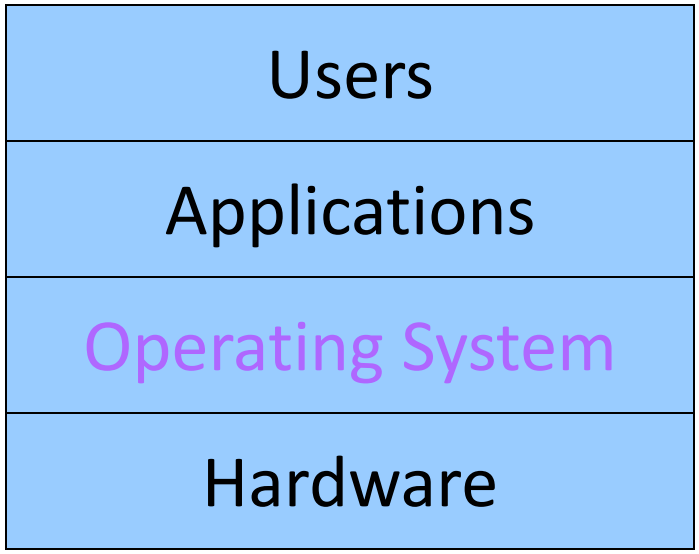
Introduction and Overview

Questions answered in this lecture:

- What is an operating system?
- What is the role of an OS?
- What is the functionality of an OS?
- How have operating systems evolved?
- Why study operating systems?

What is an Operating System?

Not easy to define precisely…



compilers
databases
word processors

CPU
memory
I/O devices

OS:

Everything in system that isn't an application or hardware

OS:

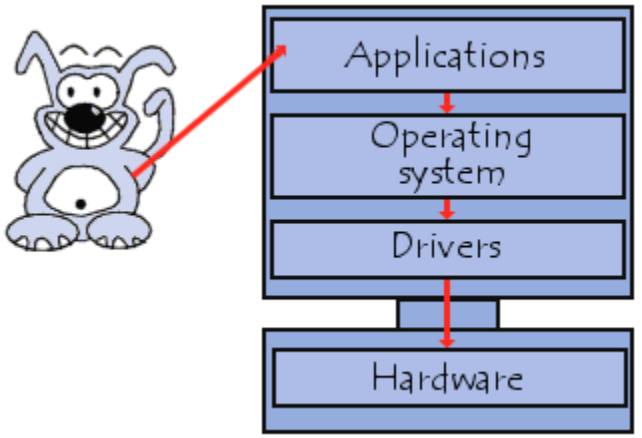
Software that converts hardware into a useful form for applications

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.

OS – Good Definition!

- The **operating system** (sometimes referred to by its abbreviation *OS*), is responsible for creating the link between the material resources, the user and the applications (word processor, video game, etc.). When a programme wants to access a material resource, it does not need to send specific information to the peripheral device but it simply sends the information to the operating system, which conveys it to the relevant peripheral via its driver. If there are no drivers, each programme has to recognize and take into account the communication with each type of peripheral!



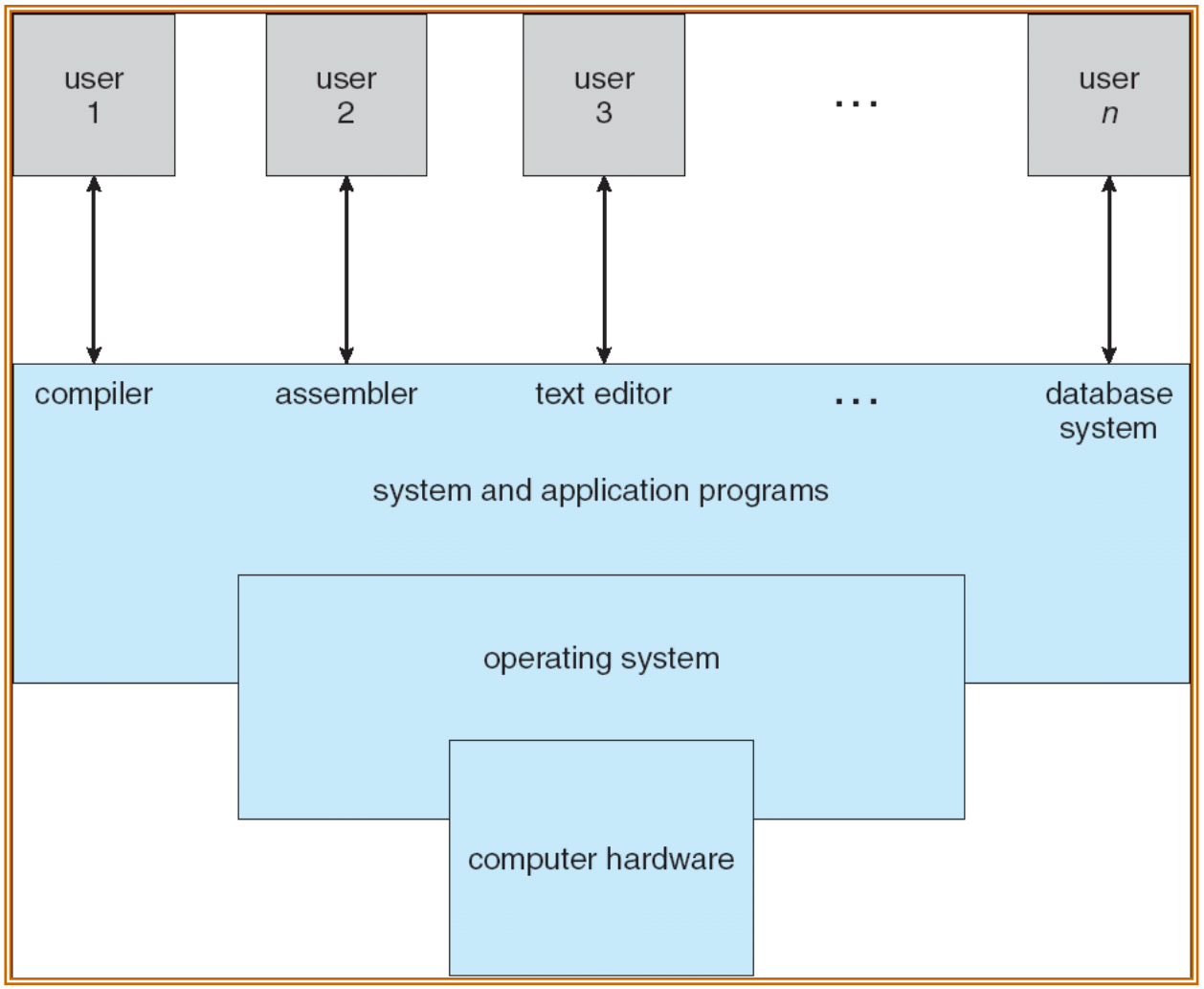
Operating System Definition

- 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

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

Four Components of a Computer System



What is the role of the OS?

Role #1: Provide standard Library (I.e., abstract resources)

What is a resource?

- Anything valuable (e.g., CPU, memory, disk)

Advantages of standard library

- Allow applications to reuse common facilities
- Make different devices look the same
- Provide higher-level abstractions

Challenges

- What are the correct abstractions?
- How much of hardware should be exposed?

What is the role of the OS?

Role #2: Resource coordinator (I.e., manager)

Advantages of resource coordinator

- Virtualize resources so multiple users or applications can share
- Protect applications from one another
- Provide efficient and fair access to resources

Challenges

- What are the correct mechanisms?
- What are the correct policies?

Operating System Services/ Functionality

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - 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 - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont.)

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

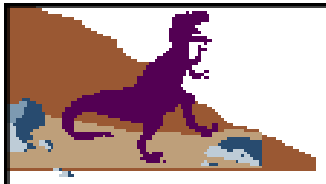
Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles,mainmemory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **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
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

History of the OS

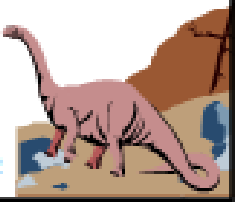
Two distinct phases of history

- Phase 1: Computers are expensive
  - Goal: Use computer's time efficiently
  - Maximize throughput (I.e., jobs per second)
  - Maximize utilization (I.e., percentage busy)
- Phase 2: Computers are inexpensive
  - Goal: Use people's time efficiently
  - Minimize response time

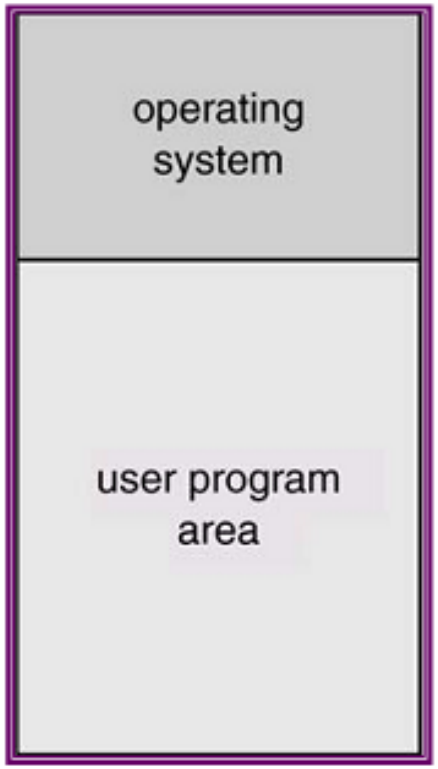


Simple Batch Systems

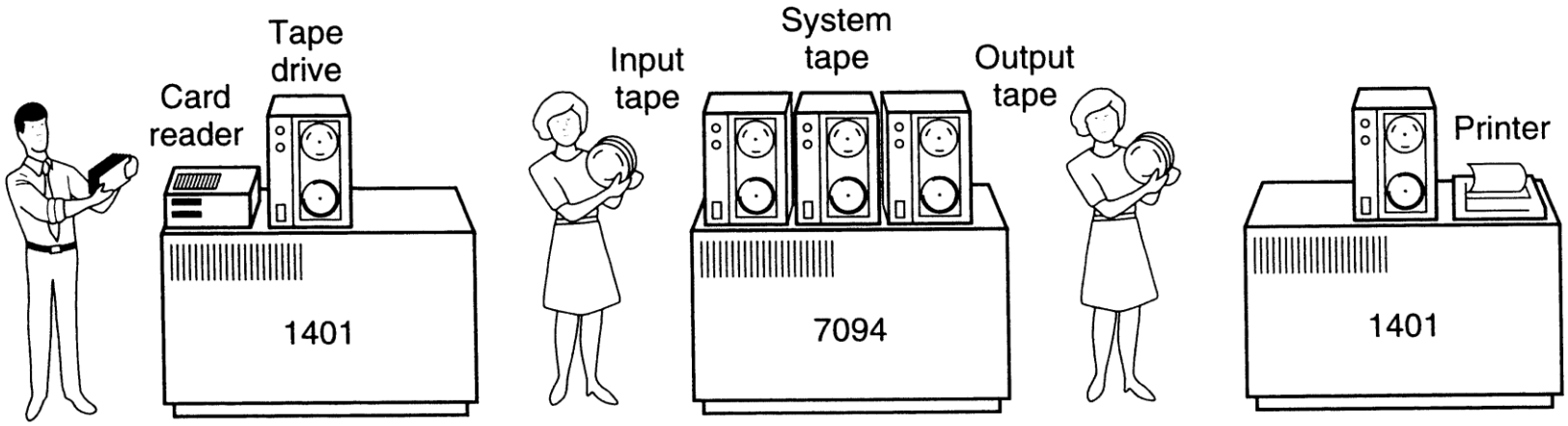
- User prepare a job and submit it to a computer operator
- User get output some time later
- No interaction between the user and the computer system
- Operator batches together jobs with similar needs to speedup processing
- Task of OS: automatically transfers control from one job to another.
- OS always resident in memory
- Disadvantages of one job at a time:
  - CPU idle during I/O
  - I/O devices idle when CPU busy



Memory Layout for a Simple Batch System



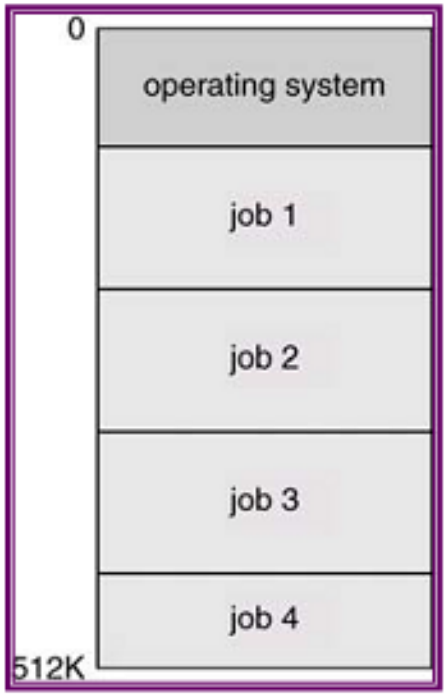
Batch Processing



Multiprogrammed Batch Systems

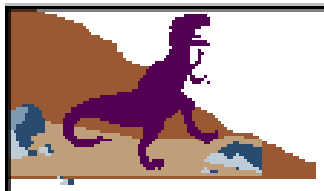
- Keep more than one job in memory simultaneously
- When a job performs I/O, OS switches to another job
  - Increase CPU utilization
- All jobs enter the system kept in the job pool on a disk, scheduler brings jobs from pool into memory

Multiprogrammed Batch Systems



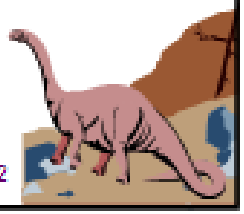
OS Features Needed for Multiprogramming

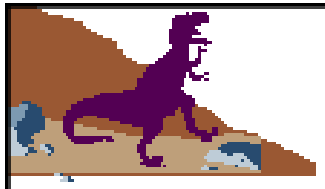
- Job scheduling: which jobs in the job pool should be brought into memory? (Ch6)
- Memory management: the system must allocate the memory to several jobs. (Ch 9, Ch 10)
- CPU scheduling: choose among jobs in memory that are ready to run. (Ch6)
- Allocation of devices: what if more than one job wants to use a device?
- Multiple jobs running concurrently should not affect one another



Time-Sharing Systems

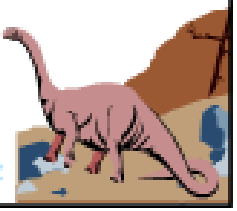
- Like multiprogrammed batch, except CPU switches between jobs occur so frequently that the users can interact with a running program
- User give instructions to the OS or a program, and wait for immediate results
  - Require low response time
- Allow many users to share the computer simultaneously, users have the impression that they have their own machine
- CPU is multiplexed among several jobs that are kept in memory and on disk





OS Features Needed for Time-Sharing

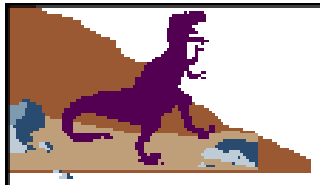
- Job synchronization and communication (Ch7)
- Deadlock handling (Ch8)
- File system (Ch11, Ch12)



Desktop Systems

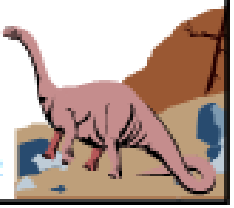
- Personal computers – computer system dedicated to a single user.
- CPU utilization not a prime concern, want maximize user convenience and responsiveness.
- Can adopt technologies developed for mainframe operating systems: virtual memory, file systems, multiprogramming
- File protection needed due to interconnections of computers
- Operating systems for PCs: Windows, Mac OS, Linux

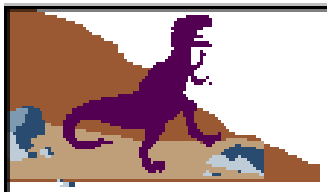
- **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**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



Multiprocessor Systems

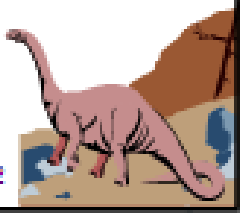
- Also known as parallel systems or tightly coupled systems
- More than one processor in close communication, sharing computer bus, clock, memory, and usually peripheral devices
  - Communication usually takes place through the shared memory.
- Advantages
  - Increased throughput: speed-up ratio with N processors < N
  - Economy of scale: cheaper than multiple single-processor systems
  - Increased reliability: graceful degradation, fault tolerant





Multiprocessor Systems

- Symmetric multiprocessing (SMP)
  - Each processor runs an identical copy of the operating system.
  - All processors are peers: any processor can work on any task
  - OS can distribute load evenly over the processors.
  - Most modern operating systems support SMP
- Asymmetric multiprocessing
  - Master-slave relationship: a master processor controls the system, assigns works to other processors
  - Each processor is assigned a specific task
    - Don't have the flexibility to assign processes to the least-loaded CPU
  - More common in extremely large systems



Clustered Systems

Clustered systems are typically constructed by combining multiple computers into a single system to perform a computational task distributed across the cluster.

Multiprocessor systems on the other hand could be a single physical entity comprising of multiple CPUs.

A clustered system is less tightly coupled than a multiprocessor system.

Clustered systems communicate using messages, while processors in a multiprocessor system could communicate using shared memory.

In order for two machines to provide a highly available service, the state on the two machines should be replicated and should be consistently updated.

When one of the machines fail, the other could then take-over the functionality of the failed machine.

Evolution of OS :

Major Phases	Technical Innovations	Operating Systems
Open Shop	**The idea of OS**	**IBM 701 open shop (1954)**
Batch Processing	**Tape batching,** **First-in, first-out scheduling.**	**BKS system (1961)**
Multi-programming	**Processor multiplexing,** **Indivisible operations,** **Demand paging,** **Input/output spooling,** **Priority scheduling,** **Remote job entry**	**Atlas supervisor (1961),** **Exec II system (1966)**

(Contd...)

Evolution of OS (contd..):

Timesharing	**Simultaneous user interaction,** **On-line file systems**	**Multics file system (1965),** **Unix (1974)**
Concurrent Programming	**Hierarchical systems, Extensible kernels, Parallel programming concepts, Secure parallel languages**	**RC 4000 system (1969),** **13 Venus system (1972),** **14 Boss 2 system (1975).**
Personal Computing	**Graphic user interfaces**	**OS 6 (1972)** **Pilot system (1980)**
Distributed Systems	**Remote servers**	**WFS file server (1979) Unix United RPC (1982)** **24 Amoeba system (1990)**

KINDS OF OS & PROPERTIES

Properties of the following types of operating systems:

- a. Batch
- b. Interactive
- c. Time sharing
- d. Real time
- e. Network
- f. Parallel
- g. Distributed
- h. Clustered
- i. Handheld

KINDS OF OS & PROPERTIES

- a. **Batch**. Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off-line operation, spooling, and multiprogramming. Batch is good for executing large jobs that need little interaction; it can be submitted and picked up later.

- b. **Interactive**. This system is composed of many short transactions where the results of the next transaction may be unpredictable. Response time needs to be short (seconds) since the user submits and waits for the result.

- c. **Time sharing**. This systems uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another. Instead of having a job defined by spooled card images, each program reads its next control card from the terminal, and output is normally printed immediately to the screen.

KINDS OF OS & PROPERTIES

- d. **Real time**. Often used in a dedicated application, this system reads information from sensors and must respond within a fixed amount of time to ensure correct performance.

- e. **Network**. Provides operating system features across a network such as file sharing.

- f. **SMP**. Used in systems where there are multiple CPU's each running the same copy of the operating system. Communication takes place across the system bus.

- g. **Distributed**. This system distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines, such as a high-speed bus or local area network.
- h. **Clustered**. A clustered system combines multiple computers into a single system to perform computational task distributed across the cluster.

- i. **Handheld**. A small computer system that performs simple tasks such as calendars, email, and web browsing. Handheld systems differ from traditional desktop systems with smaller memory and display screens and slower processors.

Why study Operating Systems?

Build, modify, or administer an operating system

Understand system performance

- Behavior of OS impacts entire machine
- Challenge to understand large, complex system
- Tune workload performance
- Apply knowledge across many areas
  - Computer architecture, programming languages, data structures and algorithms, and performance modeling

Reading Assignment:

1. Under what circumstances would a user be better off using a timesharing system rather than a PC or single-user workstation?

2. Which of the functionalities listed below need to be supported by the operating system for the following two settings: (a) handheld devices and (b) real-time systems.
a. Batch programming
b. Virtual memory
c. Time sharing

3. How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.

UNIT - I

Design Issues of an OS

The Various Design Issues

- Efficiency
- Robustness
- Flexibility
- Portability
- Security
- Compatibility

The Various Design Issues

- **Efficiency**

  Operating system efficiency is characterized by the amount of useful work accomplished by system compared to the time and resources used.

  The ratio of actual operating time to scheduled operating time of a computer system. In time-sharing system, the ratio of user time to the sum of user time plus system time.

- **Robustness**

  The word *robust*, when used with regard to computer software, refers to an operating system or other program that performs well not only under ordinary conditions but also under unusual conditions that stress its designers' assumptions.

  A major feature of Unix-like operating systems is their robustness. That is, they can operate for prolonged periods (sometimes years) without *crashing* (i.e., stopping operating) or requiring *rebooting* (i.e., restarting). And although individual application programs sometimes crash, they almost always do so without affecting other programs or the operating system itself.

The Various Design Issues

- **Flexibility**

  The ease with which a system or component can be modified for use in application or environment other than those for which it was designed.

  Eg. The RAM Capacity of a desktop computer can be expanded only if the hardware and OS were specifically designed to accommodate it.

- **Portability**

  Portability is the ability to run a program on different platforms.

  Portability of the OS means the operating system itself can be run easily (after re-compilation) on a different hardware platform

- **Compatability**

  A family of computer models is said to be **compatible** if certain software that runs on one of the models can also be run on all other models of the family.

  The main Compatibility goal for Windows 7 is to make sure that most all applications which work on Windows Vista will continue to work seamlessly on Windows 7

The Various Design Issues

- **Security**

  One use of the term computer security refers to technology to implement a secure operating system.

  Microsoft operating system updates are releases of new software that upgrade, add features to, and protect Microsoft Windows, the software that runs your computer.

Design Issues of Distributed System

- **Transparency** – the distributed system should appear as a conventional, centralized system to the user
- **Fault tolerance** – the distributed system should continue to function in the face of failure
- **Scalability** – as demands increase, the system should easily accept the addition of new resources to accommodate the increased demand
- **Clusters** – a collection of semi-autonomous machines that acts as a single system

Robustness

- Failure detection
- Reconfiguration

Failure Detection

- Detecting hardware failure is difficult
- To detect a link failure, a handshaking protocol can be used
- Assume Site A and Site B have established a link
  - At fixed intervals, each site will exchange an *I-am-up* message indicating that they are up and running
- If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost
- Site A can now send an *Are-you-up?* message to Site B
- If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B

Failure Detection (cont)

- If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred

- Types of failures:
  - Site B is down
  - The direct link between A and B is down
  - The alternate link from A to B is down
  - The message has been lost

- However, Site A cannot determine exactly **why** the failure has occurred

Reconfiguration

- When Site A determines a failure has occurred, it must reconfigure the system:

  1. If the link from A to B has failed, this must be broadcast to every site in the system

  2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available

- When the link or the site becomes available again, this information must again be broadcast to all other sites

UNIT I

OS STRUCTURE + API & DESIGN OF AN API + COMPONENTS

OS Abstraction

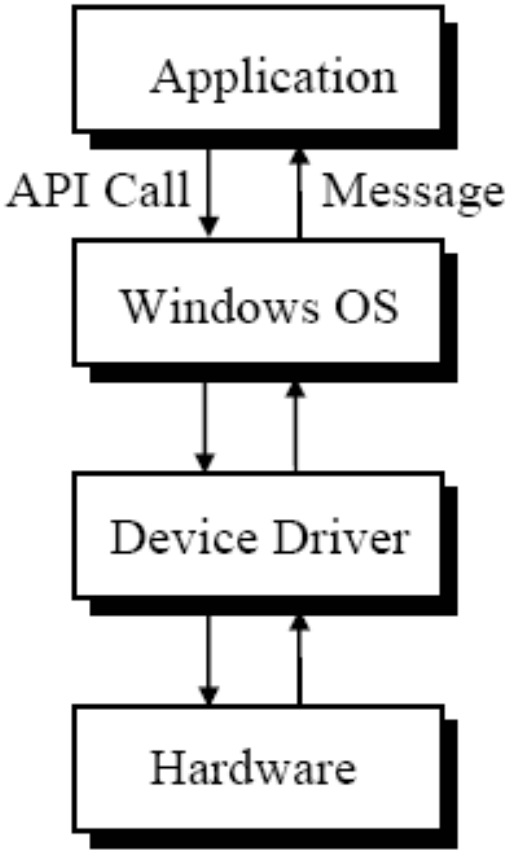
- The operating system provides a layer of abstraction between the user and the bare machine. Users and applications do not see the hardware directly, but view it through the operating system.
- This abstraction can be used to hide certain hardware details from users and applications. Thus, changes in the hardware are not seen by the user (even though the OS must accommodate them).
- This is particularly advantageous for venders that want offer a consistent OS interface across an entire line of hardware platforms.
- For example, certain operations such as interaction with 3D graphics hardware can be controlled by the operating system. When an instruction pertaining to the hardware is executed and if the hardware is present then all is fine. However, if the hardware is not present then a trap is generated by the illegal instruction. In this case the OS can emulate the desired instruction in software.
- Another way that abstraction can be used is to make related devices appear the same from the user point of view. For example, hard disks, floppy disks, CD-ROMs, and even tape are all very different media, but in many operating systems they appear the same to the user.
- Unix, and increasingly Windows NT, take this abstraction even further. From a user and application programmer standpoint, Unix is Unix regardless of the CPU make and model.

