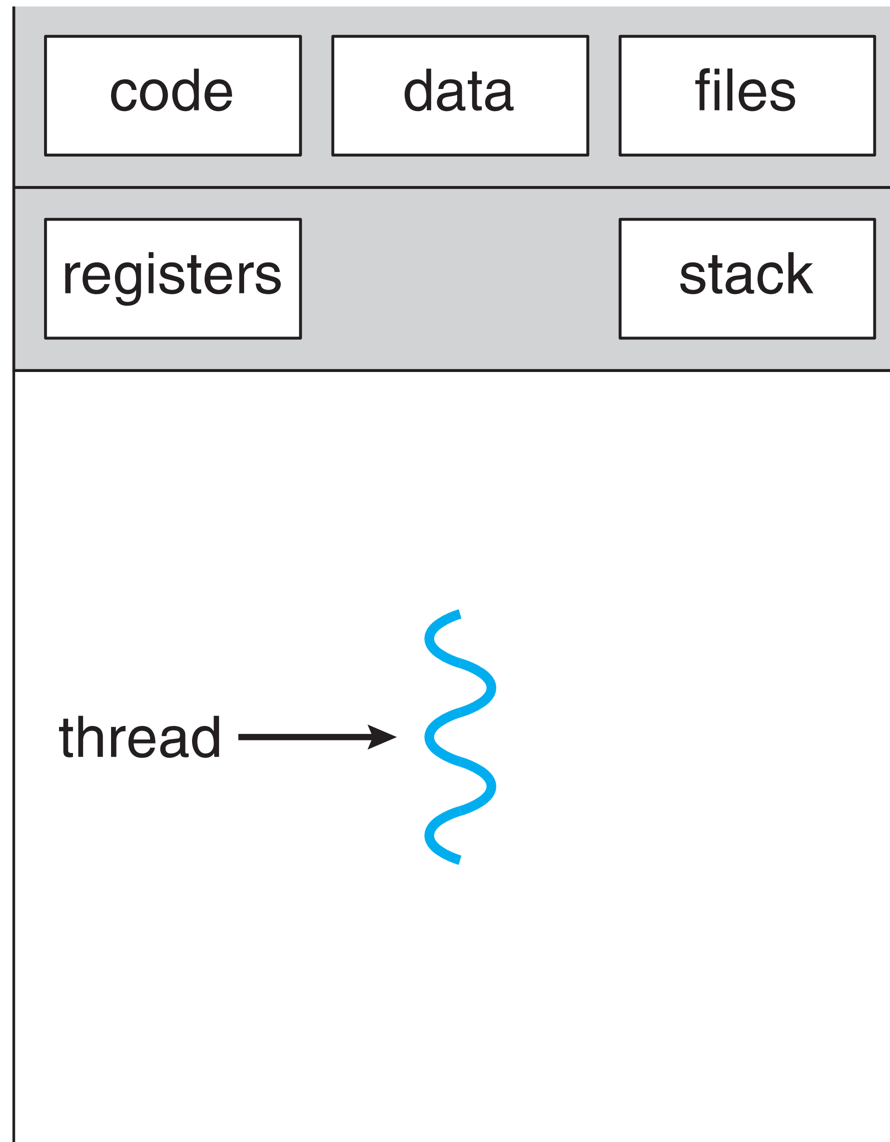
Motivation

- 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
- Most modern applications are multithreaded
- Kernels are generally multithreaded

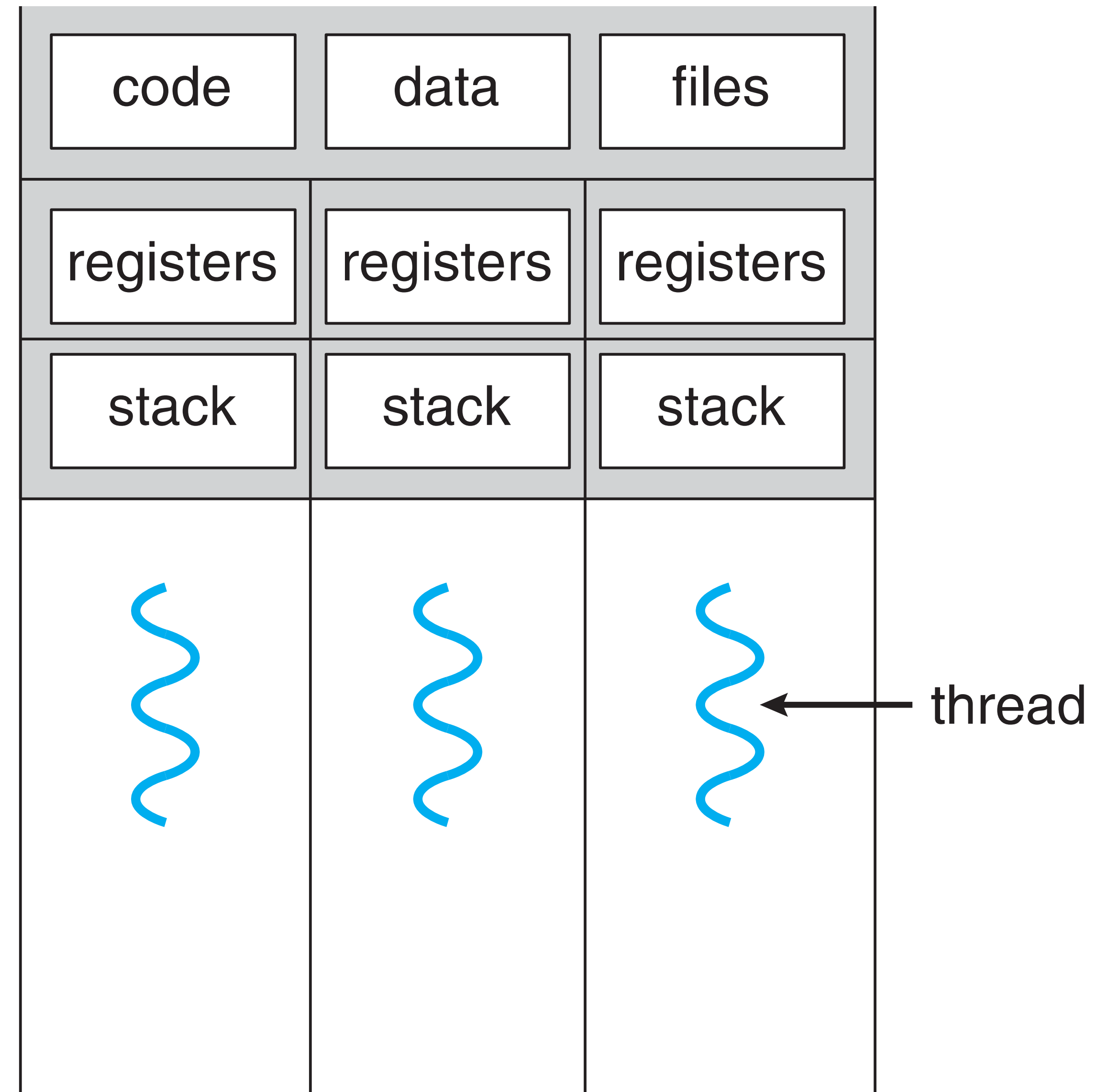
Threads

- Threads in operating systems are **lightweight processes that improve application speed** by executing concurrently within the same process
- A thread is a basic unit of CPU utilization
- **Threads** are a way for a program to divide ("split") itself into two or more simultaneously running tasks