What is an API?

- API, an abbreviation of ***a****pplication* ***p****rogram* ***i****nterface*, is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

- Most operating environments, such as MS-Windows, provide an API so that programmers can write applications consistent with the operating environment. Although APIs are designed for programmers, they are ultimately good for users because they guarantee that all programs using a common API will have similar interfaces. This makes it easier for users to learn new programs.

- An API can be created for applications, libraries, operating systems, etc, as a way to define their "vocabularies" and resources request conventions (e.g. functions calling conventions). It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer program and the implementer program of the API


Application
API Call
Message
Windows OS
Device Driver
Hardware

What is an API?

- An API is an abstraction that describes an interface for the interaction with a set of functions used by components of a software system. The software providing the functions described by an API is said to be an *implementation* of the API.

An API can be:

- general, the full set of an API that is bundled in the libraries of a programming language, e.g. Standard Template Library in C++ or Java API.
- specific, meant to address a specific problem, e.g. Google Maps API or Java API for XML Web Services.
- language-dependent, meaning it is only available by using the syntax and elements of a particular language, which makes the API more convenient to use.
- language-independent, written so that it can be called from several programming languages.

WINDOWS API

- **Purpose**

  The Microsoft Windows application programming interface (API) provides services used by all Windows-based applications.

  You can provide your application with a graphical user interface; access system resources such as memory and devices; display graphics and formatted text; incorporate audio, video, networking, or security.

- **Where Applicable**

  The Windows API can be used in all Windows-based applications. The same functions are generally supported on 32-bit and 64-bit Windows.

- **Developer Audience**

  This API is designed for use by C/C++ programmers. Familiarity with the Windows graphical user interface and message-driven architecture is required.

Application Programming Interface (API)

API s are implement using 3 Libraries in windows.

- **KERNEL**
- **USER**
- **GDI**

Kernel

It is library named KERNEL32.DLL, which supports capabilities associated with OS such as

- **Process Loading.**
- **Context switching.**
- **File I/O.**
- **Memory Management.**

User

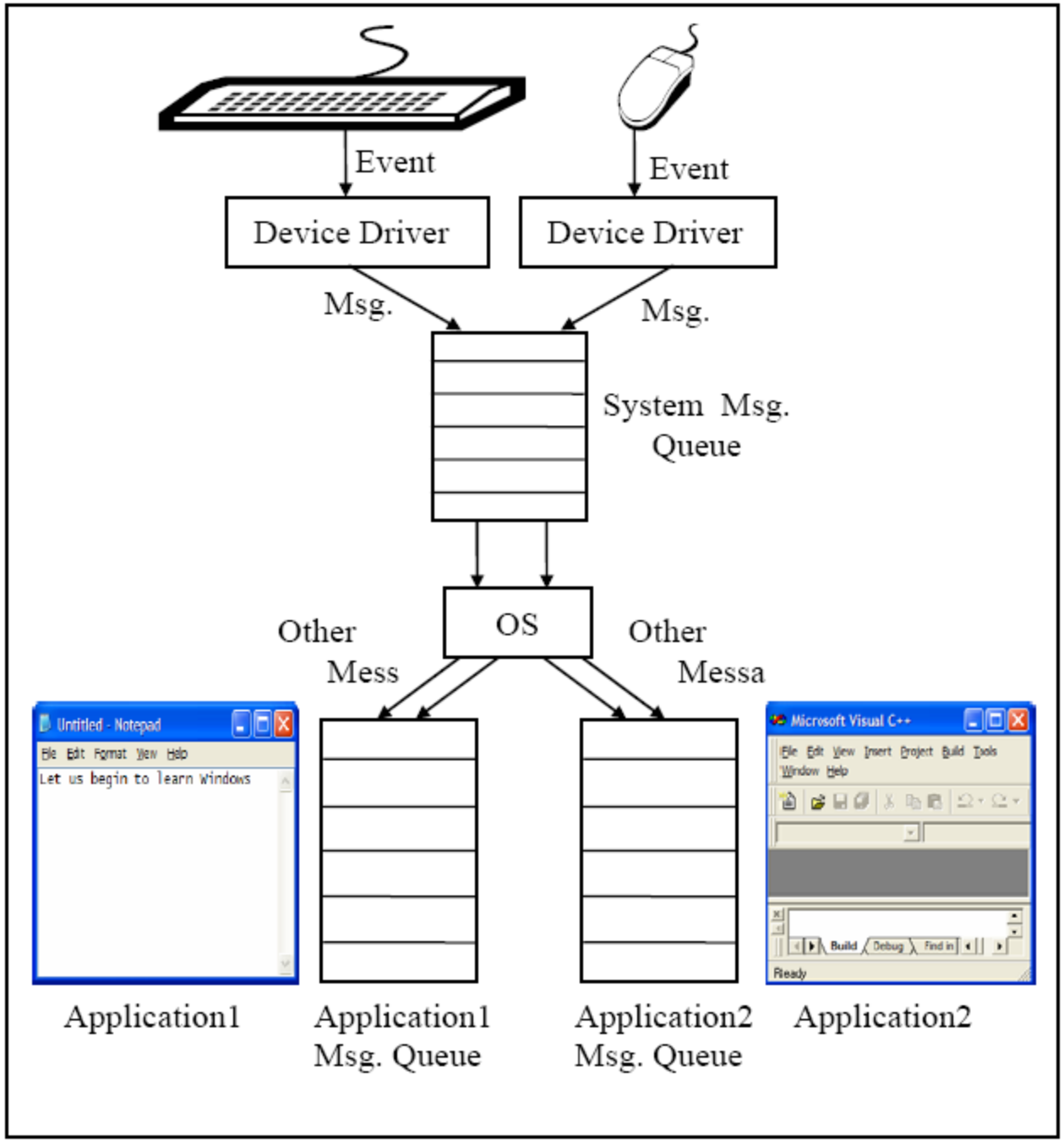
It is library named USER32.DLL, which allows managing the user interface such as

- **Windows.**
- **Menus.**
- **Dialog Boxes.**
- **Icon.**

GDI

It is library named GDI32.DLL, using GDI windows draws windows, menus and dialog boxes

- **It can create graphical output.**
- **Also use for storing Graphical Images.**


Event
Event
Device Driver
Device Driver
Msg.
Msg.
System Msg.
Queue
OS
Other
Mess
Other
Messa
Untitled - Notepad
File Edit Format View Help
Let us begin to learn Windows
Microsoft Visual C++
File Edit View Insert Project Build Tools
Window Help
Build
Debug
Find in
Ready
Application1
Application1
Msg. Queue
Application2
Msg. Queue
Application2

EXAMPLE WIN32 API

- **Application window is created by calling the API function CreateWindow().**

- **Create Window with the comments identifying the parameter.**

```
hwnd = CreateWindow
 (“classname”,                          // window class name
TEXT ("The First Program"),             // window caption
WS_OVERLAPPEDWINDOW,                    // window style
CW_USEDEFAULT,                          // initial x position
CW_USEDEFAULT,                          // initial y position
CW_USEDEFAULT,                          // initial x size
CW_USEDEFAULT,                          // initial y size
NULL,                                   // parent window handle
NULL,                                   // window menu handle
hInstance,                              // program instance handle
NULL) ;    // creation parameters, may be used to point some data for reference.
```

- **Overlapped window will be created, it includes a title bar, system menu to the left of title bar, a thick window sizing border, minimize, maximize and close button to the right of the title bar.**

- **The window will be placed in default x, y position with default size. It is a top level window without any menu.**
- **The CreateWindow() will returns a handle which is stored in hwnd.**

Why is API Design Important?

- APIs can be among a company's greatest assets
  - Customers invest heavily: buying, writing, **learning**
  - Cost to stop using an API can be prohibitive
  - Successful public APIs capture customers
- Can also be among company's greatest liabilities
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward
- Public APIs are forever - one chance to get it right

Why is API Design Important *to You*?

- If you program, you are an API designer
  - Good code is modular–each module has an API
- Useful modules tend to get reused
  - Once module has users, can't change API at will
  - Good reusable modules are corporate assets
- Thinking in terms of APIs improves code quality

Characteristics of a Good API

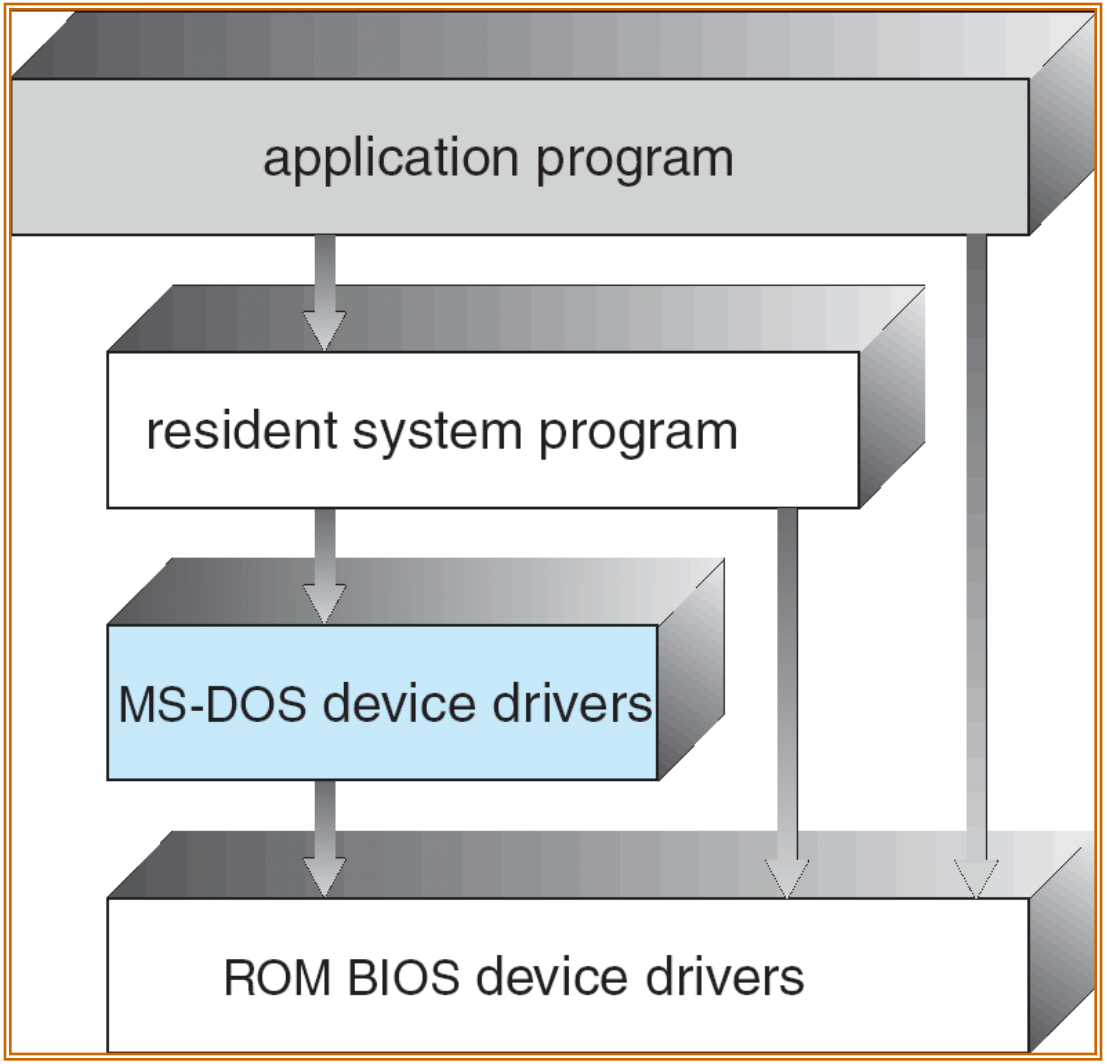
- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Operating system structures

Simple Structure

- 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

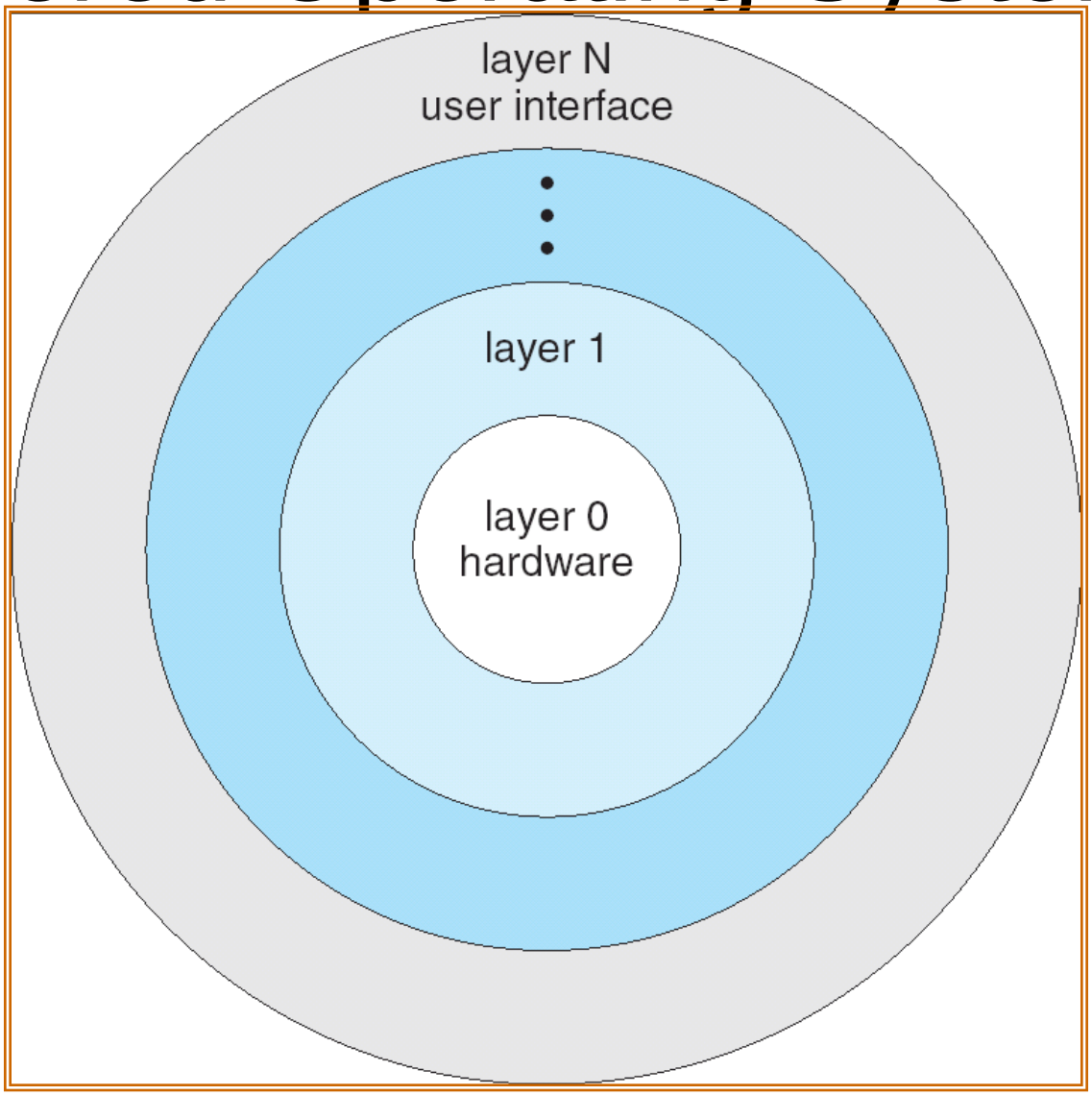
MS-DOS Layer Structure



Layered Approach

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

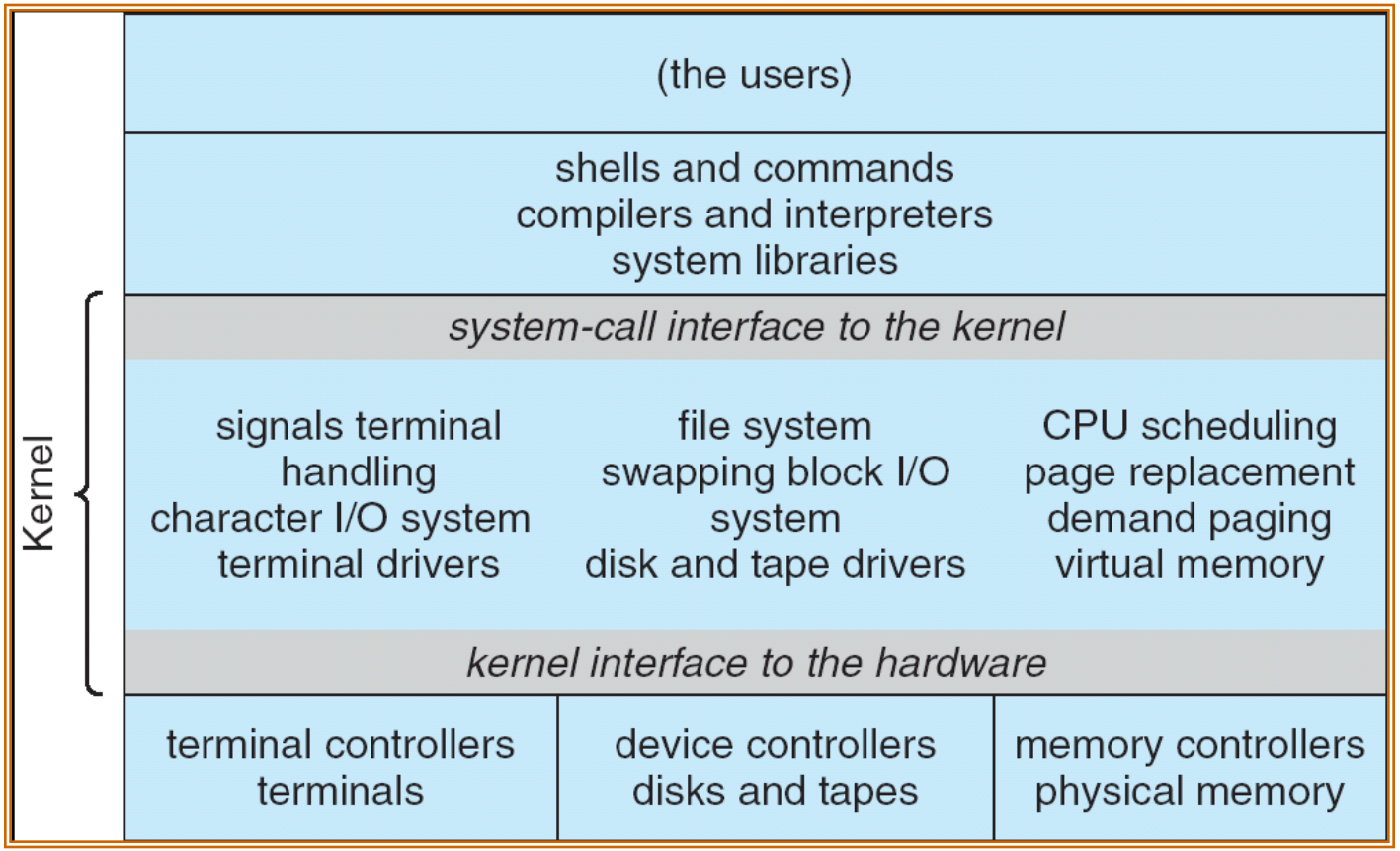
Layered Operating System



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

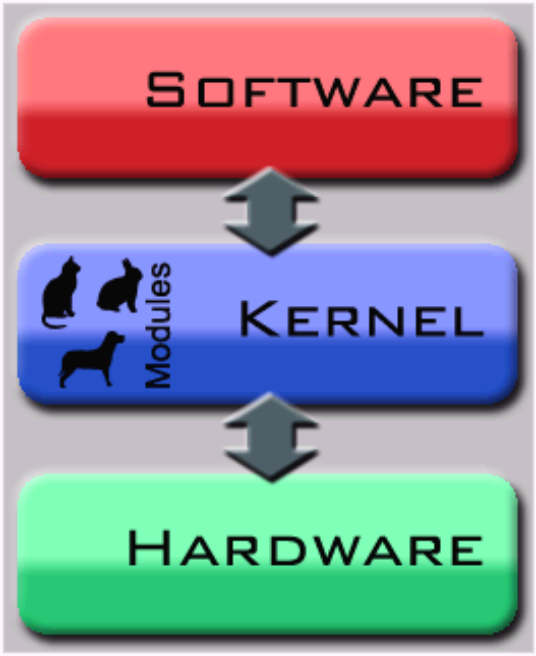
UNIX System Structure



Microkernel System Structure

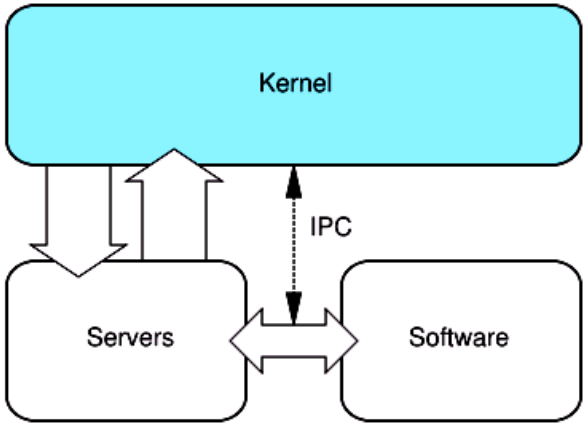
- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure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

- Monolithic Kernels
  - Execute all of their code in the same address space (kernel space)
  - Rich and powerful hardware access



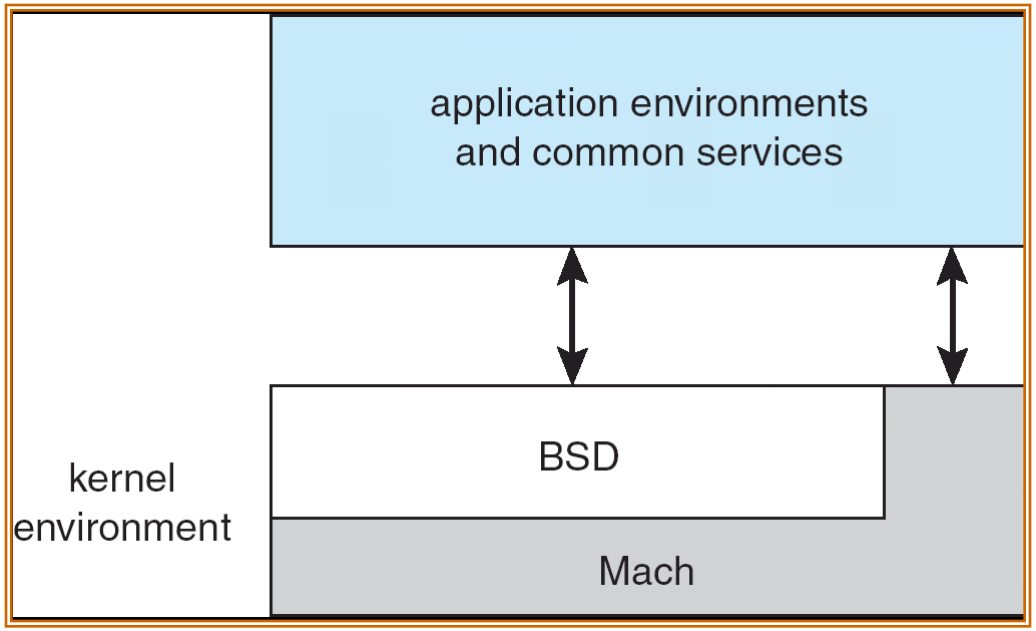
  - Disadvantages
    - The dependencies between system components
    - A bug in a driver might crash the entire system
    - Large kernels → very difficult to maintain

- Microkernels
  - Run most of their services in user space
    →improve maintainability and modularity
  - A simple abstraction over the hardware
  - A set of primitives or system calls
    - Memory management
    - Multitasking
    - IPC
  - Disadvantages
    - #(system calls) ↑
    - #(context switches) ↑



- Monolithic Kernels Versus Micro kernels
  - Most of the field-proven reliable and secure computer systems use a more microkernel-like approach
  - Micro kernels are often used in embedded robotic or medical computers where crash tolerance is important
  - Performances
    - Monolithic model is more efficient
      - IPC by: Shared kernel memory
        instead of message passing
        (Microkernel)

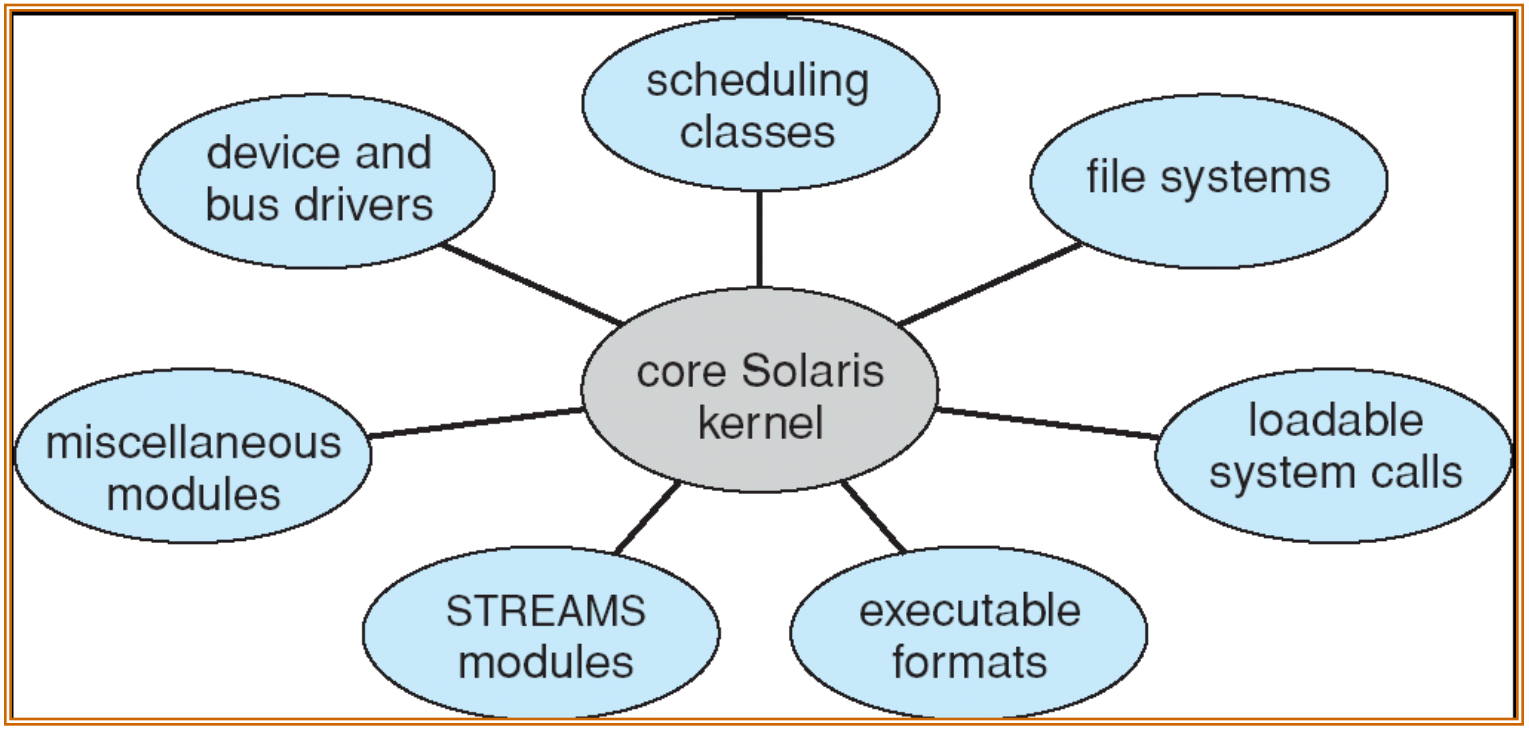
Mac OS X Structure



Modules

- Most modern operating systems implement 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

Solaris Modular Approach



Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware

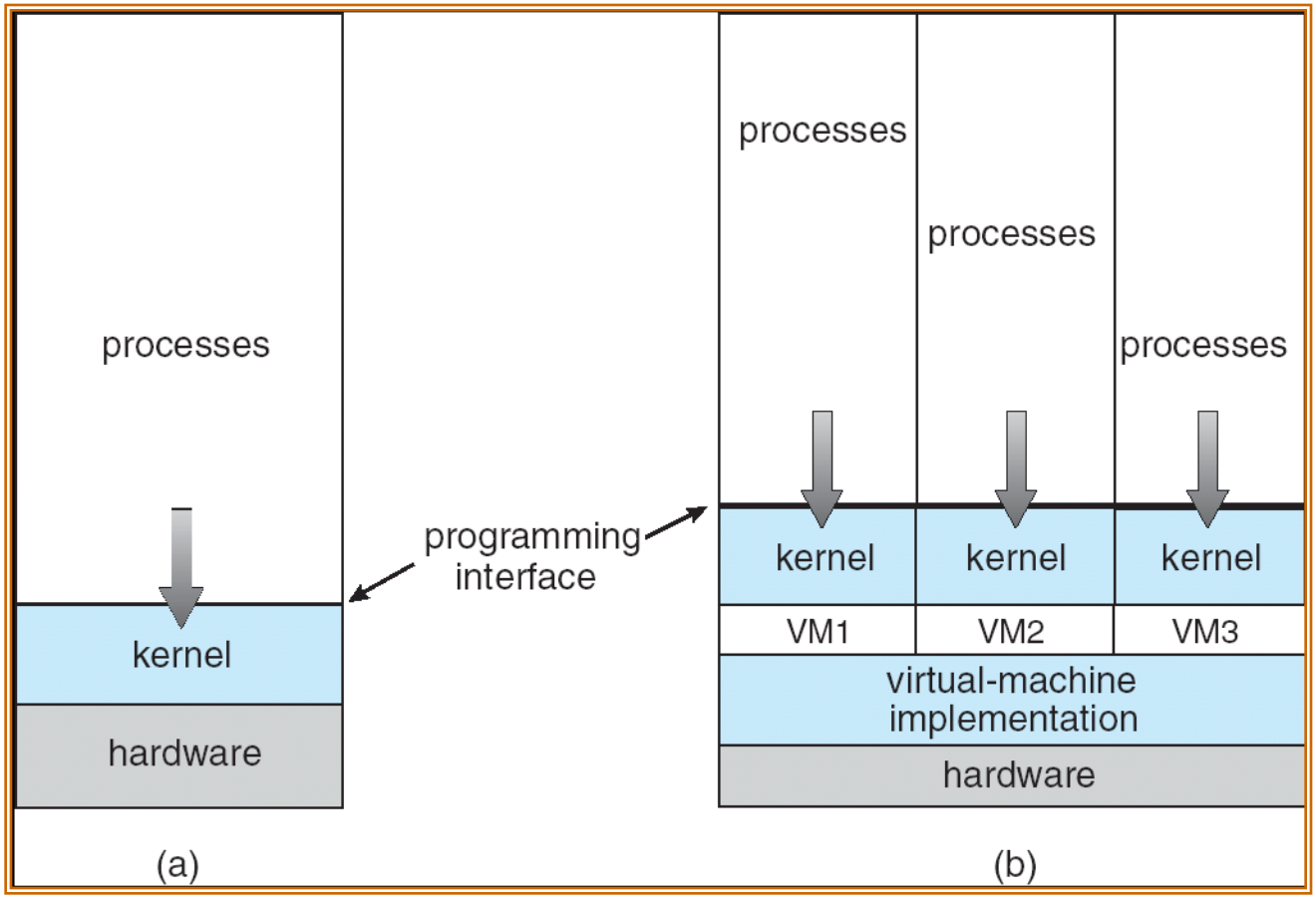
- A virtual machine provides an interface *identical* to the underlying bare hardware

- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

Virtual Machines (Cont.)

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual card readers and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator's console

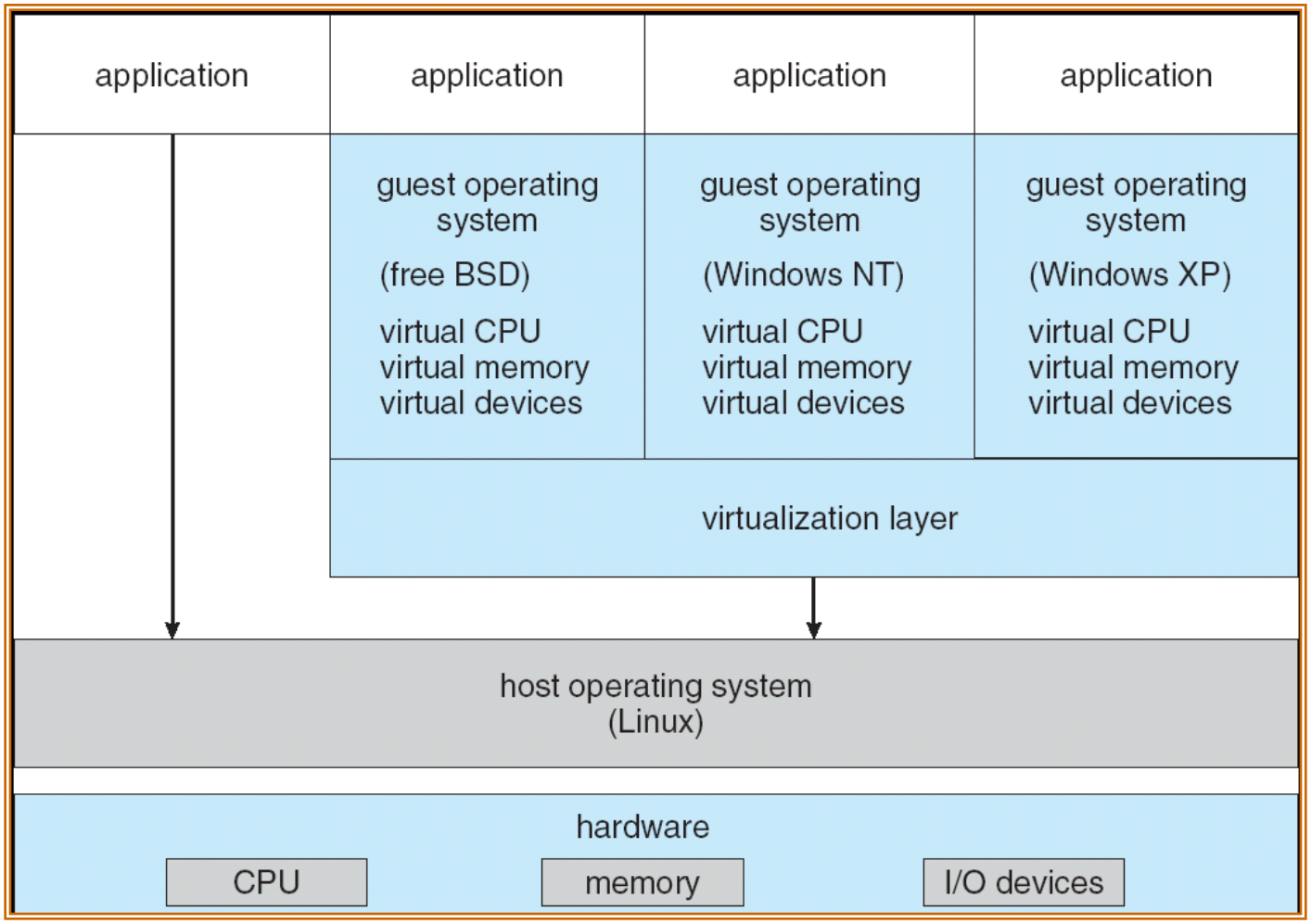
Virtual Machines (Cont.)



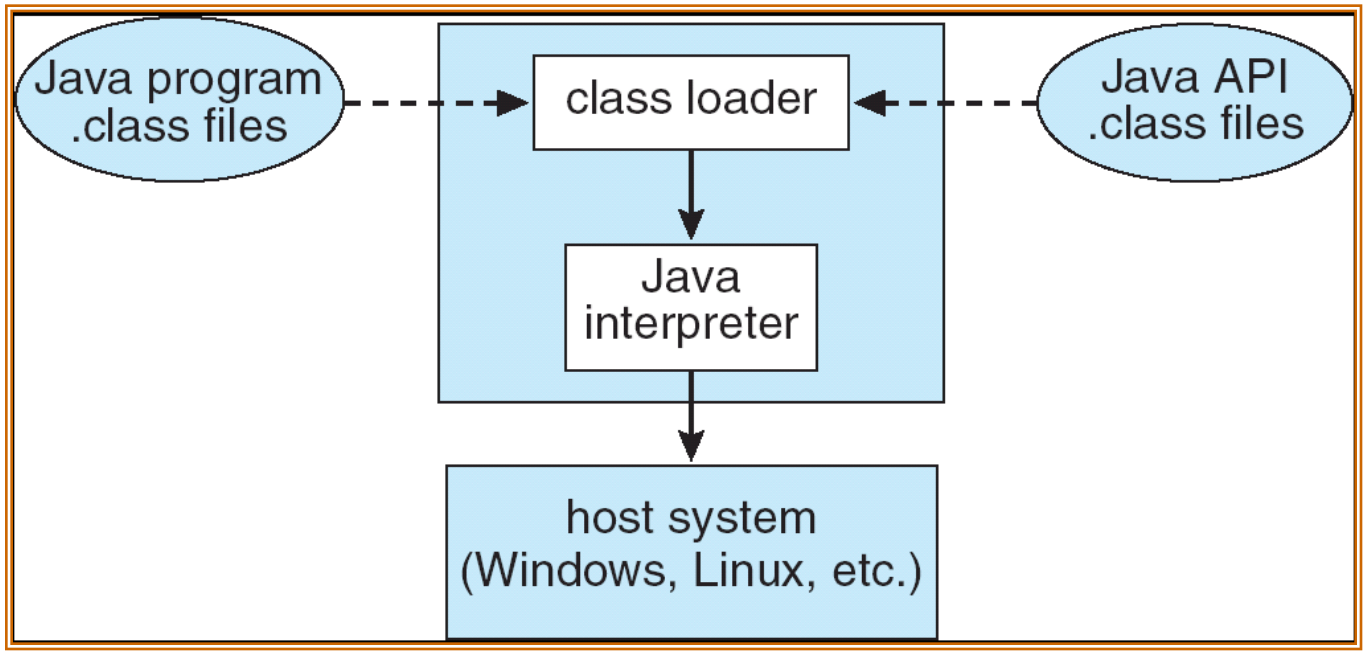
Virtual Machines (Cont.)

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine

VMware Architecture



The Java Virtual Machine



System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code

Operating system Components

OPERATING SYSTEM STRUCTURES

SYSTEM COMPONENTS

These are the pieces of the system we'll be looking at:

Process Management
Main Memory Management
File Management
I/O System Management
Secondary Management
Networking
Protection System
Command-Interpreter System

OPERATING SYSTEM STRUCTURES

SYSTEM COMPONENTS

PROCESS MANAGEMENT

A **process** is a **program** in execution: (A program is passive, a process active.)

A process has resources (CPU time, files) and attributes that must be managed.

Management of processes includes:

- Process Scheduling (priority, time management, . . .)
- Creation/termination
- Block/Unblock (suspension/resumption)
- Synchronization
- Communication
- Deadlock handling
- Debugging

OPERATING SYSTEM STRUCTURES

System Components

MAIN MEMORY MANAGEMENT

- Allocation/de-allocation for processes, files, I/O.
- Maintenance of several processes at a time
- Keep track of who's using what memory
- Movement of process memory to/from secondary storage.

FILE MANAGEMENT

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

The operating system is responsible for the following activities in connections with file management:

- File creation and deletion.
- Directory creation and deletion.
- Support of primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- File backup on stable (nonvolatile) storage media.

OPERATING SYSTEM STRUCTURES

System Components

I/O MANAGEMENT

- Buffer caching system
- Generic device driver code
- Drivers for each device - translate read/write requests into disk position commands.

SECONDARY STORAGE MANAGEMENT

- Disks, tapes, optical, ...
- Free space management (paging/swapping)
- Storage allocation (what data goes where on disk)
- Disk scheduling

OPERATING SYSTEM STRUCTURES

System Components

NETWORKING

- Communication system between distributed processors.
- Getting information about files/processes/etc. on a remote machine.
- Can use either a message passing or a shared memory model.

PROTECTION

- Of files, memory, CPU, etc.
- Means controlling of access
- Depends on the attributes of the file and user

How Do These All Fit Together?

In essence, they all provide services for each other.

SYSTEM PROGRAMS

- Command Interpreters -- Program that accepts control statements (shell, GUI interface, etc.)
- Compilers/linkers
- Communications (ftp, telnet, etc.)

OPERATING SYSTEM STRUCTURES

System Tailoring

Modifying the Operating System program for a particular machine. The goal is to include all the necessary pieces, but not too many extra ones.

- Typically a System can support many possible devices, but any one installation has only a few of these possibilities.
- **Plug and play** allows for detection of devices and automatic inclusion of the code (drivers) necessary to drive these devices.
- A **sysgen** is usually a link of many OS routines/modules in order to produce an executable containing the code to run the drivers.

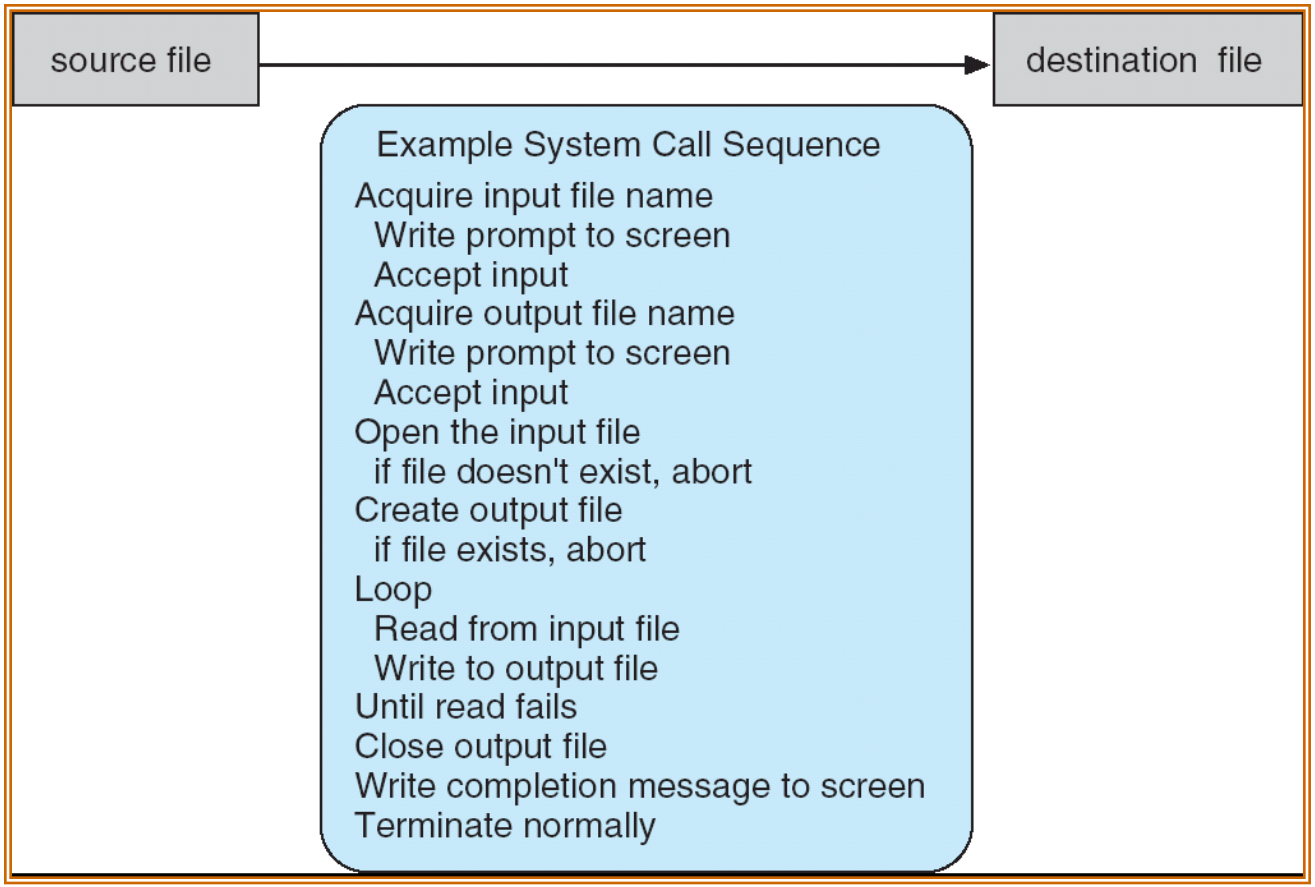
SYSTEM CALLS

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 Program Interface (API)** rather than direct system call use
- 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), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