Single and Multithreaded Processes



single-threaded process



multithreaded process

Benefits

- **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

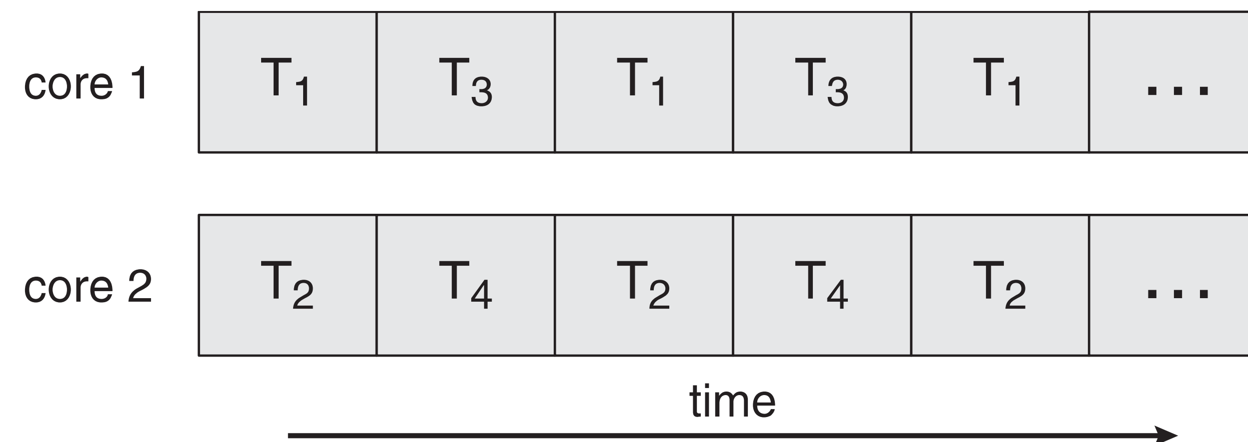
Multicore Programming

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

Concurrency vs. Parallelism

- Concurrent execution on single-core system:

- Parallelism on a multi-core system:



Programming Challenges

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - **Identifying Tasks**
  - **Balance:** tasks perform equal work of equal value
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

User Threads and Kernel Threads

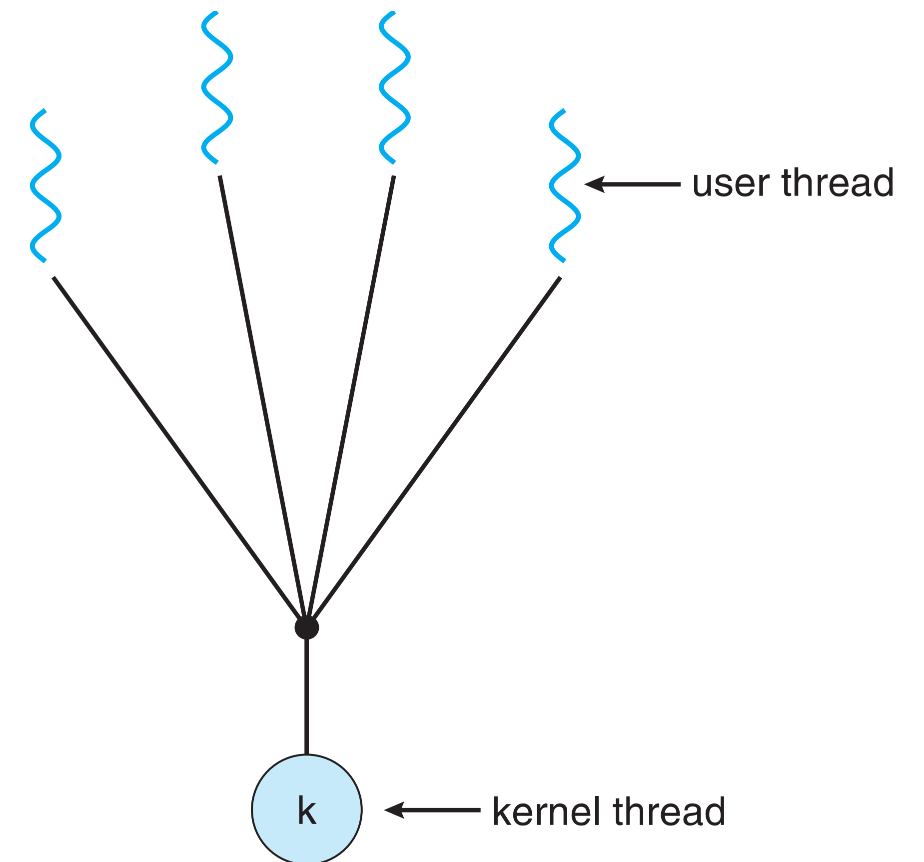
- User threads - management done by user-level threads library
- Kernel threads - Supported by the Kernel