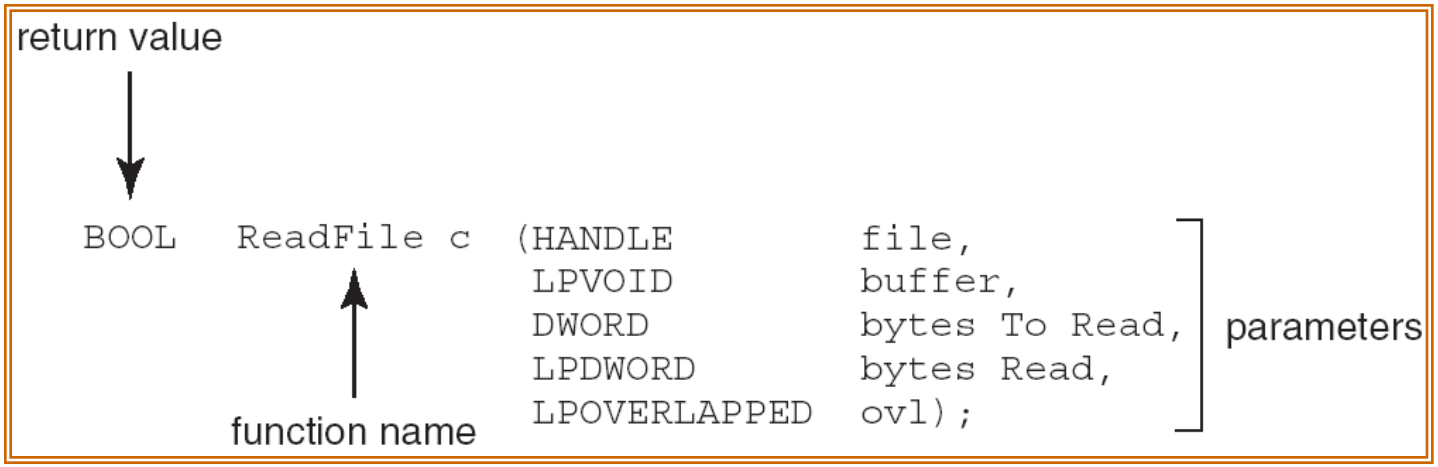
Example of System Calls

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



Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file

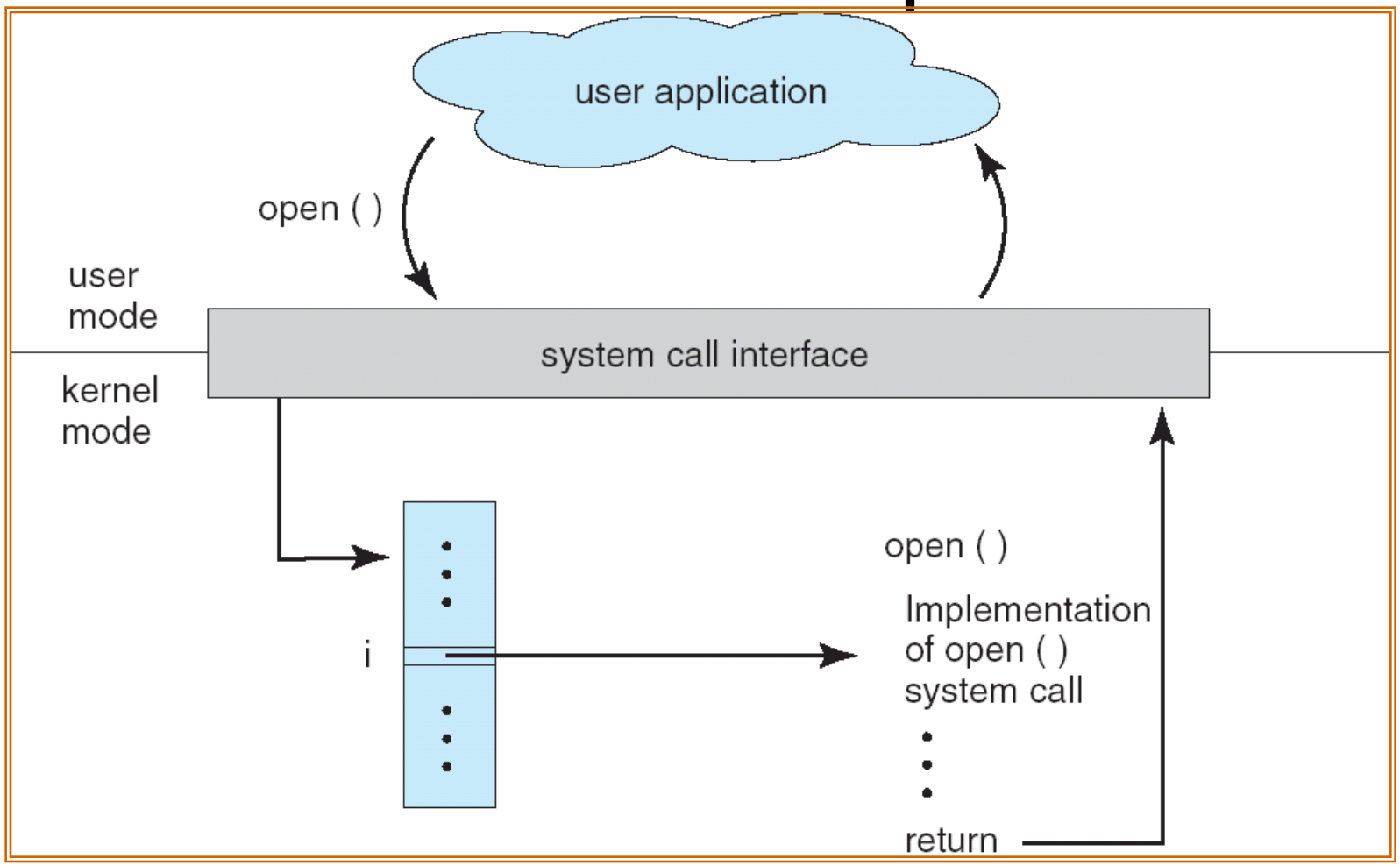


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

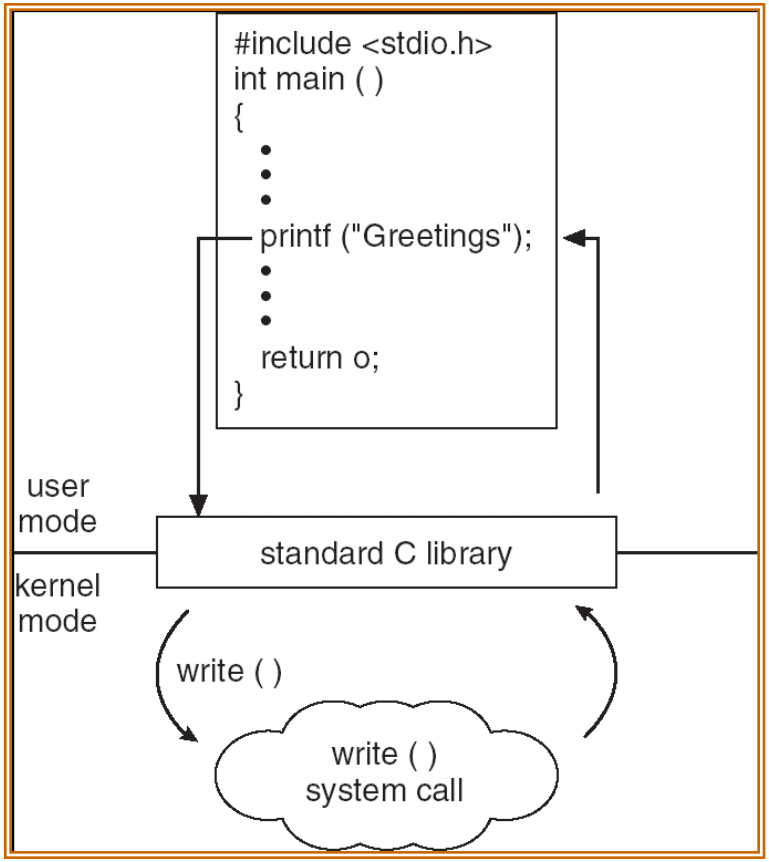
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about 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)

API – System Call – OS Relationship



Standard C Library Example

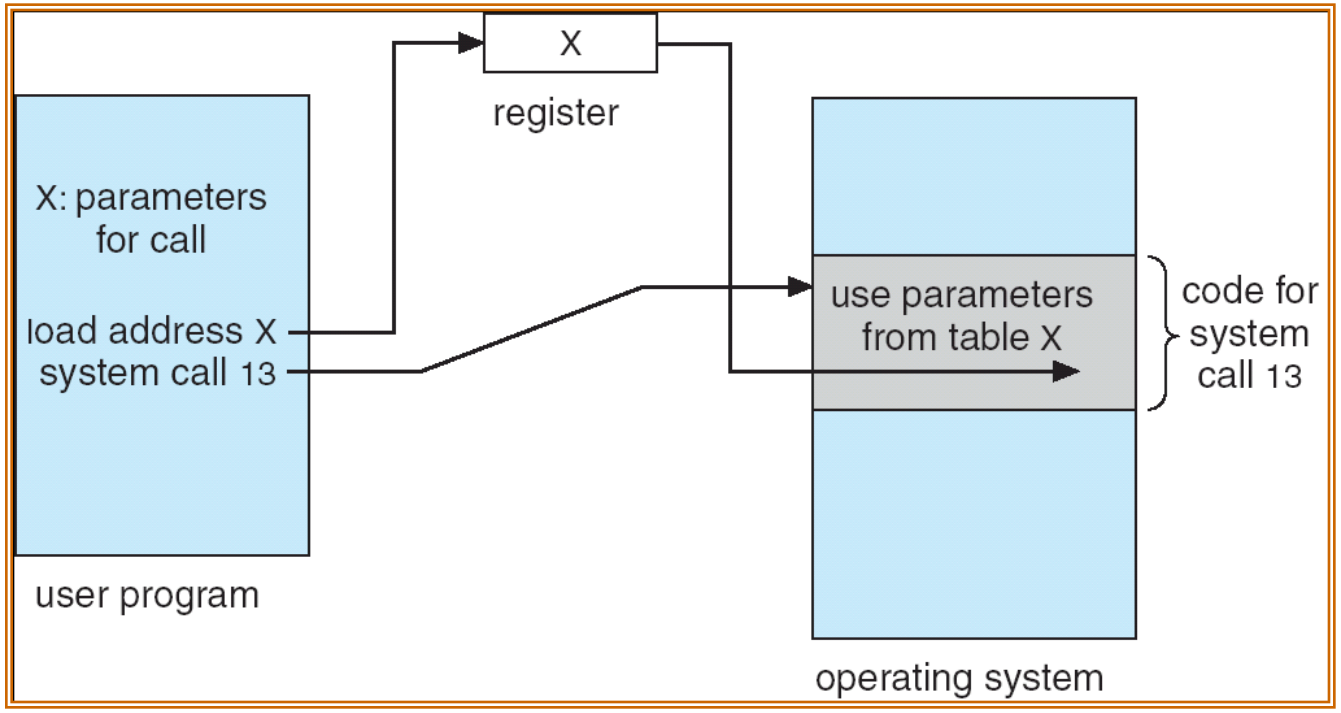
- C program invoking printf() library call, which calls write() system call



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
- 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

Parameter Passing via Table



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

Reading Assignment

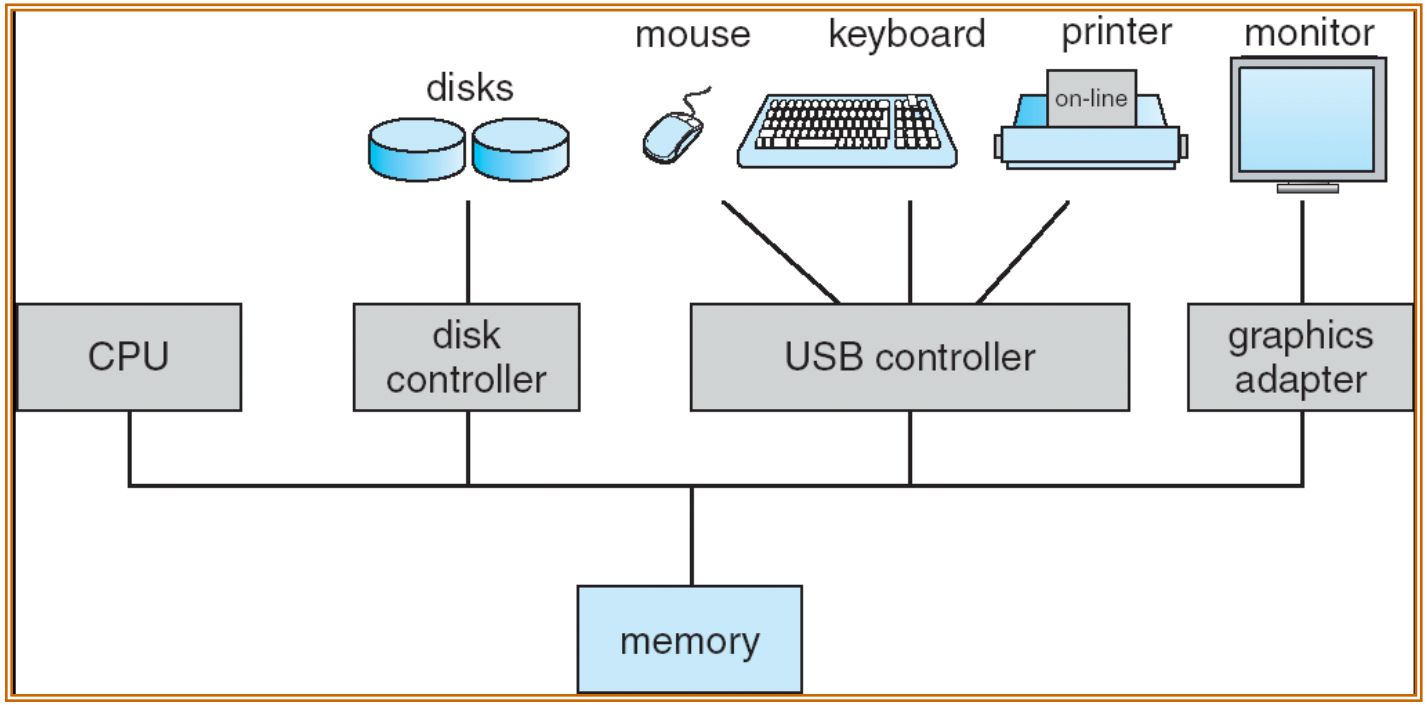
- What is the difference Between System Call & API?
- Can we interpret push & pop as an API in a stack program?
- Differentiate between Micro kernel & Monolithic kernel.

Unit I

Device organization + Interrupts + User / Kernel State Transitions

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-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*.

Interrupt

- An *interrupt* is an external or internal event that interrupts the microcontroller to inform it that a device needs its service.
- Classification of Interrupts
  - Interrupts can be classified into two types:
    - Maskable Interrupts (Can be delayed or Rejected)
    - Non-Maskable Interrupts (Can not be delayed or Rejected)
- Interrupts can also be classified into:
    - Vectored (the address of the service routine is hard-wired)
    - Non-vectored (the address of the service routine needs to be supplied externally by the device)

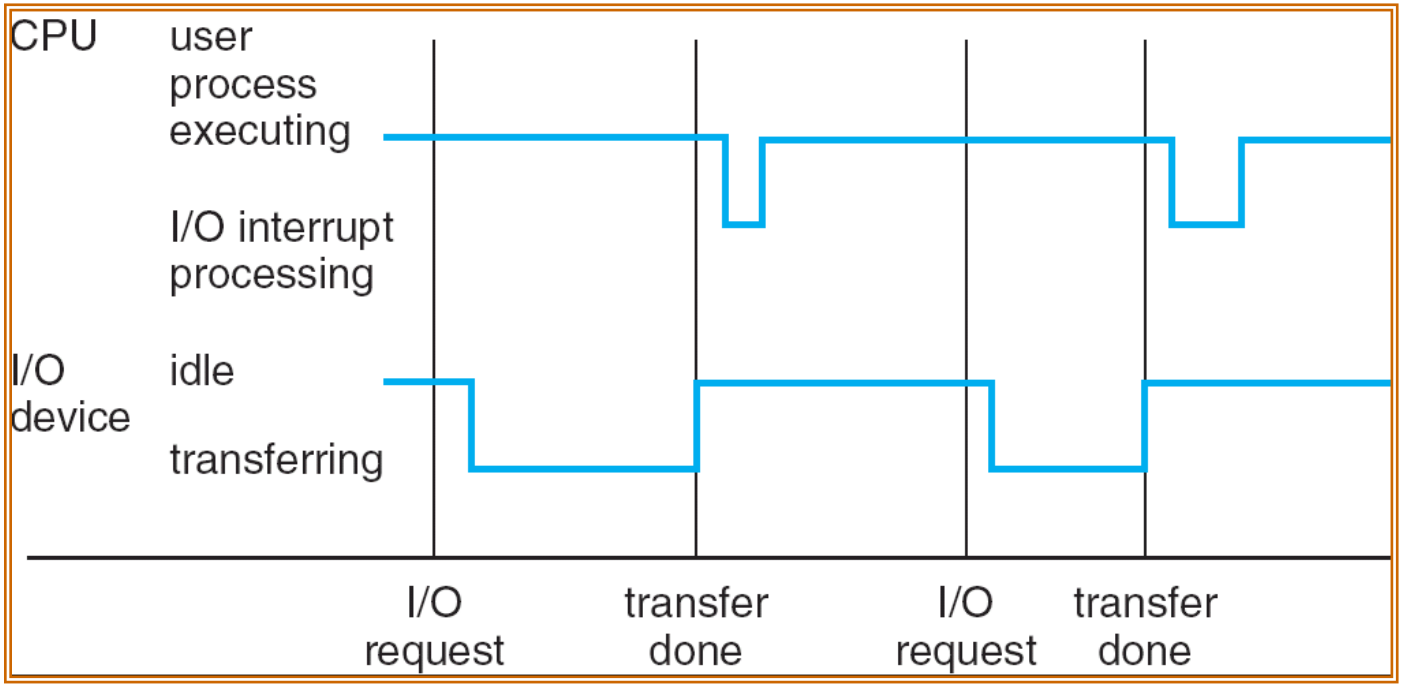
Common Functions of Interrupts

- 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.
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.
- A *trap* is a software-generated interrupt caused either by an error or a user request.
- An operating system is *interrupt* driven.

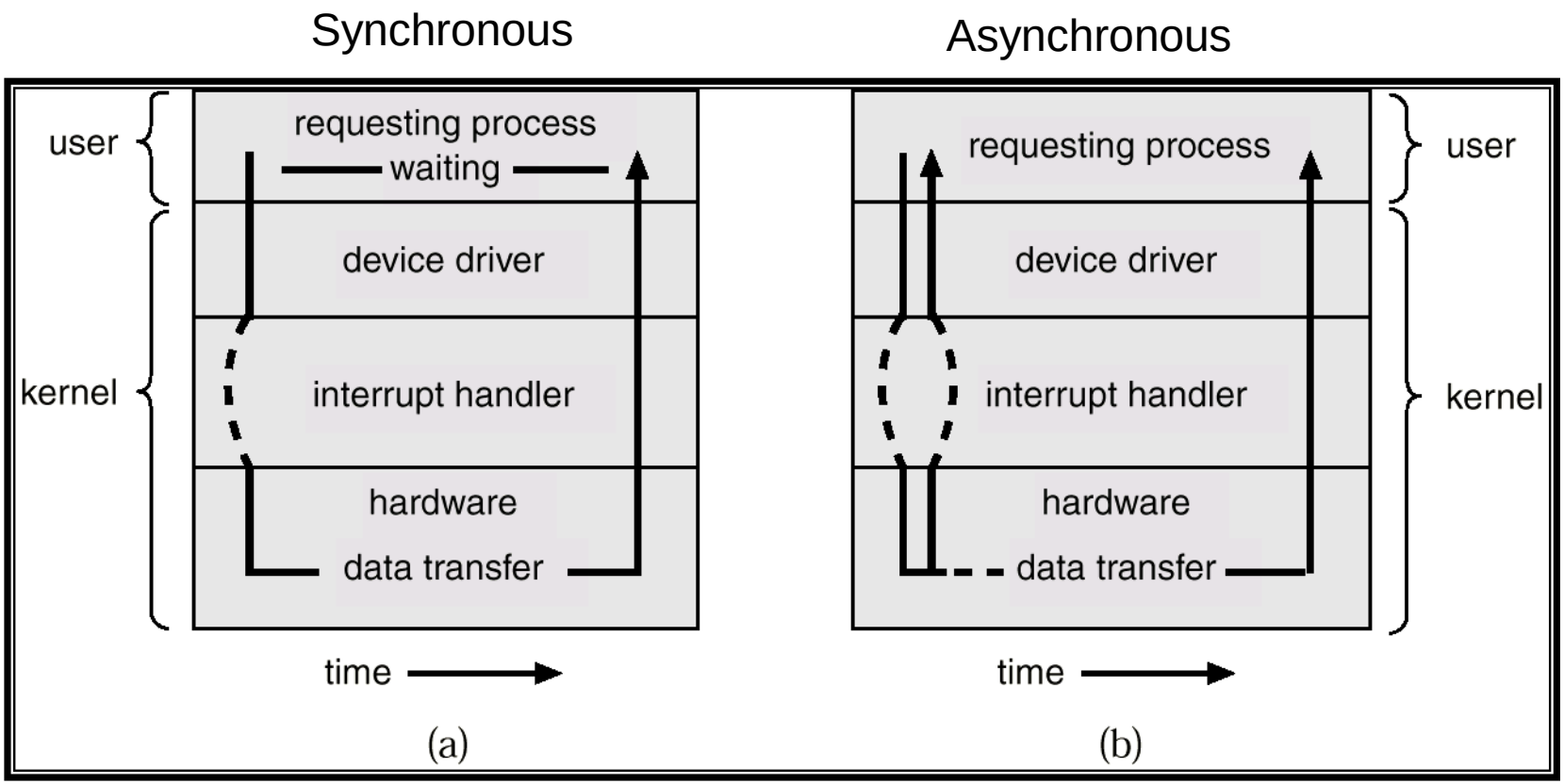
Interrupt Handling

- 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

Interrupt Timeline



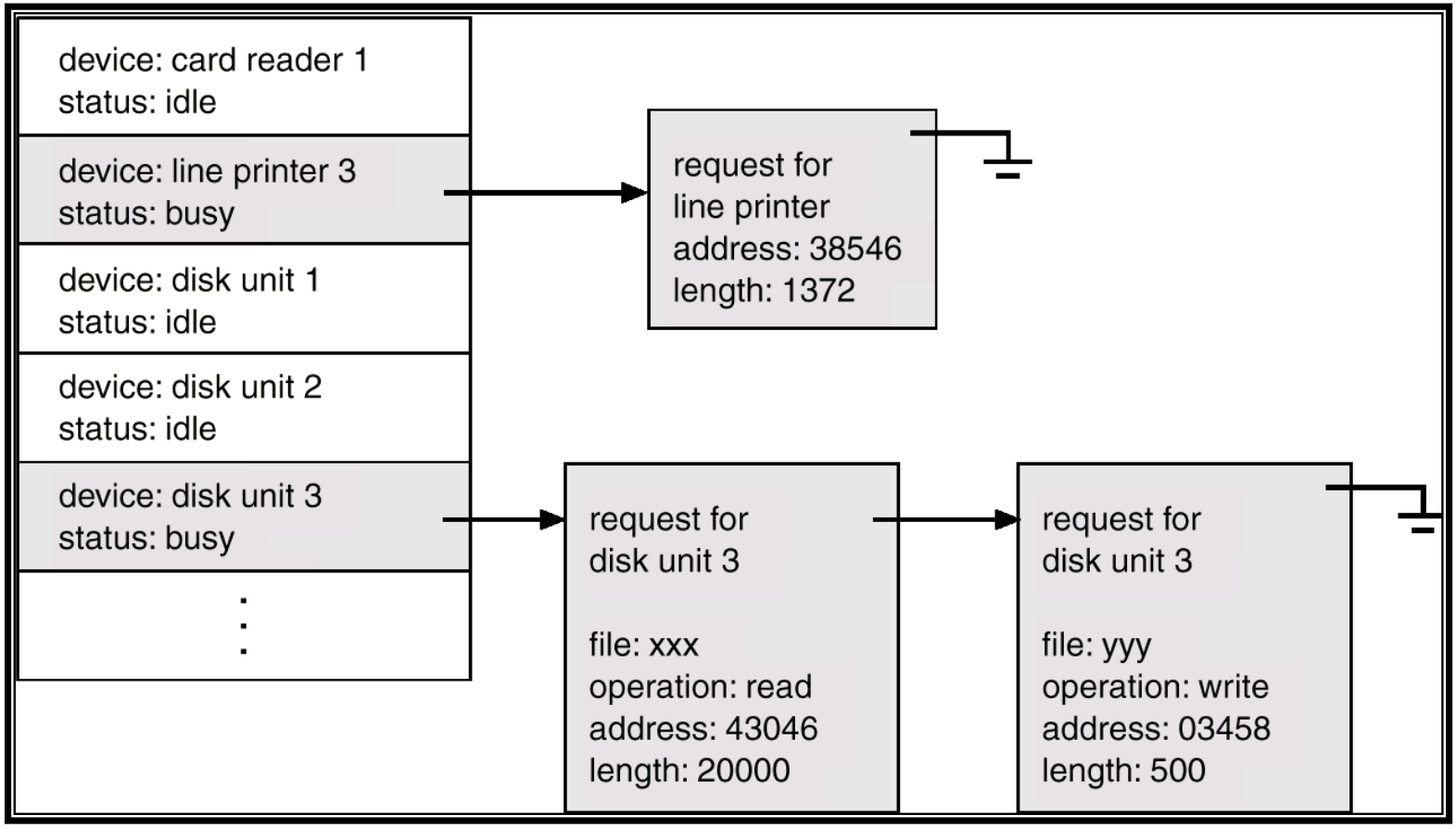
Two I/O Methods



I/O Structure

- After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access).
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
  - *System call* – request to the operating system to allow user to wait for I/O completion.
  - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Device-Status Table

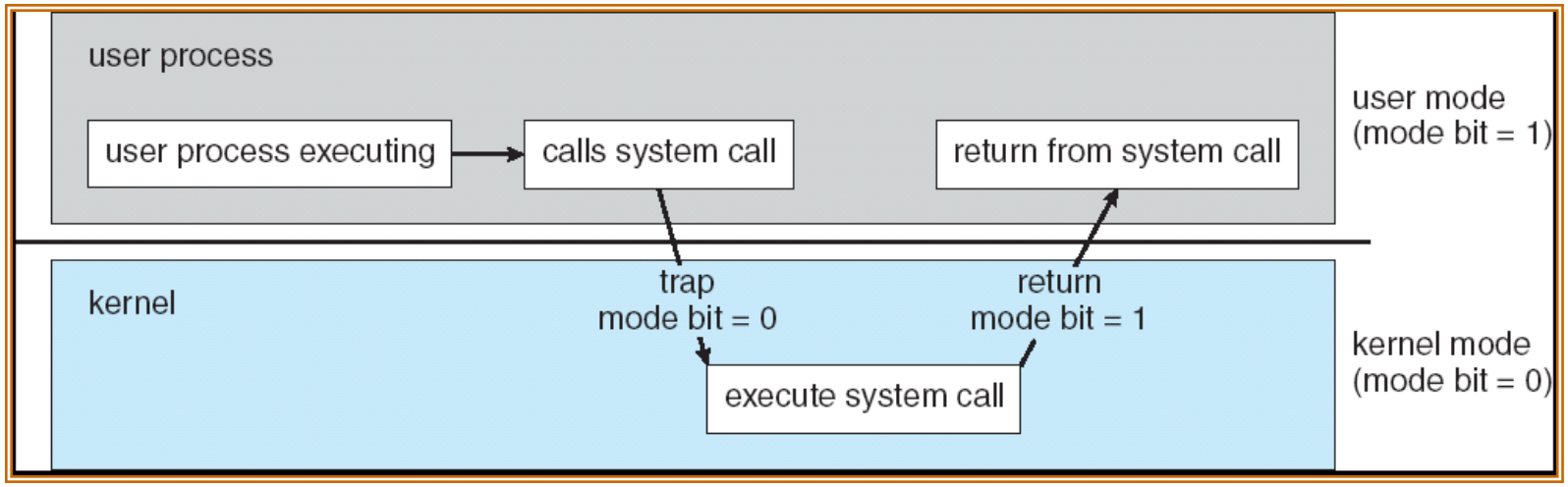


Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **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

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time



Threads

Threads

- A **thread** is a basic unit of CPU utilization
- We have considered only the single-threaded processes (heavyweight processes) so far
- We have not separated out a thread from a process
- However, a process can be multithreaded

Threads

- What is a thread?
- Why the multithreaded processes have a number of advantages in comparison with the single-threaded processes?
- How threads are organized?

Processes and Threads

- Process abstraction combines two concepts
  - Concurrency
    - Each process is a sequential execution stream of instructions
  - Protection
    - Each process defines an address space
    - Address space identifies all addresses that can be touched by the program

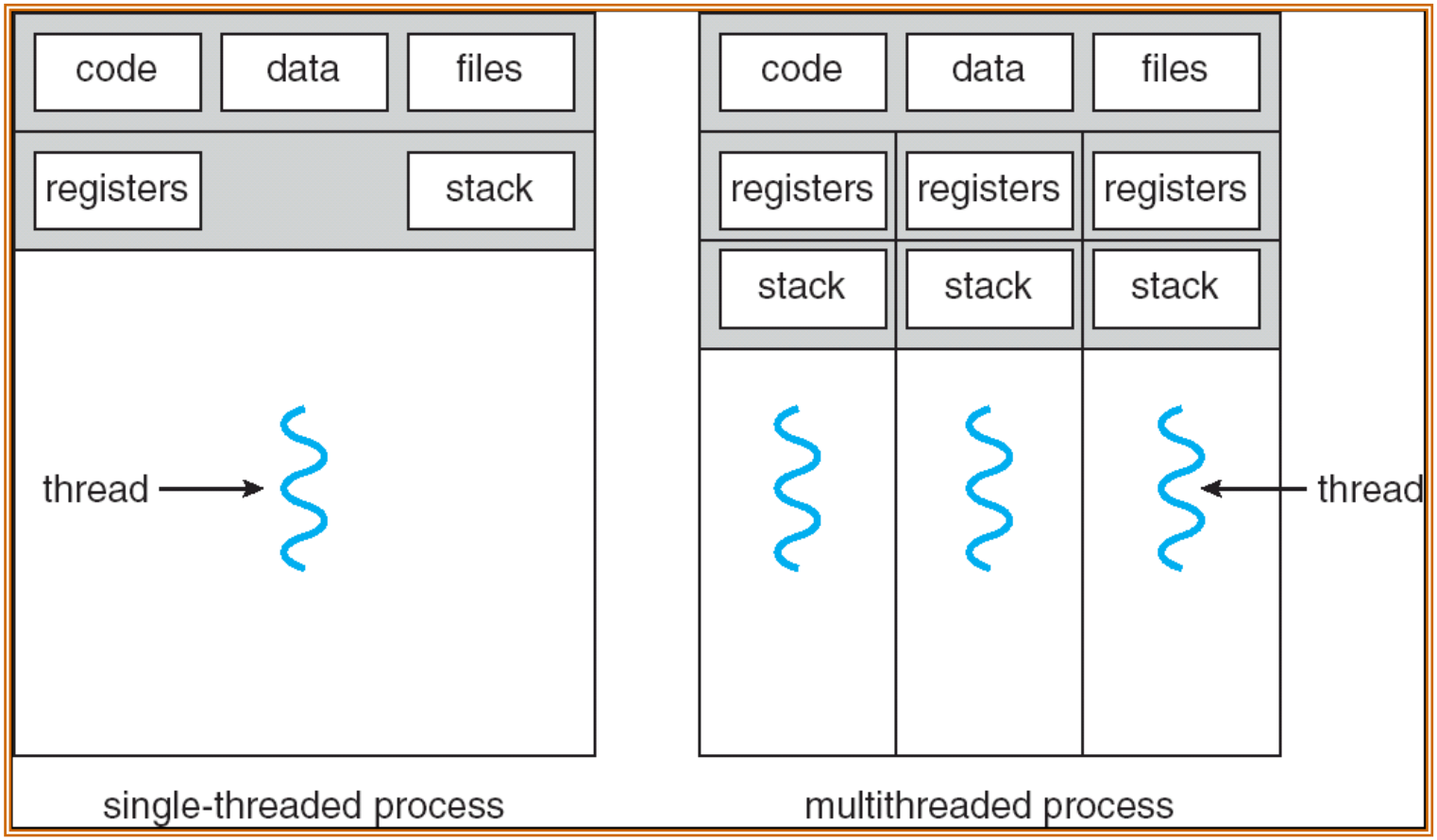
- **Threads**
  - **Key idea**: separate the concepts of concurrency from protection
  - A thread represents a sequential execution: stream of instructions
  - A process defines the address space that may be shared by multiple threads that **share code/data/heap**

Introducing Threads

- A **thread** represents an abstract entity that executes a sequence of instructions
  - **It has its own ID**
  - **It has its own Program Counter**
  - **It has its own set of CPU registers**
  - **It has its own stack**
  - There is no thread-specific heap or data segment, code section and other operating system resources (unlike process)

- Threads are lightweight
  - Creating a thread more efficient than creating a process.
  - Communication between threads easier than btw. processes.
  - Context switching between threads requires fewer CPU cycles and memory references than switching processes.
  - Threads only track a subset of process state (share list of open files, pid, …)

Single and Multithreaded Processes

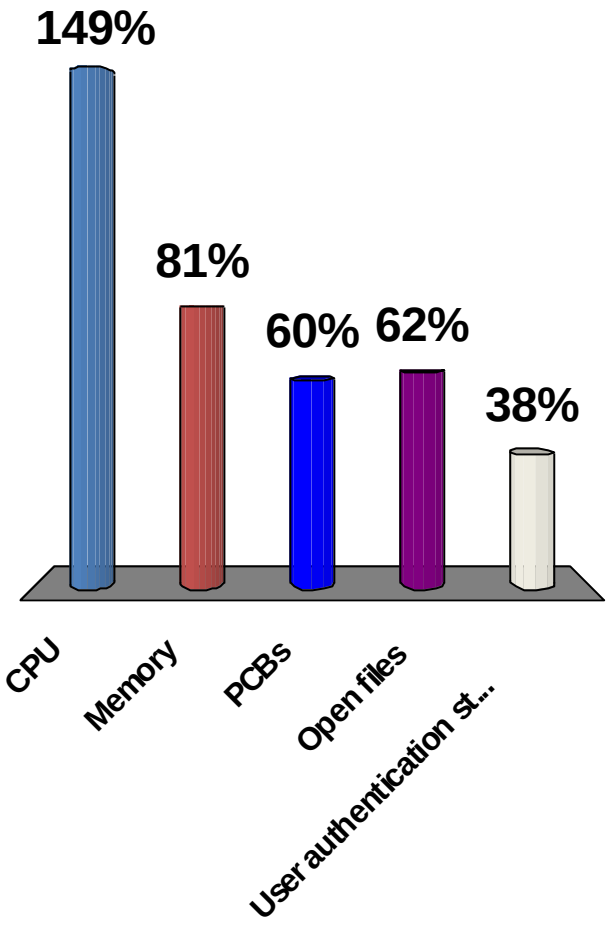


Benefits

- **Responsiveness**. Multithreading may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.
- **Resource Sharing**. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Utilization of Multiprocessor Architectures**. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

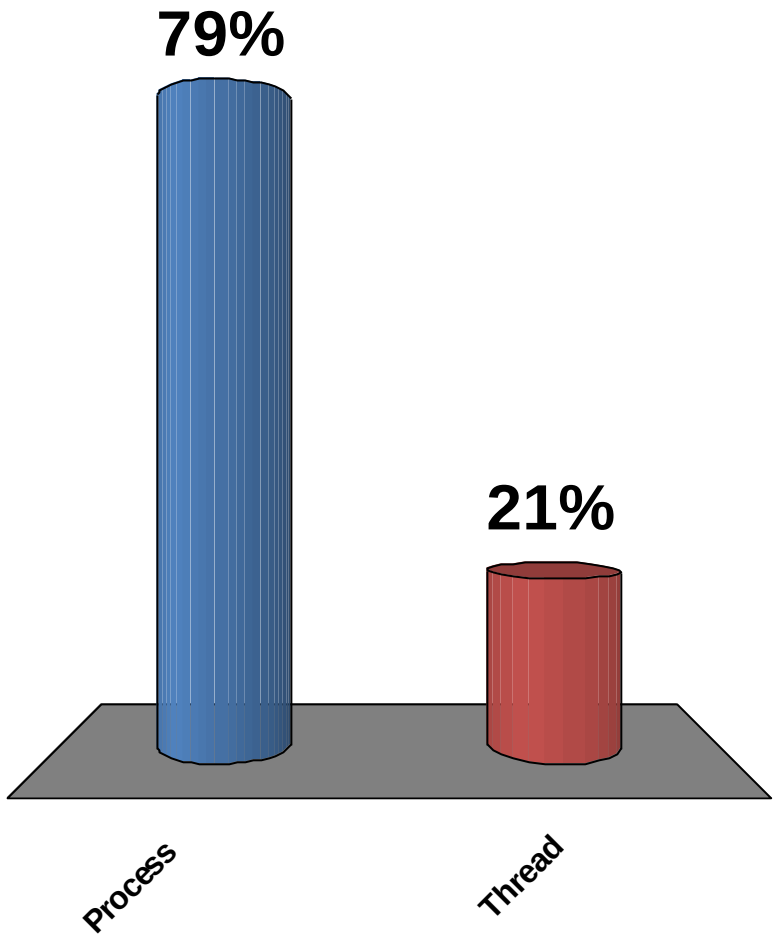
Threads allow you to multiplex which resources?

1. CPU
2. Memory
3. PCBs
4. Open files
5. User authentication structures



Context switch time for which entity is greater?

1. Process
2. Thread



Threads vs. Processes

Threads

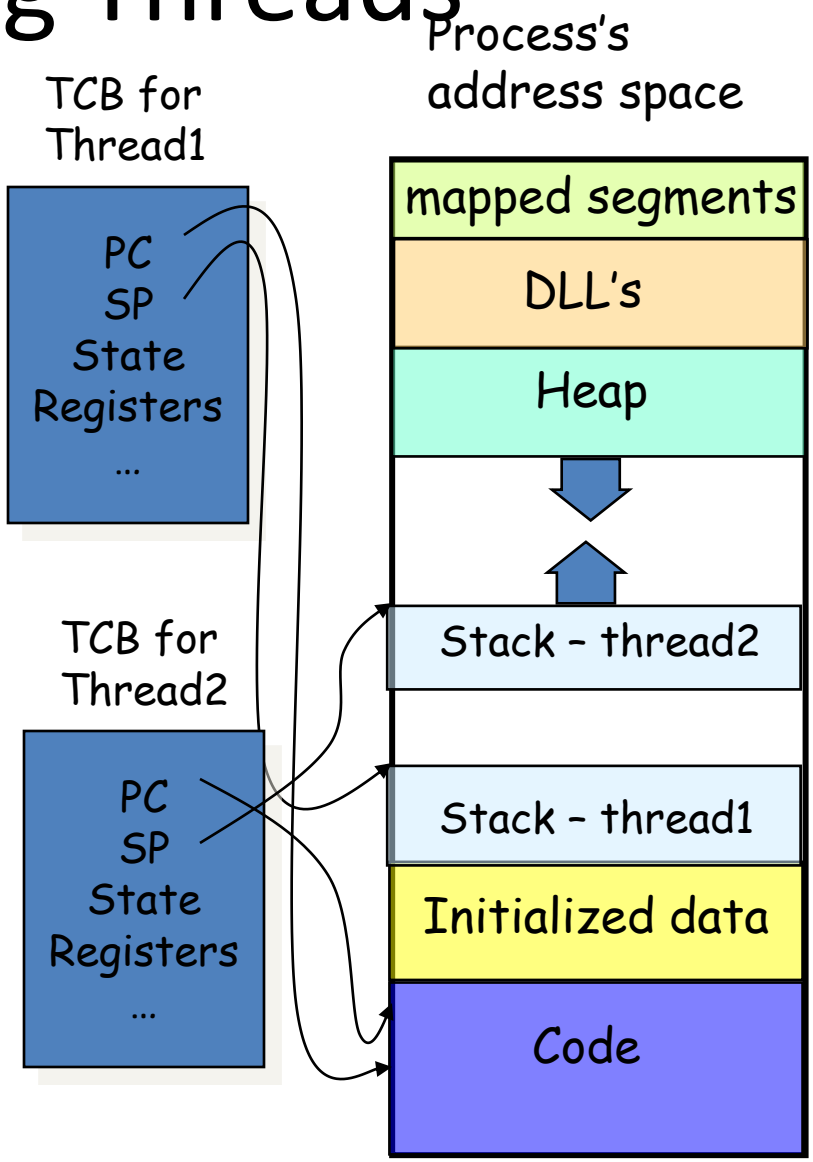
- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory.

Processes

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die
- Inter-process communication via OS and data copying.

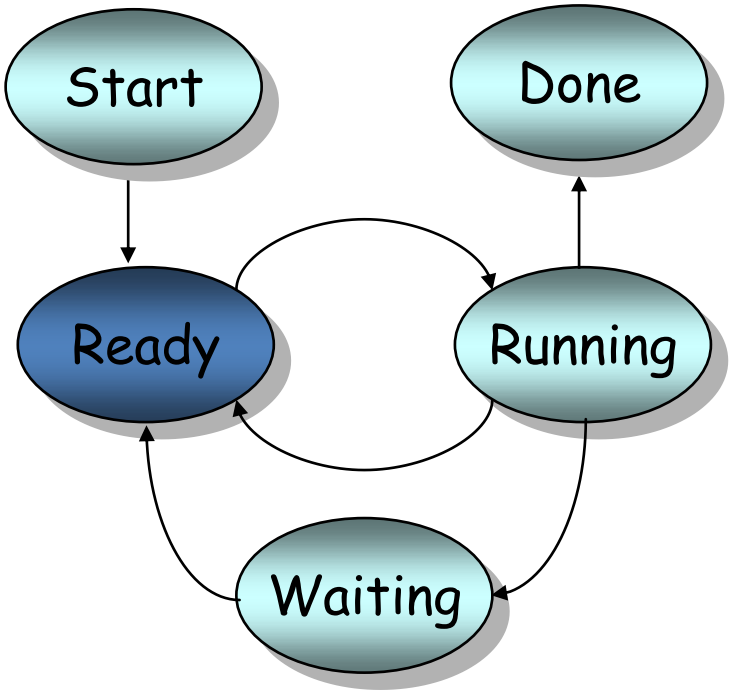
Implementing Threads

- Processes define an address space; threads share the address space
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Threads' Life Cycle

- Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states



User and Kernel Threads

- Thread management done by user-level threads library above the kernel
- A thread library provides the programmer an API for creating and managing threads.
- Two approaches of implementing a thread library:
  - the library is in user space with no kernel support (all code and data structures for the library exist in the user space, above the kernel)
  - a kernel-level library supported directly by the operating system.
- Three primary thread libraries:
  - POSIX Pthreads (user- or kernel- level - UNIX, Linux)
  - Win32 threads (kernel-level - Windows)
  - Java threads (thread creation and management in Java programs)

Multithreading Models

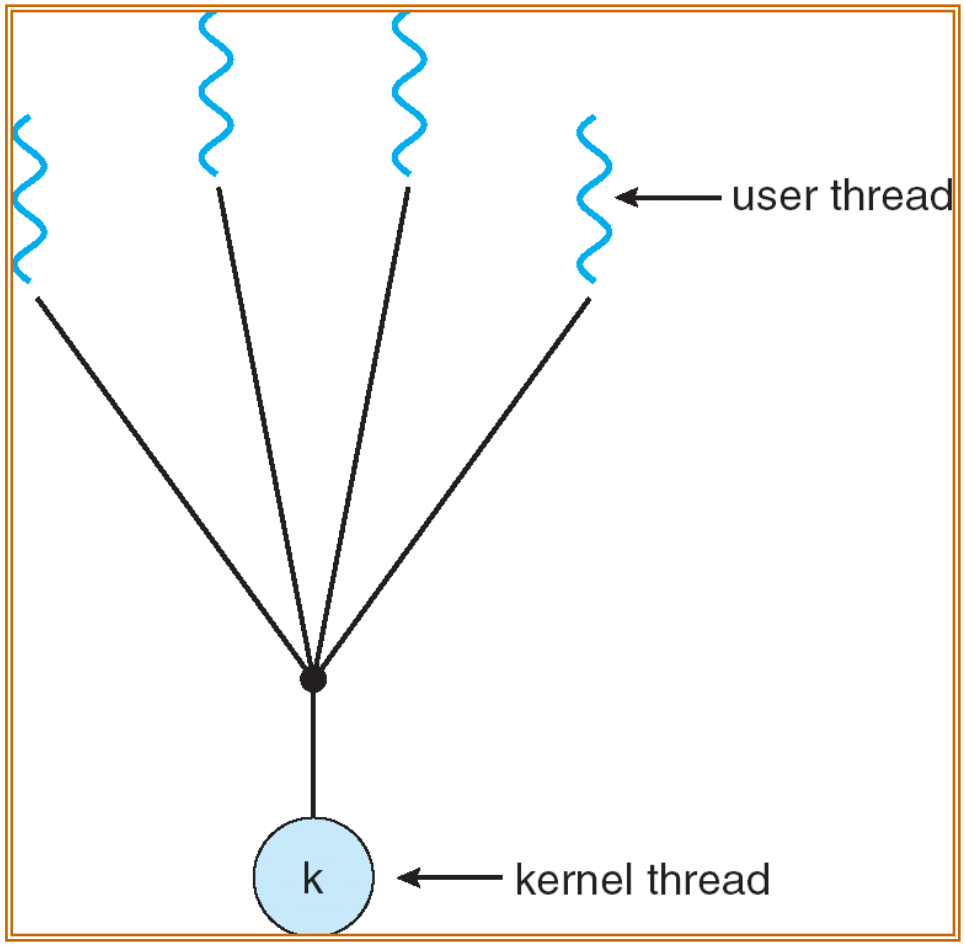
- A relationship between user threads and kernel threads

- Many-to-One model maps many user-level threads to one kernel thread. Multiple threads are unable to run in parallel, but the entire process will block if any thread makes a blocking system call. Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- One-to-One model maps each user thread to a separate kernel thread and allows multiple threads to run in parallel on multiprocessors and another thread to run when a thread makes a blocking system call
- Many-to-Many model multiplexes many user-level threads to a smaller or equal number of kernel threads that can run in parallel on a multiprocessor

Many-to-One

- Many user-level threads mapped to a single kernel thread
- Multiple threads are unable to run in parallel, but the entire process will block if any thread makes a blocking system call.
- Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

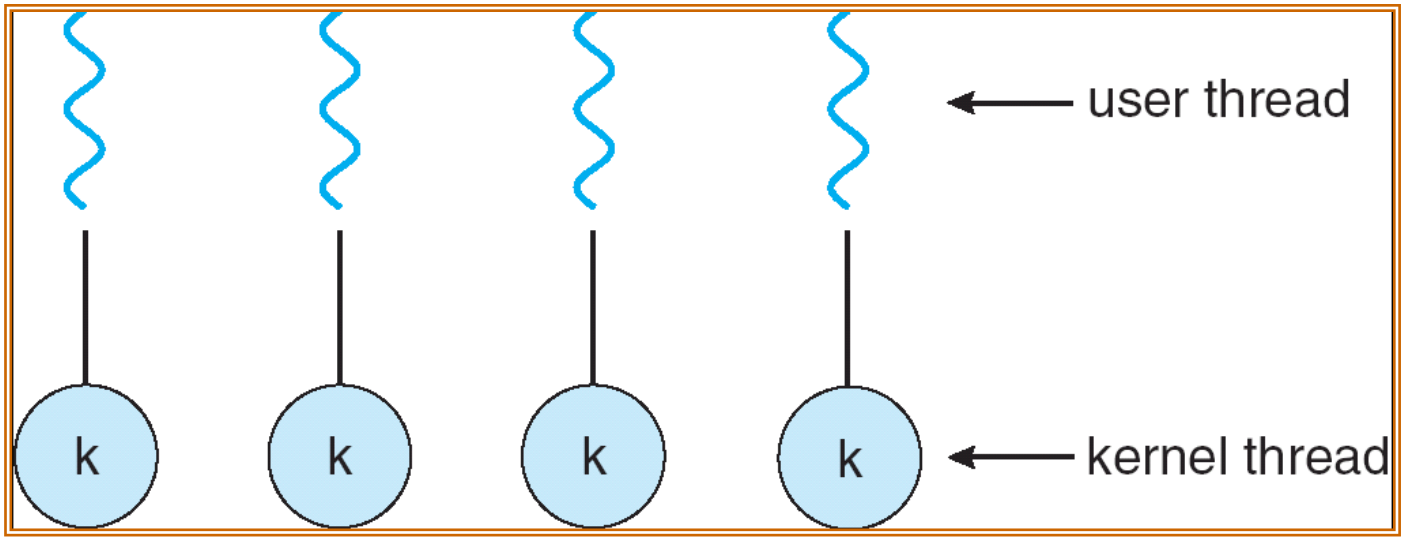
Many-to-One Model



One-to-One

- Each user-level thread maps to a separate kernel thread
- allows multiple threads to run in parallel on multiprocessors
- allows another thread to run when a thread makes a blocking system call
- **Drawback**: because the overhead of creating kernel threads can burden the performance of the application, most implementations of this model restrict the number of threads supported by the system
- Examples
  - Windows NT/XP/2000, 95, 98
  - Linux
  - Solaris 9 and later