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

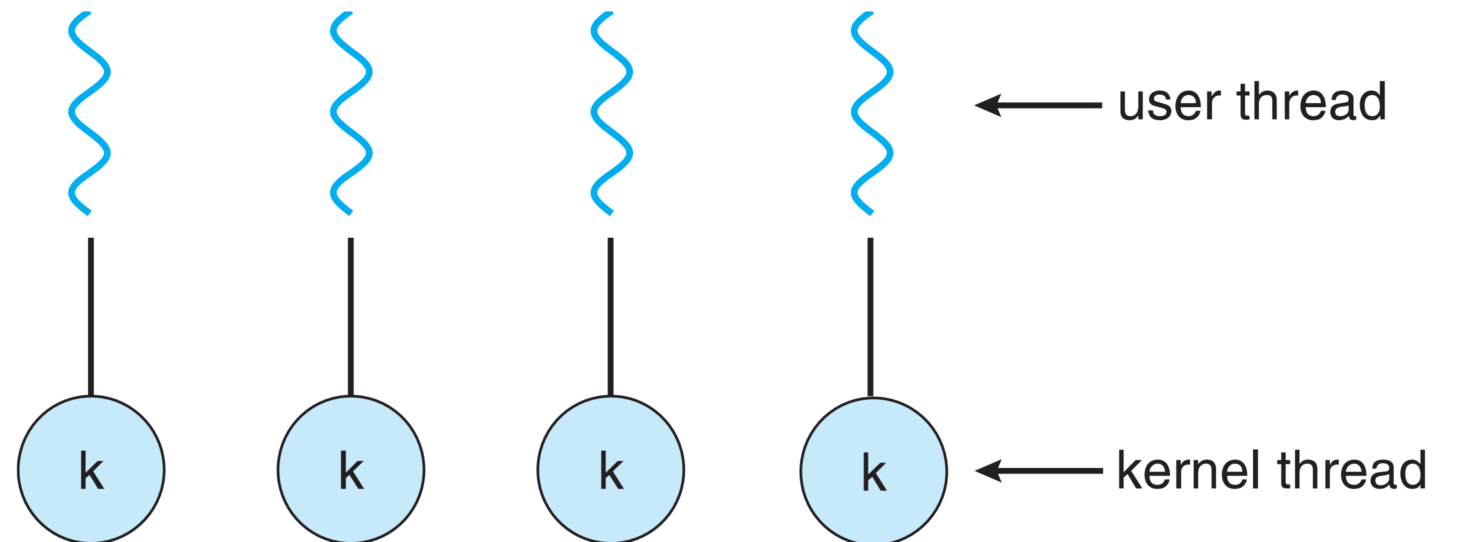
Many-to-One

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



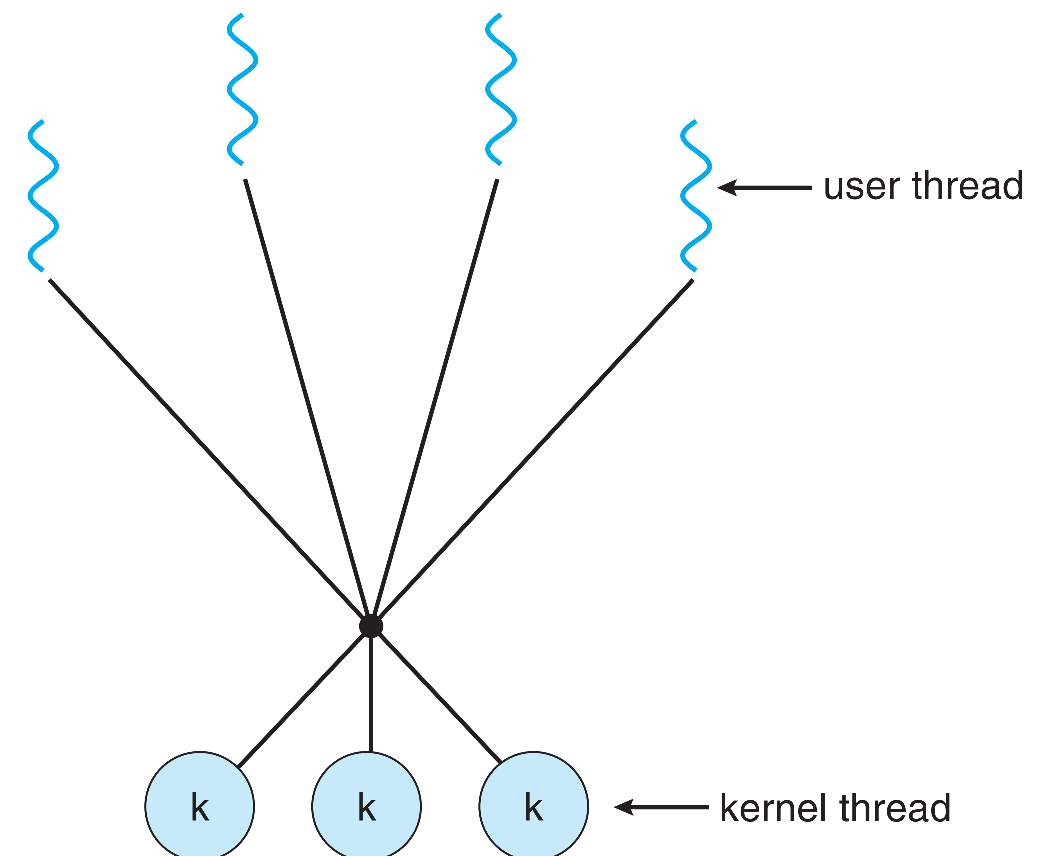
One-to-One

- Each user-level thread maps to 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
  - Solaris 9 and later



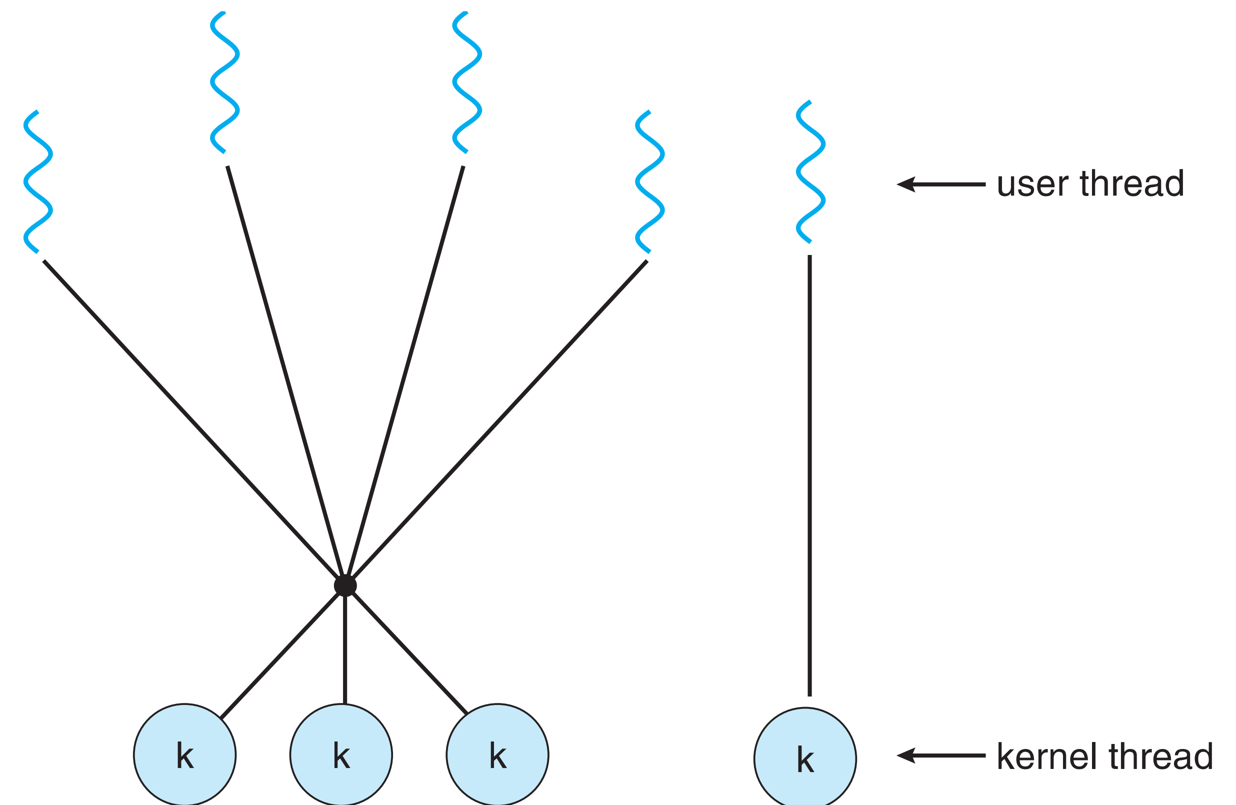
Many-to-Many Model

- 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

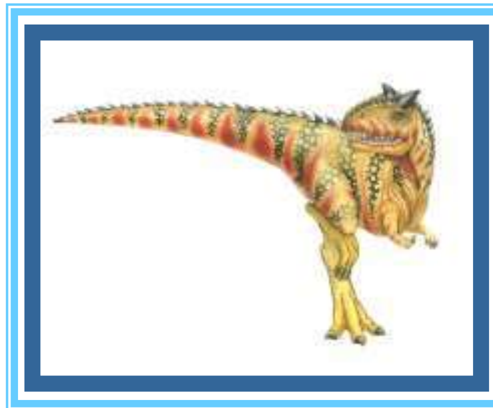


Two-level Model

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



Chapter 2: Operating-System Structures



Chapter 2: Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- OS services that are helpful to the user:
  - **User interface:**
    - **Command-Line (CLI)**, **Batch, Graphics User Interface (GUI)**,
  - **Program execution**
  - **I/O operations**
  - **File-system manipulation**
  - **Communications**
  - **Error detection**

Operating System Services (Cont.)

- OS functions for ensuring the efficient operation of the system
  - **Resource allocation**
  - **Accounting**
  - **Protection and security**
    - **Protection**
    - **Security**

A View of Operating System Services

user and other system programs

GUI | batch | command line

user interfaces

system calls

program execution

I/O operations

file systems

communication

resource allocation

accounting

error detection

protection and security

services

operating system

hardware

User and Operating System Interface

- Command Interpreters
- Graphical User Interfaces
- Touchscreen Interfaces

Command Interpreters

CLI or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Primarily fetches a command from user and executes it
- **Shells**

Bourne Shell Command Interpreter

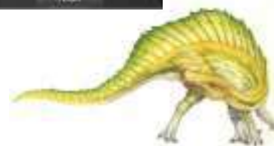
```
Default
New   Info   Close   Execute   Bookmarks
Default   Default
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                14:34      50 -
pbg      s000     -                15:05       - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1           disk10       cpu     load average
    KB/t tps  MB/s     KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30    64.31  14  0.88    39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65     0.00   0  0.00     0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37     0.00   0  0.00     0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$
```

Graphical User Interfaces

- 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
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

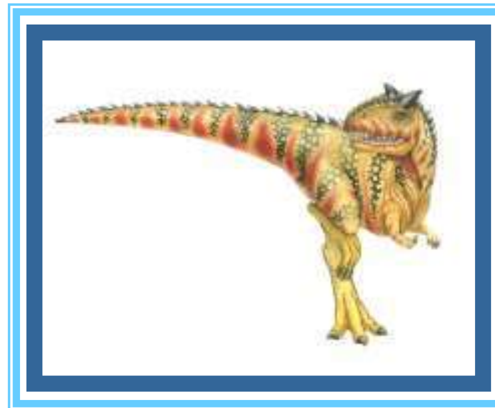
Touchscreen Interfaces

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



The Mac OS X GUI

System Calls



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 use
- Three most common APIs are
  - Win32 API for Windows,
  - POSIX API for UNIX, Linux, and Mac OS X
  - Java API for the Java virtual machine (JVM)

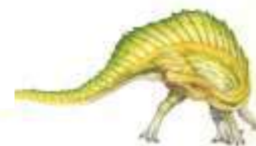
Example of System Calls

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

source file → destination file

Example System Call Sequence

Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally



System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values

Handling System Calls

1. User Process Requests a System Call

- A program executes a system call (e.g., reading a file) using a library function like printf() in C, which internally calls write().

2. Transition from User Mode to Kernel Mode

- The request triggers a software interrupt or trap, causing the CPU to switch from **user mode** to **kernel mode**.
- The OS takes control and verifies the validity of the request.

Handling System Calls

3. Execution of System Call in the Kernel

- The OS identifies the requested system call using a system call number.
- The appropriate system call handler in the kernel executes the requested operation.

4. Returning the Result to the User Process

- After execution, the OS switches back to **user mode** and returns the result to the calling process.
- If an error occurs, an error code is returned.

API – System Call – OS Relationship

user application

open ( )

user mode

kernel mode

system call interface

i

open ( )

Implementation of open ( ) system call

•
•
•

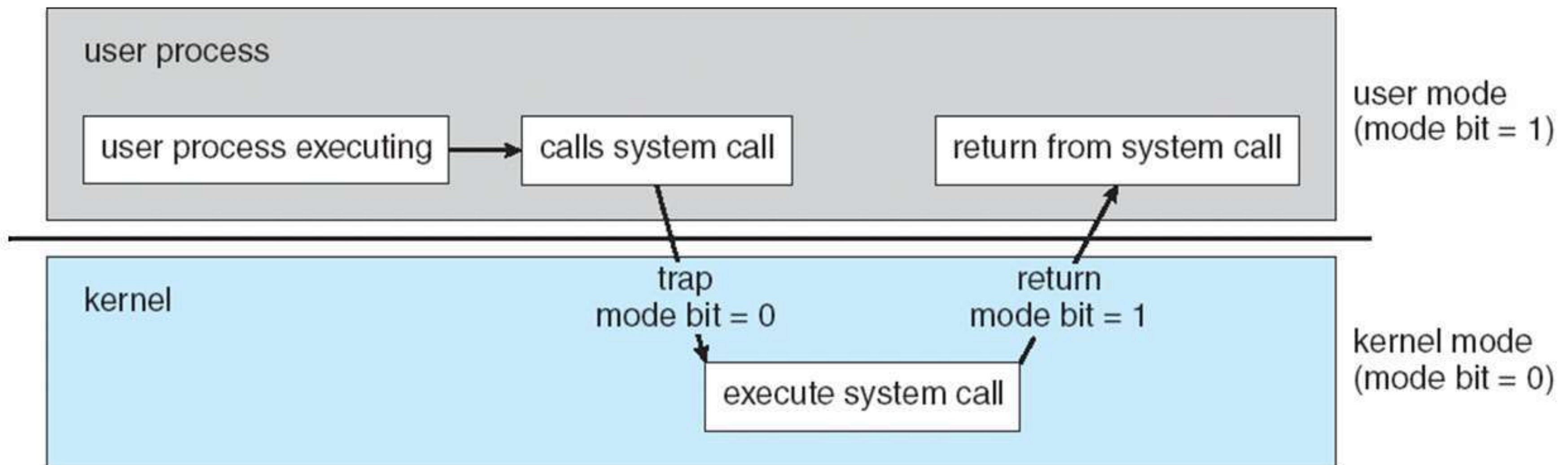
return

System Call Implementation

- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

OS Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware

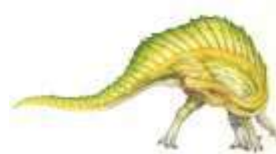


System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call

System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - pass the parameters in **registers**: Simplest
    - 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



Parameter Passing via Table

X

register

X: parameters
for call

load address X
system call 13

user program

use parameters
from table X

code for
system
call 13

operating system

Types of System Calls

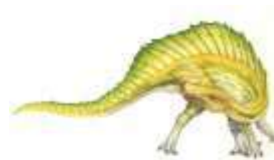
- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error

Types of System Calls

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

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes



Types of System Calls (Cont.)

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

Types of System Calls (Cont.)

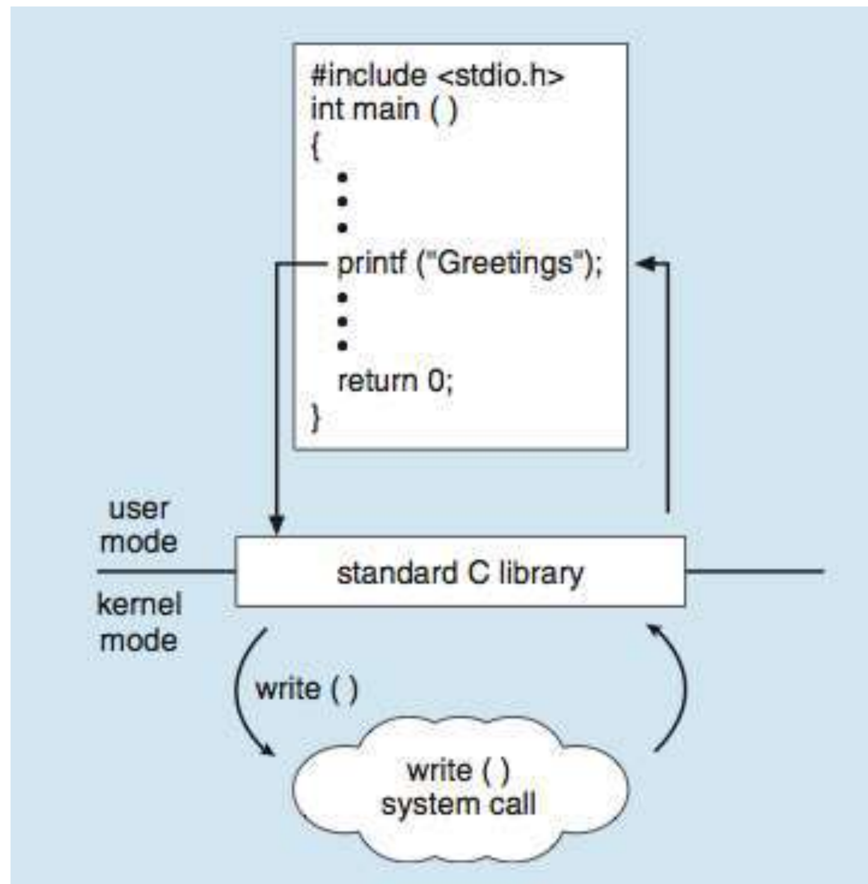
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

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

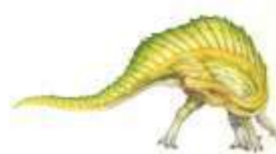


System Programs

- System programs provide a convenient environment for program development and execution. 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

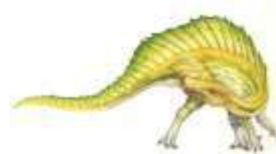
System Programs

- **File management** - Create, delete, copy, rename, print, dump, list, and manipulate files & directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Some systems implement a **registry** - used to store and retrieve configuration information



System Programs (Cont.)

- **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



System Programs (Cont.)

- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another’s screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Programs (Cont.)