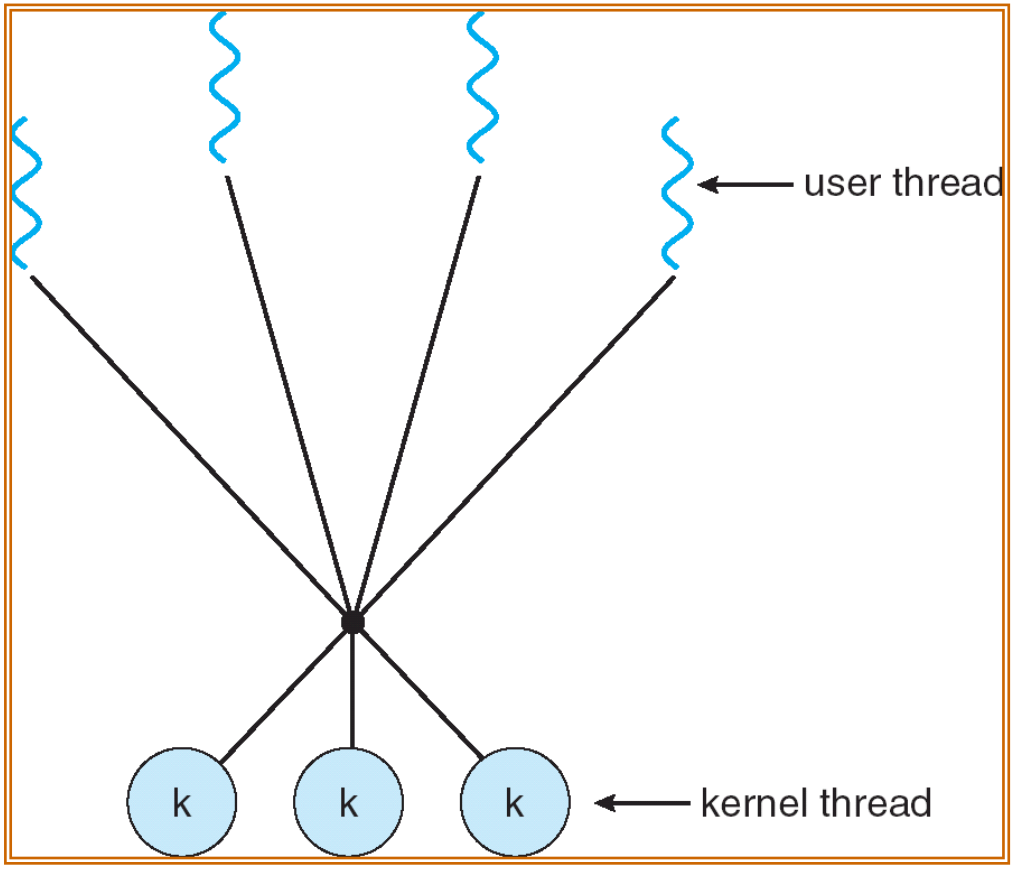
One-to-one Model



Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads that can run in parallel on a multiprocessor
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- The number of kernel threads may be specific to either particular application or a particular computer
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



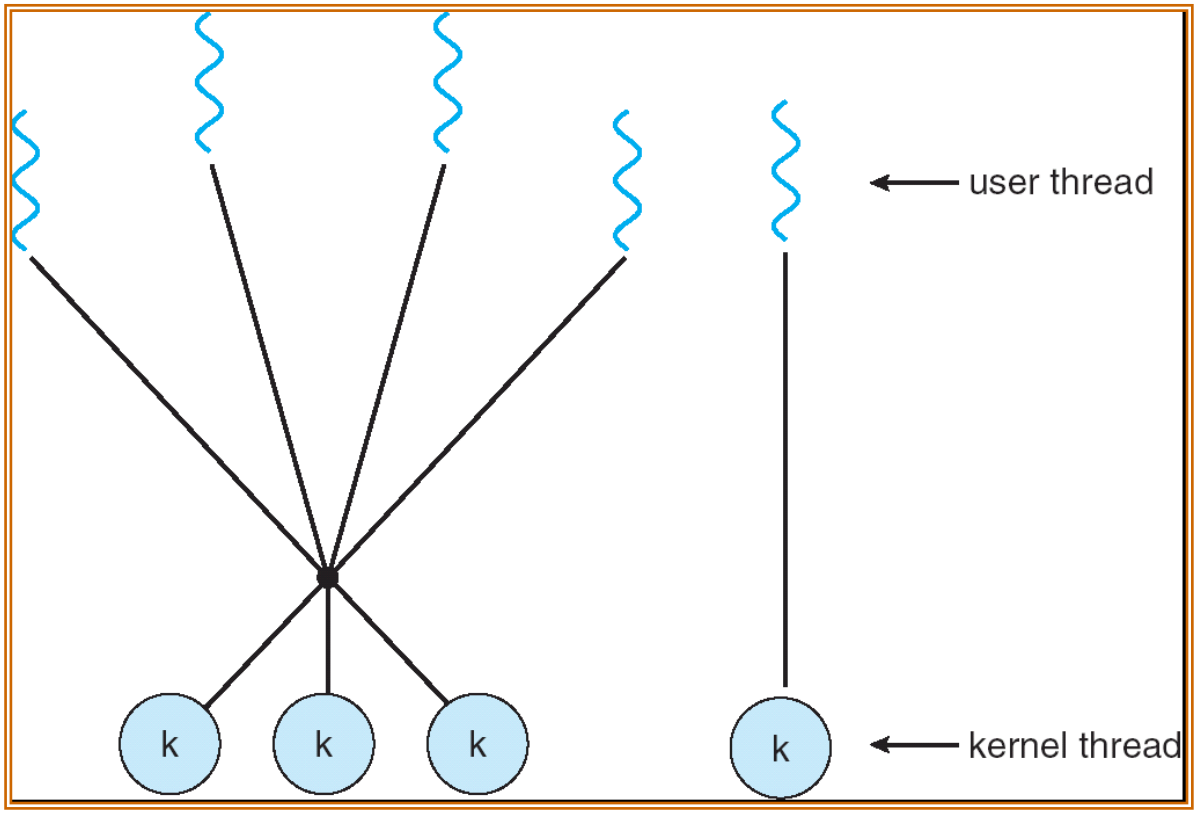
Multithreading Models: Comparison

- The **many-to-one** model allows the developer to create as many user threads as he/she wishes, but true concurrency can not be achieved because only one kernel thread can be scheduled for execution at a time
- The **one-to-one** model allows more concurrence, but the developer has to be careful not to create too many threads within an application
- The **many-to-many** model does not have these disadvantages and limitations: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor

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

Two-level Model



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
  - Two versions of fork() in UNIX:, one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
  - If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process – including all threads.
  - If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.
  - If the separate process does not call exec() after forking, the separate process should duplicate all threads.

Thread Cancellation

- Thread cancellation is the task of terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Pools

- The general idea is to create a number of threads at process startup and place them into a **pool** where they sit and wait for 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

Thread Specific Data

- 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)

End of Chapter 4

Chapter 3: Processes

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

Process Concept

- 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
- A process includes:
  - program counter
  - stack
  - data section

Process Concept

- ***Orphan Process***

  An **orphan process** is a computer process whose parent process has finished or terminated, though itself remains running.

  In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called *re-parenting* and occurs automatically. Even though technically the process has the "init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

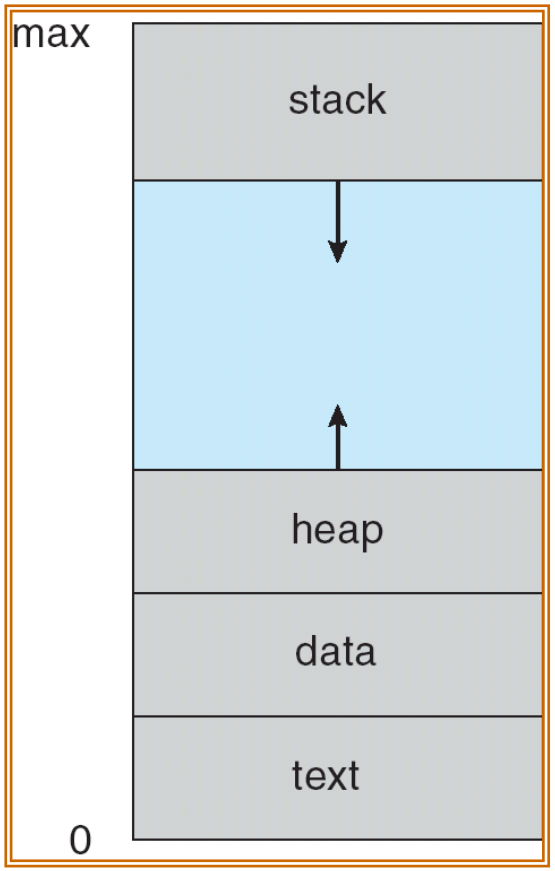
- ***Zombie Process***

  On Unix and Unix-like computer operating systems, a **zombie process** or **defunct process** is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the process that started the (now zombie) process to read its exit status. The term *zombie process* derives from the common definition of zombie—an undead person.

- ***Daemon Process***

  In Unix and other computer multitasking operating systems, a **daemon** (pronounced /ˈdeɪmən/ or /ˈdiːmən/)[1] is a computer program that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

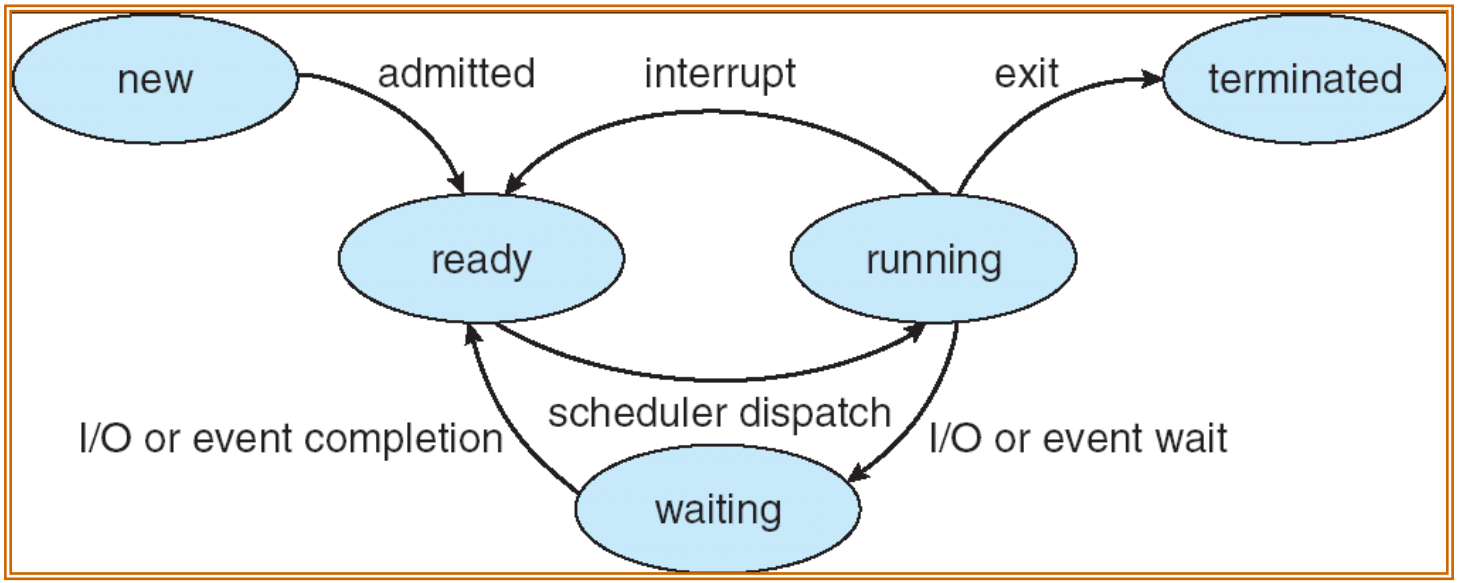
Process in Memory



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 process
  - **terminated**: The process has finished execution

Diagram of Process State

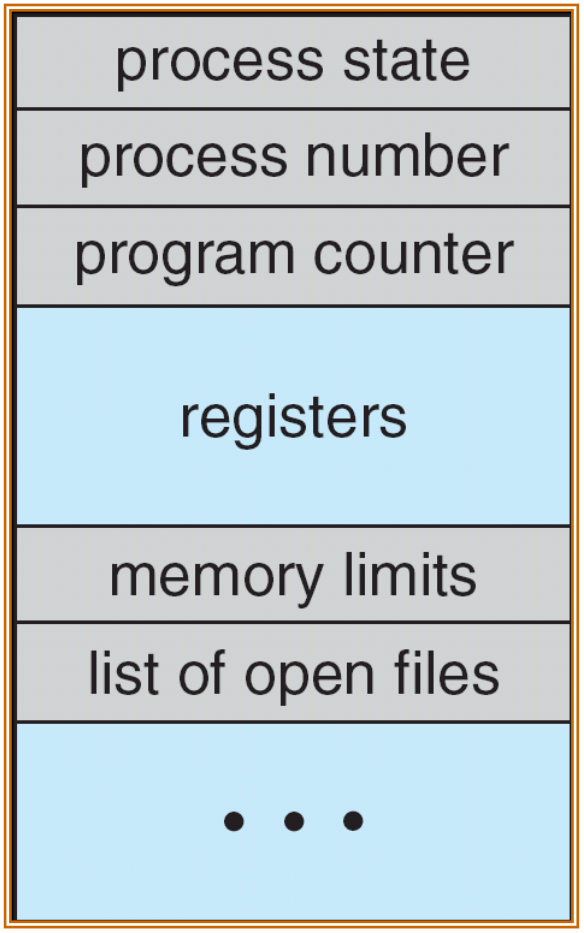


Process Control Block (PCB)

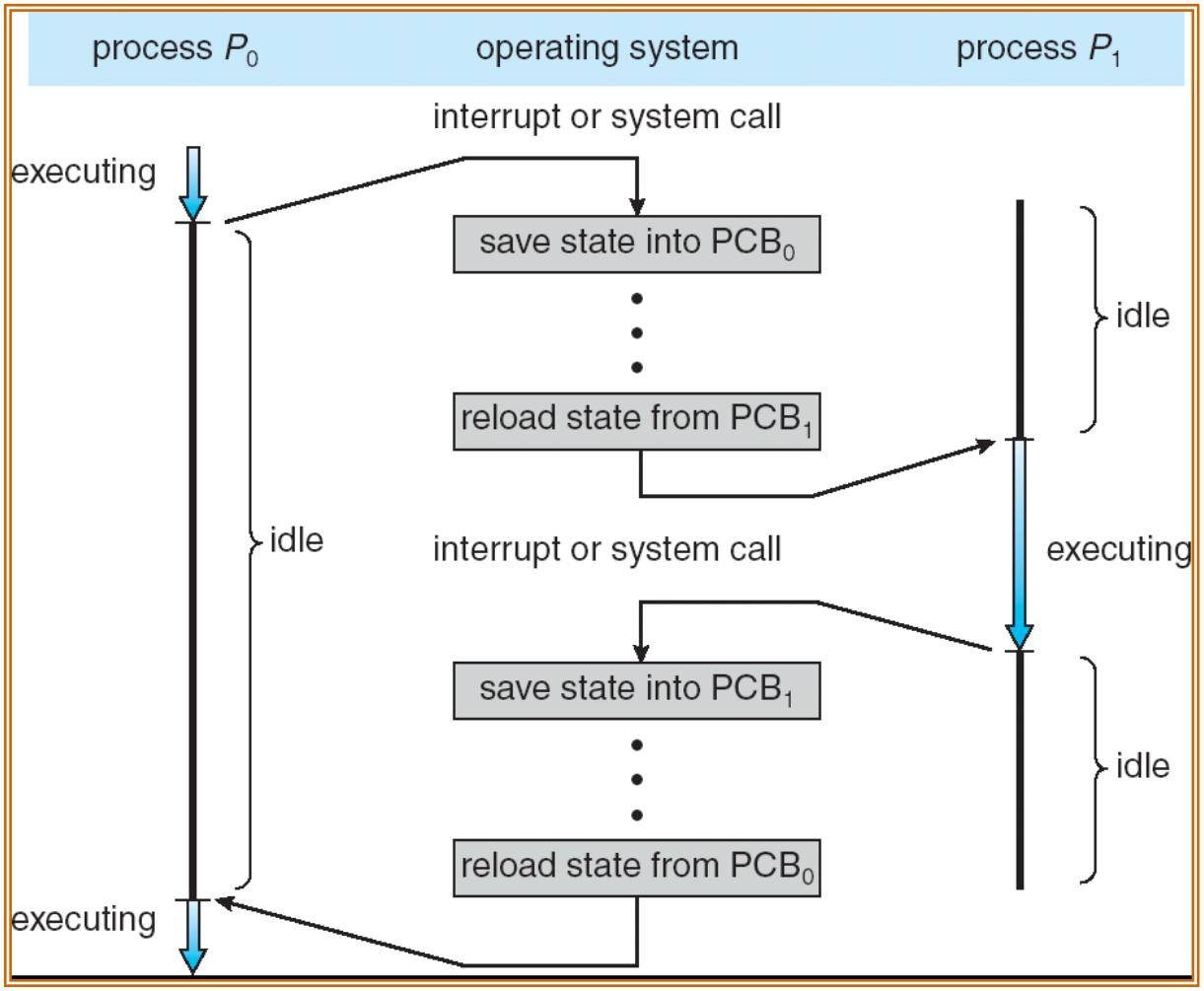
Information associated with each process

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

Process Control Block (PCB)



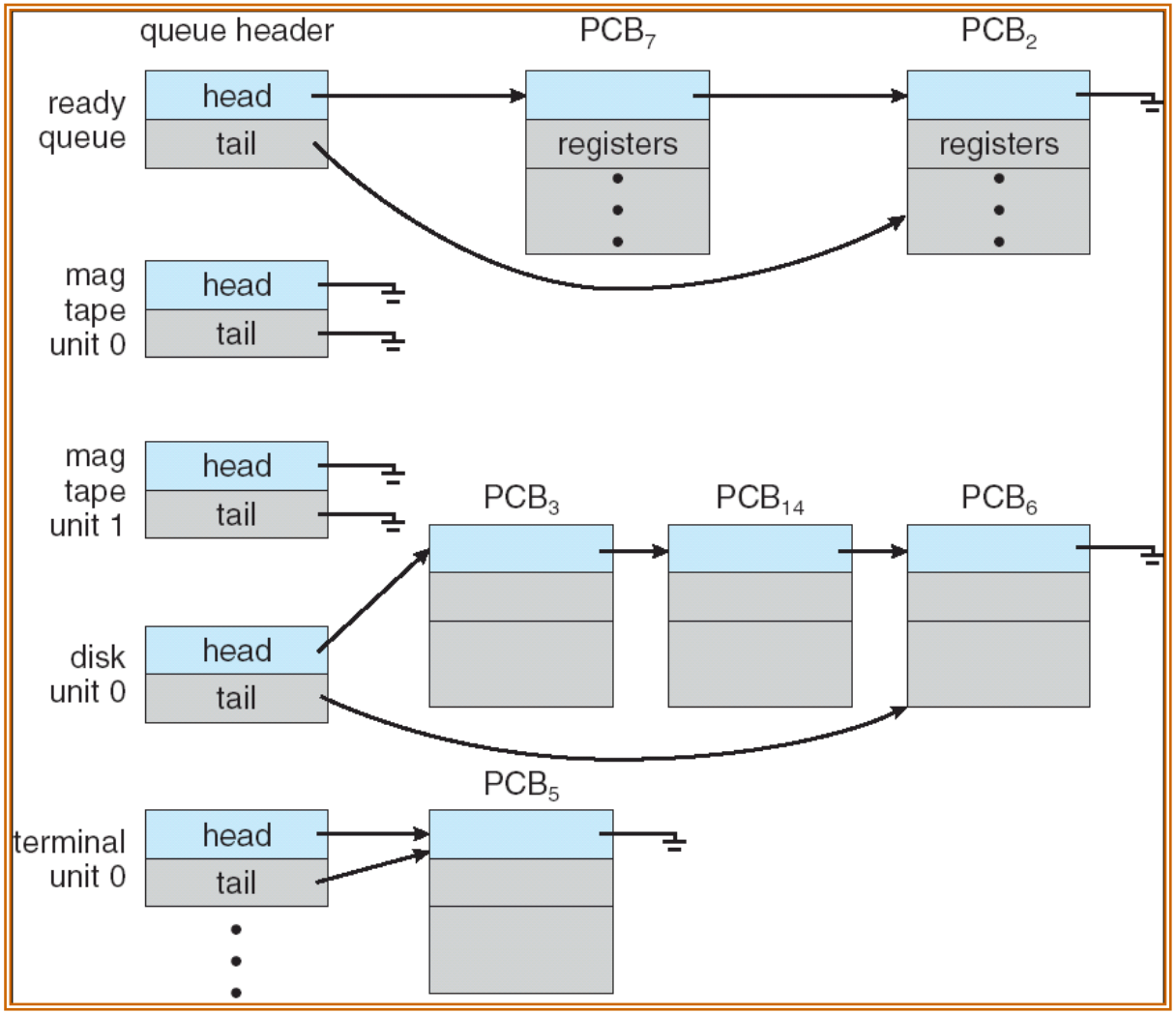
CPU Switch From Process to Process



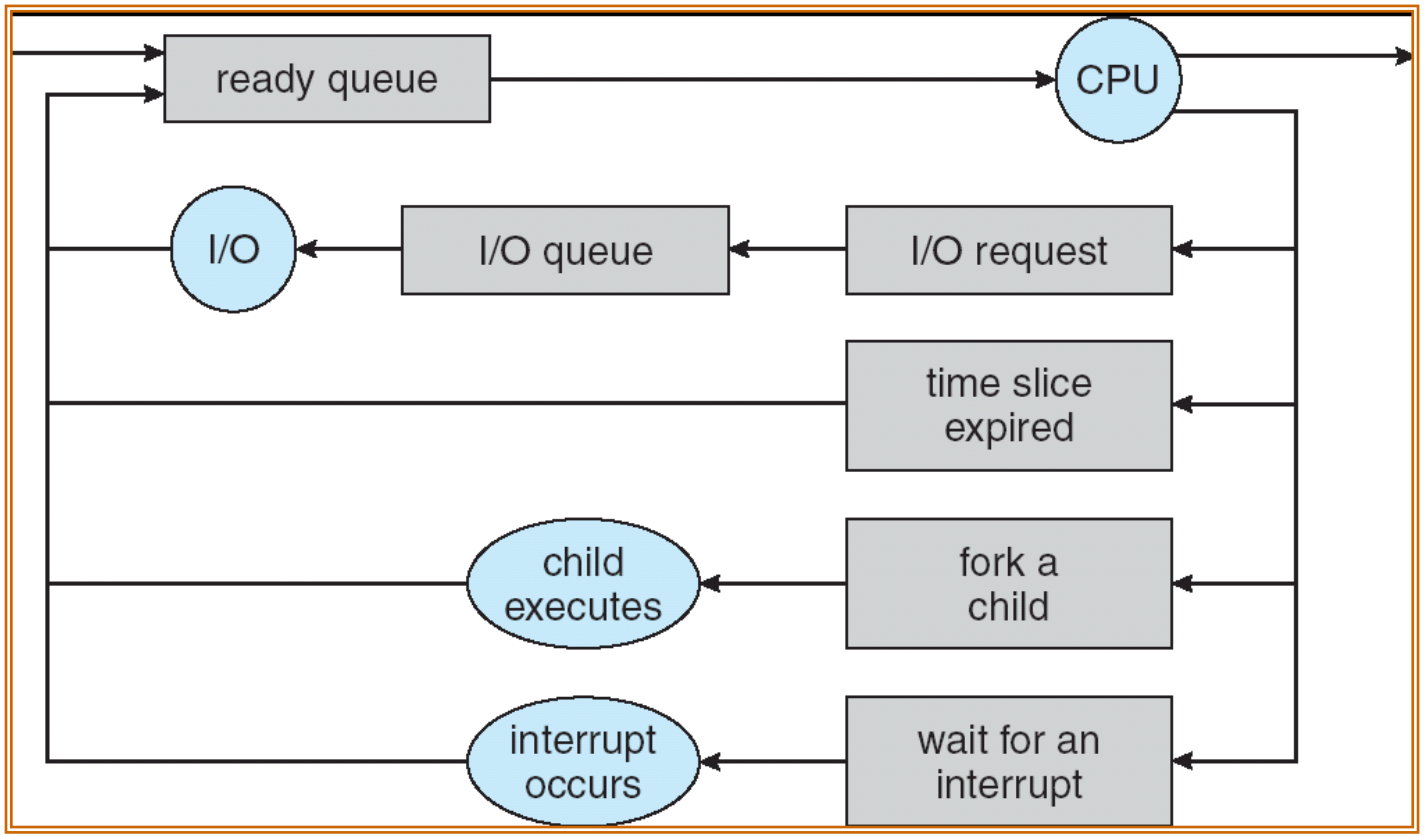
Process Scheduling Queues

- **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 And Various I/O Device Queues