- **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

- Design Goals
  - Main Problem to design an OS is to define goals
  - Choice of hardware
  - Type of system: Batch, Single system, multiuser etc
  - Hard to specify the requirements
- Specifying and designing an OS is highly creative task of **software engineering**

Operating System Design and Implementation

- Start the design by defining goals and spec's
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate

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

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility

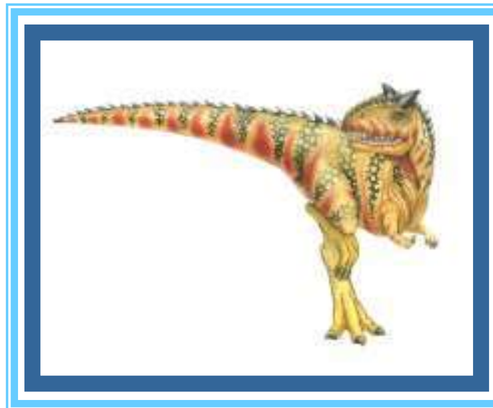
Implementation

- Much variation
  - Early OSes in assembly language
  - Then system pgming lang's like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

Implementation

- More high-level language
  - easier to **port** to other hardware
  - Written faster, compact and easy to understand
  - Changes can be done easily with recompilation
  - But slower and need more storage
- **Emulation** can allow an OS to run on non-native hardware

Operating System Structure

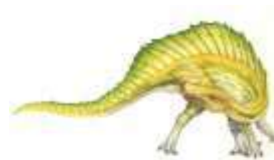


Operating System Structure

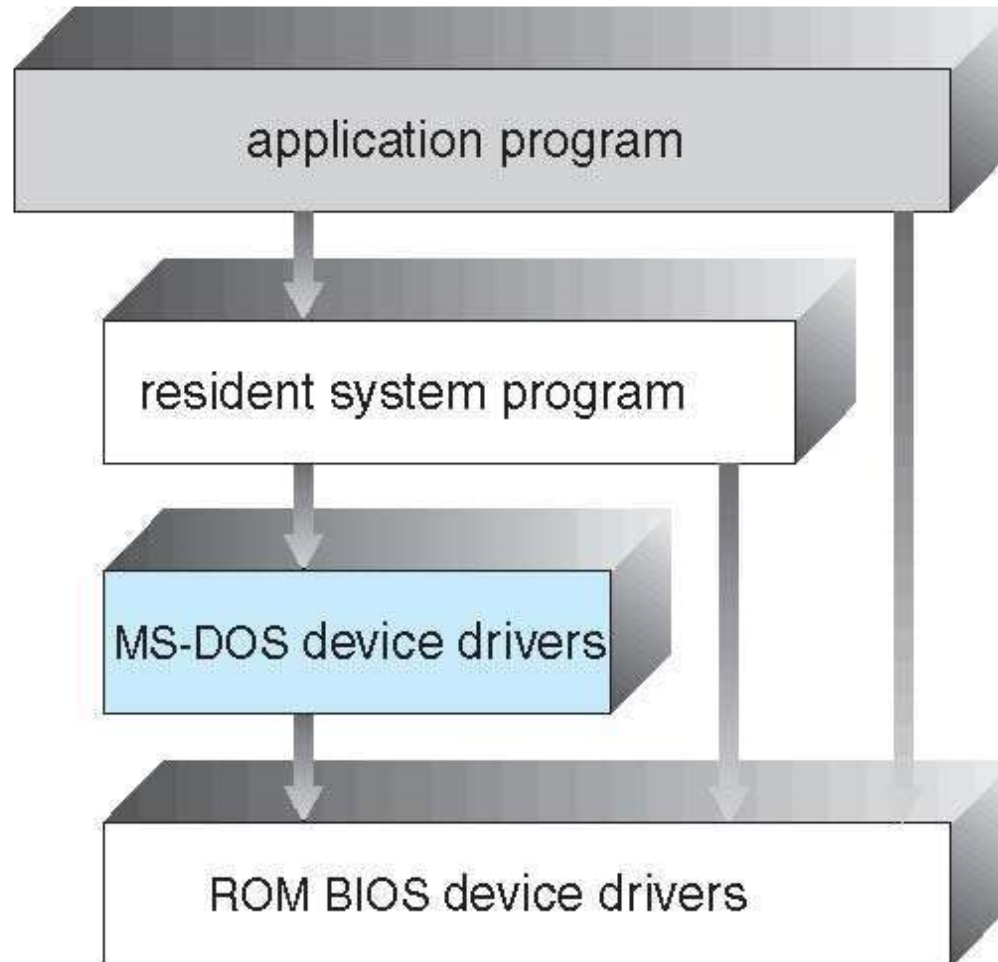
- General-purpose OS is very large and complex
- It should be engineered carefully so that
  - it function properly and
  - Can be modified easily

Operating System Structure

- Various ways to structure ones
  1. Simple structure – MS-DOS
  2. Monolithic structure – UNIX
  3. Layered – an abstraction
  4. Microkernel – Mach
  5. Modules
  6. Hybrid systems



Simple Structure -- MS-DOS



Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Vulnerable to malicious programs

Traditional UNIX System Structure

Beyond simple but not fully layered

(the users)

shells and commands
compilers and interpreters
system libraries

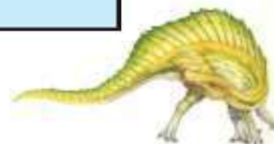
Kernel:

system-call interface to the kernel

signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory

kernel interface to the hardware

terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory



Monolithic Structure -- UNIX

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

Monolithic Structure -- UNIX

- Very difficult to implement and maintain
- It has a distinct performance advantage:
  - there is very little overhead in the system call interface or in communication within the kernel

Layered Approach

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

Layered Approach

layer N
user interface

layer 1

layer 0
hardware

Layered Approach

- Benefits:
  - Simplicity in construction and debugging
  - Protection to data structures in each layer
- Demerits:
  - How to define the layers
  - Less efficient : each layer put an overhead during system call

Microkernel System Structure

Application Program

File System

Device Driver

user mode

messages

messages

Interprocess Communication

memory managment

CPU scheduling

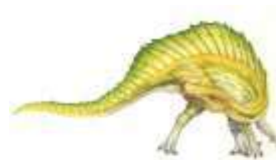
kernel mode

microkernel

hardware

Microkernel System Structure

- Moves as much from the kernel into user space
- Communication takes place between user modules using **message passing**



Microkernel System Structure

- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode, no crashing problem)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Solaris Modular Approach

Modules

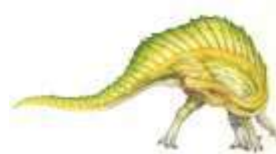
- Many modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Each module can req service from any other module directly
- Use advantages of Layered model and micro kernel
  - Linux, Solaris, etc

Hybrid Systems

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

Mac OS X Structure

graphical user interface	Aqua	
application environments and services	Java, Cocoa, Quicktime, BSD	
kernel environment	Mach	BSD
I/O kit		kernel extensions



iOS

- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel

Cocoa Touch
Media Services
Core Services
Core OS

Android

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

Android Architecture

Application Framework	
Libraries: SQLite, openGL, surface manager, media framework, webkit, libc	**Android runtime**: Core Libraries, Dalvik virtual machine

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory

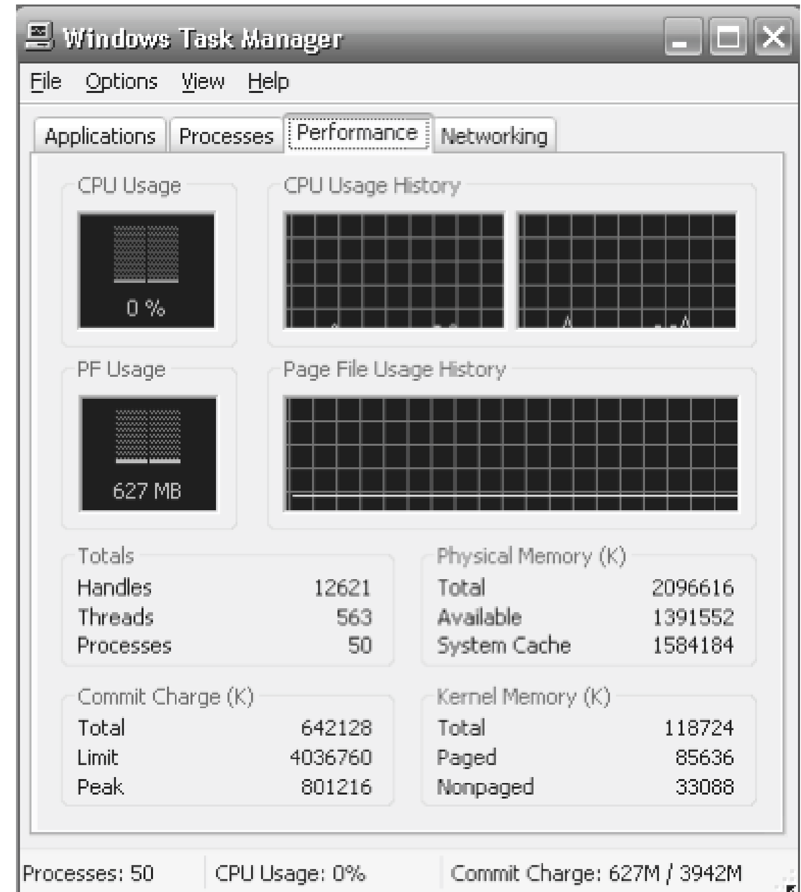
Operating-System Debugging

- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **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."