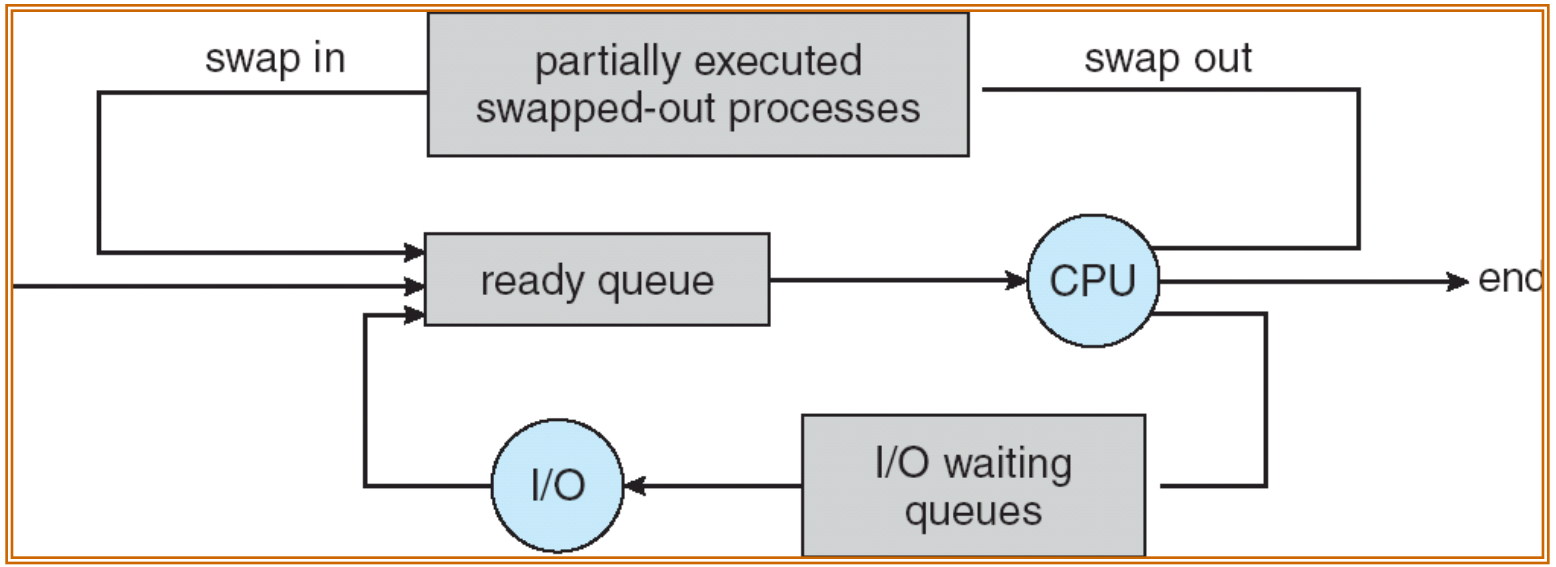
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



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*
- 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

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

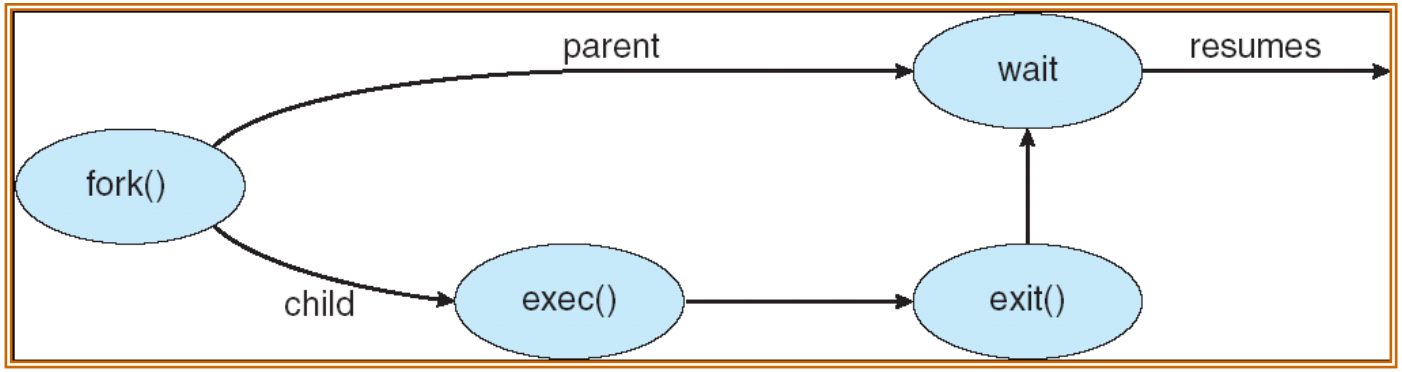
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- 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
- 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

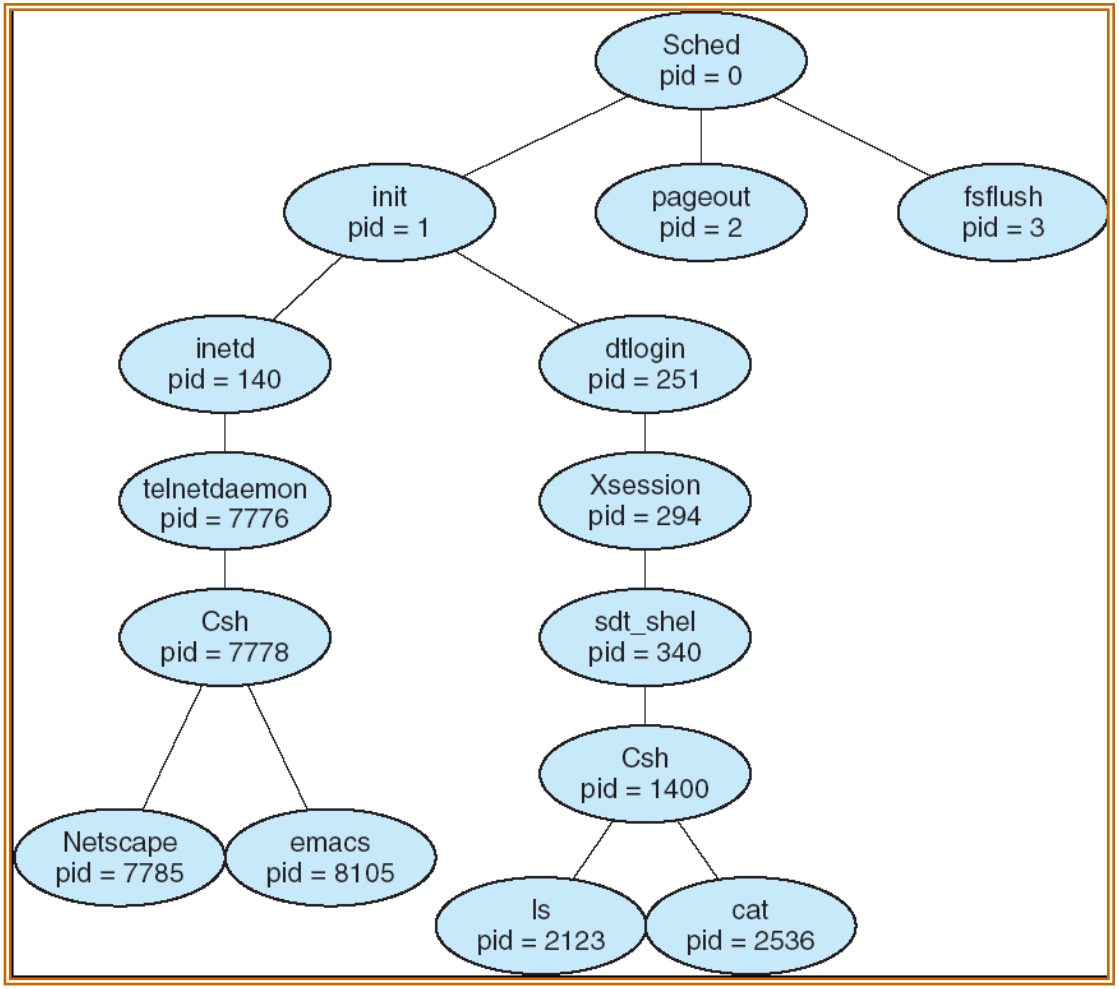
Process Creation



C Program Forking Separate Process

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

A tree of processes on a typical Solaris



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
- 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 system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

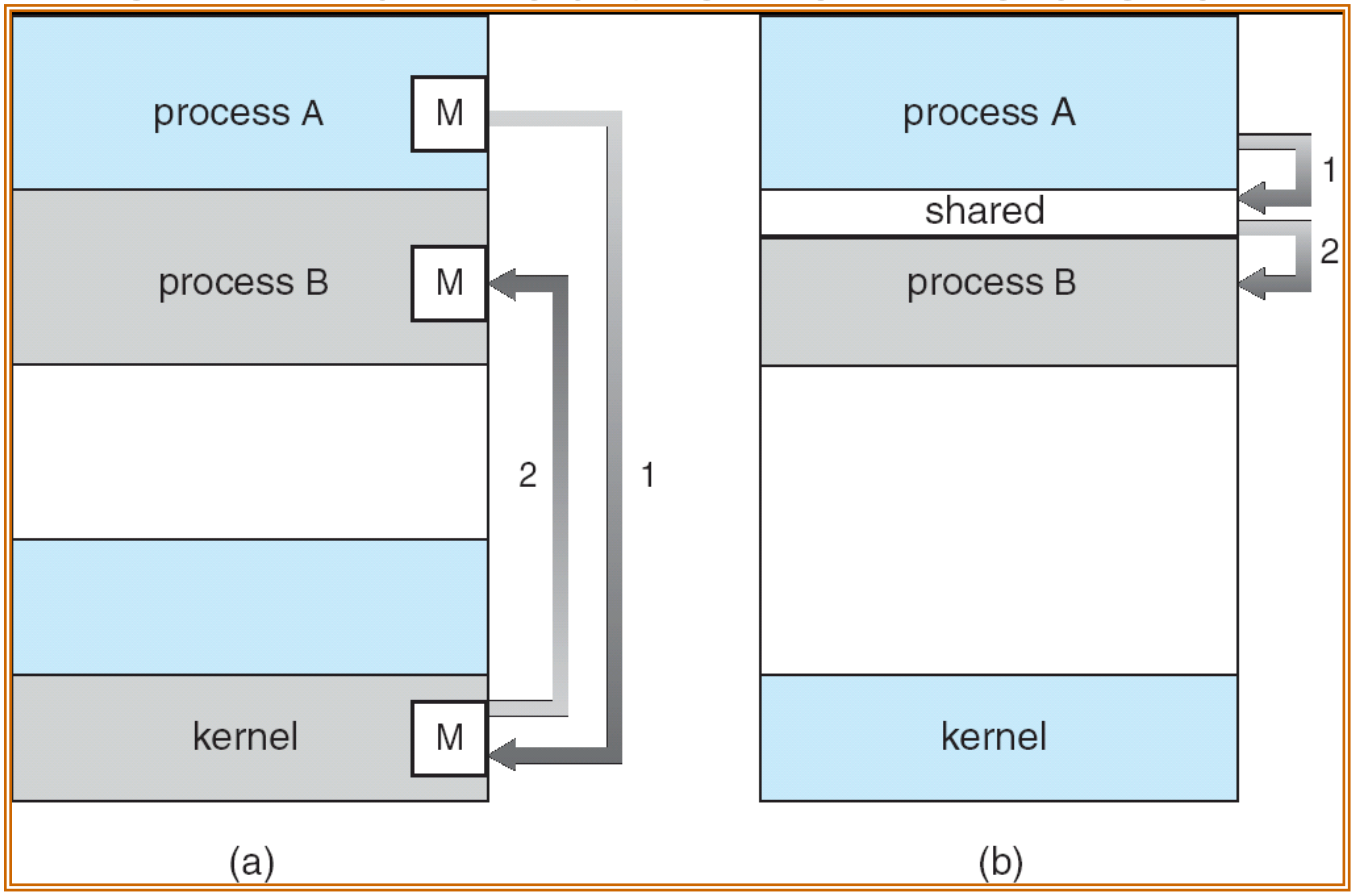
Interprocess Communication (IPC)

- 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*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Communications Models



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

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
- 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

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

- 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.

Synchronization

- Message passing may be either blocking or non-blocking
- **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

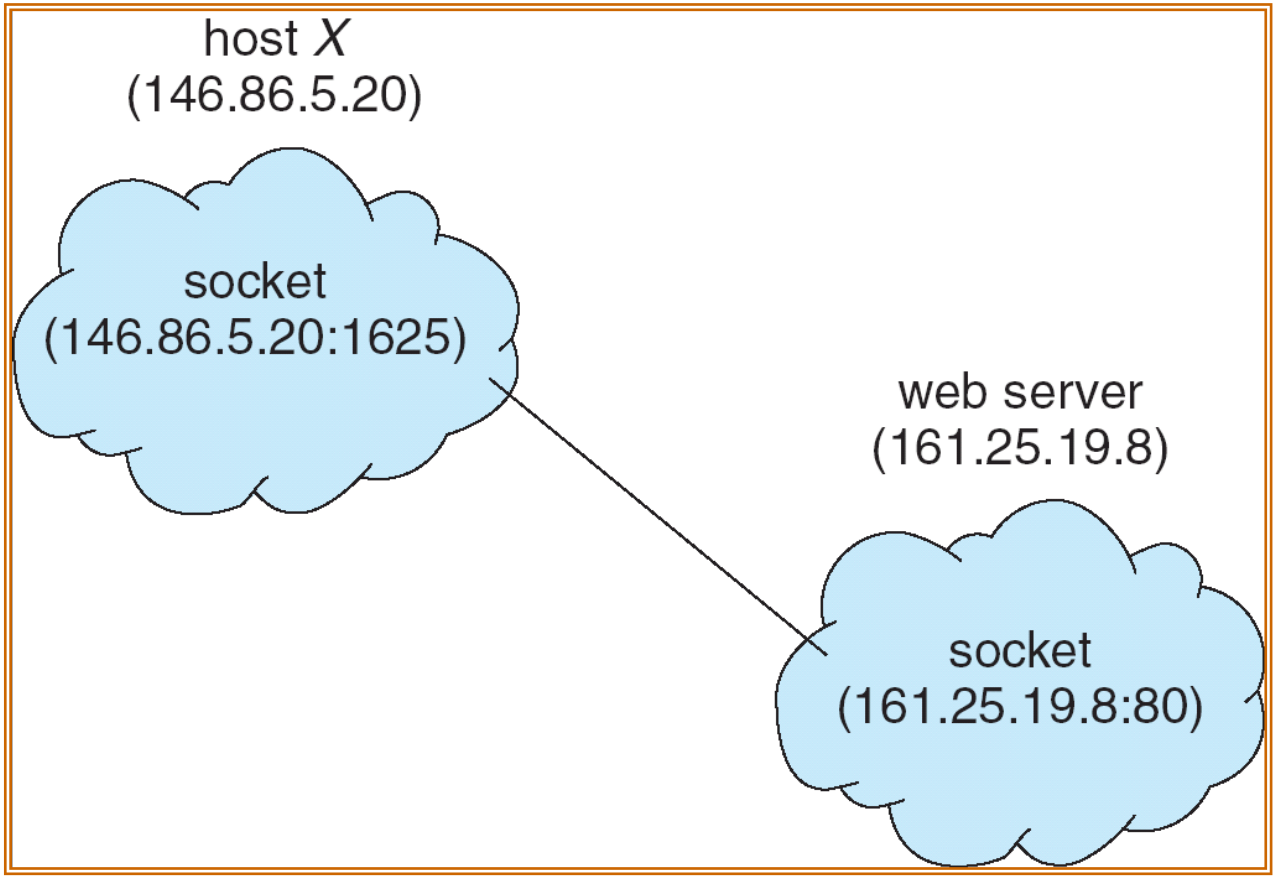
Client-Server Communication

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

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

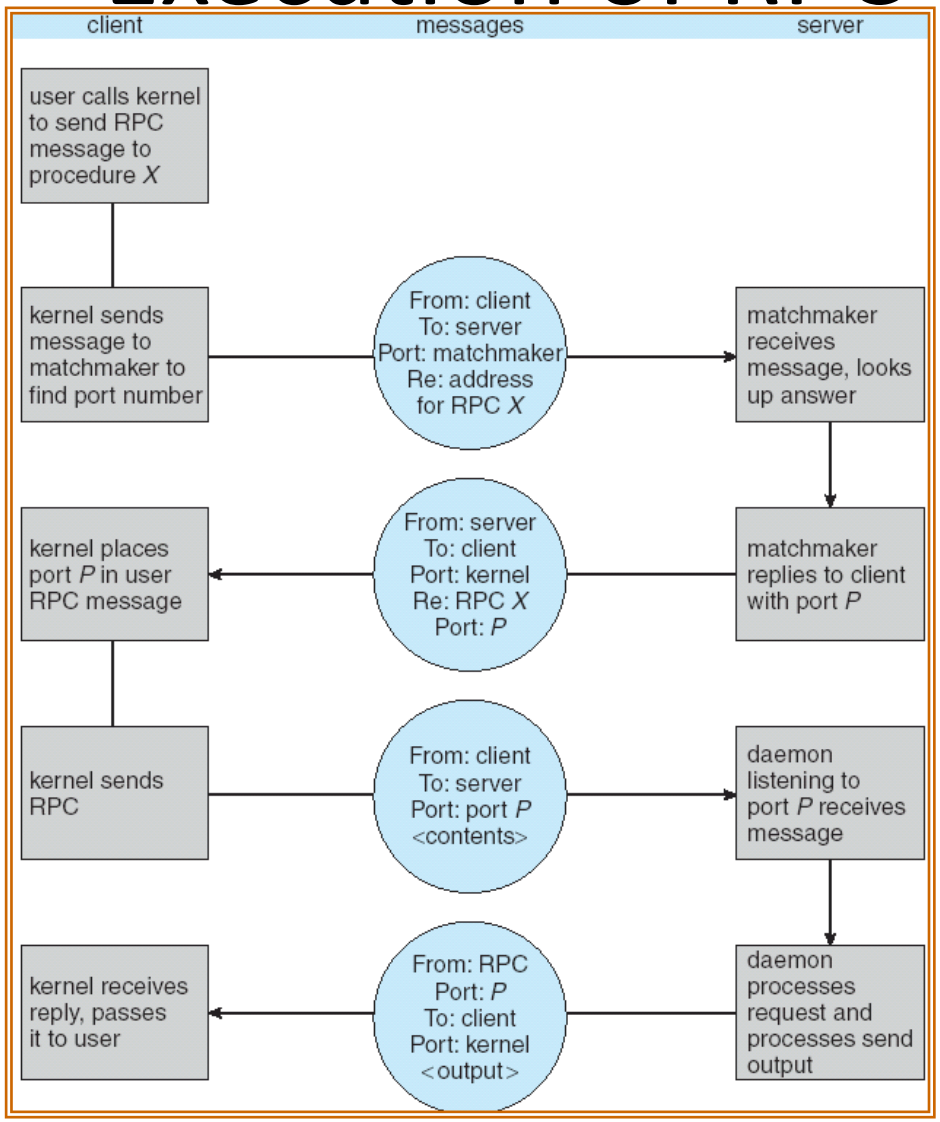
Socket Communication



Remote Procedure Calls

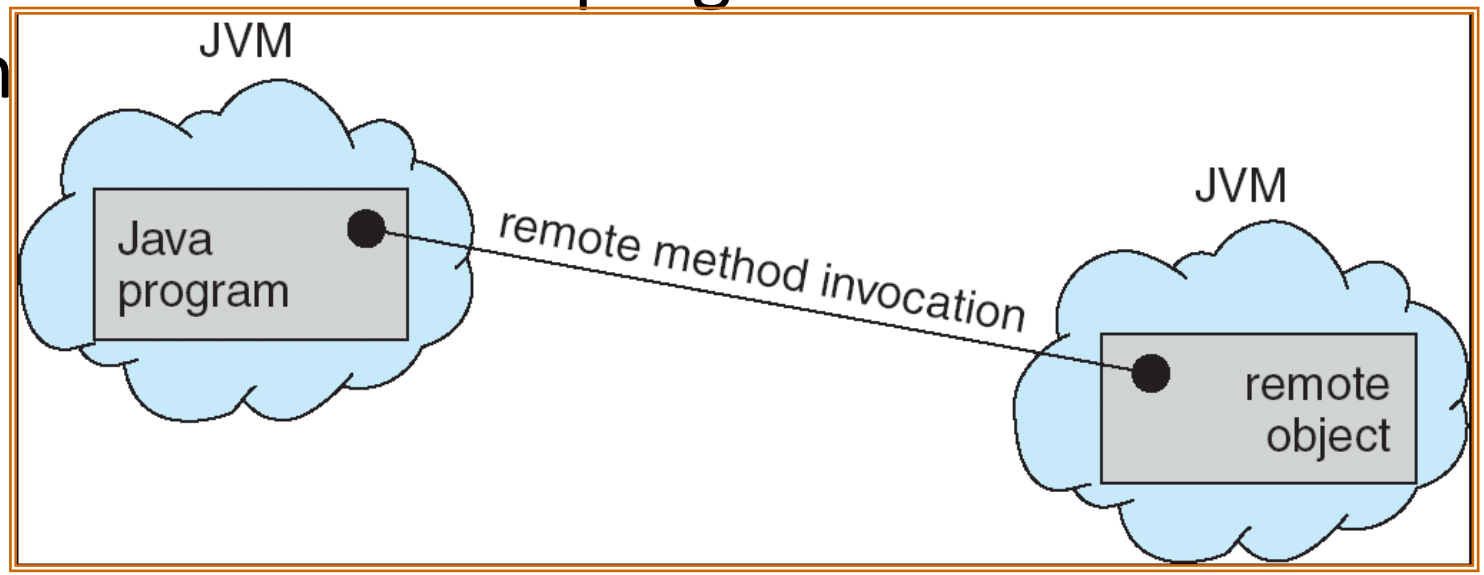
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