Performance Tuning

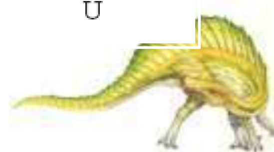
- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                              U
  0       -> ioctl                            K
  0         -> getf                           K
  0           -> set_active_fd                K
  0           <- set_active_fd                K
  0         <- getf                           K
  0         -> get_udatamodel                 K
  0         <- get_udatamodel                 K
...
  0         -> releasef                       K
  0           -> clear_active_fd              K
  0           <- clear_active_fd              K
  0           -> cv_broadcast                 K
  0           <- cv_broadcast                 K
  0         <- releasef                       K
  0       <- ioctl                            K
  0     <- ioctl                              U
  0   <- _XEventsQueued                       U
  0 <- XEventsQueued                          U
```



Dtrace (Cont.)

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

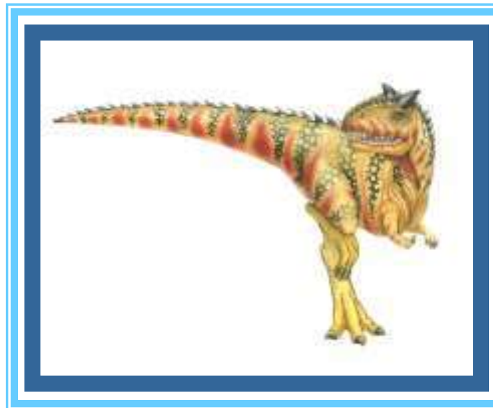
```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
   gnome-settings-d        142354
   gnome-vfs-daemon        158243
   dsdm                    189804
   wnck-applet             200030
   gnome-panel             277864
   clock-applet            374916
   mapping-daemon          385475
   xscreensaver            514177
   metacity                539281
   Xorg                   2579646
   gnome-terminal         5007269
   mixer_applet2          7388447
   java                  10769137
```

Figure 2.21 Output of the D code.

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

System Boot



System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- The procedure of starting a computer by loading the kernel is called **Booting** the system
  - 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 **bootstrap loader** fetches more complex **boot program** from disk, which loads kernel

System Boot

- Bootstrap program has a variety of tasks:
  - run diagnostics
  - initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory
  - Then it starts the operating system

System Boot

Loads bootstrap address → Run bootstrap → System Diagnostics (POST) → Initialize System → Load Boot Block (Block 0) into memory → Run Boot Block (loads OS) → OS starts

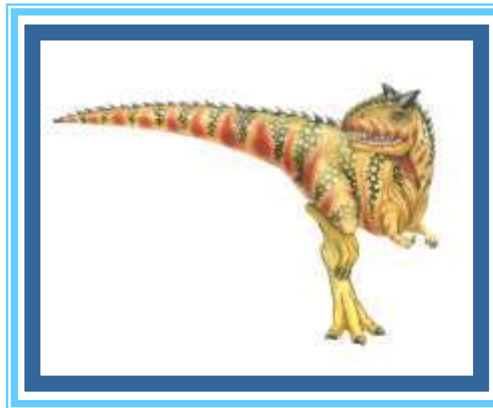
System Boot

- Large OS, bootstrap loader in firmware and OS on disk
  1. Bootstrap runs diagnostics
  2. Read a single block at a fixed location (usually zero) from disk called **boot block**
  3. Executing boot block loads the entire OS into memory
  4. Then OS begins execution (It is only at this point the system is said to be **Running**)
- Common bootstrap loader for LINUX, **GRUB**,
  - allows selection of kernel from multiple disks, versions, kernel options

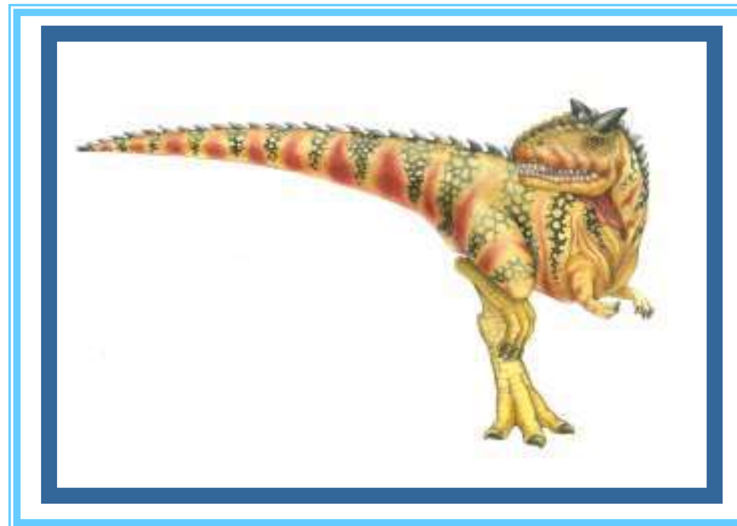
System Boot

- Small Operating System can be kept :
  - ROM (Cell phones etc)
  - EEPROM
  - Firmware
    - Can be copied to RAM during execution, coz it is slow

End of Module 1



Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept

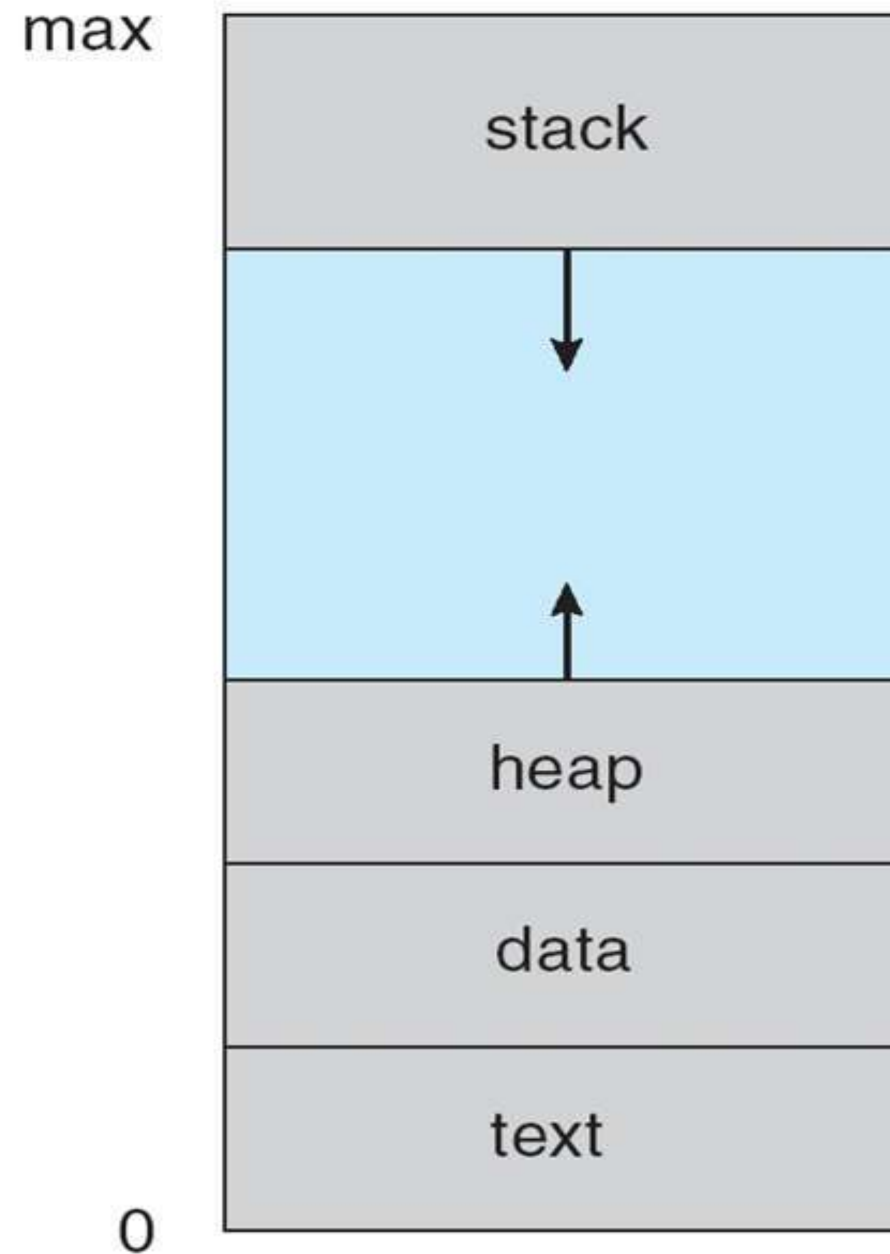
- Textbook uses the terms ‘*job*’ and ‘*process*’ almost interchangeably
- **Process – a program in execution**; process execution must progress in sequential fashion

- Program is **passive** entity, process is **active**
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

The Process

- 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

Process in Memory



Process State

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

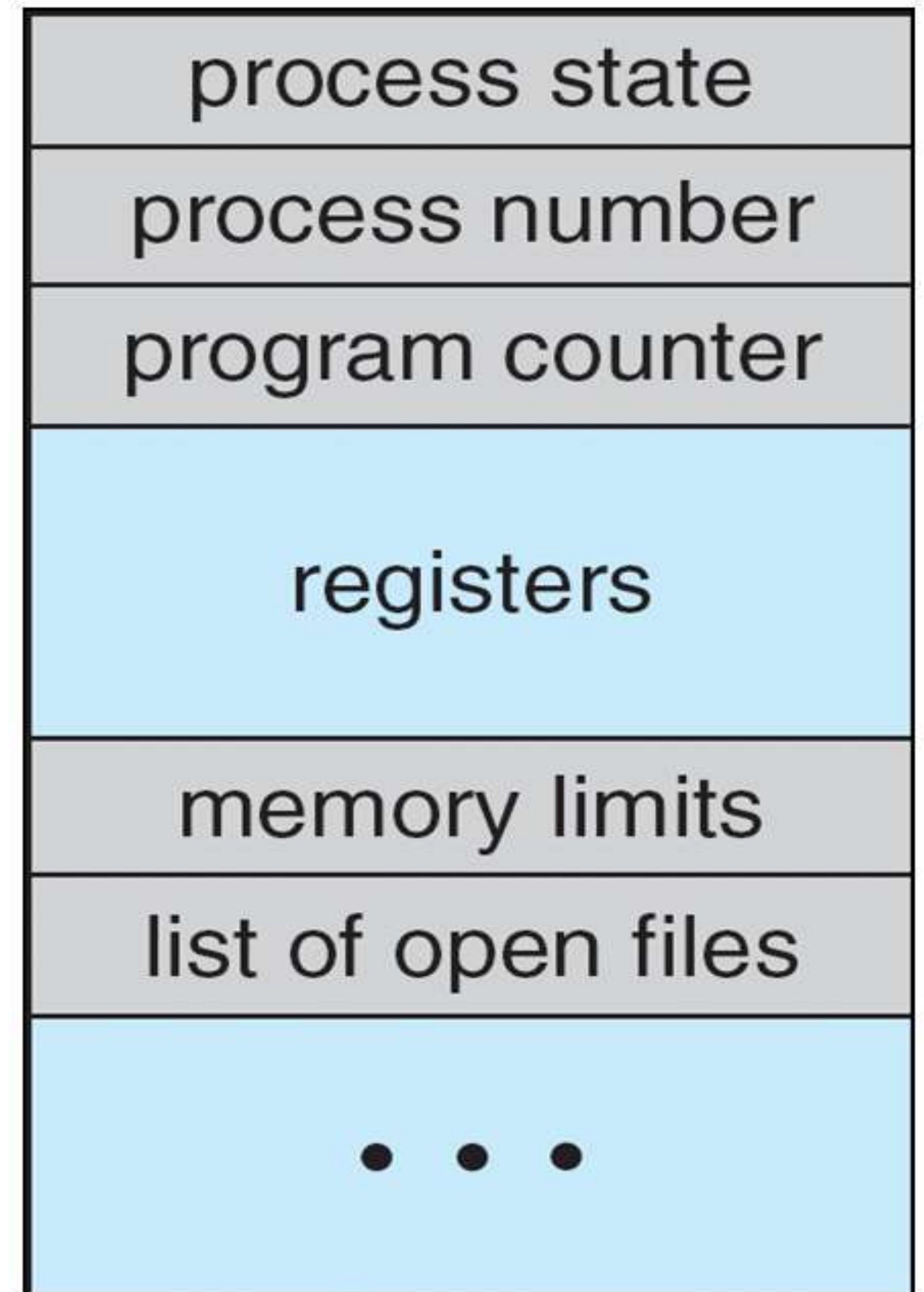
Diagram of Process State

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



CPU Switch From Process to Process

process P_0 | operating system | process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

interrupt or system call

executing

save state into PCB_1

idle

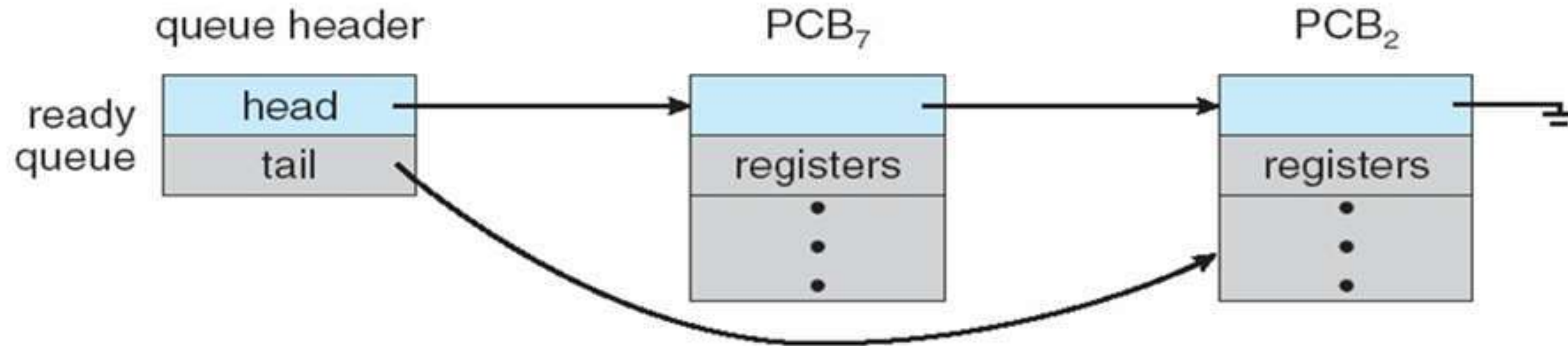
reload state from PCB_0

executing

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue



Ready Queue And Various I/O Device Queues

queue header | PCB_7 | PCB_2

ready queue: head, tail → PCB_7 (registers …) → PCB_2 (registers …)

mag tape unit 0: head, tail

mag tape unit 1: head, tail

PCB_3 | PCB_{14} | PCB_6

disk unit 0: head, tail → PCB_3 → PCB_{14} → PCB_6

PCB_5

terminal unit 0: head, tail → PCB_5

…

Representation of Process Scheduling

ready queue

CPU

I/O

I/O queue

I/O request

time slice expired

child executes

fork a child

interrupt occurs

wait for an interrupt

Queueing – Diagram

Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the memory
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

Schedulers (Cont.)

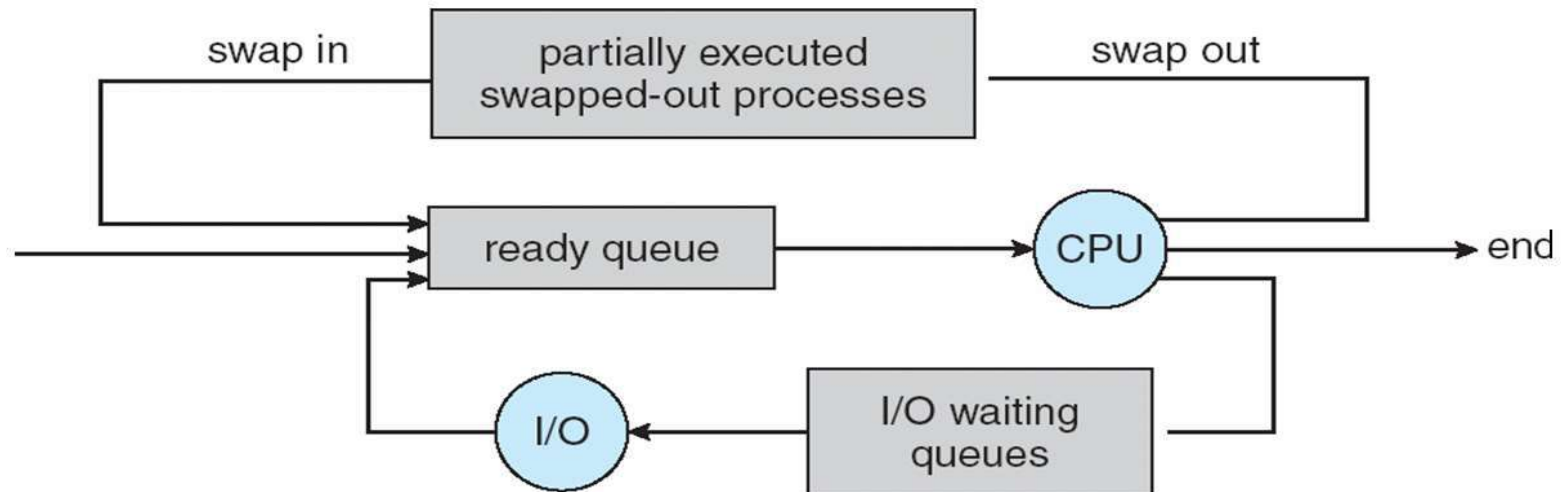
- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the ***degree of multiprogramming*** (the number of processes in memory)

Schedulers (Cont.)

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler select a good process mix of I/O-bound process and CPU-bound process

Image Name	PID	User Name	CPU	Memory (Private Working Set)	Page Faults	I/O Reads	I/O Writes	Description
csrss.exe	416		00	1,036 K	28,143	124,932	0	
dwm.exe	2228	dell	01	7,528 K	188,865	1	0	Desktop Window Manager
explorer.exe	2268	dell	00	16,652 K	74,467	16,641	52	Windows Explorer
GrooveMonito...	2840	dell	00	1,056 K	3,571	42	0	GrooveMonitor Utility
imageCreator...	2468	dell	00	6,936 K	168,630	16,276	18,998	Image Creator
jusched.exe	3088	dell	00	236 K	1,202	0	3	Java Update Scheduler
NMBgMonitor....	3192	dell	00	688 K	4,424	0	0	Nero Home
NMIndexStor...	4008	dell	00	2,908 K	7,597	98	429	Nero Home
ONLINENT.EXE	3072	dell	00	5,076 K	48,292	43,008	731	Online Protection
sttray.exe	2864	dell	00	488 K	4,062	1	0	Sigmatel Audio system tray ...
taskhost.exe	2100	dell	00	900 K	3,315	4	2	Host Process for Windows T...
taskmgr.exe	4084	dell	01	2,704 K	2,923	2	0	Windows Task Manager
WebcamDell2....	2888	dell	00	856 K	3,124	5	0	WebcamDell2.exe
winlogon.exe	496		00	672 K	4,717	9	0	

Addition of Medium-Term Scheduling



Context Switch

process P_0 | operating system | process P_1

interrupt or system call

executing

save state into PCB_0

•
•
•

reload state from PCB_1

idle

idle

executing

interrupt or system call

save state into PCB_1

•
•
•

reload state from PCB_0

idle

executing

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 overhead; the system does no useful work while switching
- Time dependent on hardware support

Operations on Processes

Operations on Processes

- Process Creation
- Process Termination

Process Creation

- A process may create several new process using create-process system call
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier (pid)**

A Tree of Processes on Solaris

- Sched pid = 0
  - init pid = 1
    - inetd pid = 140
      - telnetdaemon pid = 7776
        - Csh pid = 7778
          - Netscape pid = 7785
          - emacs pid = 8105
    - dtlogin pid = 251
      - Xsession pid = 294
        - sdt_shel pid = 340
          - Csh pid = 1400
            - ls pid = 2123
            - cat pid = 2536
  - pageout pid = 2
  - fsflush pid = 3

Process Creation (Cont.)

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

Process Creation (Cont.)

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

C Program Forking Separate Process

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t  pid;
    pid = fork();                       /* fork another process */
    if (pid < 0) {                      /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {                /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else {                              /* parent process */
        wait (NULL);                    /* parent will wait for the child */
        printf ("Child Complete");
    }
    return 0;
}
```

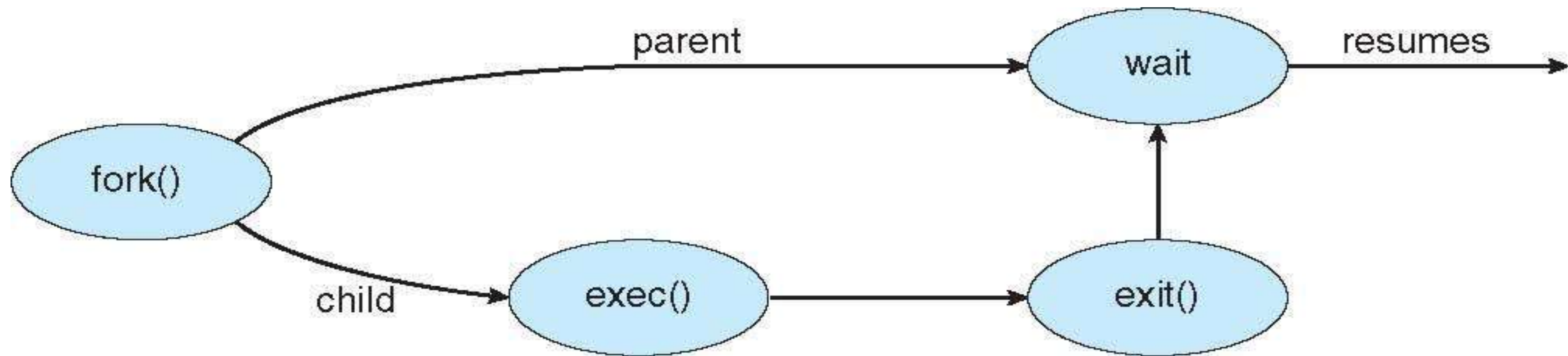
Parent Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

Child Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

Process Creation (Cont.)



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system

Process Termination (Cont.)

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**