Execution of RPC

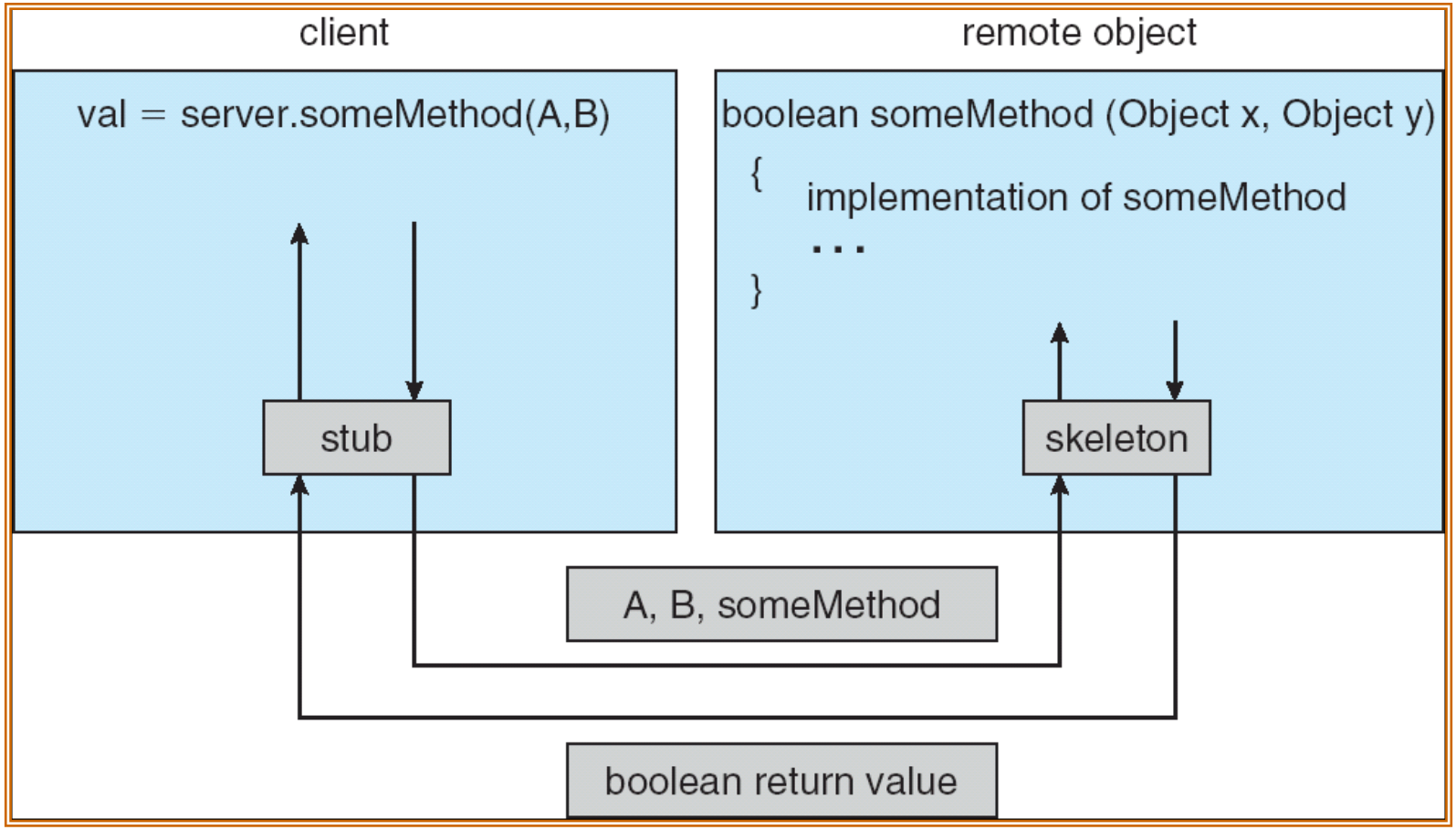


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to in



Marshalling Parameters



End of Chapter 3

Chapter 5: CPU Scheduling

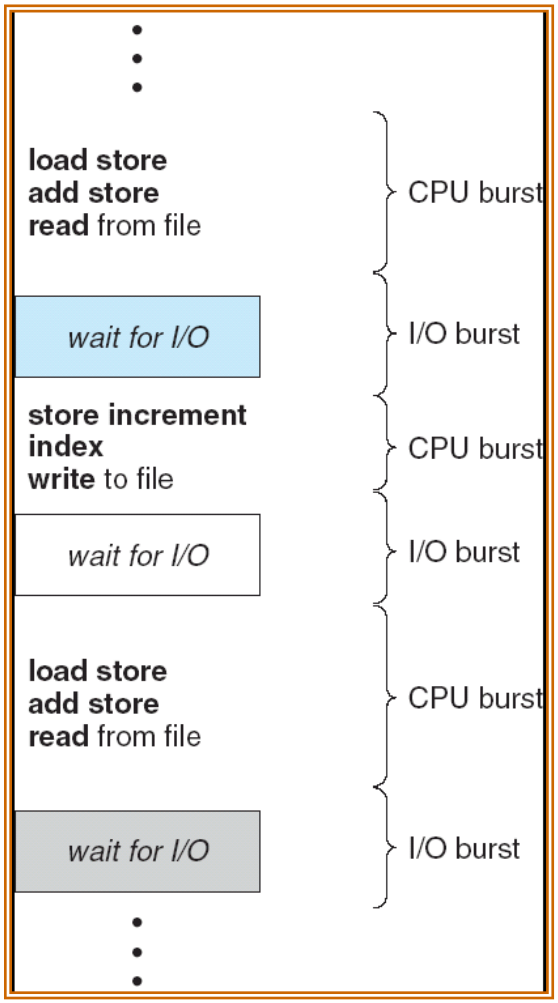
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Thread Scheduling
- Operating Systems Examples
- Java Thread Scheduling
- Algorithm Evaluation

Basic Concepts

- **Scheduling** is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. **Scheduling** refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs.
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution
- Burst time is an assumption of how long a process requires the cpu between I/O waits.It can not be predicted exactly, before a process starts.
  It means the amount of time a process uses the CPU for a single time. (A process can use the CPU several times before complete the job)
  Read more: http://wiki.answers.com/Q/What_is_CPU_burst_time#ixzz1BNfLOziS

Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

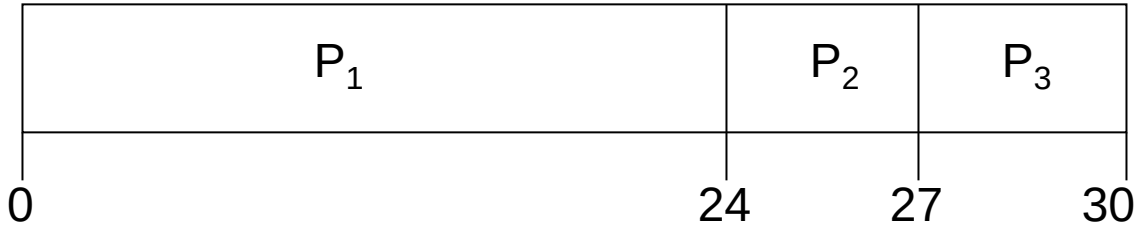
Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

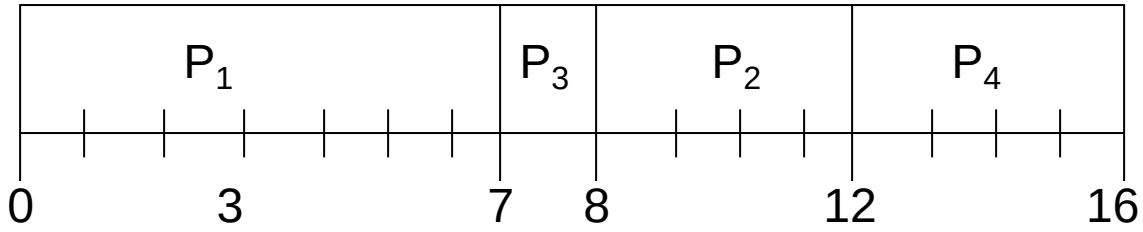
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

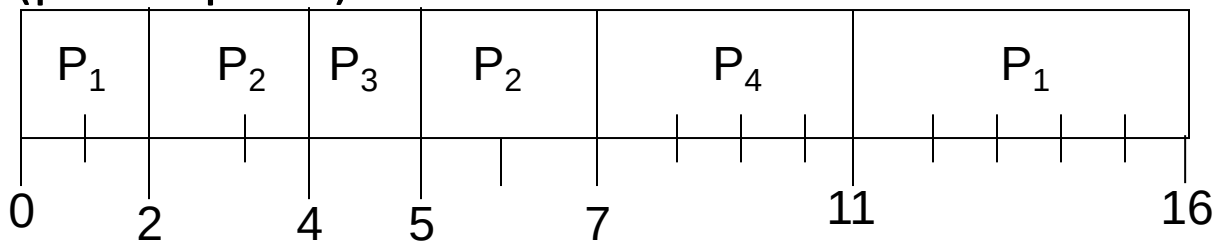


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

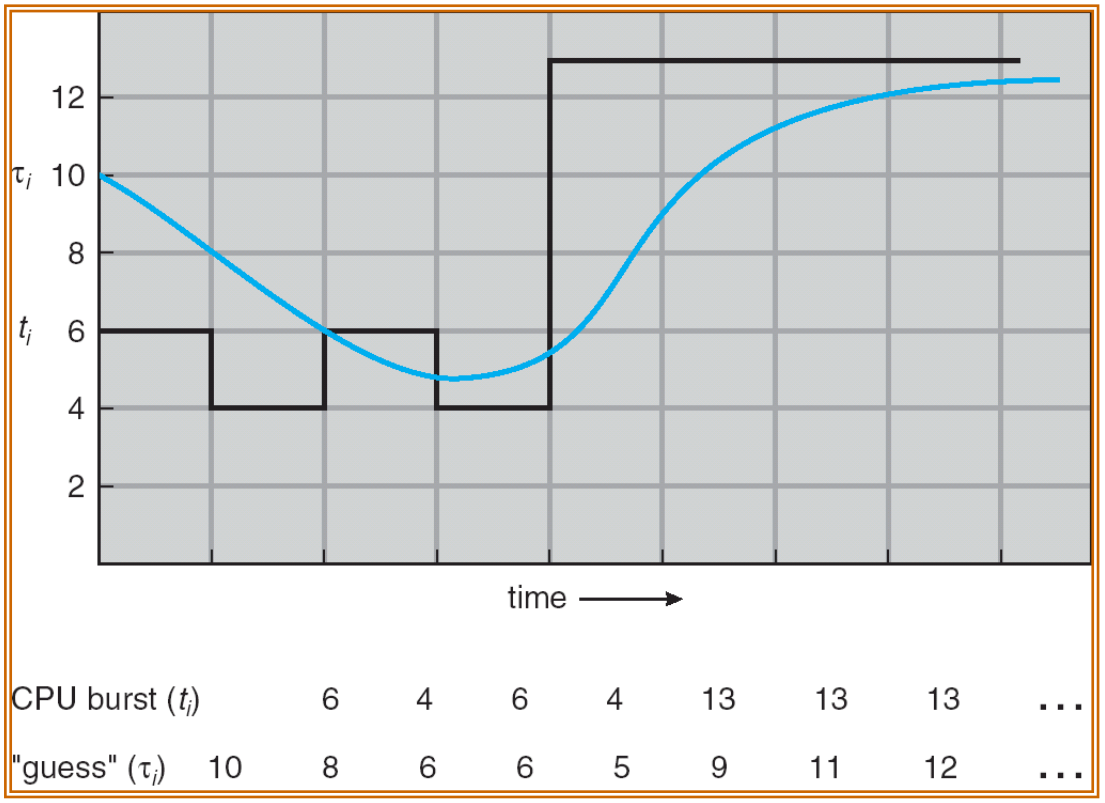
Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

Prediction of the Length of the Next CPU Burst



Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

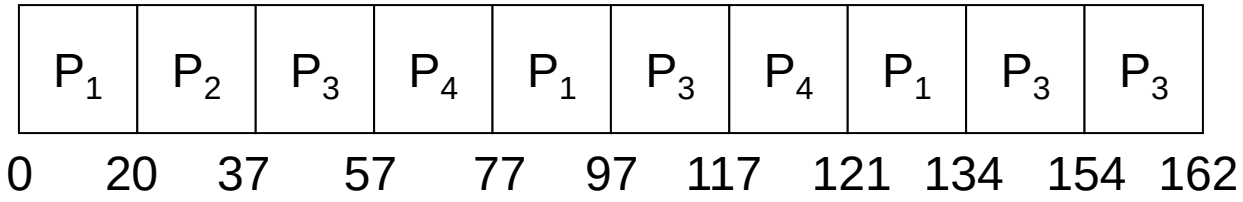
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

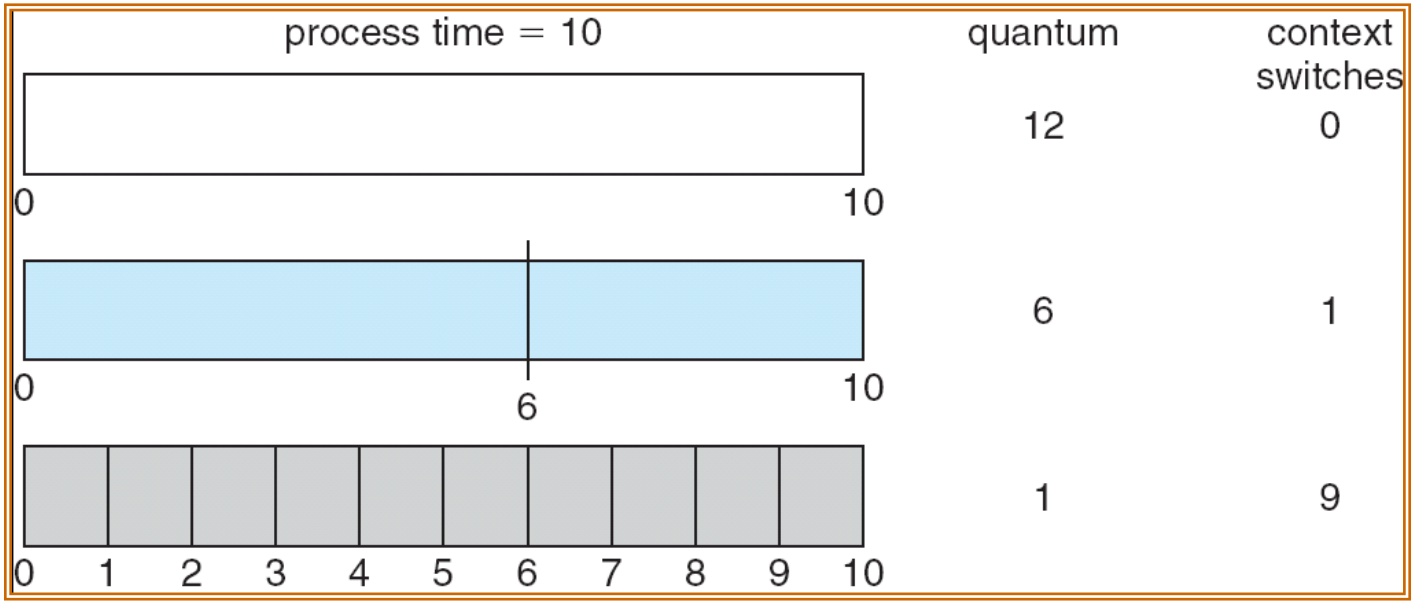
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

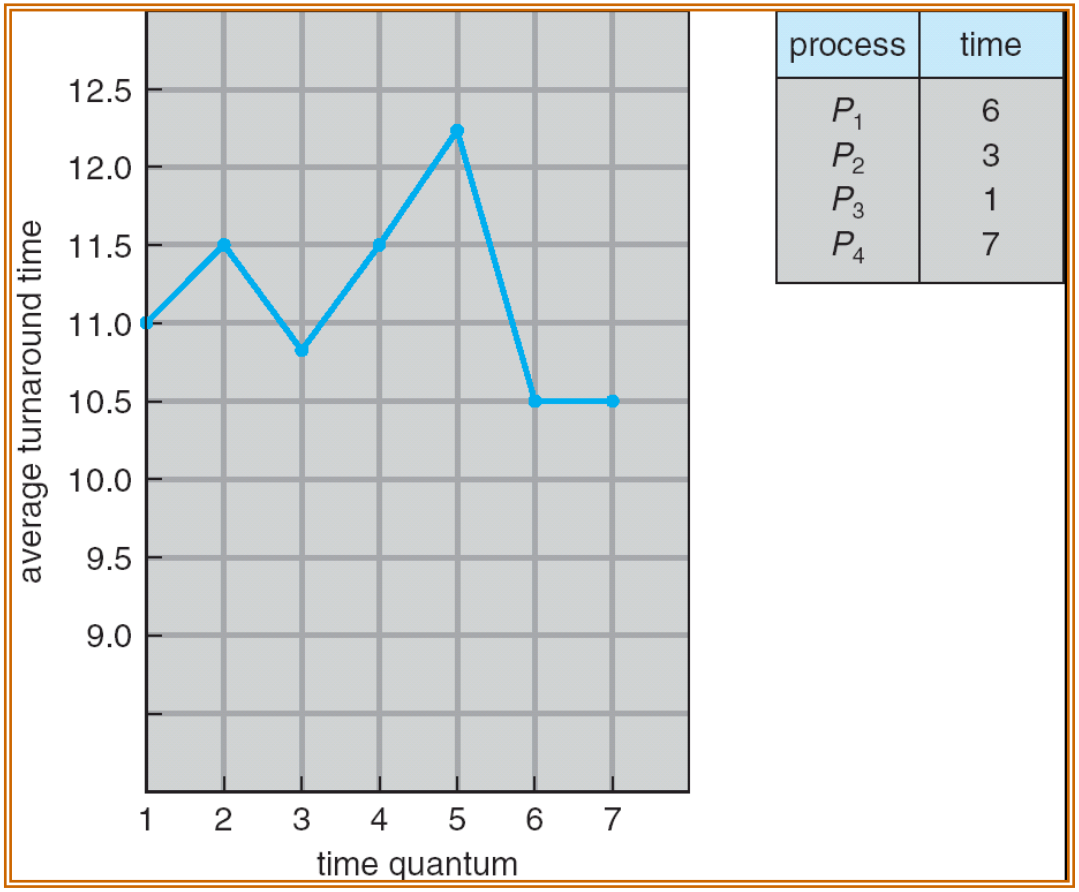


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

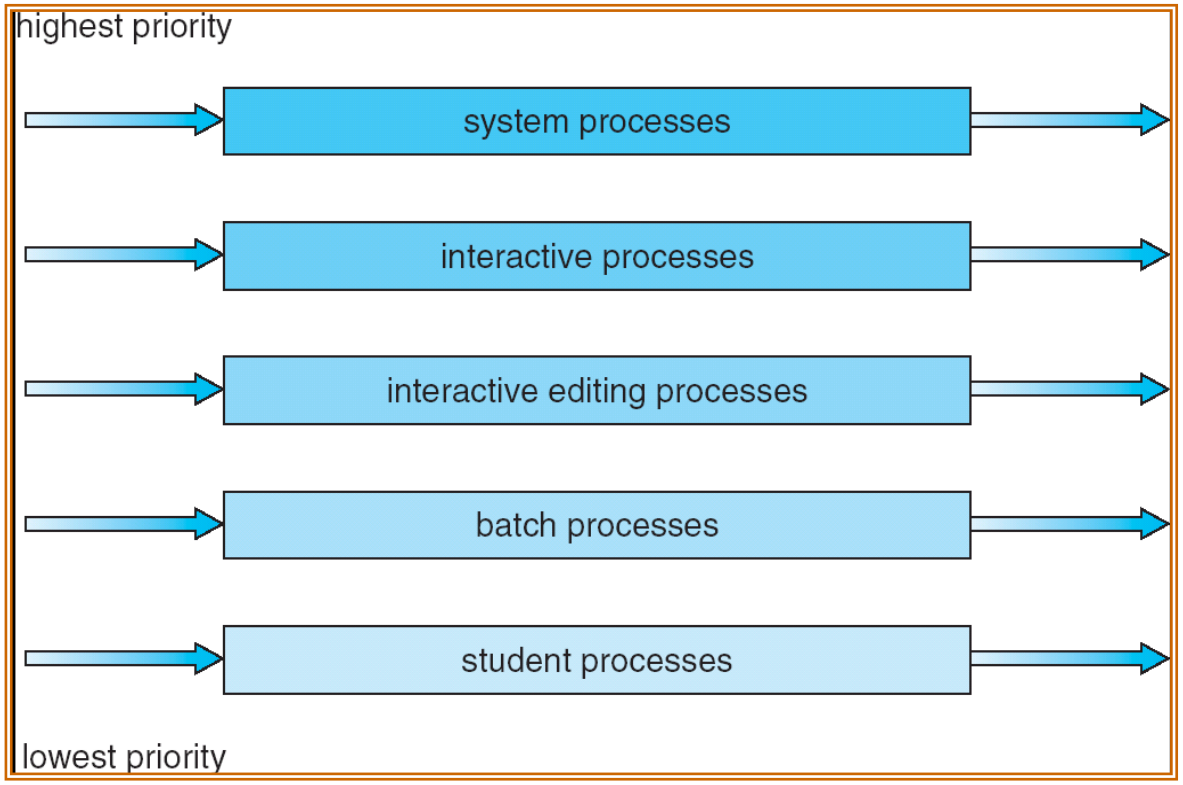


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



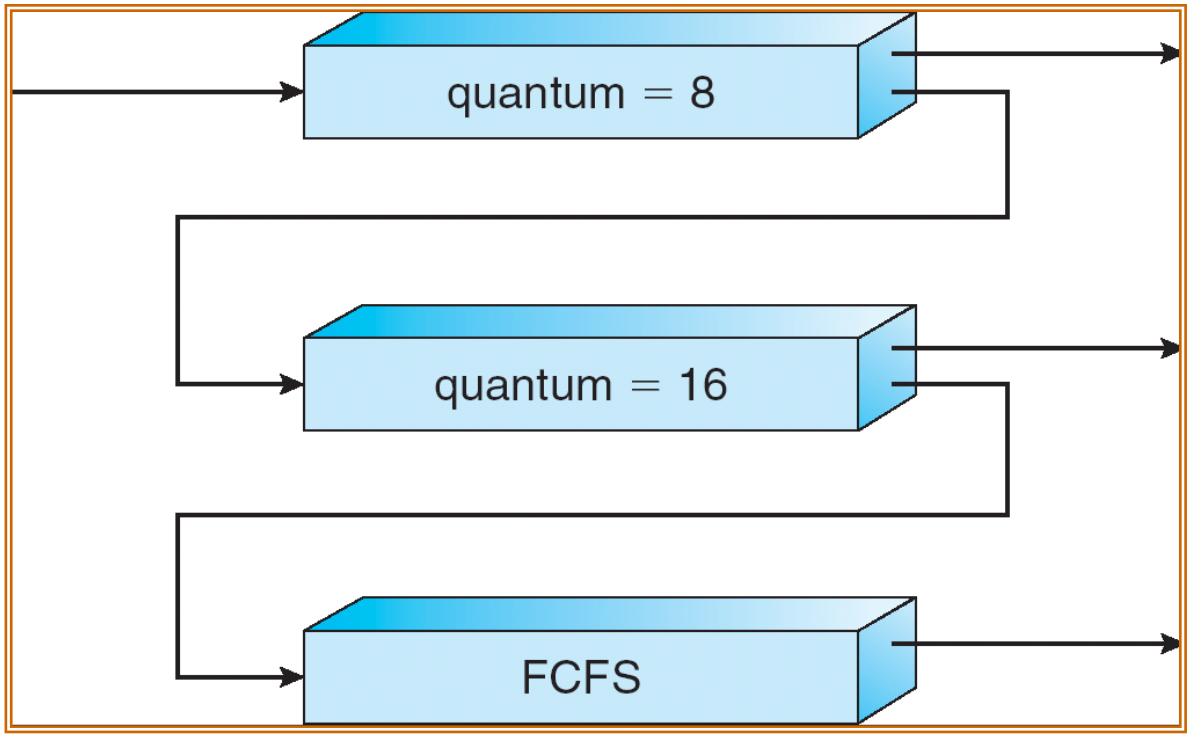
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones