Operating Systems
Module-1

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.

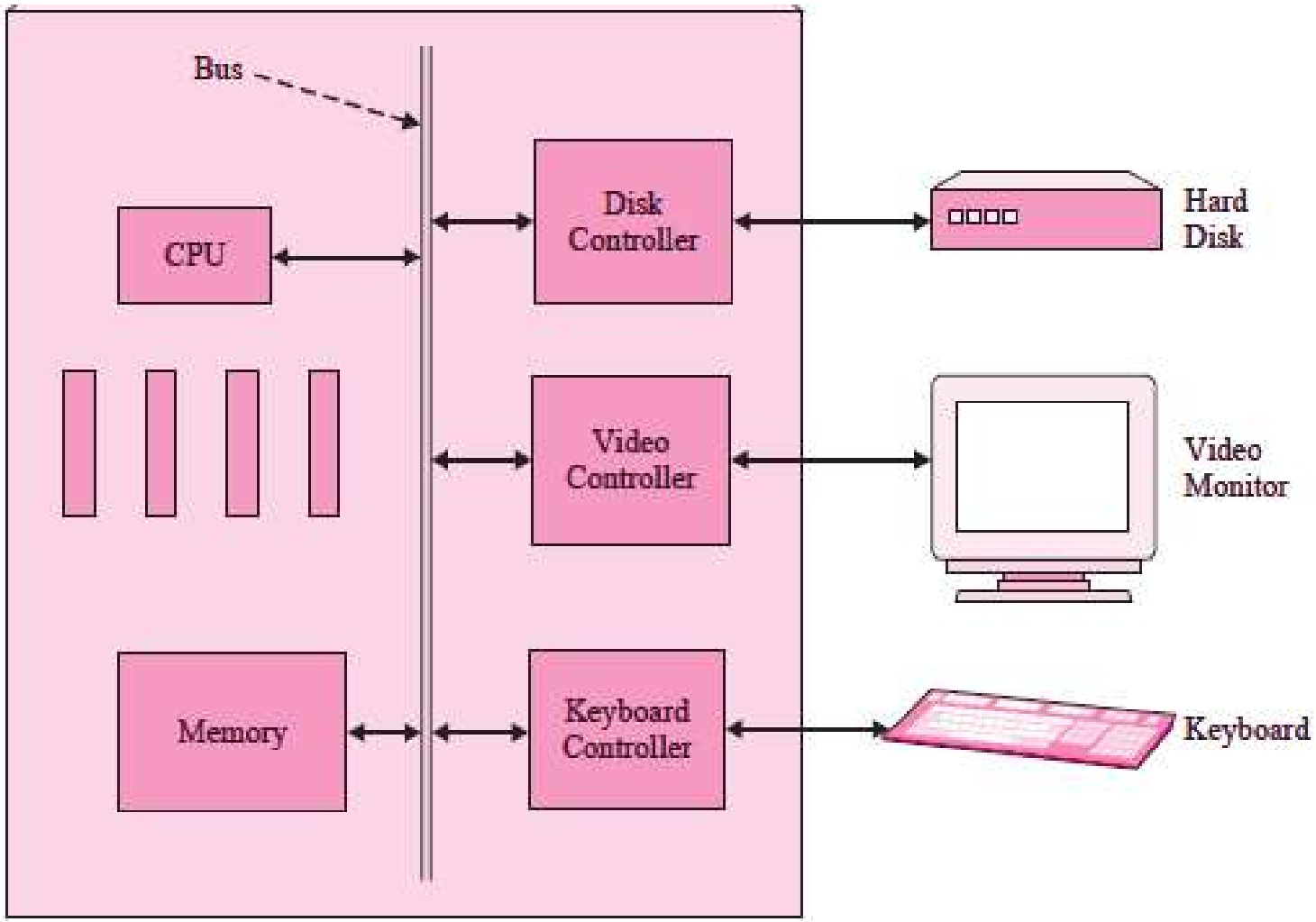
Definition

- A computer program that acts as an interface between the user and computer hardware and controls and manages the overall resources of the computer system.

A few questions…..

- Have you written any programs?
  - Is it platform-dependent or platform-independent?
- Where do you execute your programs?
- Did you use any compilers?
- Can I copy and execute my compiled code from my desktop to laptop?
- Does the mother board contain only one component?
  - If not, who is going to manage all the components?
- What is abstraction and interface?

Hardware: A very simplistic view of a small personal computer



- Hardware resources of a computer:
  - CPUs (processors)
  - Main memory and caches,
  - Secondary storage,
  - I/O devices at the lowest level,
  - Network access

  Who is going to manage these resources?
  - A User or Software?
    - Software – Operating Systems
- Why to manage these resources?

- Operating System
  - The OS is a **collection of one or more software modules** that **manages and controls the resources of a computer** or other computing or electronic device, and **gives users and programs an interface to utilize these resources**. The managed resources include memory, processor, files, input or output devices, and so on.
  - Responsibilities/ Services
    - Memory management
    - Device management
    - Processor management
    - Security
    - Error detection
    - Job accounting
    - File management

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)
  - **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.

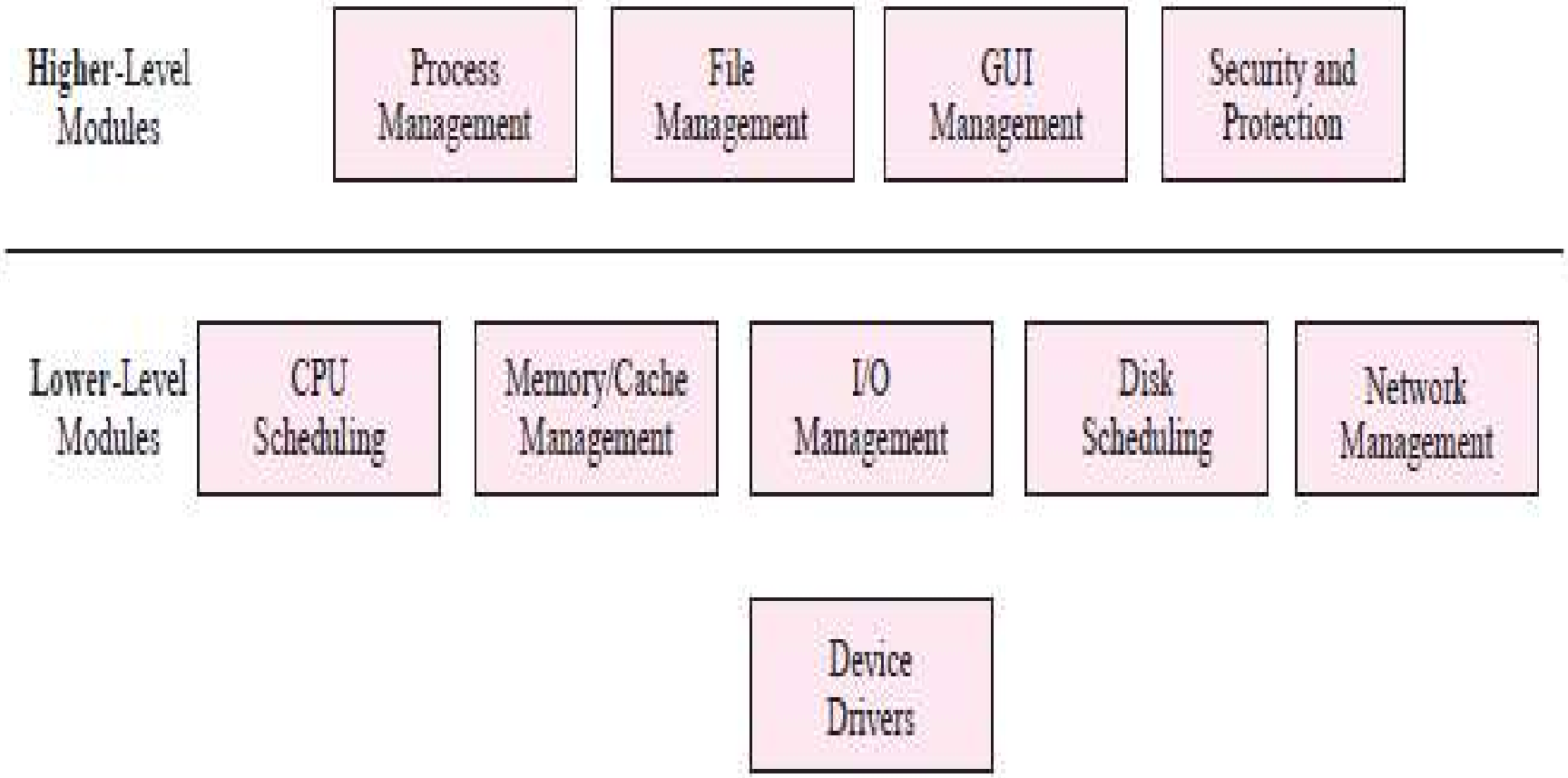
Operating System Services (Cont.)

- 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

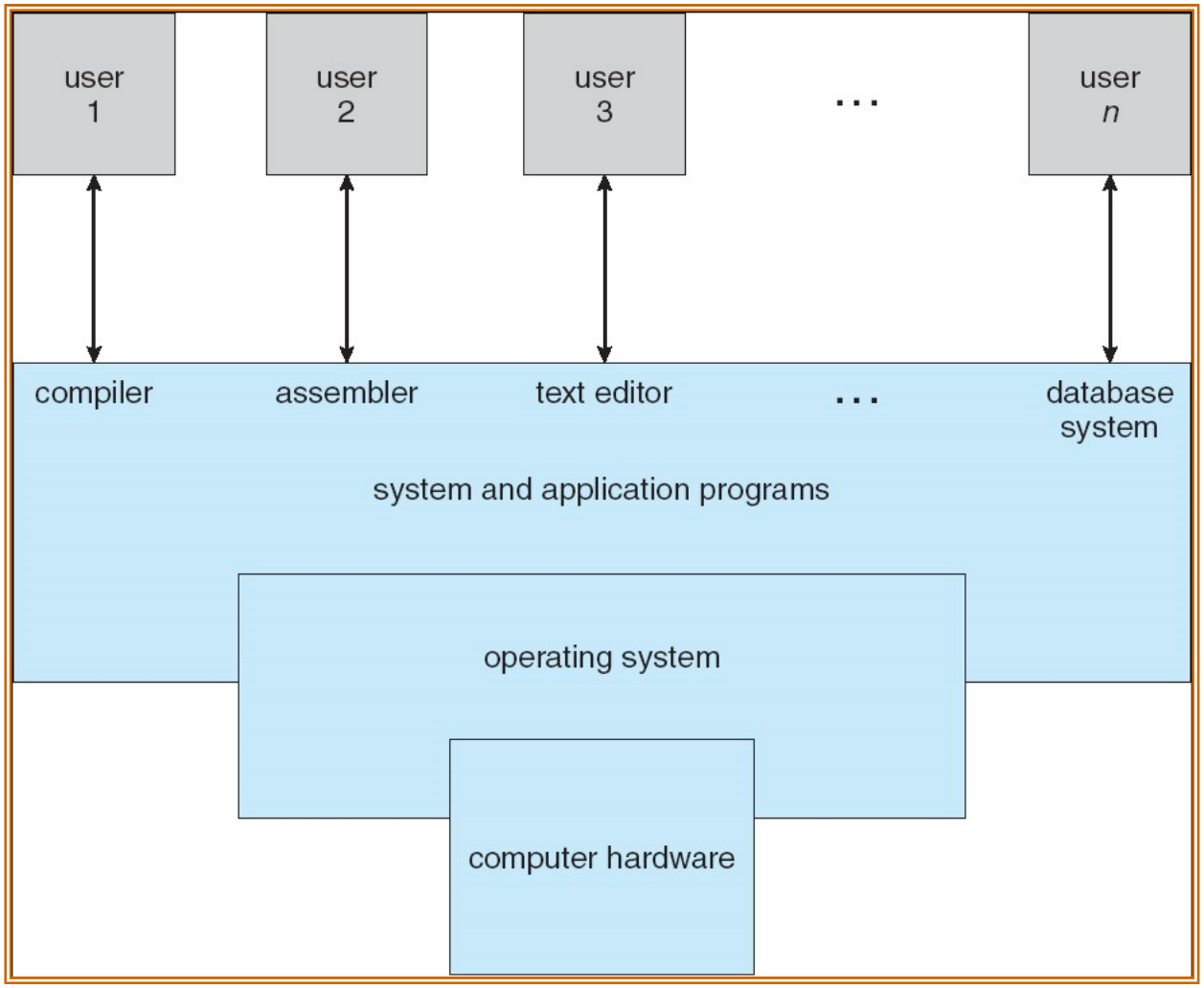
Operating System Services

- 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.

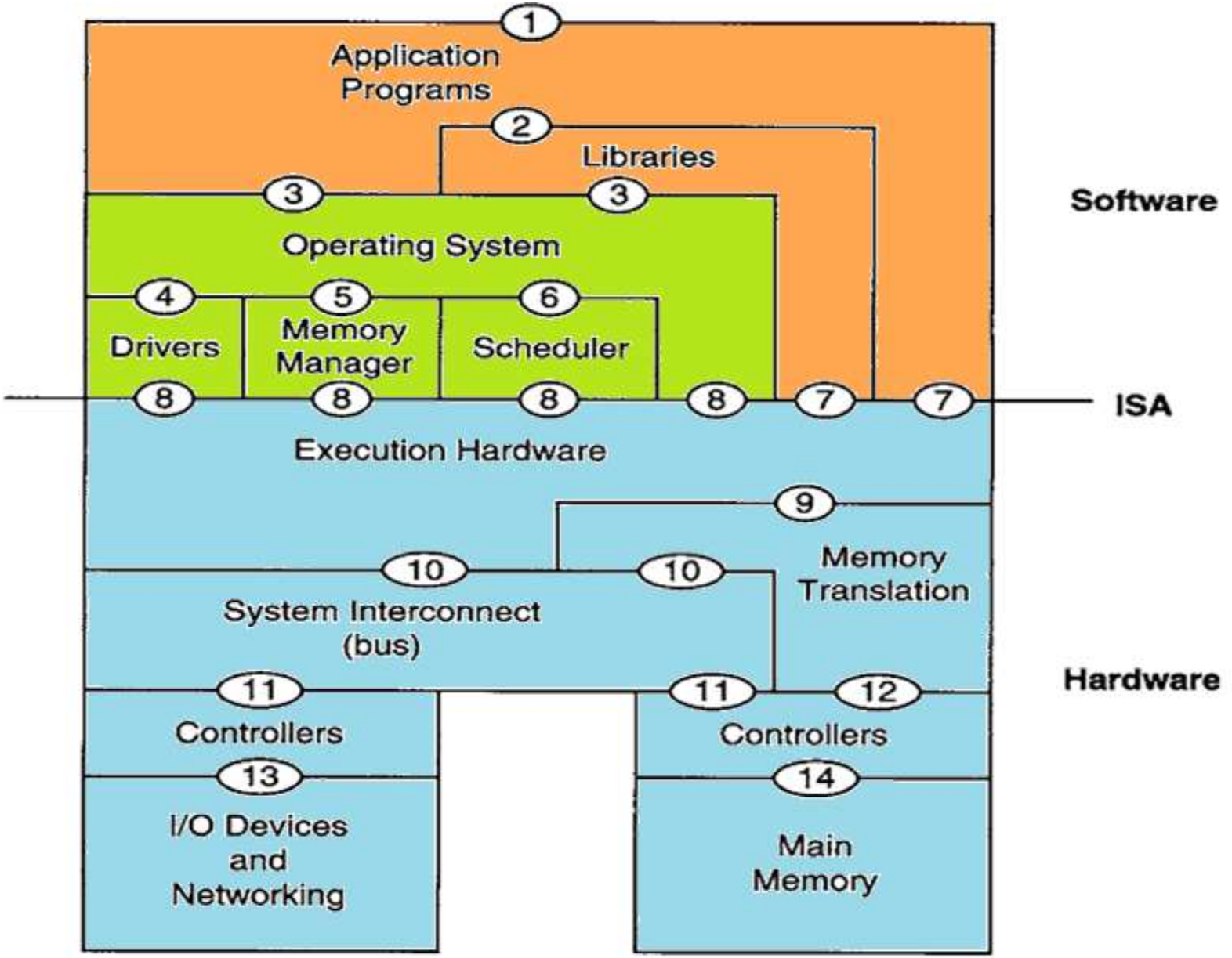
Major OS modules



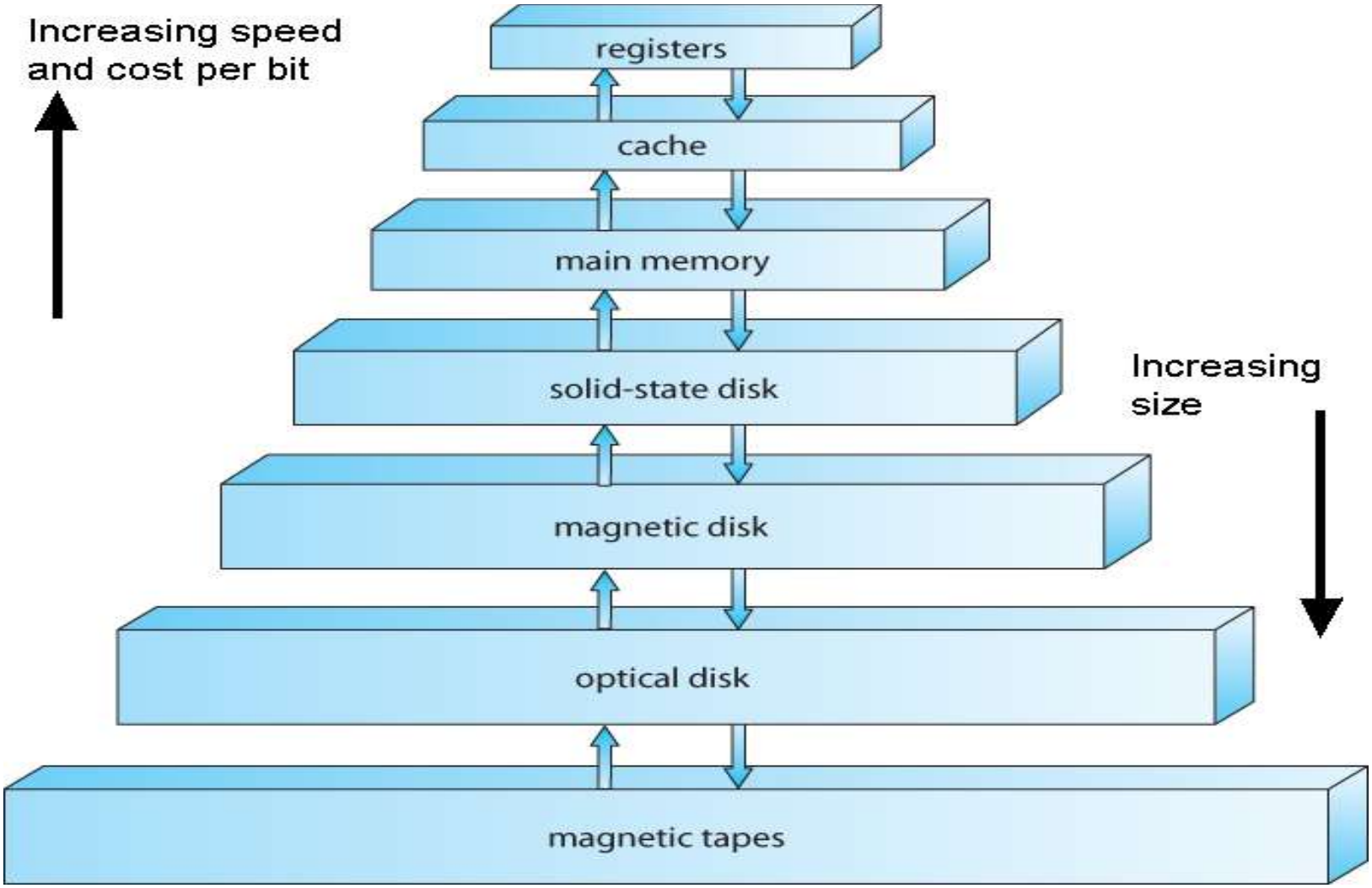
Four Components of a Computer System



Computer System Architecture: Abstraction and Interfaces



Storage Structure



Memory – Array of words

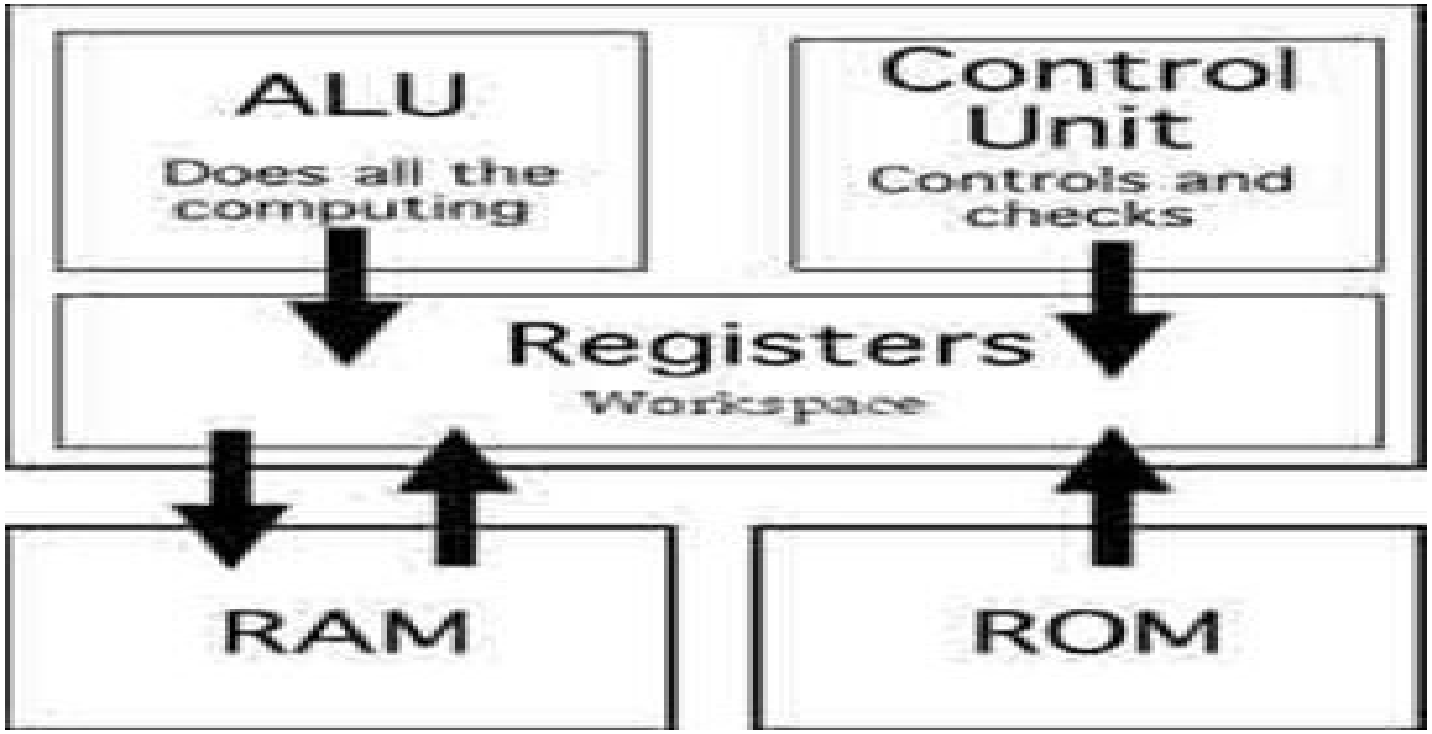
- Word - Largest size of data that can be transferred in and out of the memory.
- Interaction occurs through LOAD and STORE

Main Memory – rewritable memory (RAM)

- Implemented in a semiconductor technology called DRAM
- Volatile, very fast and expensive

Registers:

Very expensive and very fast.



Storage contd..

Secondary memory

- Electronic disk
- Magnetic disk
- Optical disk
- Magnetic tapes

Cache Memory

- Random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM.
- Most frequently used instructions are stored in the cache.

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.

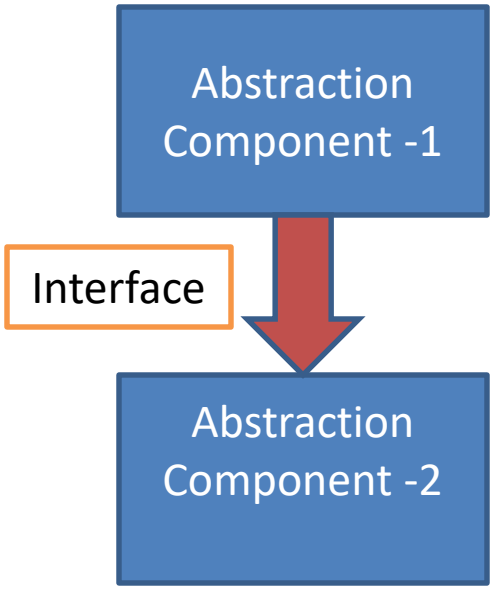
- This abstraction can be used to hide certain hardware details from users and applications. Thus, changes in the hardware are not seen by the user (even though the OS must accommodate them).

- This is particularly advantageous for vendors that want offer a consistent OS interface across an entire line of hardware platforms.
-

- 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.

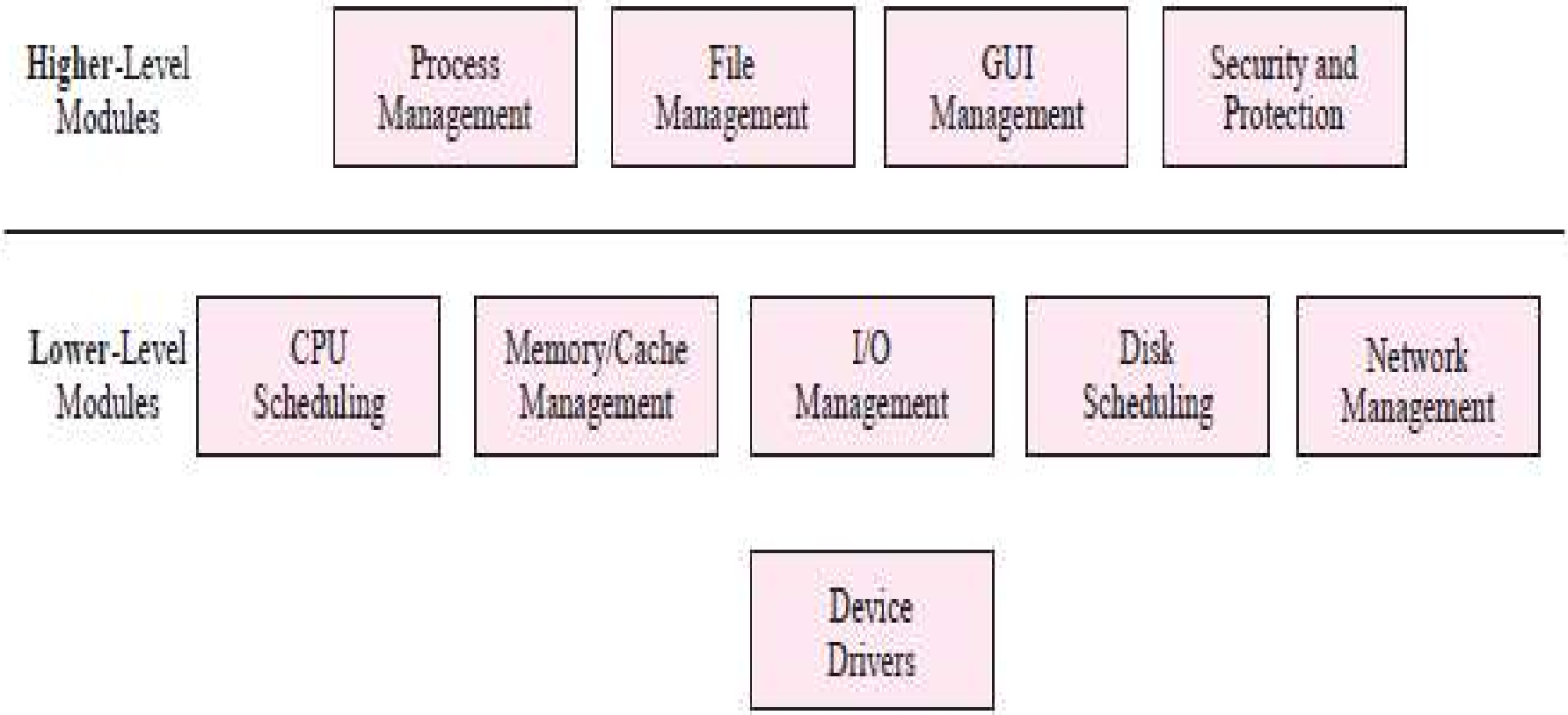
- Unix, and increasingly Windows NT, take this abstraction even further. From a user and application programmer standpoint, Unix is Unix regardless of the CPU make and model.

- Abstraction
  - Hide the implementation details of a component
  - Freedom to modify the implementation without affecting other underlying modules
  - Ex: int add(int a, int b) {
    return a + b; // Implementation details
    }
- Interface
  - To hide the details of lower level components
  - To enable the communication / data passing between the components without knowing details of their implementation
  - Ex: printf("%d, %d", add(5,10), add(10,10))



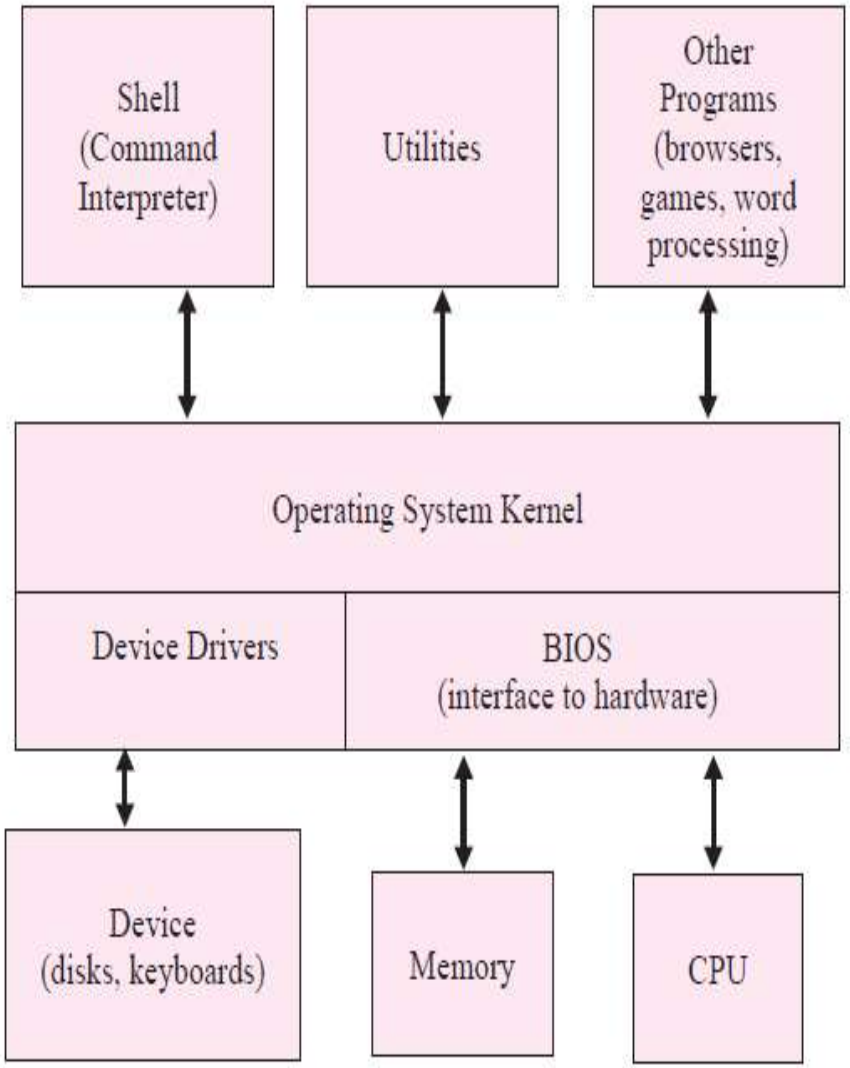
Take away:
If you introduce an interface,
you want to hide the implementation details below it

Major OS modules



Each of the responsibilities is implemented as separate Modules (**abstraction**) which are combined together through appropriate **interfaces** to formulate an Operating System software

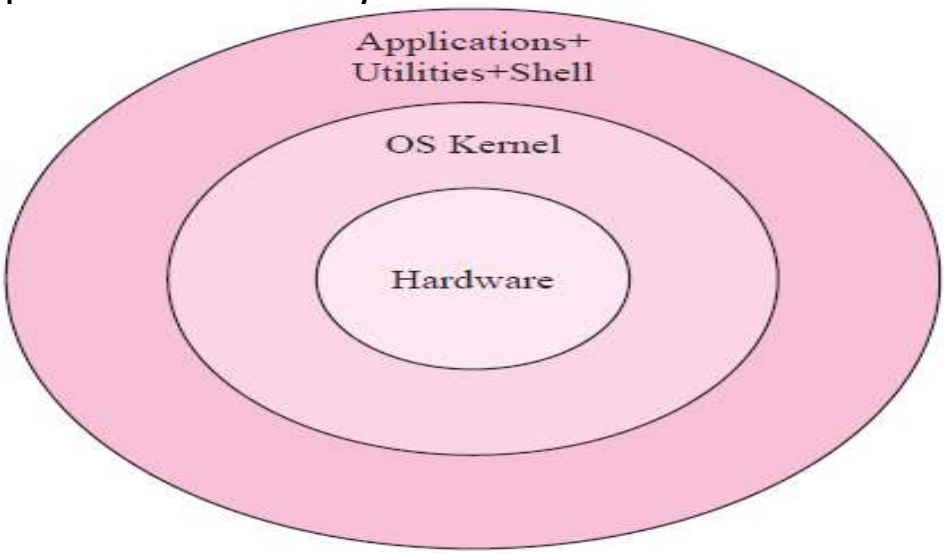
OS in relationship to hardware



Shell: programs that are not part of the OS core (or kernel), but work closely with the kernel to provide ease of use or access to system information. A **shell** or **command interpreter** is an example of a utility

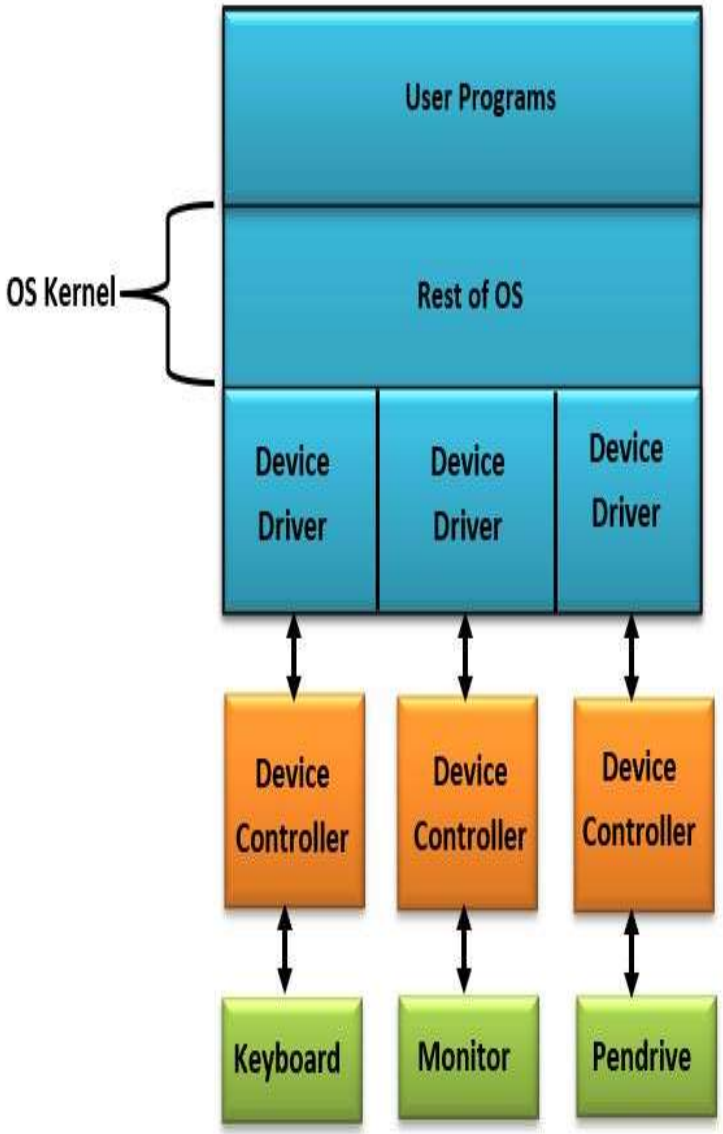
Service: Services are functions that the OS kernel provides to users, mostly through APIs via OS calls. Ex: file manipulation services (create, read, copy), memory allocation services (get, free)

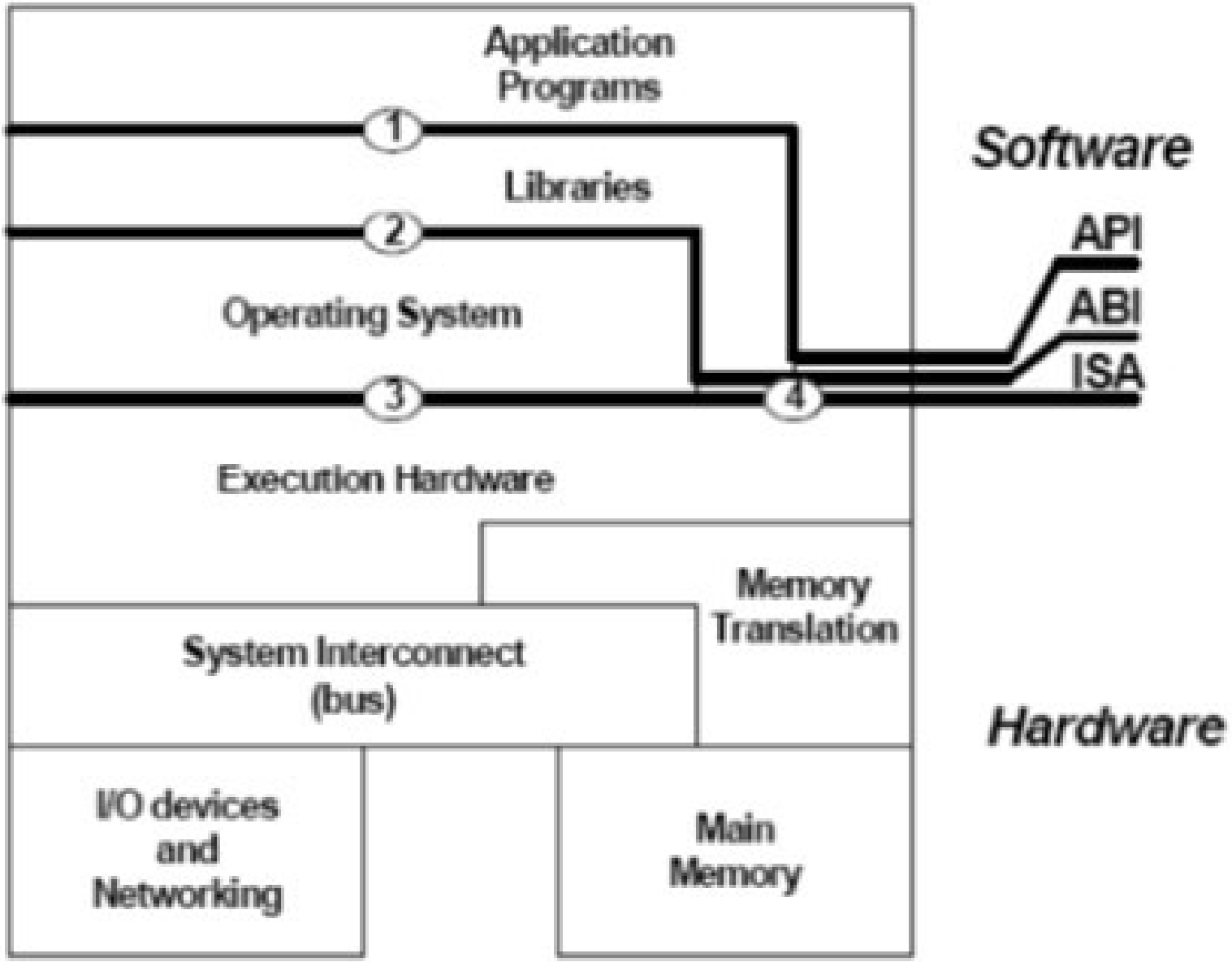
Kernel: refers to that part of the OS that implements basic functionality and is always present in memory



Layered or Levels approach

- Role of device, device controller and device driver
  - Device (Hardware)
    - A device is a piece of hardware connected tc the main computer system hardware
      - Hard disks, DVDs, and video monitors
  - Device Controller (Hardware)
    - Usually, every device has a special electronic hardware interface, called a **device controlle** which helps connect a device or a group of similar devices to a computer system throug **bus.**
      - hard disk controllers and video monitor controllers
      - converts a serial bit stream to block of bytes and perform error correction as required
      - Record and save the data
  - Device driver (Software)
    - A device driver is a software routine that is p of the OS, and is used to communicate with control a device through its device controller
      - works as a translator between the hardware device and the application or the operating system that uses it




Application
Programs
1
Libraries
2
Operating System
3
4
Execution Hardware
Memory
Translation
System Interconnect
(bus)
I/O devices
and
Networking
Main
Memory
Software
API
ABI
ISA
Hardware

What is the role of the OS?

Role #1: 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

What is the role of the OS?

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

Advantages of resource coordinator

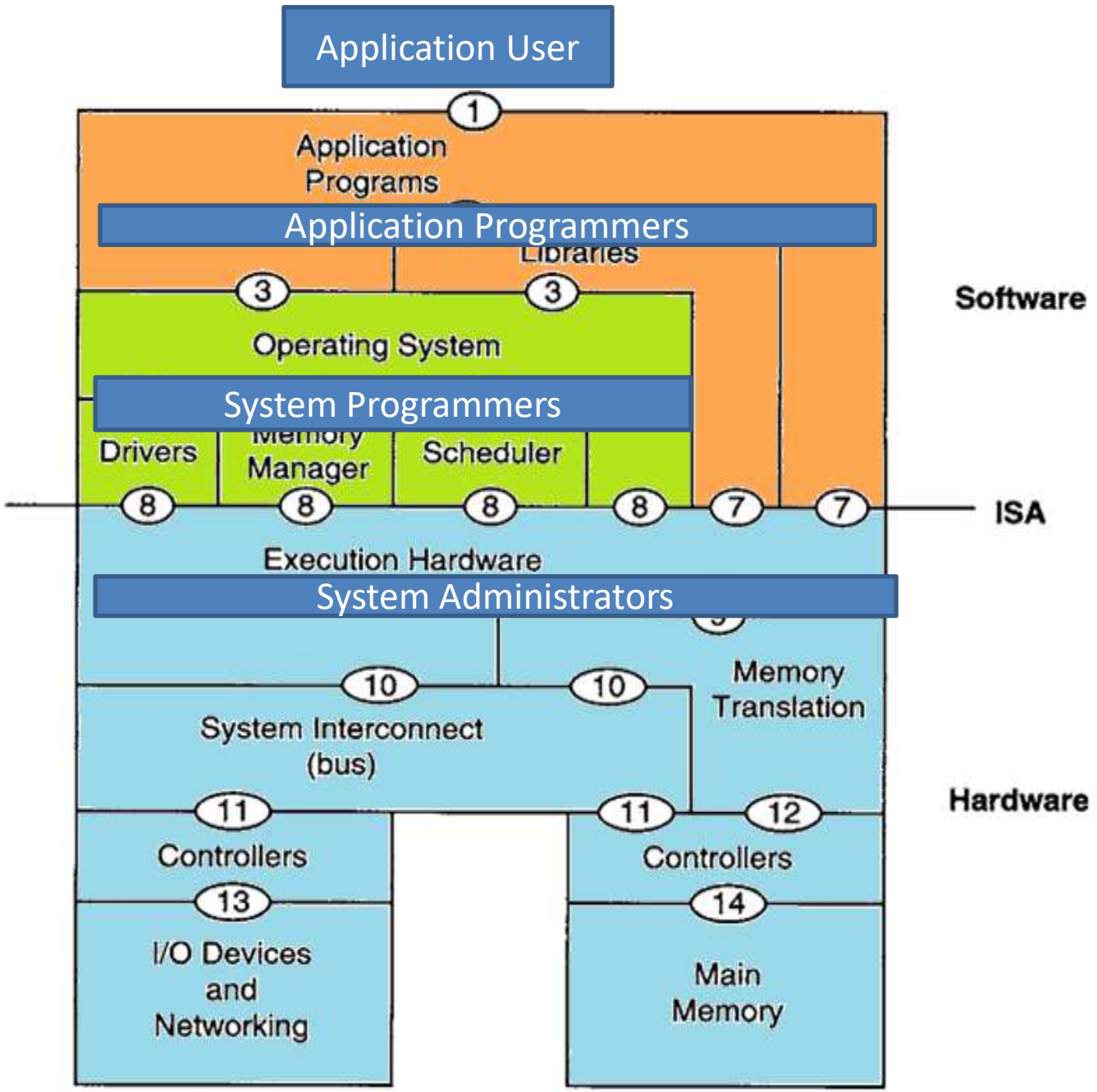
- Virtualize resources so multiple users or applications can share
- Protect applications from one another
- Provide efficient and fair access to resources

- Desirable qualities of an OS
  - Certain OS parts those handle the user and application program interaction should never be exposed to the users
    - Abstraction and Interface
  - OSs should provide new features and be easier to work with.
    - Updated periodically
  - OSs must provide support for old features
    - Backward compatibility
  - OSs must be able to connect, adapt and handle new devices that are not yet available and not even have been thought of when the OS was created
    - Extendable

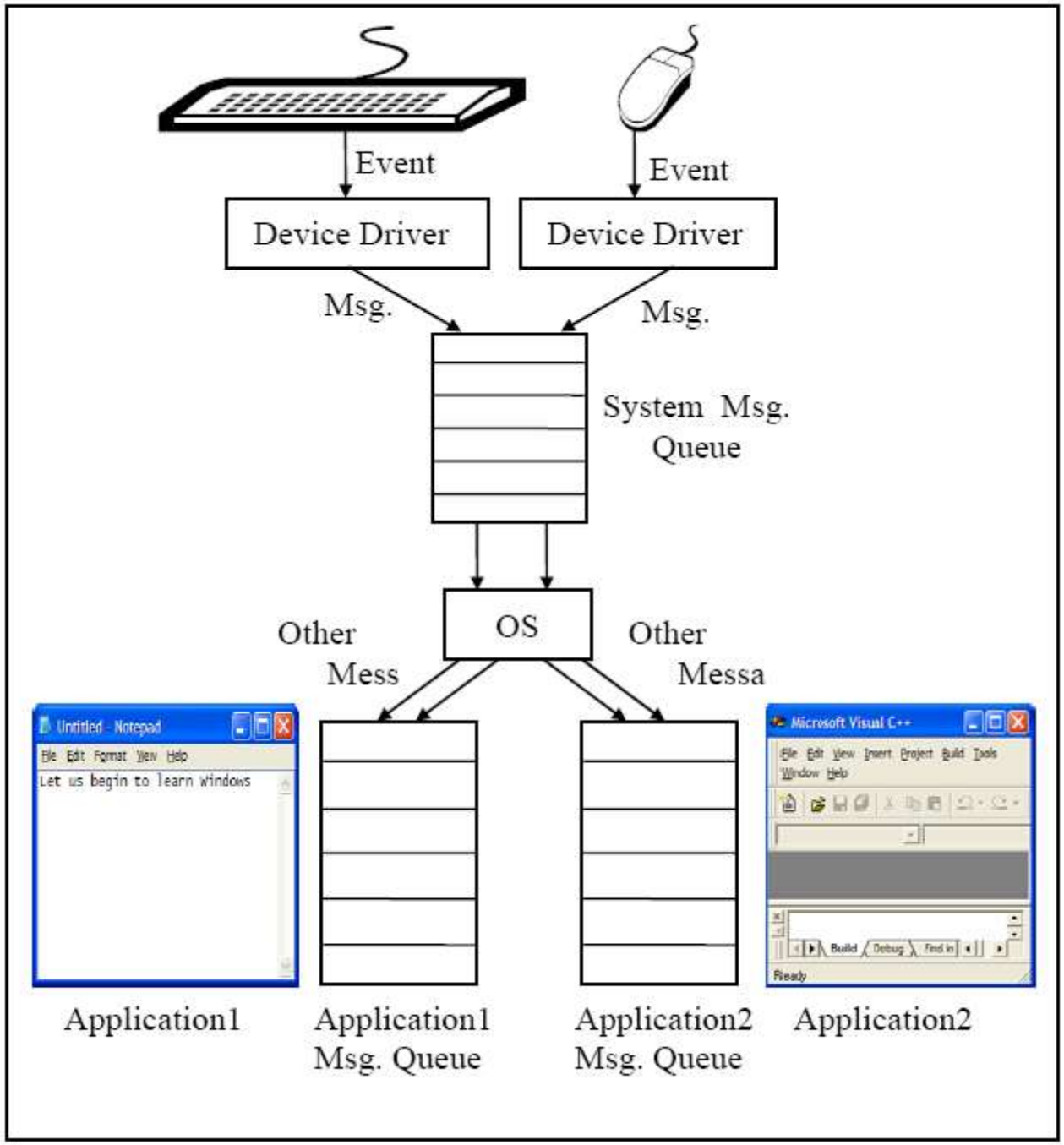
- User and System view of an OS
  - User View: How users or programs utilize the OS?
    - Type of users
      - Application Users / End Users
      - Application Programmers
  - System View: How the OS software actually does the required action?
    - how it gets keystrokes, separates out special ones like shift, and makes them available to the user or program.
    - Type of Users
      - System programmers
      - System Administrators

- User View – Types of Users
  - Application User / End User
    - people who use (or run) application or system programs
      - expect a quick, reliable response (to keystrokes or mouse movement
  - Application Programmers
    - people who write application programs
      - system calls or an API (application program interface)
- System View –Types of Users
  - Systems Programmers
    - these are the people who write software—either programs or components—that is closely tied to the OS.
      - have a detailed understanding of the internal functioning of the OS
      - A utility that shows the status of the computer's network connection or an installable driver for a piece of hardware are examples of systems programs.
  - System Administrators
    - people who manage computer facilities, responsible for installing and upgrading the OS, as well as other systems programs and utilities

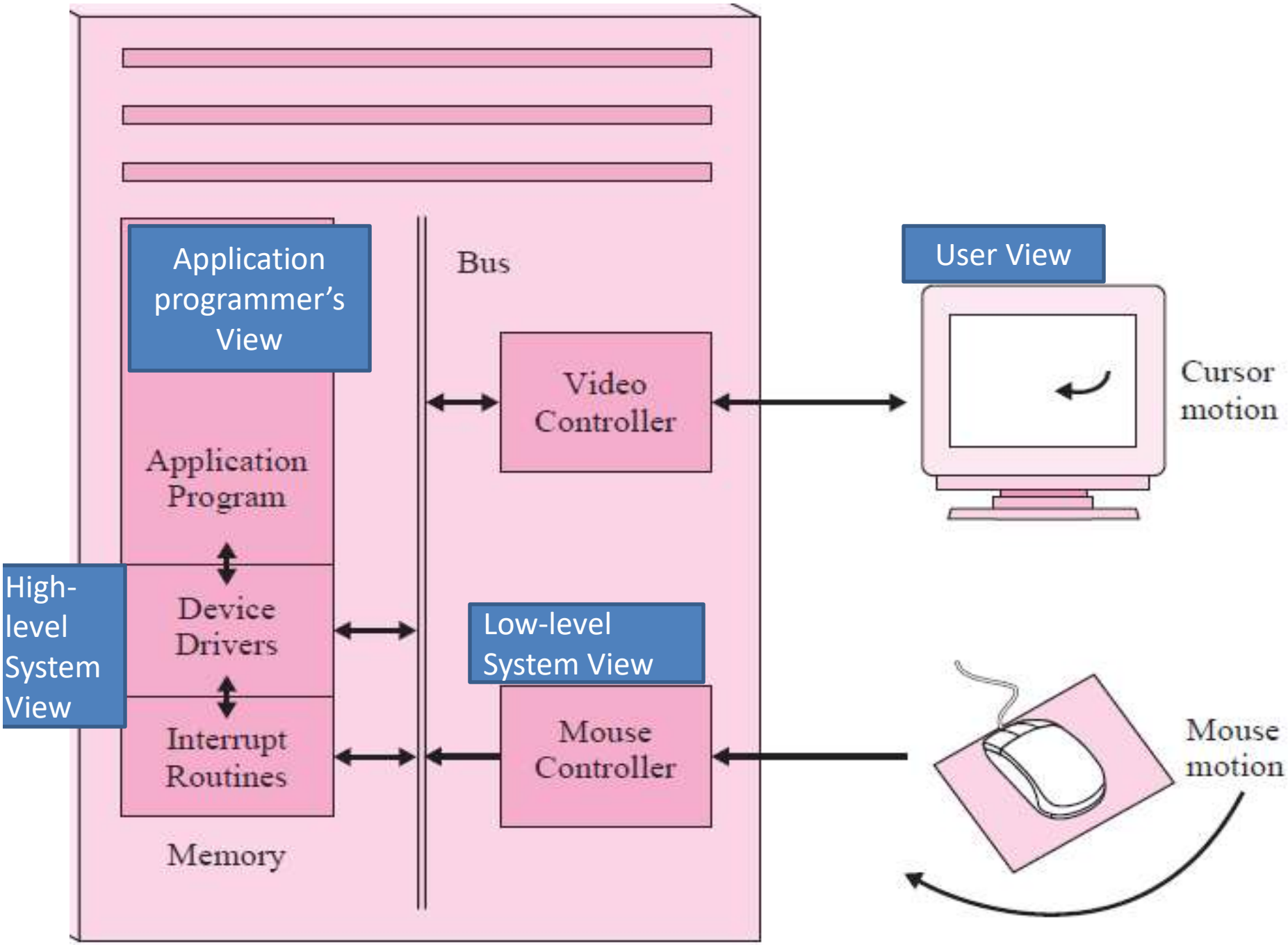
End Users	Easy to use and learn Adapts to user's style of doing things Lively response to input Provides lots of visual cues Free of unpleasant surprises (e.g., deleting a file without warning) Uniform ways to do the same thing (e.g., moving an icon or scrolling down a window—in different places) Alternative ways to do one thing (e.g., some users like to use the mouse, others like to use the keyboard)
Application Programmers	Easy to access low-level OS calls by programs (e.g., reading keystrokes, drawing to the screen, getting mouse position) Provide a consistent programmer view of the system Easy to use higher-level OS facilities and services (e.g., creating new windows, or reading from and writing to the network) Portability to other platforms
Systems Programmers	Easy to create correct programs Easy to debug incorrect programs Easy to maintain programs Easy to expand existing programs
System Managers and Administrators	Easy addition or removal of devices such as disks, scanners, multimedia accessories, and network connections Provide OS security services to protect the users, system, and data files Easy to upgrade to new OS versions Easy to create and manage user accounts Average response is good and predictable System is affordable


Application User
1
Application Programs
Application Programmers
Libraries
3
3
Operating System
System Programmers
Drivers
Memory Manager
Scheduler
8
8
8
8
7
7
ISA
Execution Hardware
System Administrators
10
10
System Interconnect (bus)
Memory Translation
11
11
12
Controllers
Controllers
13
14
I/O Devices and Networking
Main Memory
Software
Hardware

- Example : M**oving a mouse (and mouse cursor)**
  - When the pointing device is moved,
    - it generates a hardware event called an **interrupt** which the OS handles.
    - OS notes the movements of the mouse in terms of some hardware-specific units:
      - **System view:** Which application gets this mouse movement if there are multiple open windows?
        - » The mouse movements may need to be queued up if there are multiple movements before the application retrieves them.
        - » movements may even be lost if the OS is busy doing other things
      - **Low-level System view:** actual software **reading the mouse movements as number of pulses** which is part of the OS, and is called a **mouse device driver**
      - **High-level system view:** device driver reads the low-level mouse movement information and another part of the OS interprets it so that it can be converted into a higher-level system view,
        - » How is the mouse movements info presented to the application programmer?
      - **Application programmers' view:** How do I get the mouse movement information in order to use it and display it in my application?
      - **User view:** user's view is that the cursor will smoothly move on the screen and that as the mouse moves greater distances faster, the screen movement will appear faster too.


Event
Event
Device Driver
Device Driver
Msg.
Msg.
System Msg.
Queue
OS
Other
Mess
Other
Messa
Application1
Application1
Msg. Queue
Application2
Msg. Queue
Application2

System View


Application programmer's View
Bus
User View
Video Controller
Application Program
Cursor motion
High-level System View
Device Drivers
Low-level System View
Interrupt Routines
Mouse Controller
Mouse motion
Memory

- Example : Files
  - **End user's view:** File names
    - Can file names contain spaces? How long can they be? Are upper and lowercase letters allowed? Are they treated as different or the same characters? How about non-English characters or punctuation?
  - **Application programmer's view:** the file system is a frequently used, critical part of the system
    - commands for creating a new file, using an existing file, reading or appending data to a file, and other file operations.
  - **System view:** file system is so large it is usually divided into subparts:
    - file naming and name manipulation (directory services),
    - file services such as locating and mapping a file name to its data (file allocation and storage),
    - trying to keep parts of open files in main memory to speed up access to its data (file buffering and caching), and
    - the actual management of the storage devices (disk scheduling).

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

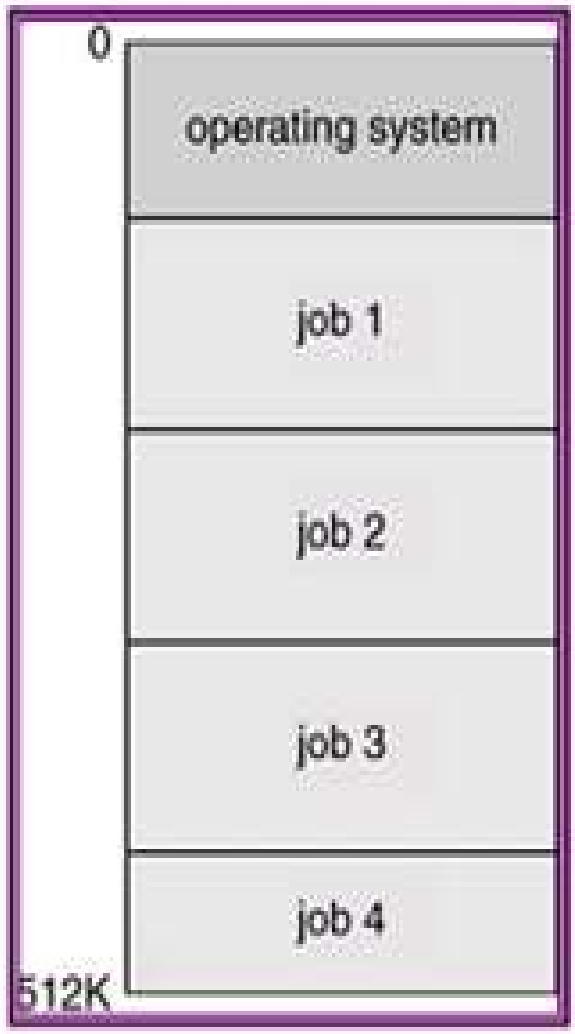
- **Single-tasking OS**
  - OS runs a single process at a time
    - Example: CP/M and MS-DOS
    - Handling I/O, starting and terminating programs
    - Fairly simple Memory management
    - no need for CPU scheduling
  - Supports batch job
    - have no live user so rapid response is not a requirement
    - less context switching is needed and more time is spent on productive computing

Single-user single-tasking Batch processing system

- 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

- **Multitasking or Multiprogramming** OS
  - OS will control multiple processes running concurrently
    - CPU scheduling component is used to choose which of the **ready to run** processes to run next
    - The CPU can switch to run another process while I/O is performed.
    - **Context Switching:** Changing from one running process to run another process is known as **context switching**
      - entire CPU state must be saved on process-1's **Process Control Block (PCB)**, execute the process-2 for a stipulated period of time, save the CPU state of process-2 in PCB of 2, switch back to process-1 and resume from stored CPU state by loading PCB-1
    - Supports both interactive and batch jobs
      - Interactive jobs are processes that handle a user interacting directly with the computer through mouse, keyboard, video monitor display, and other interactive I/O devices whereas batch jobs do not support user interaction with computers.
  - Advantages
    - Improve processor utilization by keeping the CPU busy while I/O is performed
      - When a process waiting for I/O, the CPU will execute other process

- Multiprogramming needed for efficiency
  - Single program 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



- Multi-programming systems demand
  - Job scheduling
    - To decide which jobs in a job pool should be brought into memory
  - Memory management
    - Allocate memory to many jobs
  - CPU scheduling
    - Choose the jobs in memory that are in ready to run state
  - Allocation of devices
    - If more than one job is competing for resources
  - Jobs that are running concurrently should not affect the other.

Timesharing Systems

- (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 i.e. process
  - If several jobs ready to run at the same time need CPU scheduling to decide the job to be executed
  - 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 system

- 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 system

- 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 System
  - 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.

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.

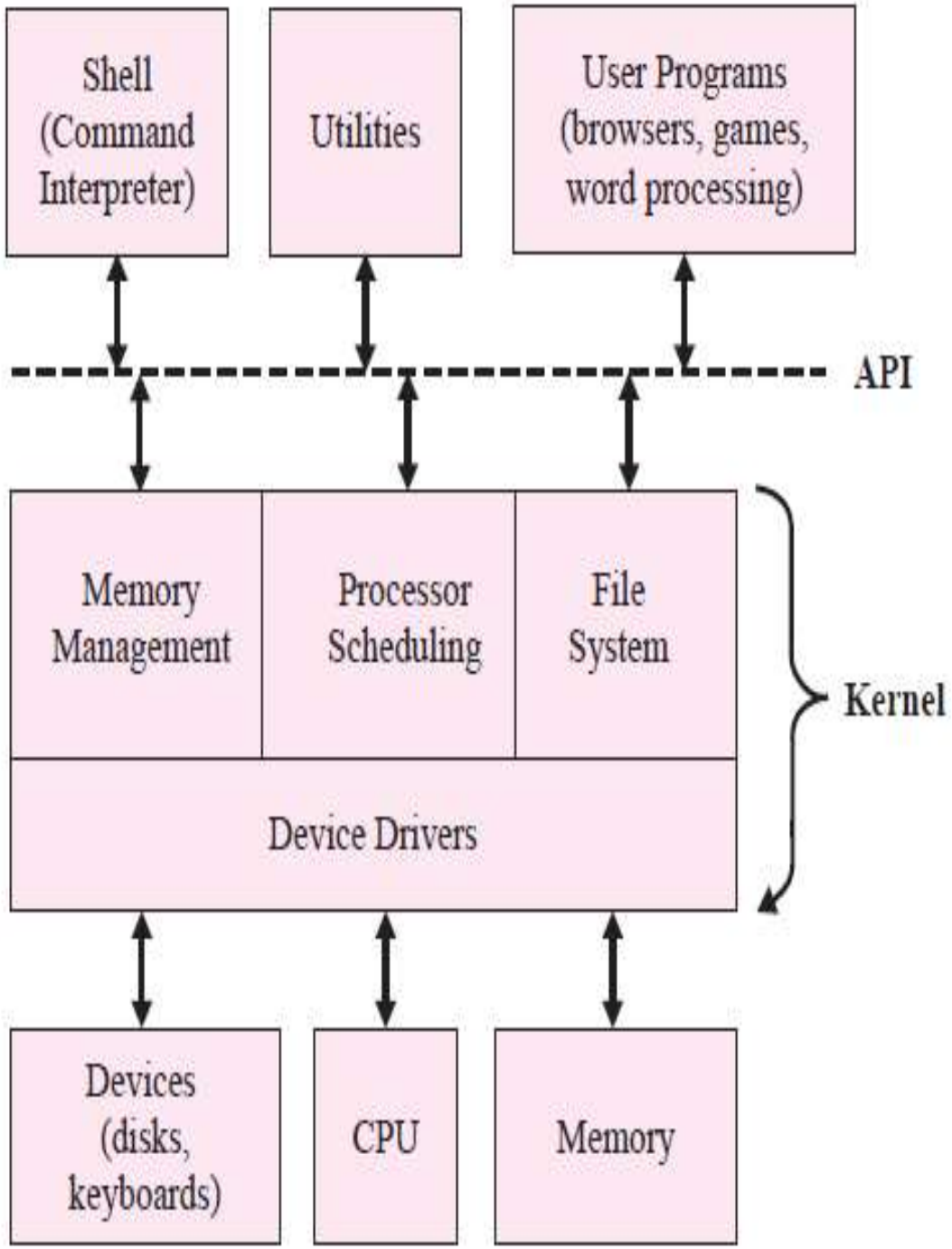
- i. **Handheld**. A small computer system that performs simple tasks such as calendars, email, and web browsing. Handheld systems differ from traditional desktop systems with smaller memory and display screens and slower processors.

ARCHITECTURAL APPROACHES TO BUILDING AN OS

1. Monolithic single-kernel OS approach
2. Layered OS approach
3. Modular approach
4. Microkernel OS approach

- **Monolithic single-kernel OS approach**
  - written as a single program (no modules at all)
    - **kernel** or **monolithic kernel**
  - **Disadvantages**
    - As monolithic kernel size grew if more and more functionalities are added, percentage of main memory occupied by the OS is too large.
    - Lead to had more bugs,
    - Difficult to maintain,
    - Difficult to add features or to fix bugs
  - Solution
    - develop OSs based on a more modular, layered design

- **Layered OS approach**
  - modular OS approach that was developed was a **layered architecture.**
  - OS is divided into modules that were limited to a specific function such as processor scheduling or memory management.
  - The modules were grouped into layers of increasing abstraction
    - each layer provides a more abstract view of the system and relies on the services of the layers below it.
    - The layered approach would hide the peculiarities and details of handling hardware devices, and provide a common abstract view to the rest of the OS.
    - Thus, when new devices entered the marketplace, new device drivers could be added to the kernel without drastically affecting the other OS modules, which provide memory management, processor scheduling, and the file system interface.
  - **Disadvantage/Critic:**
    - OS design should return to a minimum amount of code in the kernel
  - **Solution**
    - Microkernel approach



Different Variations

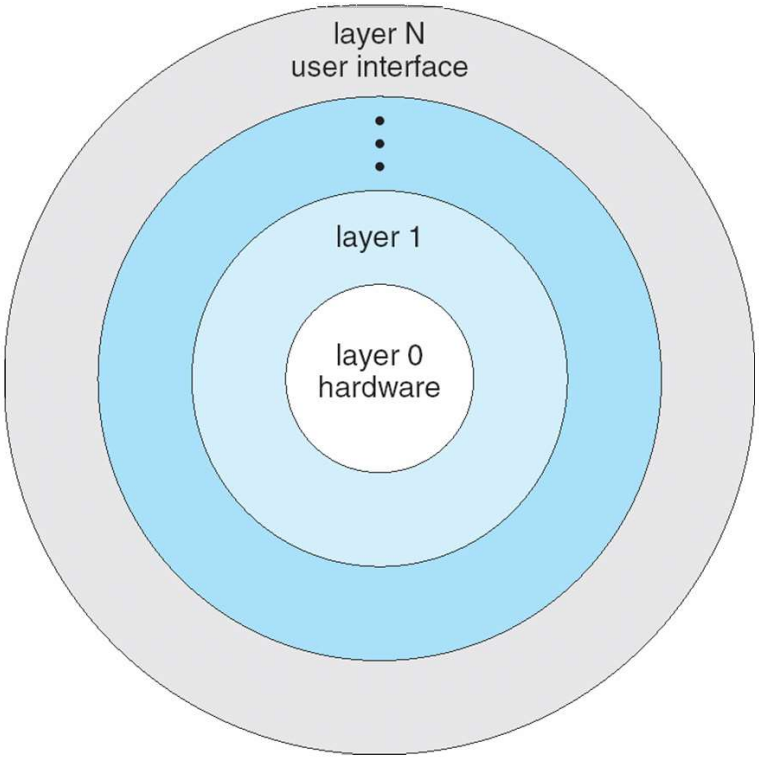
1. Allow modules at **layer *n*** to call only the modules in the next lower **layer *n-1***

2. allow modules **at layer *n*** to call modules at any of the lower l**ayers (*n-1, n-2,* and so on**).

3. allow **level *n* modules** to interact with other **level *n* modules**

Layered Approach

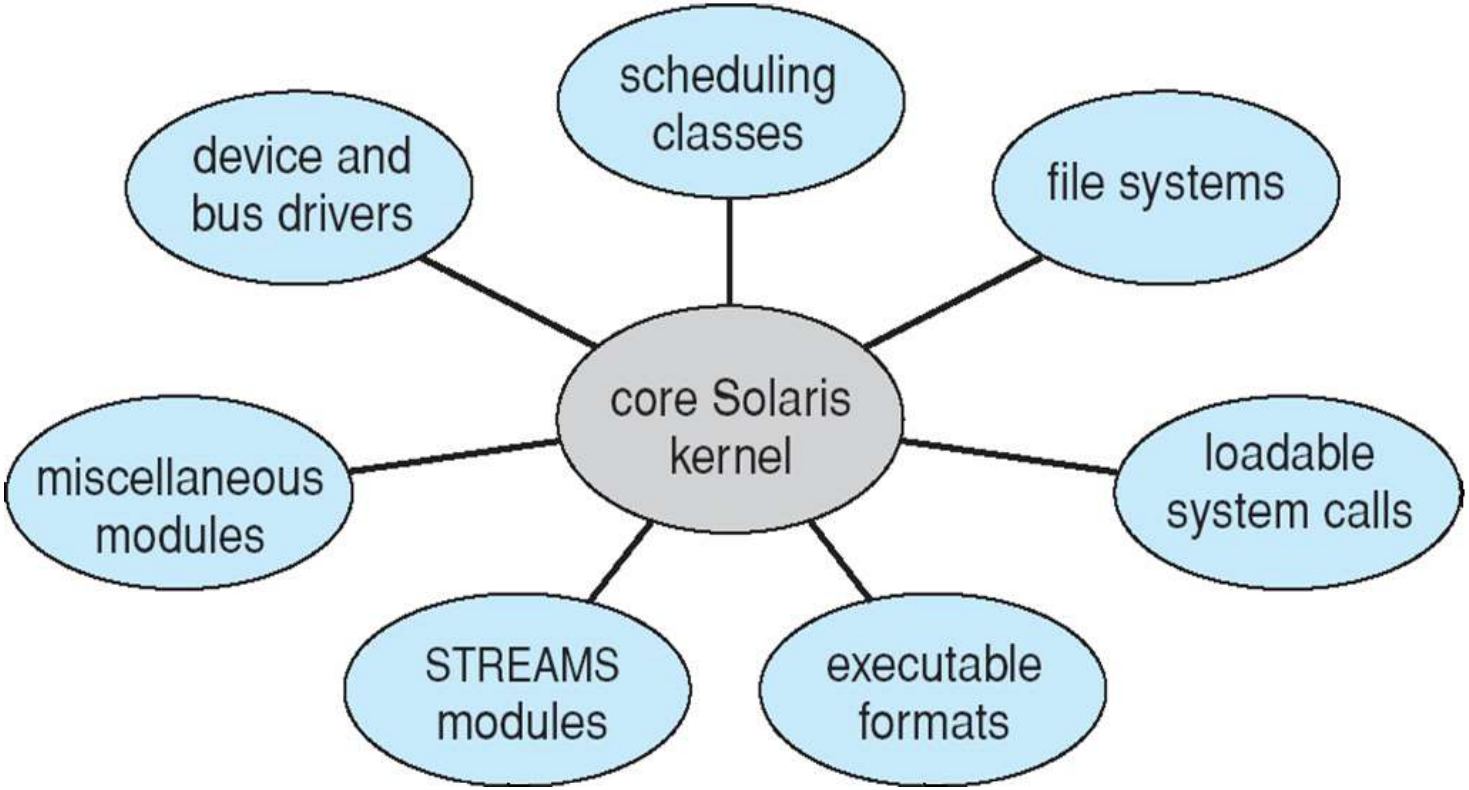
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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



Modules

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

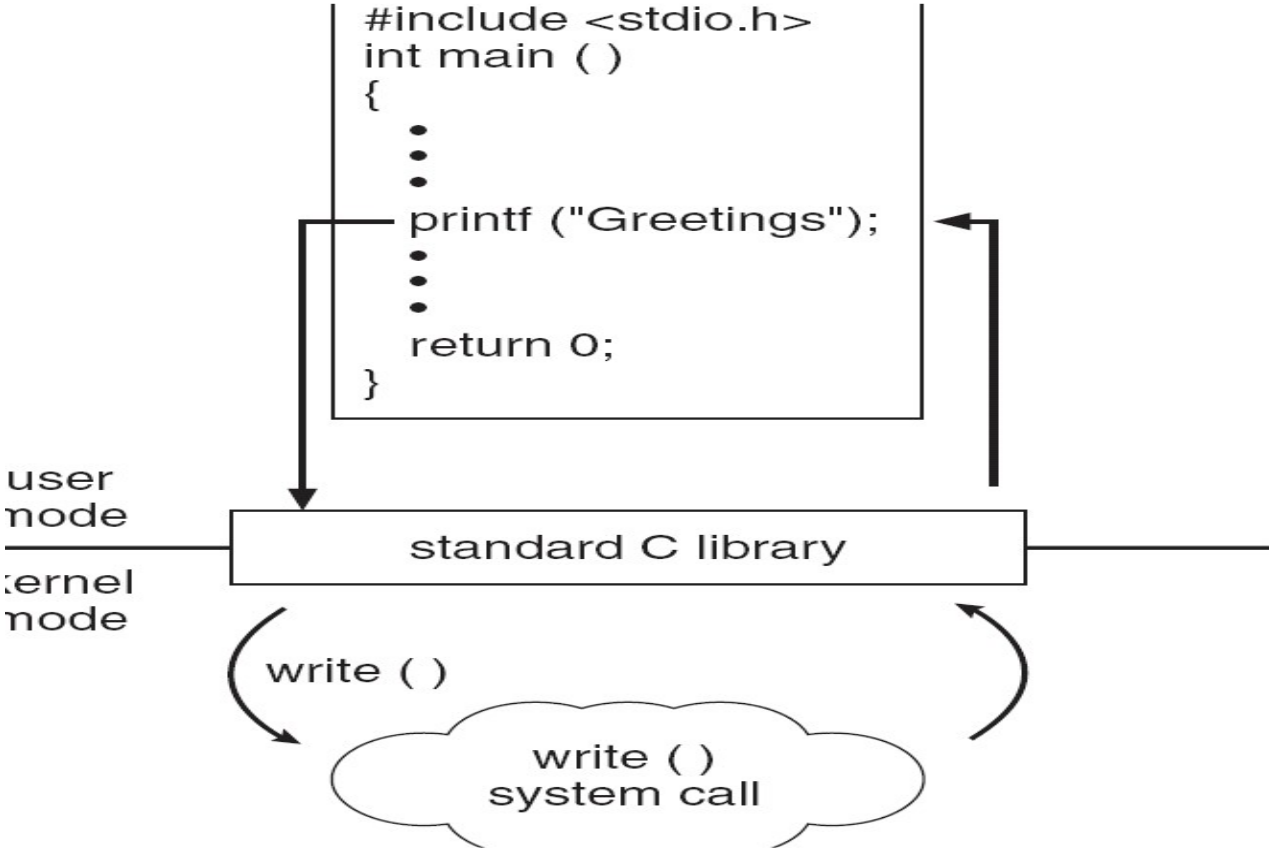
Solaris Modular Approach

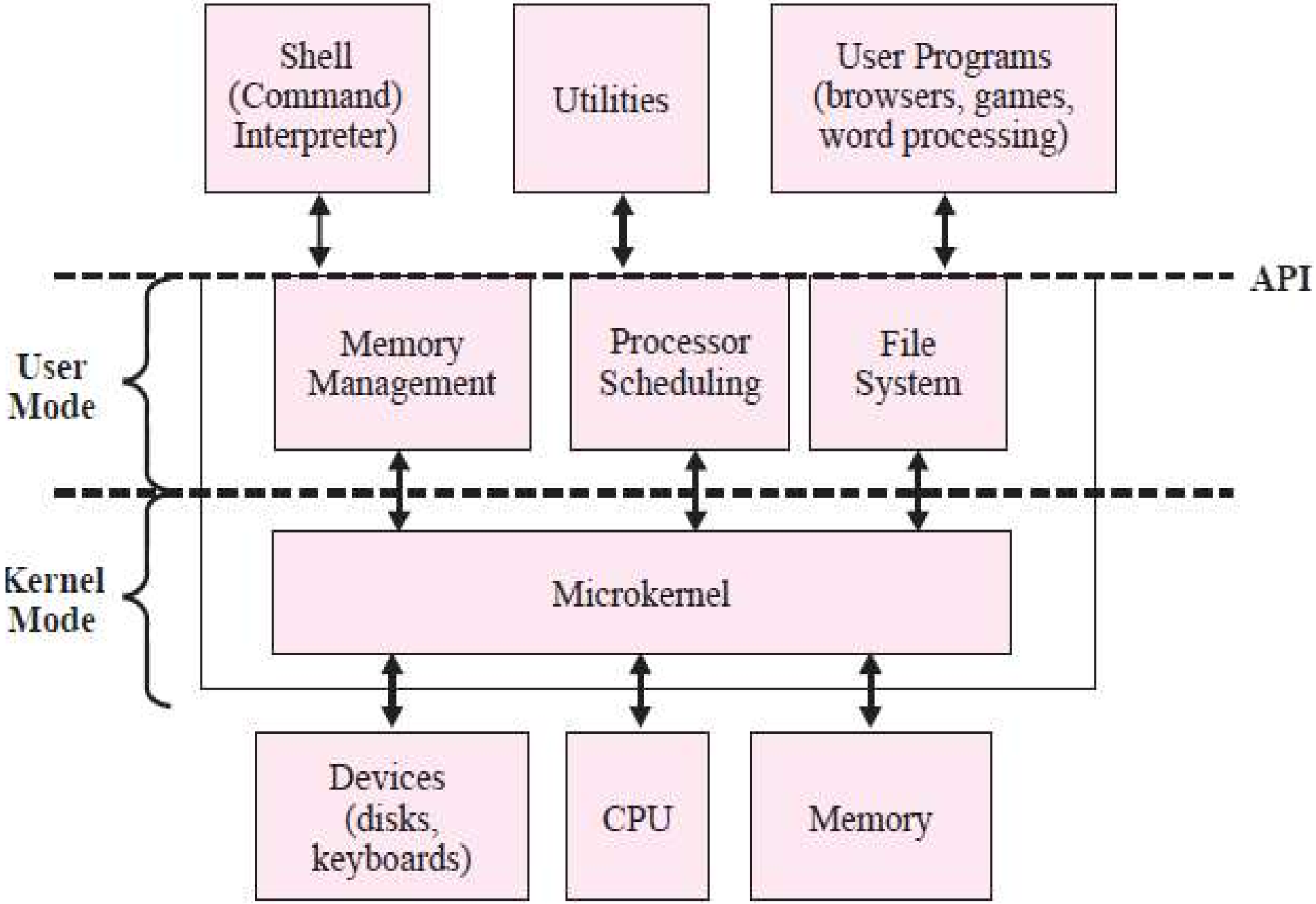


- **Microkernel OS approach**
  - **More robust** – As the amount of code that is running in supervisor mode is smaller
    - easier to inspect for flaws
    - easier to port a small microkernel to a new platform
  - Only basic functionality, usually **the interfaces to the various types of device drivers**, is included in the microkernel
    - **code that must run in supervisor mode** because it actually uses privileged resources such as protected instructions or accesses memory not in the kernel space
    - **Code running in protected mode** literally can do anything, so an error in this code **can do more damage** than code running in user mode
  - **Remainder of the OS functions** are still **part of the resident OS**, but they **run in user mode** rather than protected mode
  - **Make use of interrupts** to **make the necessary calls** from the **user mode portions** of the OS to the **supervisor mode portions**.
  - **Disadvantage/Critics:**
    - Makes a microkernel OS run more slowly due to context switching and not yet resolved

Standard C Library Example

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




Shell (Command) Interpreter)
Utilities
User Programs (browsers, games, word processing)
API
User Mode
Memory Management
Processor Scheduling
File System
Kernel Mode
Microkernel
Devices (disks, keyboards)
CPU
Memory

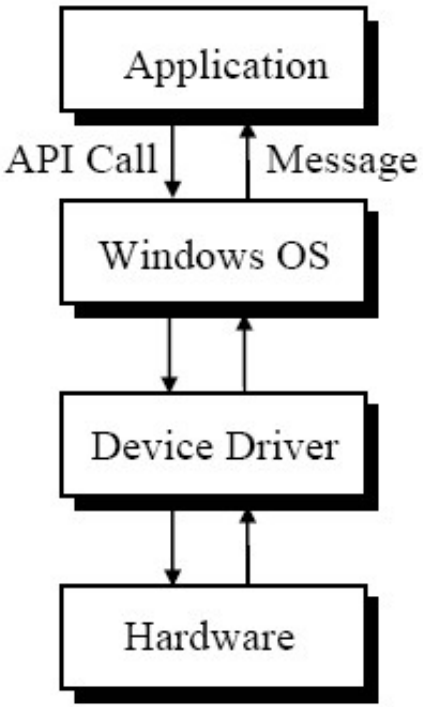
- OS Design Issues
  - **Minimalist**
    - only those things that really must go into the kernel (or microkernel) are included in the OS.
    - Other components may be added into **library routines** or as **“user” programs but not written by the user.**
    - **Claims:**
      - Load when it is really needed
      - Write and integrate new components
      - Easy to design
      - Elegant or cleaner
  - **Maximalist**
    - philosophy—to put most of the commonly used services in the OS
    - **Claims:**
      - Consistent look and feel

Operating Systems-Module 2

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.


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.

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.**

Characteristics of a Good API

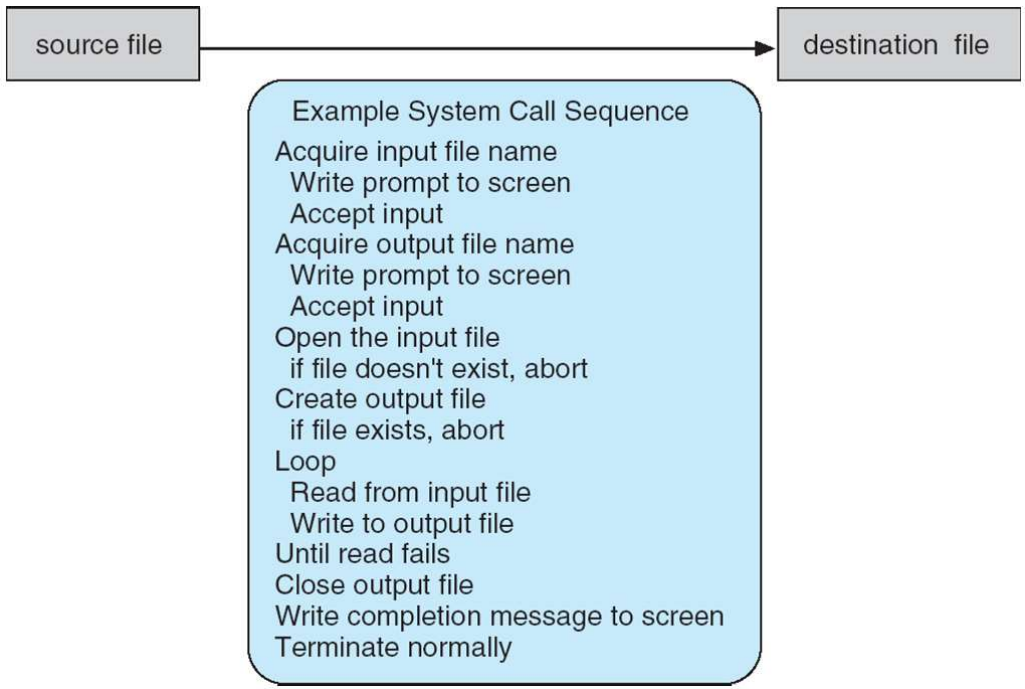
- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

System Calls

- Programming interface to the services provided by the OS
- The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system.
- A system call is how a program requests a service from an operating system's kernel.
- System calls provide an essential interface between a process and the operating system.
- Typically written in a high-level language (C or C++) can be written in assembly language for low level tasks.
- 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)

Example of System Calls

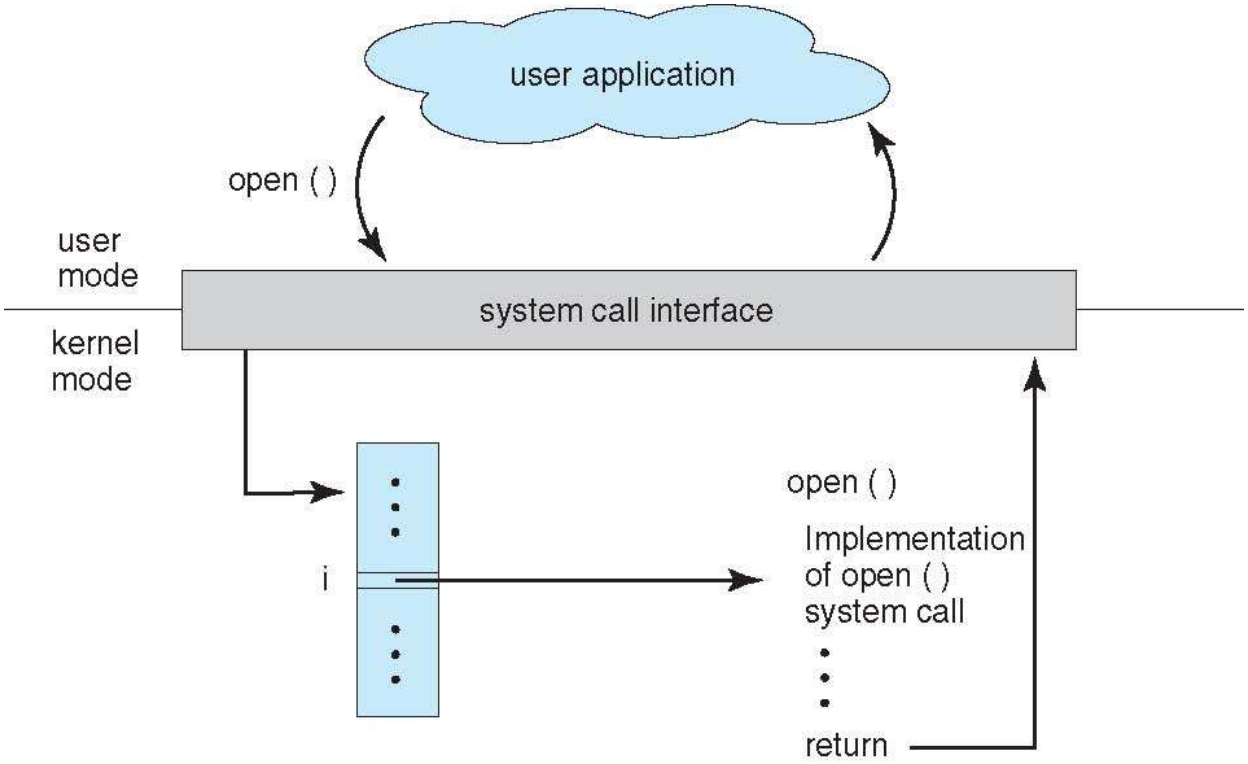
- System call sequence to copy the contents of one file to another file



System Call Implementation

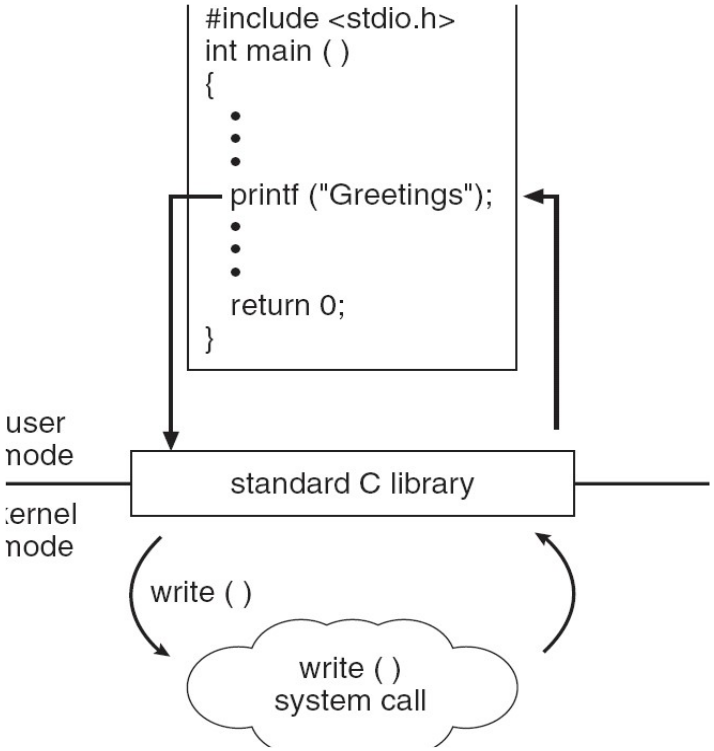
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about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

API – System Call – OS Relationship



Standard C Library Example

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



System calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

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

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;
- The entity that can be assigned to and executed on a processor
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

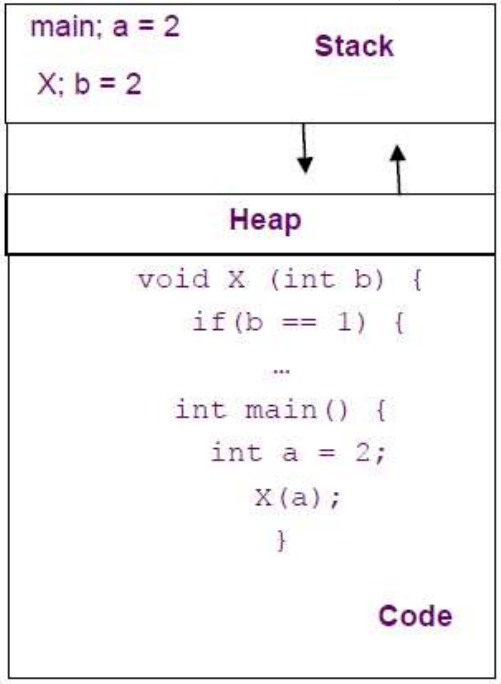
Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

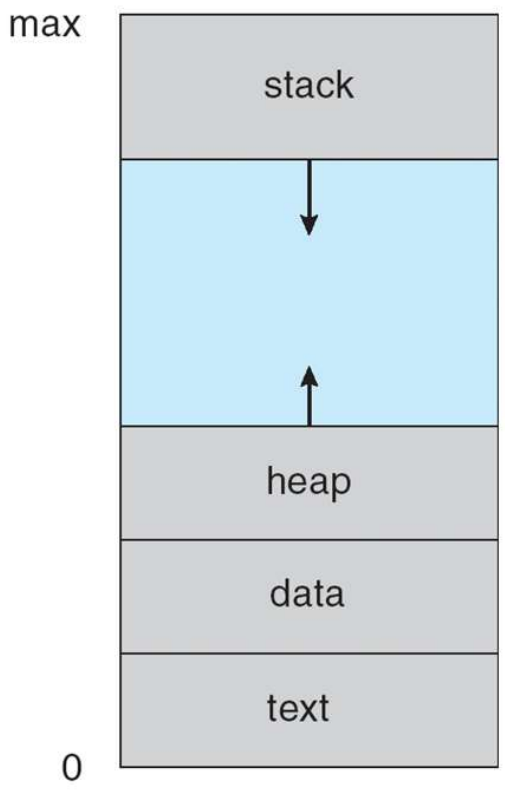
What you wrote

```
void X (int b) {
  if(b == 1) {
    …
int main() {
  int a = 2;
    X(a);
    }
```

What is in memory.



Process in Memory



Process Concept (Cont.)

- Program is ***passive*** entity stored on disk , process is ***active***
  - Program becomes process when file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc

Program, executable and process

In order to execute a program, the operating system must first create a process and make the process execute the program.

Program

A set of instructions which is in human readable format. A passive entity stored on secondary storage.

Executable

A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage.

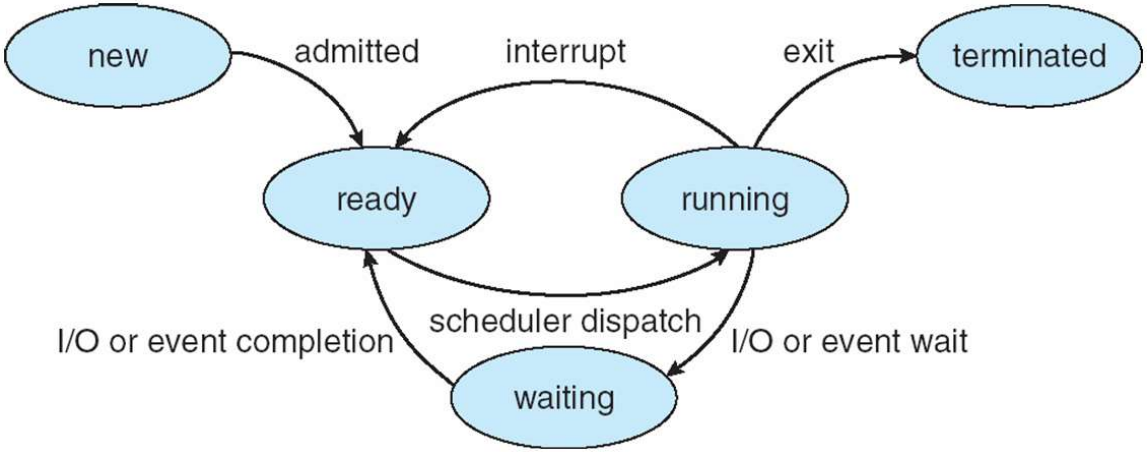
Process

A program loaded into memory and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting). A process is an active entity and needs resources such as CPU time, memory etc to execute.

Process State

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

Diagram of Process State



Process Control Block (PCB)

Information associated with each process

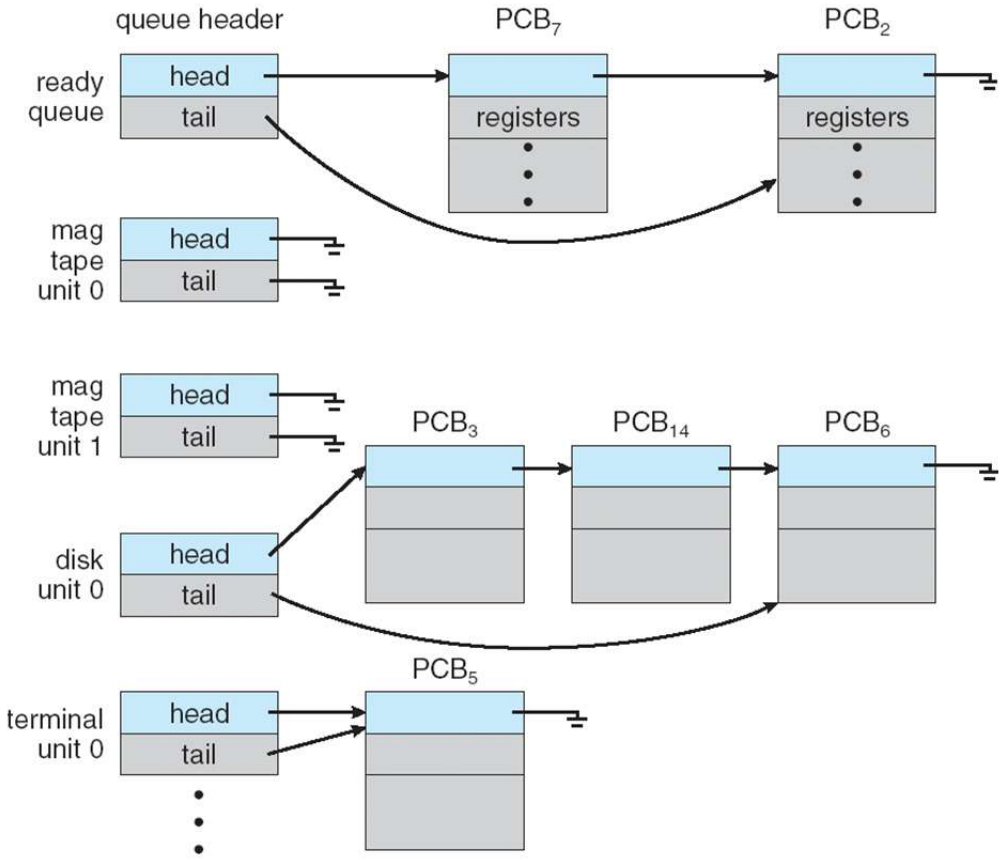
(also called **task control block**)

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

Process Scheduling

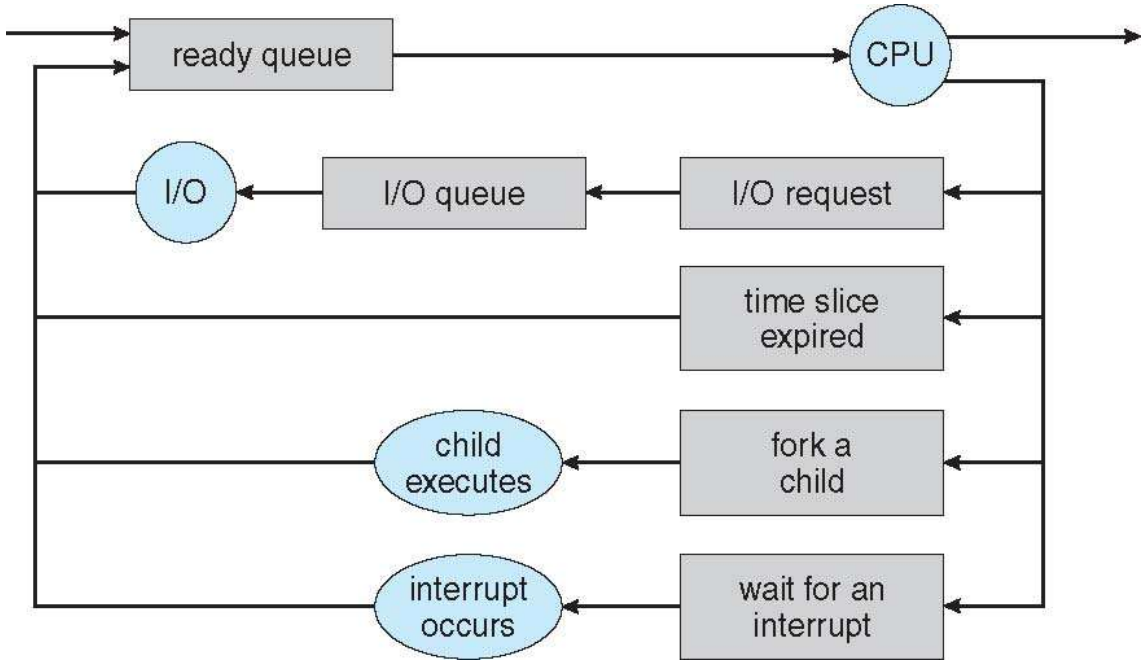
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

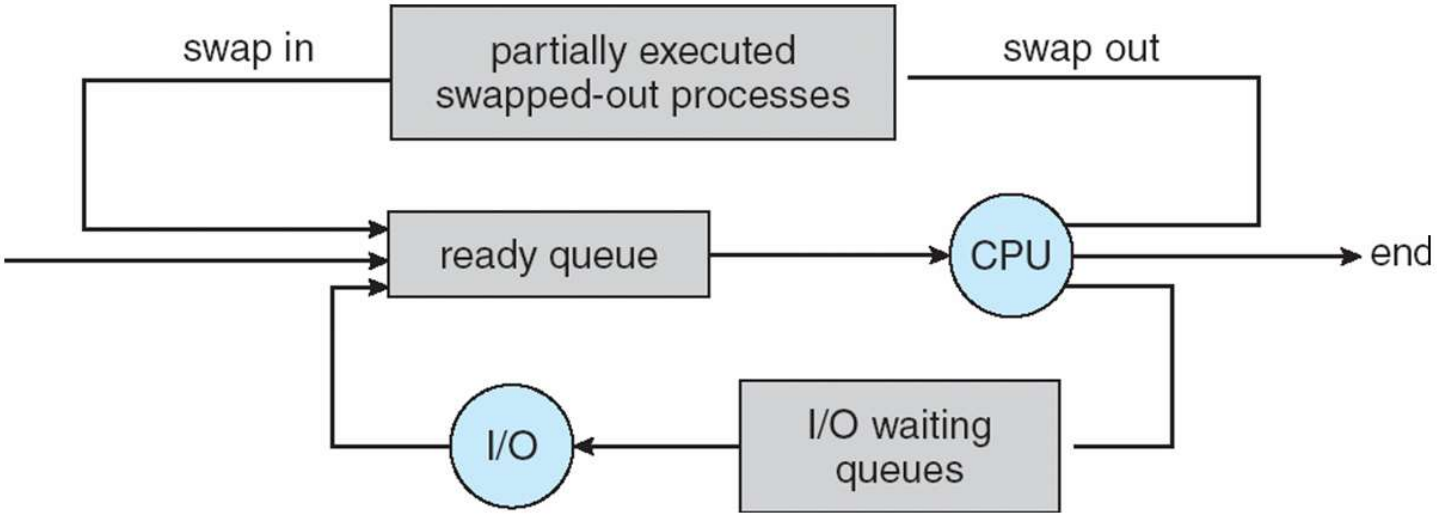


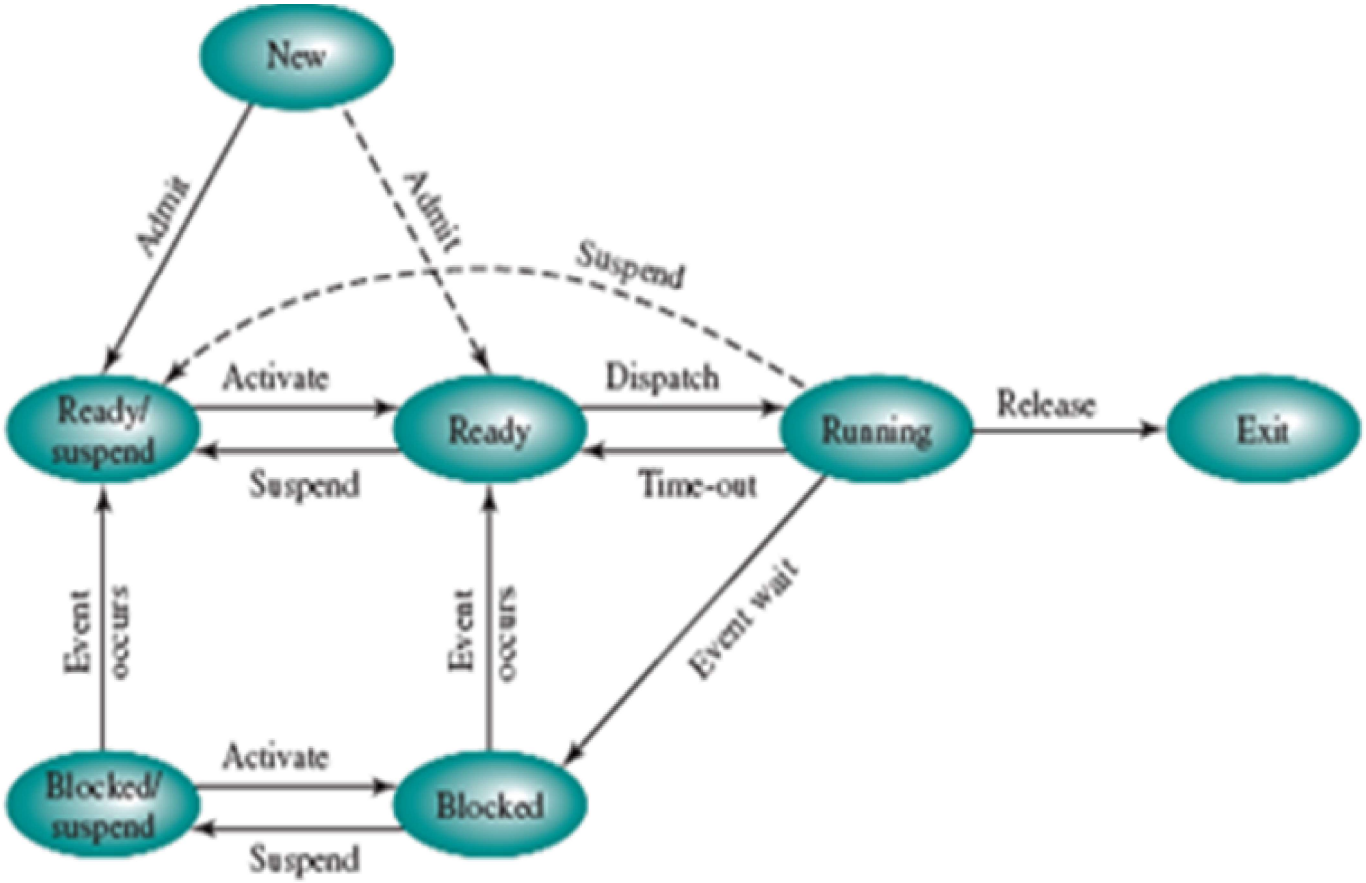
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked 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
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

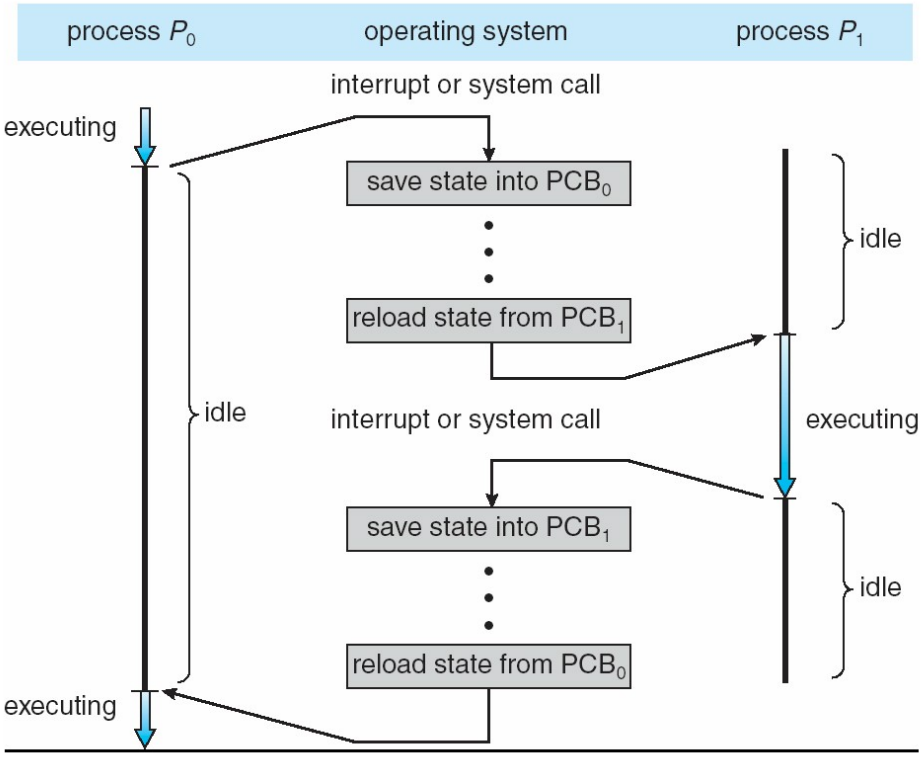



New
Admit
Admit
Suspend
Activate
Dispatch
Release
Ready/
suspend
Ready
Running
Exit
Suspend
Time-out
Event
occurs
Event
occurs
Event wait
Activate
Blocked/
suspend
Blocked
Suspend

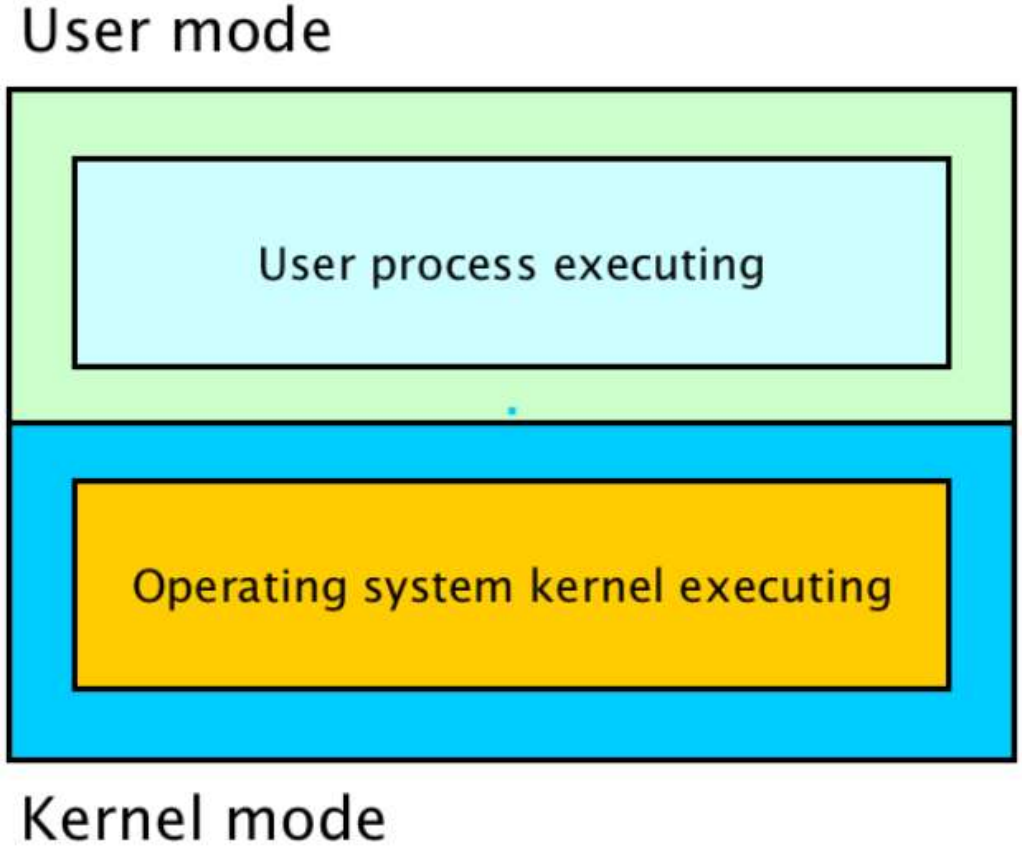
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support

CPU Switch From Process to Process



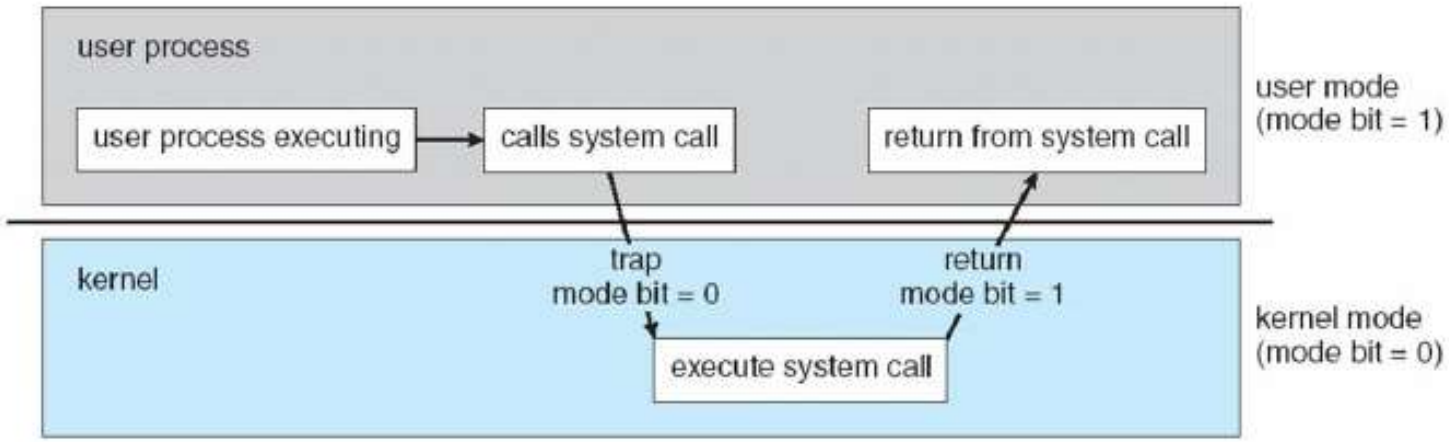
Operating system operations



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



Exception and interrupts

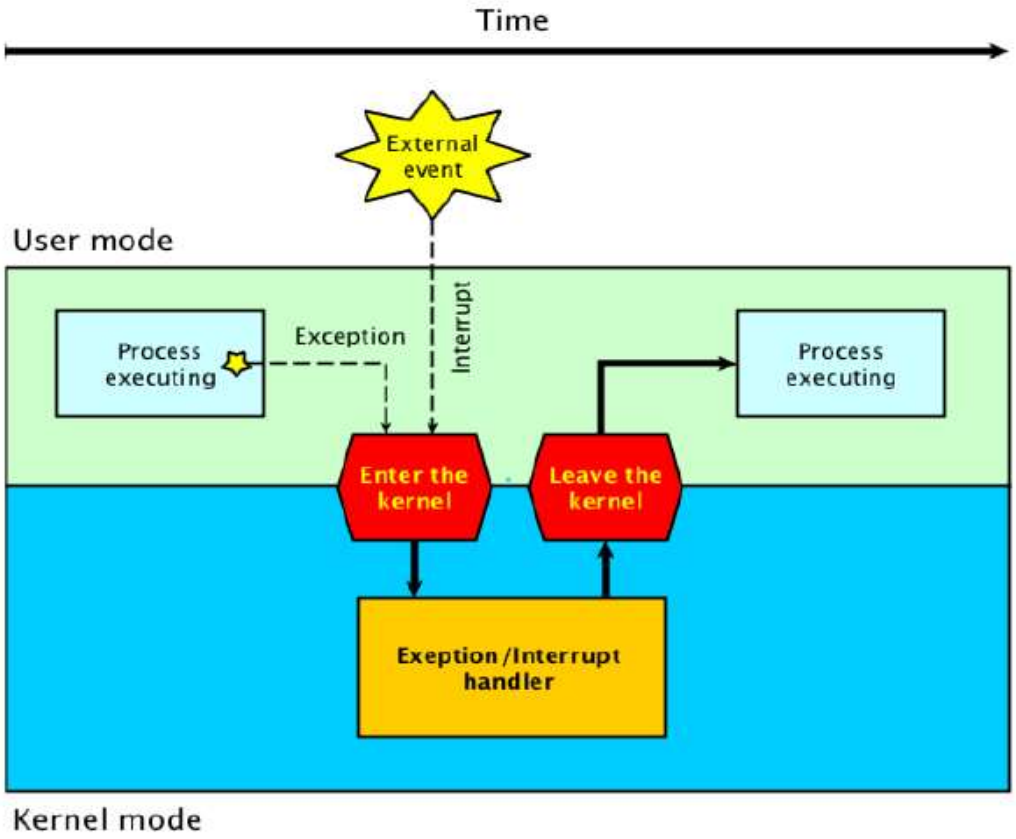
Exceptions are internal and synchronous

- Exceptions are used to handle **internal program errors**.
- Overflow, division by zero and bad data address are examples of internal errors in a program.
- Another name for exception is trap. **A trap (or exception) is a software generated interrupt**.
- Exceptions are produced by the CPU control unit while executing instructions and are considered to be synchronous because the control unit issues them only after terminating the execution of an instruction.

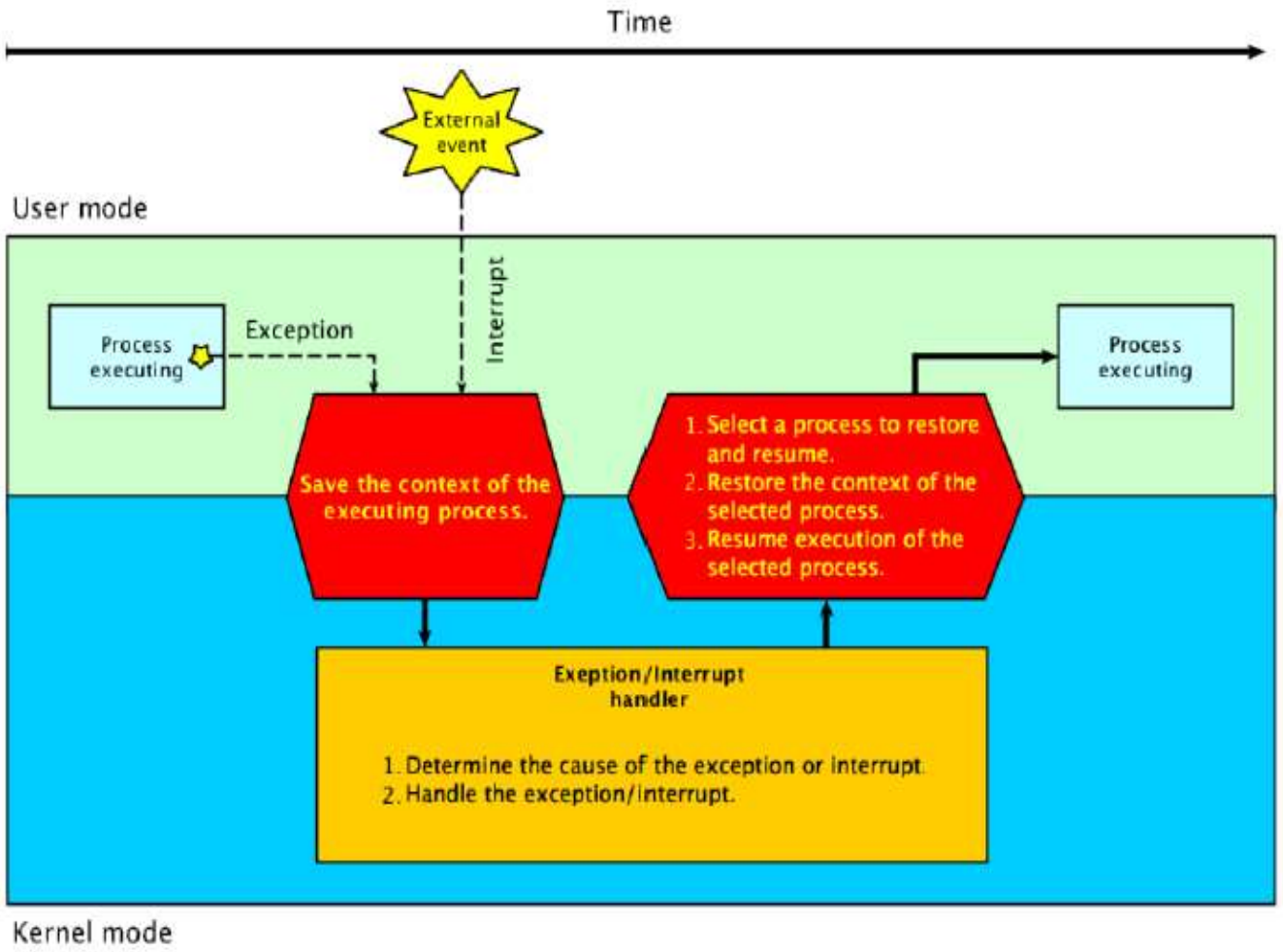
Interrupts are external and asynchronous

- Interrupts are used **to notify the CPU of external events**.
- Interrupts are **generated by hardware devices** outside the CPU at arbitrary times with respect to the CPU clock signals and are therefore considered to be asynchronous.
- Read and write requests to disk is similar to key presses.

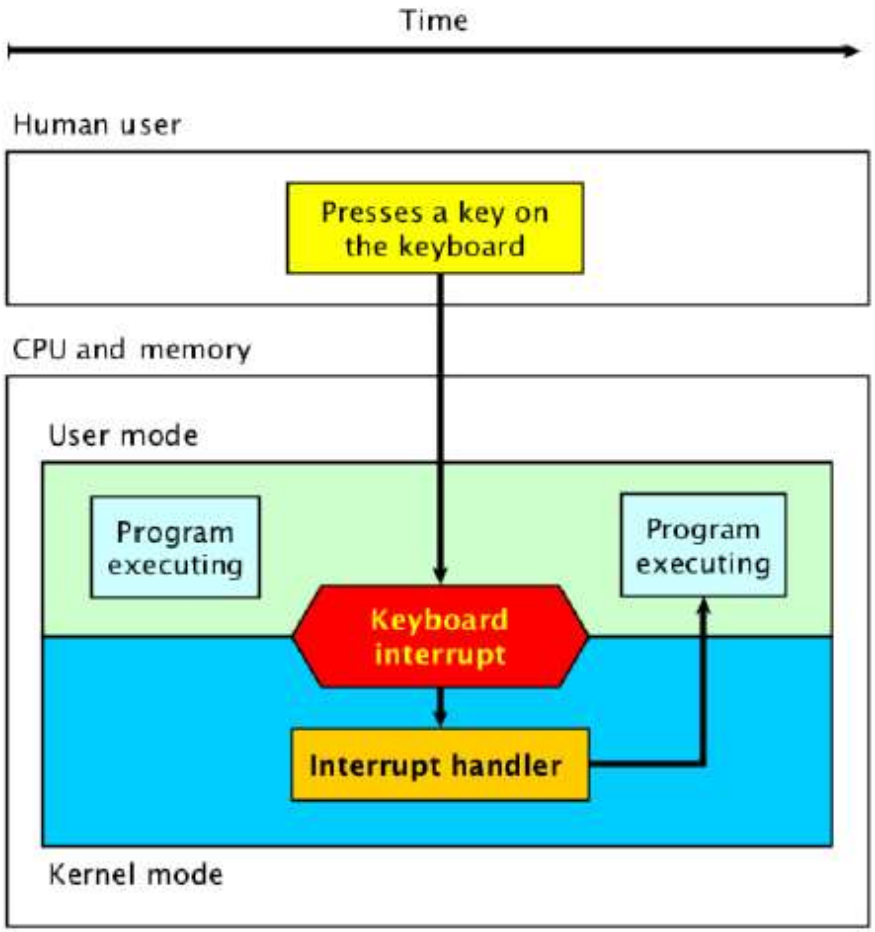
Exception and interrupt handler

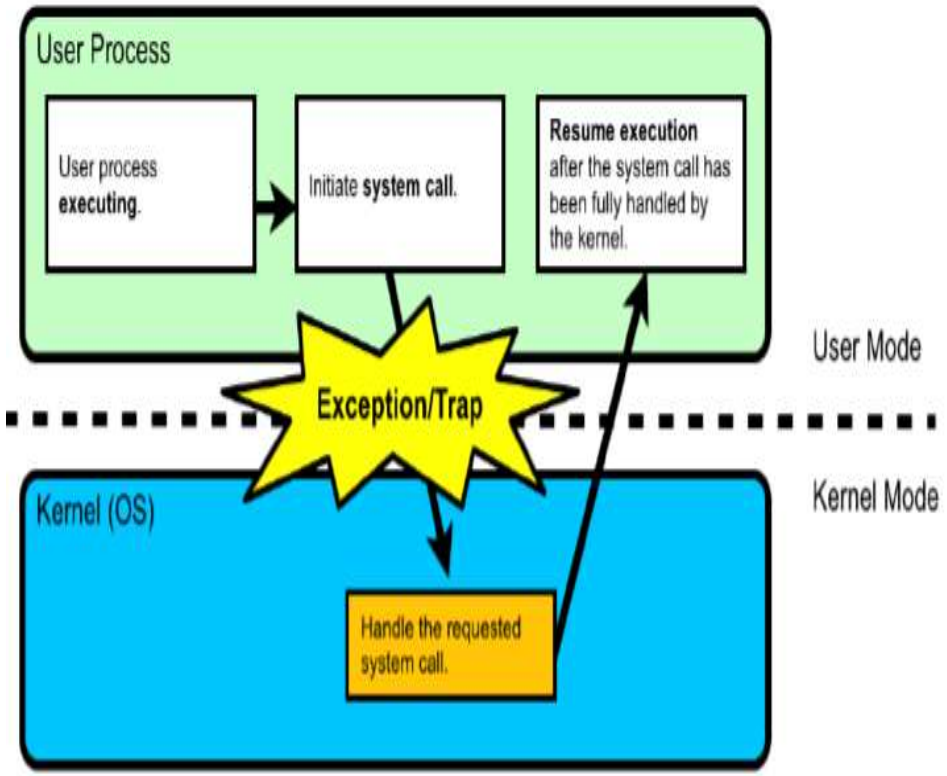


- When an exception or interrupt occurs, execution transition from user mode to kernel mode where the exception or interrupt is handled.
- When the exception or interrupt has been handled execution resumes in user space.



- When an exception or interrupt occurs, execution transition from user mode to kernel mode where the exception or interrupt is handled.
- While entering the kernel, **the context (values of all CPU registers)** of the currently executing process must first be saved to memory.
  - The kernel is now ready to handle the exception/interrupt.
  - Determine the cause of the exception/interrupt.
  - Handle the exception/interrupt.
- When the exception/interrupt have been handled the kernel performs the following steps:
  - Select a process to restore and resume.
  - Restore the context of the selected process.
  - Resume execution of the selected process.


Time
Human user
Presses a key on the keyboard
CPU and memory
User mode
Program executing
Program executing
Keyboard interrupt
Interrupt handler
Kernel mode


User Process
User process executing.
Initiate system call.
Resume execution after the system call has been fully handled by the kernel.
User Mode
Exception/Trap
Kernel Mode
Kernel (OS)
Handle the requested system call.

Operations on Processes

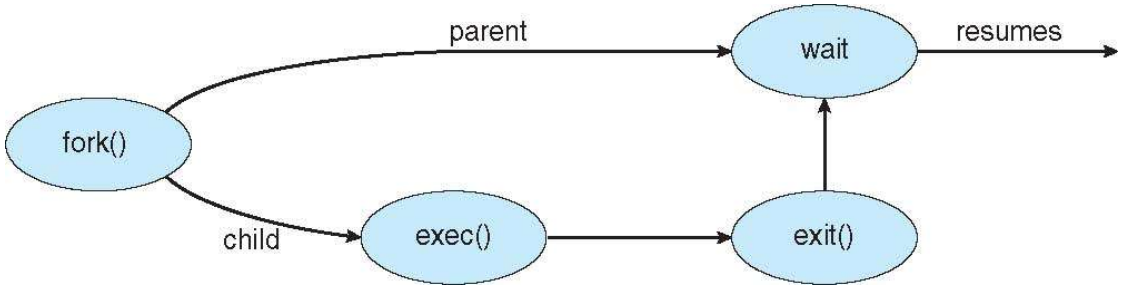
- System must provide mechanisms for:
  - process creation,
  - process termination

Process Creation

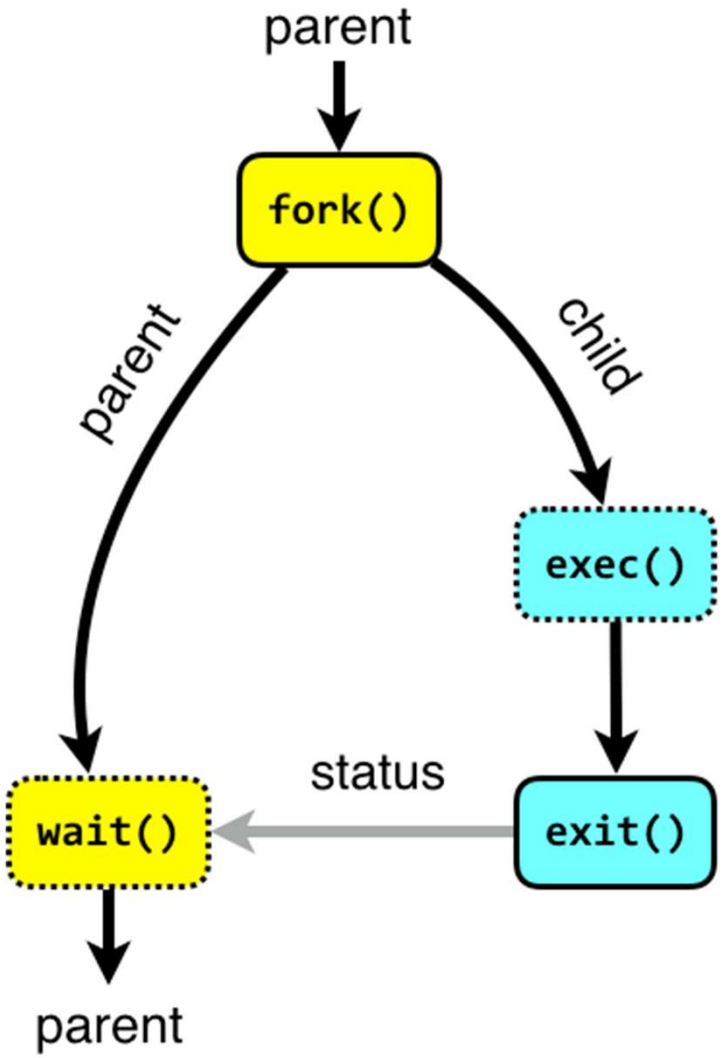
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

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 Management - System calls



The following system calls are used for basic process management.

fork

- A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

exec

- Replaces the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different program than the parent.

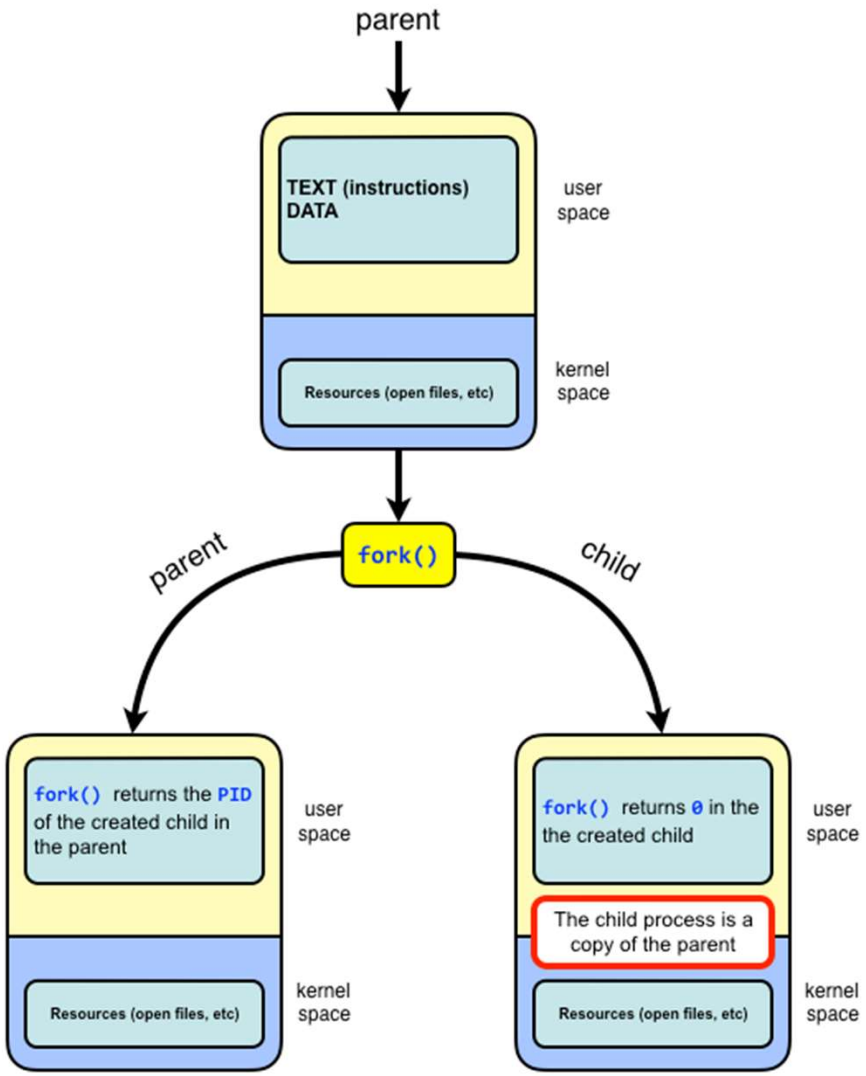
exit

- Terminates the process with an exit status.

wait

- The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child

Process Mgt. - System Calls



#include <unistd.h>

pid_t fork(void);

- On success, the PID of the child process is returned in the parent, and 0 is returned in the child.
- On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Fork returns twice on success

- On success fork returns twice: once in the parent and once in the child.
- After calling fork, the program can use the fork return value to tell whether executing in the parent or child.
  - If the return value is 0 the program executes in the new child process.
  - If the return value is greater than zero, the program executes in the parent process and the return value is the process ID (PID) of the created child process.
  - On failure fork returns -1.

fork() - example

```
int main(void) {
  pid_t pid;

  switch (pid = fork()) {
    case -1:
      // On error fork() returns -1.
      perror("fork failed");
      exit(EXIT_FAILURE);
    case 0:
      // On success fork() returns 0 in the child.
      child();
    default:
      // On success fork() returns the pid of the child to the
parent.
      parent(pid);
  }
}
```

```
void child()
{
printf(" CHILD <%ld> I'm alive! My PID is <%ld> and my parent got  PID
<%ld>.\n", (long) getpid(), (long) getpid(), (long)            getppid());
printf(" CHILD <%ld> Goodbye!\n", (long)     getpid());
exit(EXIT_SUCCESS);
}
```

```
void parent(pid_t pid)
{
printf("PARENT <%ld> My PID is <%ld> and I spawned a child with  PID
<%ld>.\n", (long) getpid(), (long) getpid(), (long) pid); printf("PARENT
<%ld> Goodbye!\n", (long) getpid()); exit(EXIT_SUCCESS);
}
```

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

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

    return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

Orphans

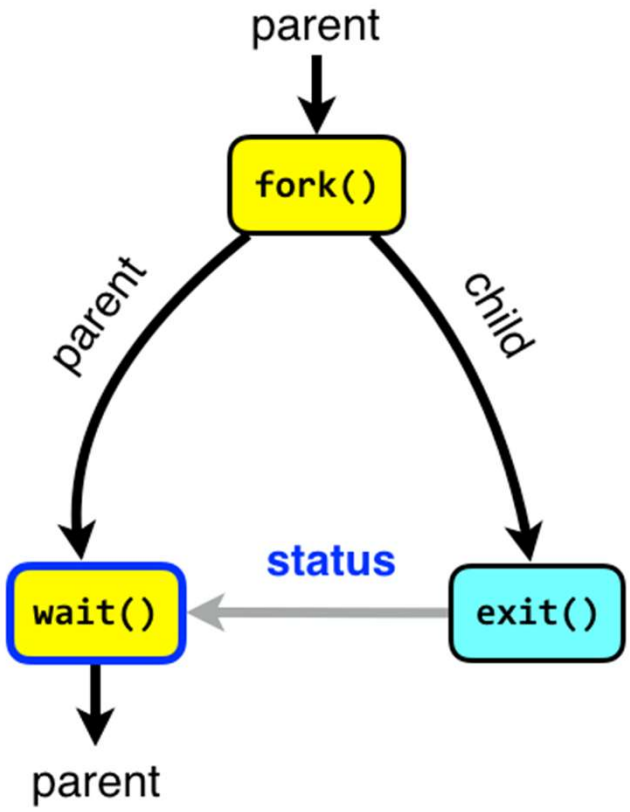
- An orphan process is a process **whose parent process has terminated**, though it remains running itself.
- Any orphaned process will be **immediately adopted by the special init system process** with PID 1.
- Processes execute **concurrently**
- Both the parent process and the child process competes for the CPU with all other processes in the system.
- The operating systems decides which process to execute when and for how long. The process in the system execute concurrently.
- In our example program:
  - most often the parent terminates before the child and the child becomes an orphan process adopted by init (PID = 1) and therefore reports PPID = 1
  - sometimes the child process terminates before its parent and then the child is able to report PPID equal to the PID of the parent.

Wait

- The wait system call blocks the caller until one of its child process terminates.
- If the caller doesn't have any child processes, wait returns immediately without blocking the caller.
- Using wait the parent can obtain the exit status of the terminated child.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

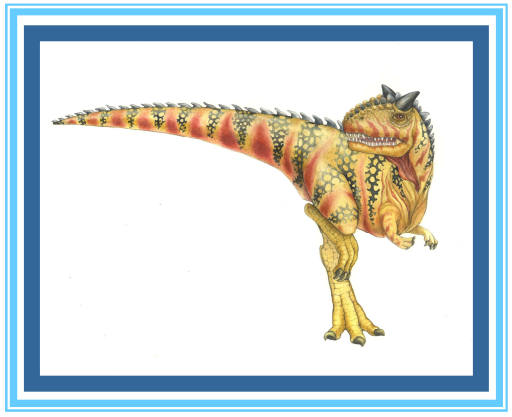
On success, wait returns the PID of the terminated child.
On failure (no child), wait returns -1.



Zombies

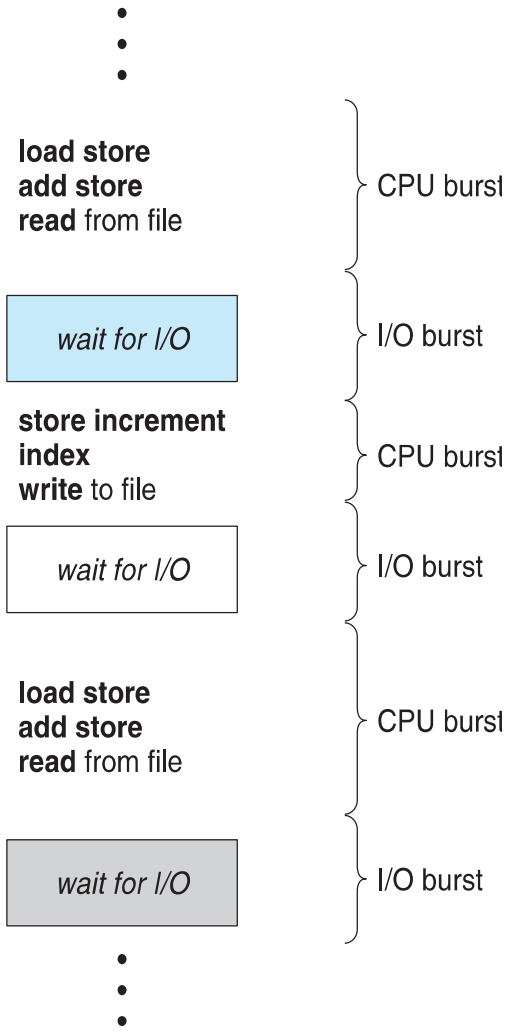
- A terminated process is said to be a zombie or defunct until the parent does wait on the child.
- When a process terminates all of the memory and resources associated with it are deallocated so they can be used by other processes.
- However, the exit status is maintained in the PCB until the parent picks up the exit status using wait and deletes the PCB.
- A child process always first becomes a zombie.
- In most cases, under normal system operation zombies are immediately waited on by their parent.
- Processes that stay zombies for a long time are generally an error and cause a resource leak.

CPU Scheduling



Basic Concepts

- 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** followed by **I/O burst**
- CPU burst distribution is of main concern



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- 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**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

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)

Scheduling Algorithm 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
  - Consider one CPU-bound and many I/O-bound processes

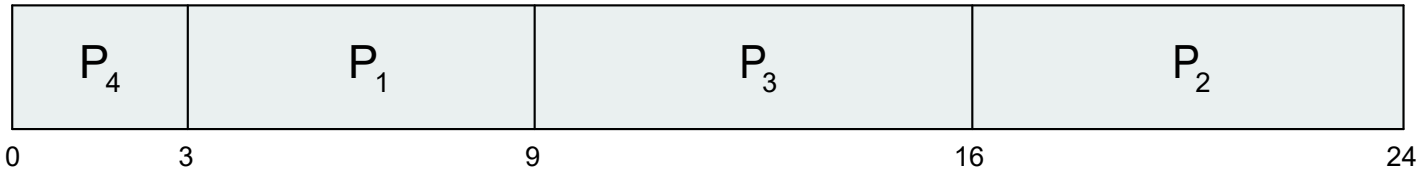
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



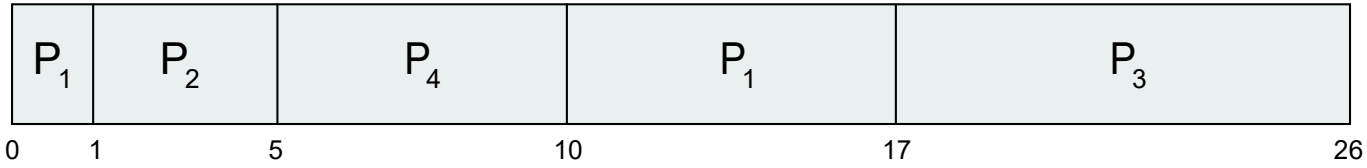
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

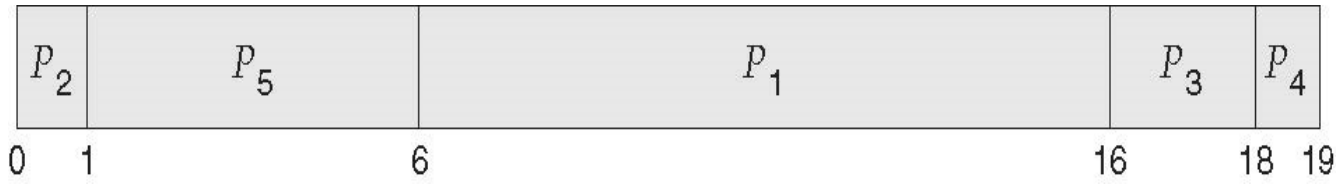
Priority Scheduling

- 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 priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

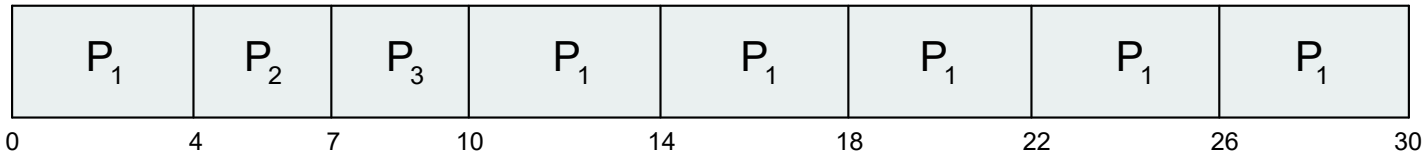
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), 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.
- Timer interrupts every quantum to schedule next process
- 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 = 4

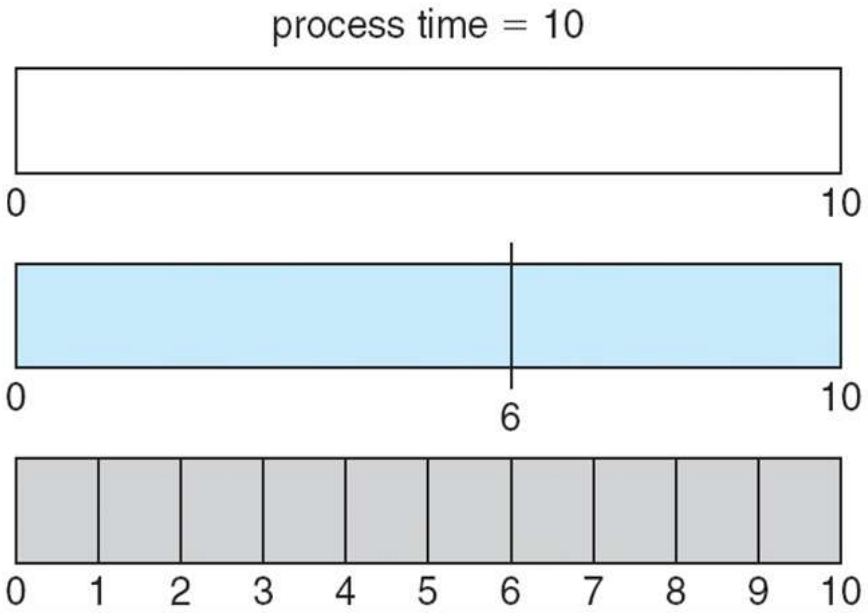
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



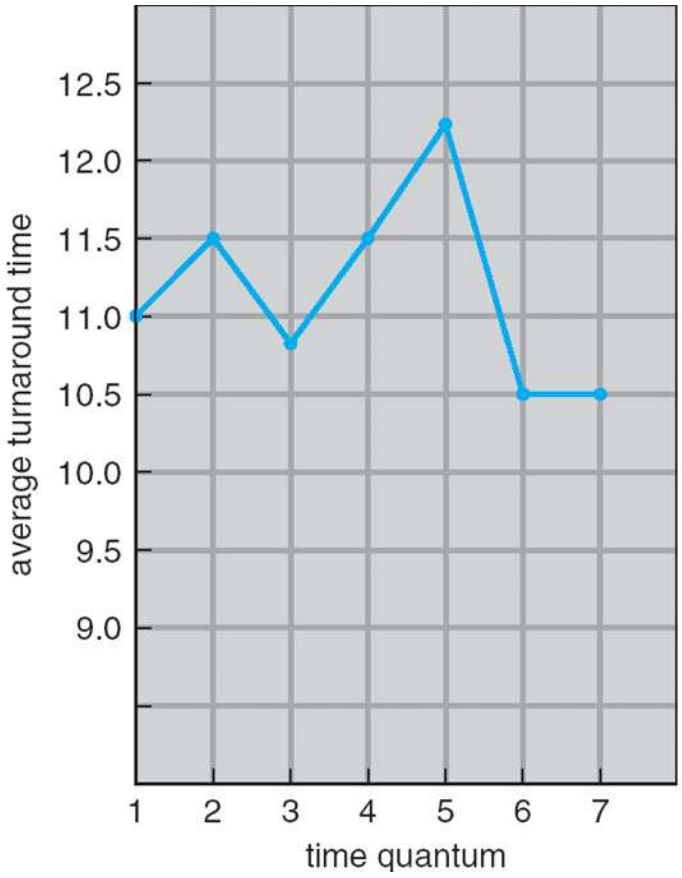
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



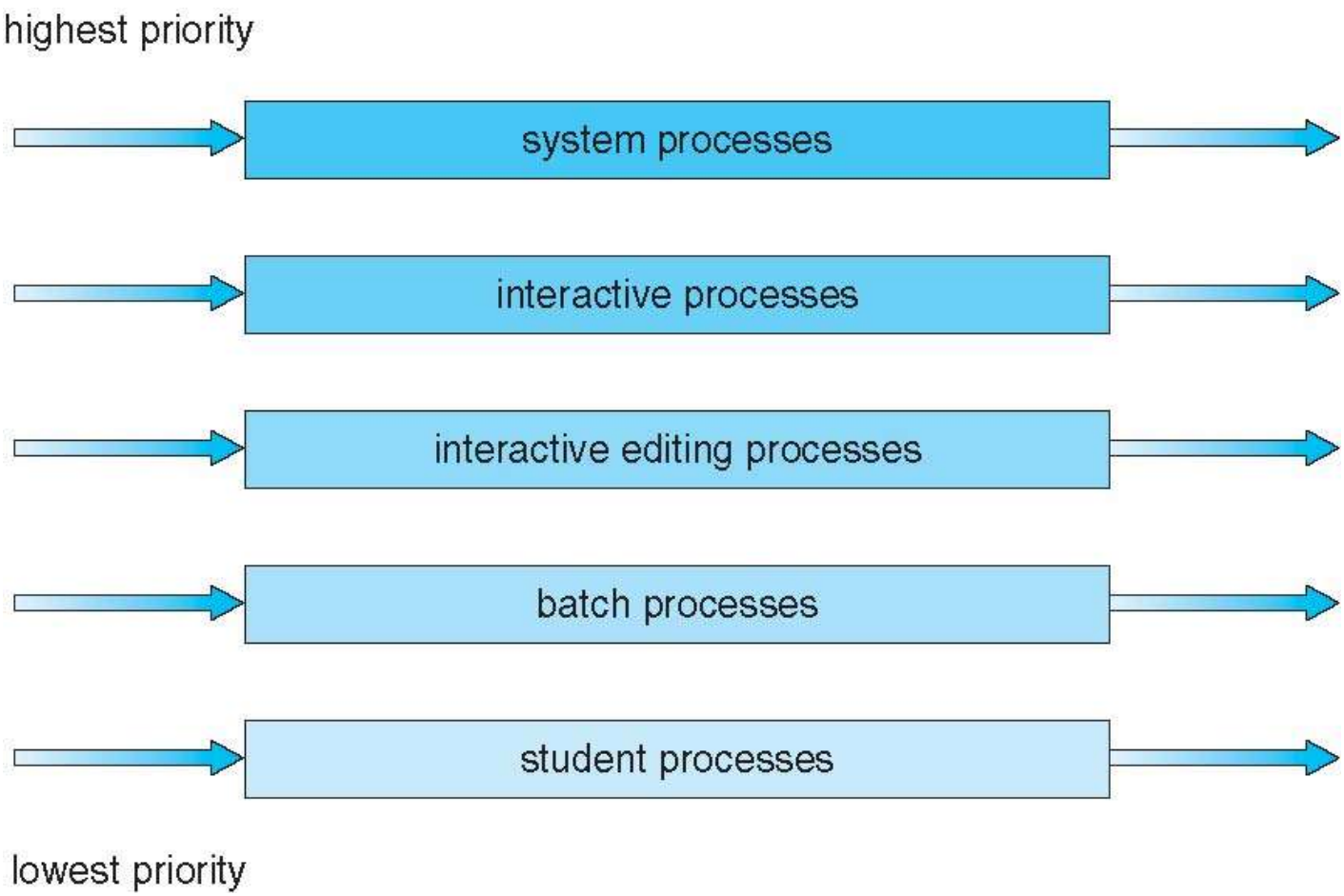
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- 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

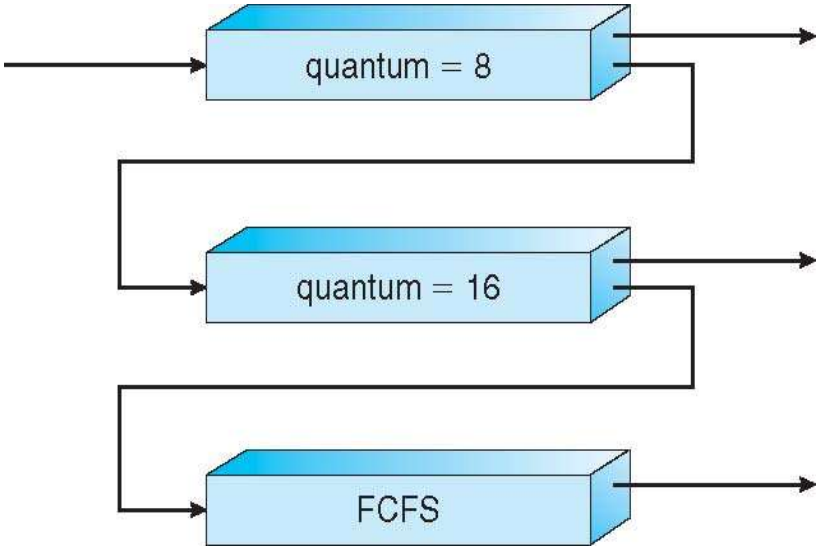


Multilevel Feedback Queue

- 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

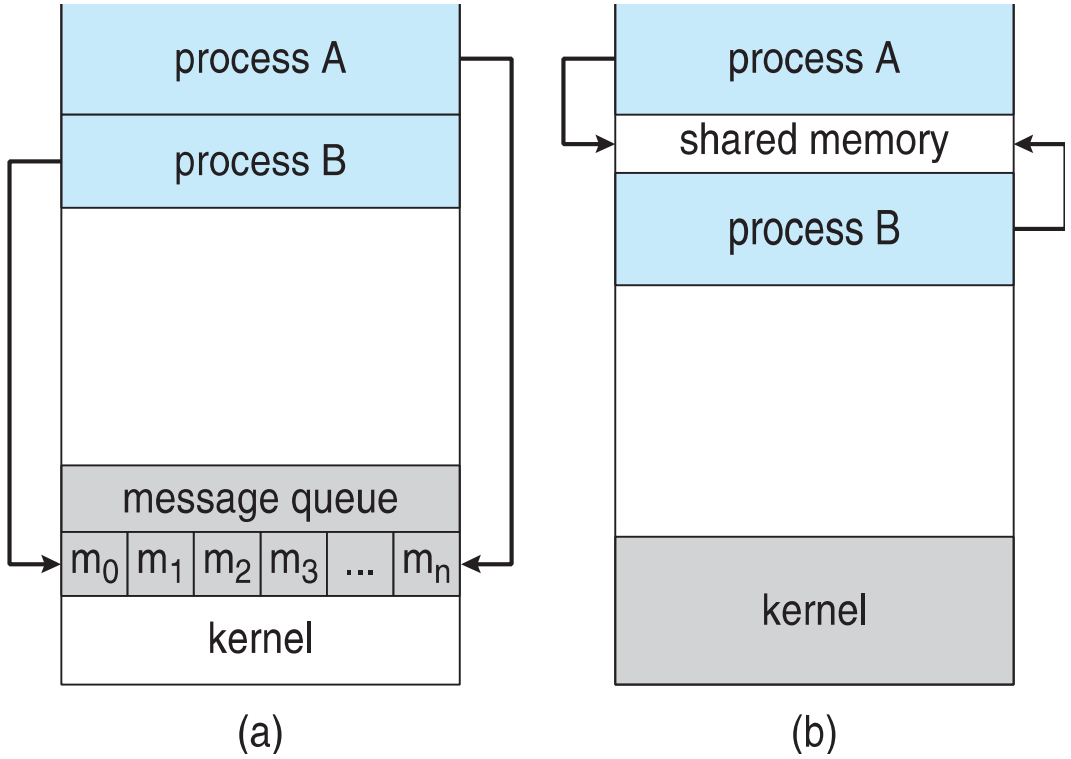


Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Producer-Consumer Problem

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

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;

while (true) {
        while (in == out)

                ; /* do nothing */
        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;


        /* consume the item in next consumed */

}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details later

Interprocess Communication – Message Passing

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

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - 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?

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 (port)
  - 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** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

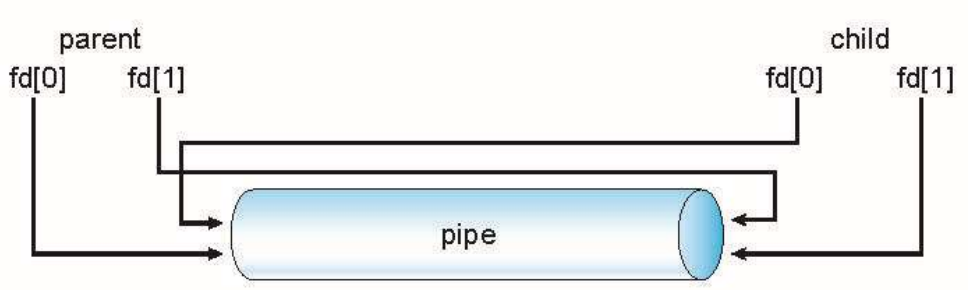
- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Pipes

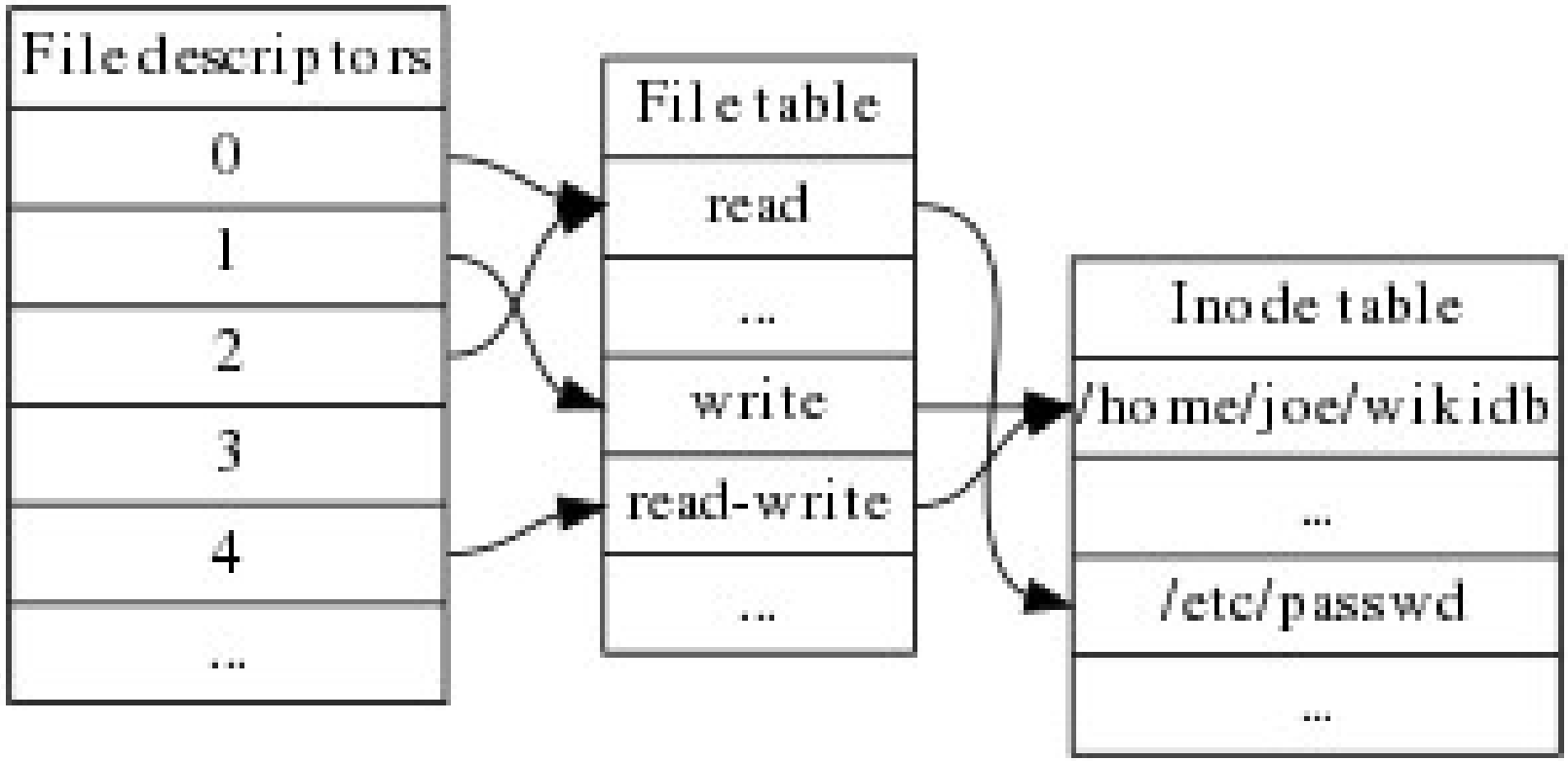
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**


File descriptors
0
1
2
3
4
...
File table
read
...
write
read-write
...
Inode table
/home/joe/wikidb
...
/etc/passwd
...

```
#include<stdio.h>
#include<sys/types.h>
int main()
{
char msg[25]=“Greetings”;
char read[25];
int fd[2];
pid_t pid;
if(pipe(fd)==-1)
{
fprintf(stderr,”Pipe failed”);
return 1;
}
pid=fork();
```

```
if(pid<0)
{
Error;
}
If(pid>0)
{
close(fd[0]);
write(fd[1],msg,strlen(msg+1));
close(fd[1]);
}
Else{
close(fd[1]);
read(fd[0],read,25);
close(fd[0]);
}return 0;}
```

Process Synchronization

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

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

Producer-Consumer Problem

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

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

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

Bounded-Buffer – Insert() Method

```
while (true) {
  /* Produce an item */
    while((((in + 1) % BUFFER SIZE count)  == out)

    // ;   /* do nothing -- no free buffers */
Busy waiting
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;

}
```

Bounded Buffer – Remove() Method

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

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
  }
```

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
       while (count == BUFFER_SIZE)
              ; // do nothing
           buffer [in] = nextProduced;
           in = (in + 1) % BUFFER_SIZE;
           count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count--;

                  /*  consume the item in nextConsumed
}
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

PROCESS SYNCHRONIZATION

Critical Sections

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

Entry Section	**Code requesting entry into the critical section.**
Critical Section	**Code in which only one process can execute at any one time.**
Exit Section	**The end of the critical section, releasing or allowing others in.**
Remainder Section	**Rest of the code AFTER the critical section.**

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] == true && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

             REMAINDER SECTION

  }
```

Process={0,1}

Flag ={ f,f}

Turn=0

Process 0	Process 1
Flag[0]=T	Flag[1]=T
Turn=1	Turn =0
While(flag[1]==t&&turn==1);	while(flag[0]=T&&turn==0);
CS	CS
Flag[0]= F	Flag[1]=F

```
var flag: array [0..1] of boolean;
turn: 0..1;
%flag[k] means that process[k] is interested in the critical section
flag[0] := FALSE;
flag[1] := FALSE;
turn := random(0..1)
```

After initialization, each process, which is called process *i* in the code (the other process is process *j*), runs the following code:

```
repeat
flag[i] := TRUE;
turn := j;
while (flag[j] and turn=j) do no-op;
CRITICAL SECTION
flag[i] := FALSE;
REMAINDER SECTION
until FALSE;
```

Information common to both processes:

```
turn = 0
flag[0] = FALSE
flag[1] = FALSE
```

EXAMPLE 1	
Process 0	Process 1
i = 0, j = 1	i = 1, j = 0
flag[0] := TRUE turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is false because flag[1] = FALSE - Since condition is false, no waiting in while loop - Enter the critical section - Process 0 happens to lose the processor	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Since condition is true, it keeps busy waiting until it loses the processor
- Process 0 resumes and continues until it finishes in the critical section - Leave critical section flag[0] := FALSE - Start executing the remainder (anything else a process does besides using the critical section) - Process 0 happens to lose the processor	
	check (flag[0] = TRUE and turn = 0) - This condition fails because flag[0] = FALSE - No more busy waiting - Enter the critical section

EXAMPLE 2	
Process 0	Process 1
i=0, j=1	i=1, j=0
flag[0] = TRUE turn = 1 - Lose processor here	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Condition is true so Process 1 busy waits until it loses the processor
check (flag[1] = TRUE and turn = 1) - This condition is false because turn = 0 - No waiting in loop - Enters critical section	

EXAMPLE 3	
Process 0	Process 1
i=0, j=1	i=1, j=0
flag[0] = TRUE - Lose processor here	
	flag[1] = TRUE turn = 0 check (flag[0] = TRUE and turn = 0) - Condition is true so, Process 1 busy waits until it loses the processor
turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is true so Process 0 busy waits until it loses the processor	
	check (flag[0] = TRUE and turn = 0) - The condition is false so, Process 1 enters the critical section

Bakery Algorithm

- N process solution

The processes share two arrays:

```
boolean flag choosing[i] initially false ;
integer num[i] initially 0
```

The entry and exit codes for process i are:

```
entry(i) {
   i.1: choosing[i] := TRUE ;
   i.2: num[i] := max( num[0], num[1], ... , num[N-1] ) + 1 ;
   i.3: choosing[i] := FALSE ;
        for p := 0 to N-1 do  {       // p is local to process i
   i.4:   while choosing[p] do skip ;
   i.5:   while num[p] != 0 and (num[p],p)<(num[i],i) do skip ;
        }
}

exit(i) {
   i.6: num[i] := 0 ;
}
```

Processes = {P1,P2,P3}

Choosing = { F, F, F}

Num= {0,0,0}

CS

Num[i]=0

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

```
boolean waiting[n];
boolean lock;
lock = FALSE; waiting[i] = FALSE;
do
{
waiting[i]=TRUE;
Key=TRUE;
While(waiting[i] && key)
        key=TestAndSet(&lock);
Waiting[i]=FALSE;
//CS
j=(i+1)%n;
While((j!=i)&&!waiting[j])
        j=(j+1)%n;
If(j==i)
        lock=FALSE;
Else
        Waiting[j]=FALSE;
//Remainder section
} while(TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );

                //    critical section

         lock = FALSE;

                //      remainder section

}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

```
        while S <= 0
            ; // no-op
          S--;
    }
```

  - signal (S) {

```
      S++;
    }
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

    Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting (spin lock)

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation

```
typedef struct
{
int value;
struct process * list;
}semaphore;
```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (semaphore *S){
    S->value--;
    if (S->value < 0) {
            add this process to S->list
             block();  }
 }
```

- Implementation of signal:

```
Signal (semaphore *S){
      S-> value++;
       if (S->value > 0) {
              remove a process P from the waiting queue
               wakeup(P);  }
}
```

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

Mutex=1, full=0,empty=N

```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

     signal (mutex);
     signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

              //  remove an item from  buffer

        signal (mutex);
        signal (empty);

              //  consume the removed item

}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.

- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

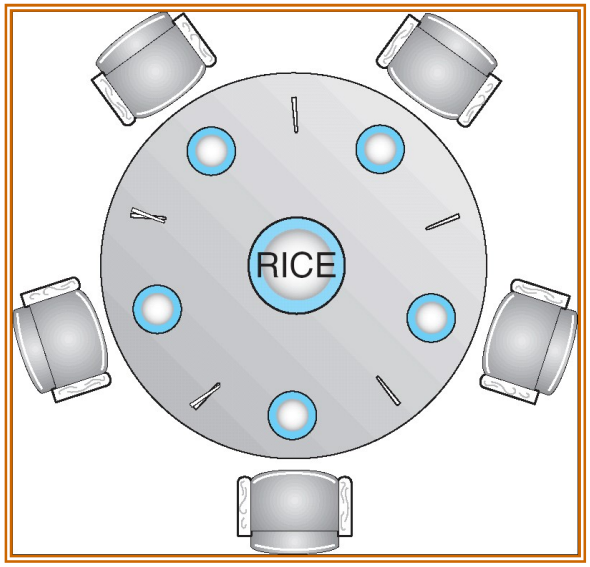
- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)  wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (redacount  == 0)  signal (wrt) ;
        signal (mutex) ;
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Philo i

Wait(chopstick[0]); //0

Wait(chopstick[1]); //0

Eat

Signal(chopstick[0]);

Signal(chopstick[1]);

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)  {
        wait ( chopstick[i] );
          wait ( chopStick[ (i + 1) % 5] );

                // eat

          signal ( chopstick[i] );
          signal (chopstick[ (i + 1) % 5] );

             // think

}
```

Problems with Semaphores

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

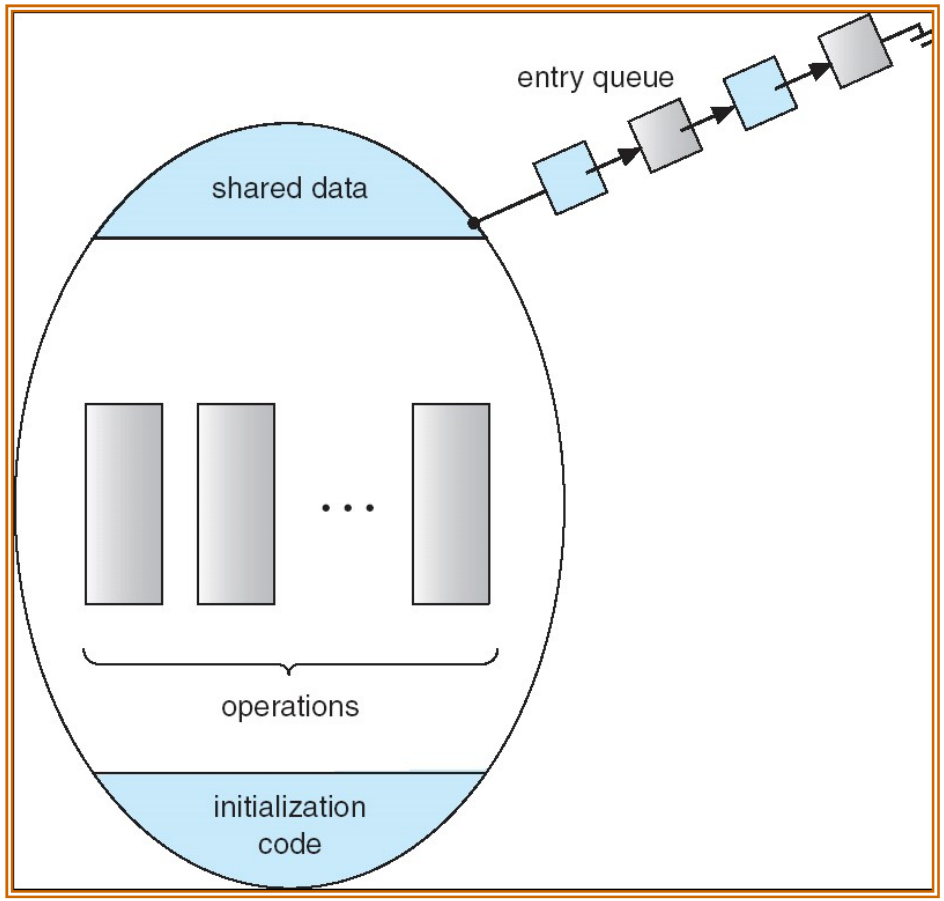
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
   }
}
```

Schematic view of a Monitor



Condition Variables

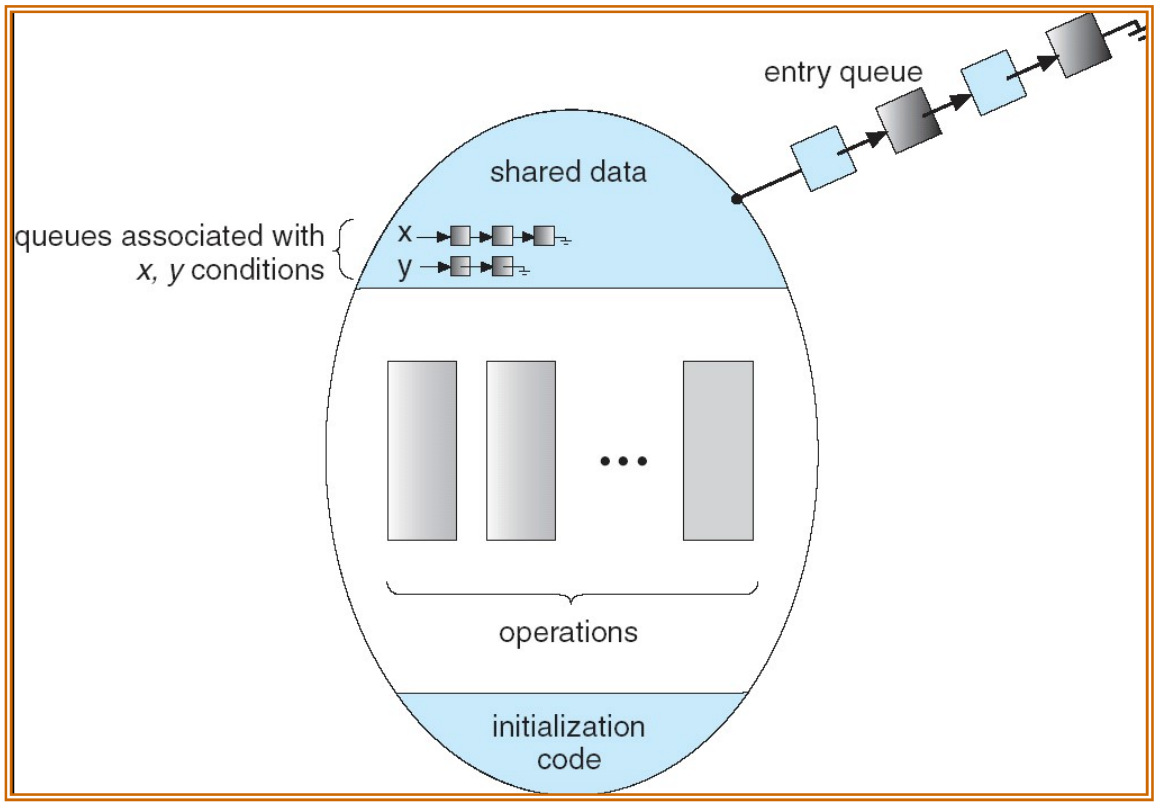
- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

  Two possibilities for signal

  1. Signal and wait: P either waits until Q leaves the monitoror waits for another condition
  2. Signal and continue: Q either waits until P leaves the monitor or waits for another condition.

Monitor with Condition Variables



Producer - Consumer

```
monitor PC
  {
int slots=0;
condition full, empty;

Void producer()
{
while(slots==N)    empty.wait();
slots++;
full.signal();
}

Void consumer()
{
While(slots==0) full.wait();
slots--;
empty.signal();
}
```

Solution to Dining Philosophers

```
monitor DP
  {
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait();
   }

      void putdown (int i) {
         state[i] = THINKING;
               // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
       }
```

Solution to Dining Philosophers (cont)

```
    void test (int i) {
          if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
              state[i] = EATING ;
                self[i].signal () ;
           }
     }

       initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
     }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Deadlocks

Deadlocks

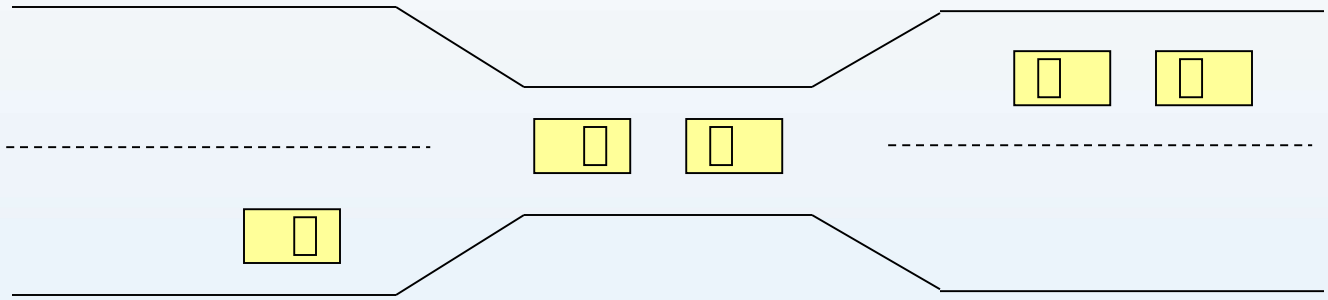
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

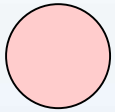
Resource-Allocation Graph

A set of vertices V and a set of edges E.

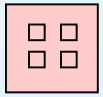
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

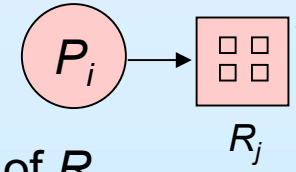
- Process



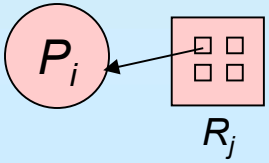
- Resource Type with 4 instances



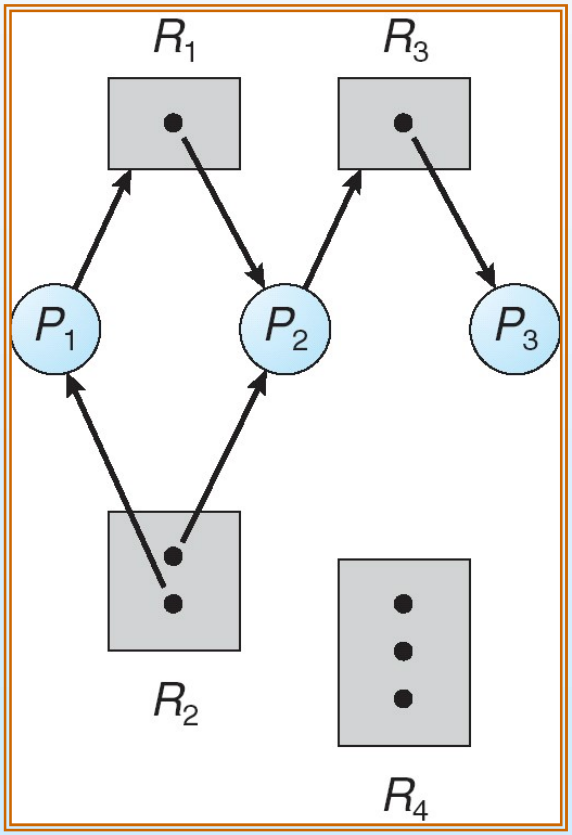
- P_i requests instance of R_j



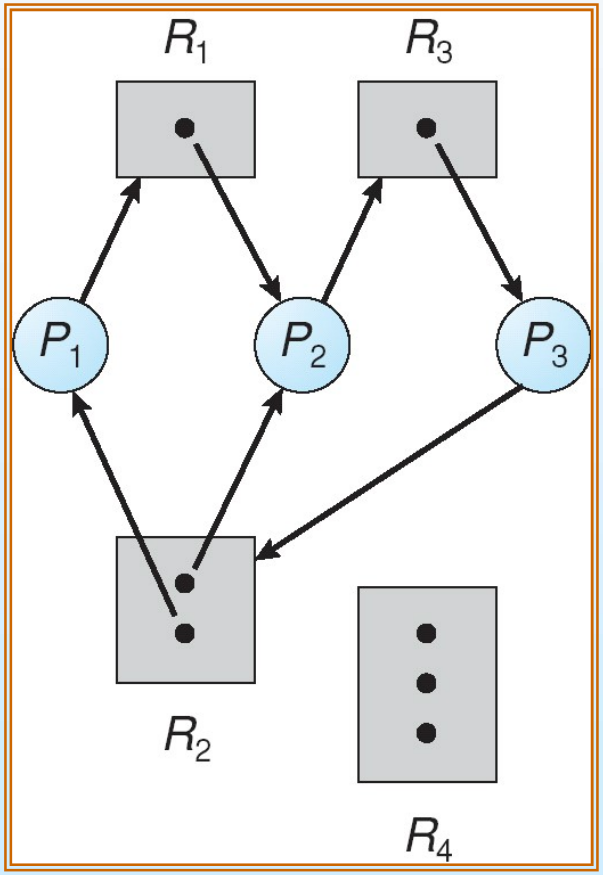
- P_i is holding an instance of R_j



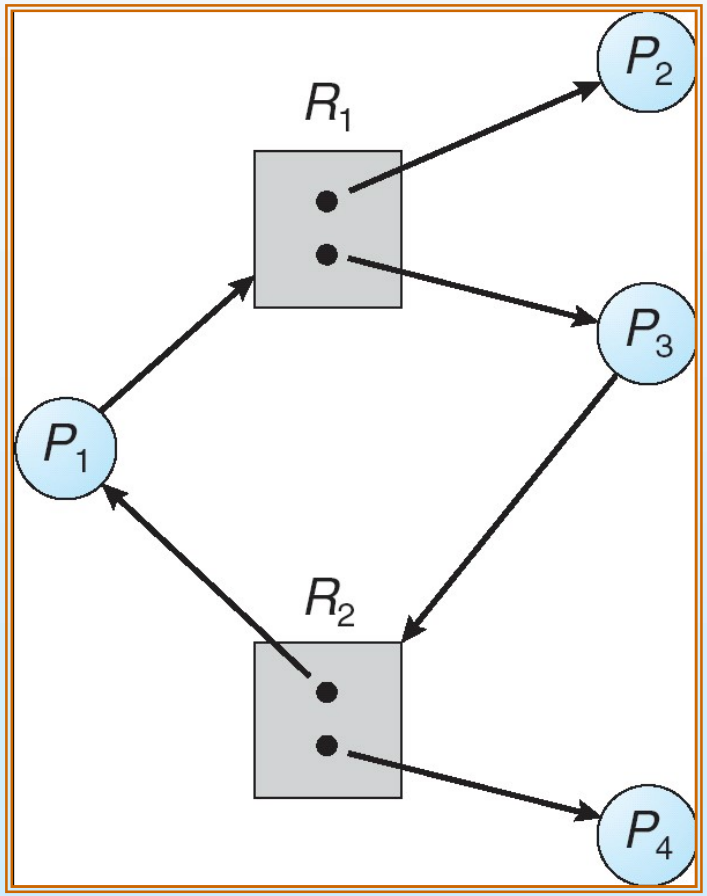
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

- Let R ={ R1,R2,R3}
- Assign a number for a resource type : f: R->N
- Eg: F(R1)=2
- Each process can request resources only in the increasing order of enumeration. ie: F(Rj) > F(Ri). This will avoid circular wait.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

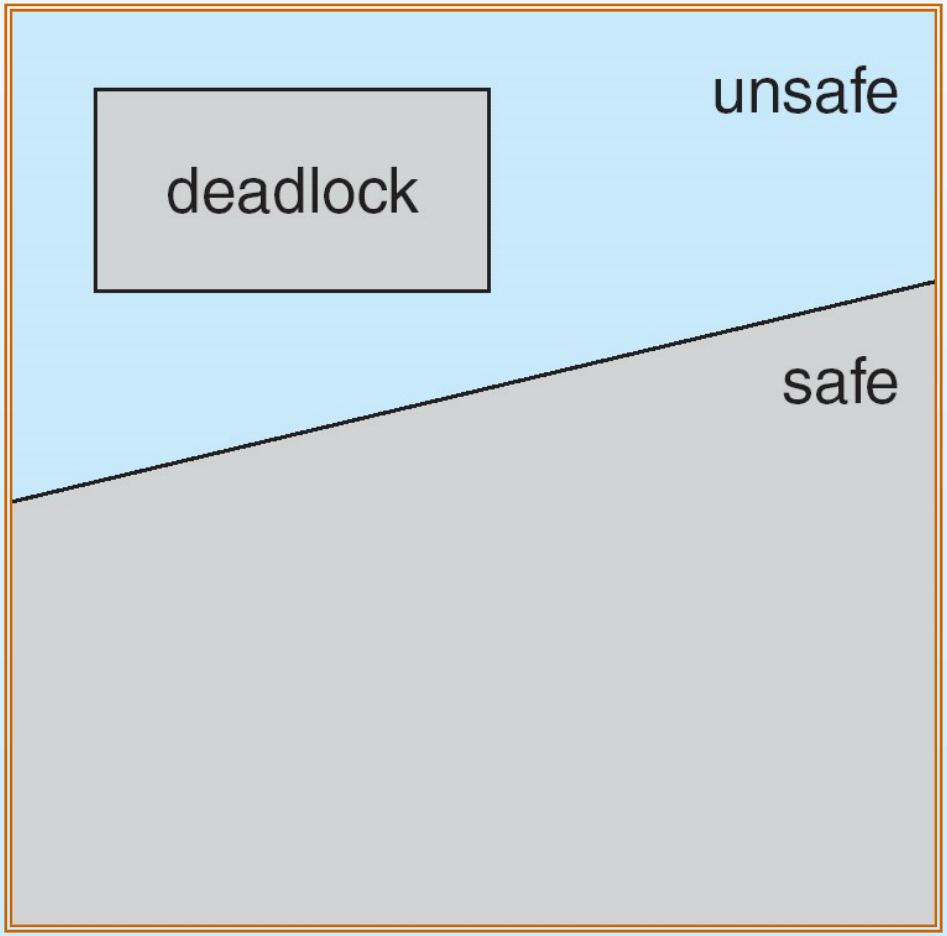
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



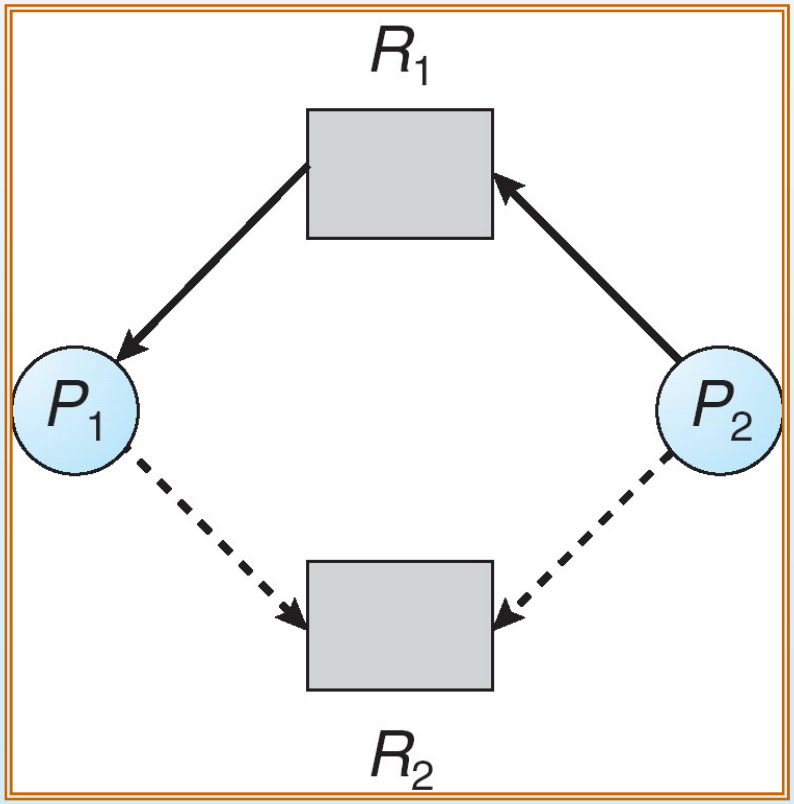
Avoidance algorithms

- Single instance of a resource type. Use a resource-allocation graph
- Multiple instances of a resource type. Use the banker's algorithm

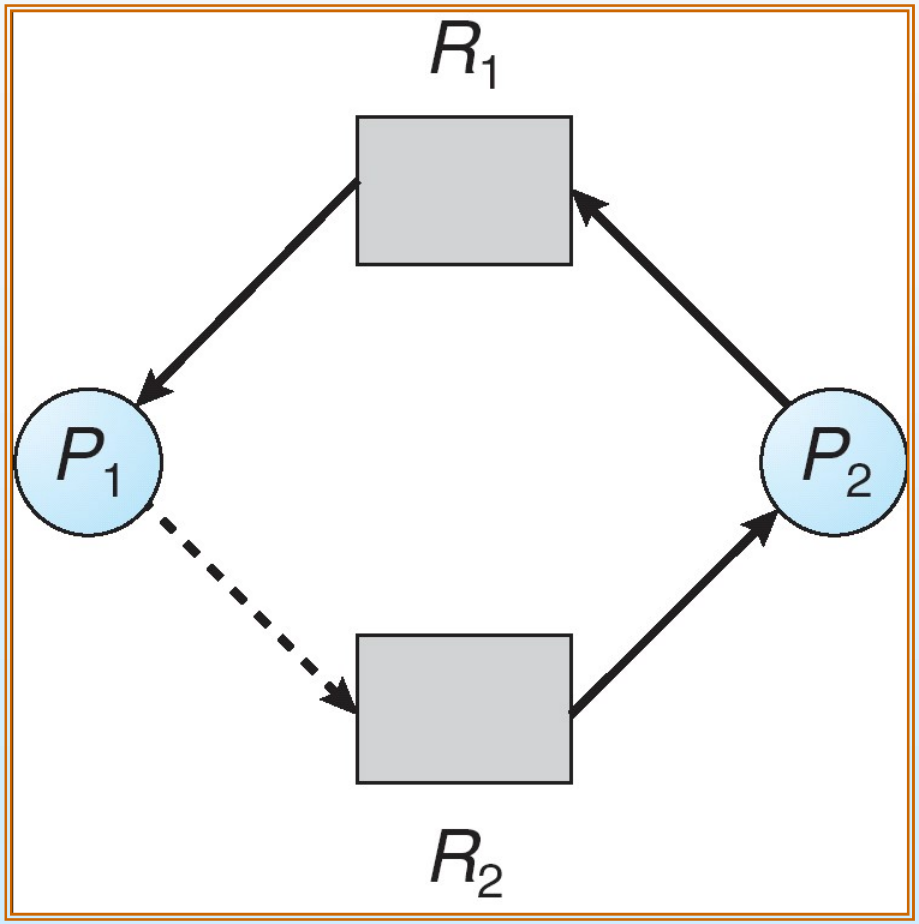
Resource-Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- ***Available***: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- ***Max***: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- ***Allocation***: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- ***Need***: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task.

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances).

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

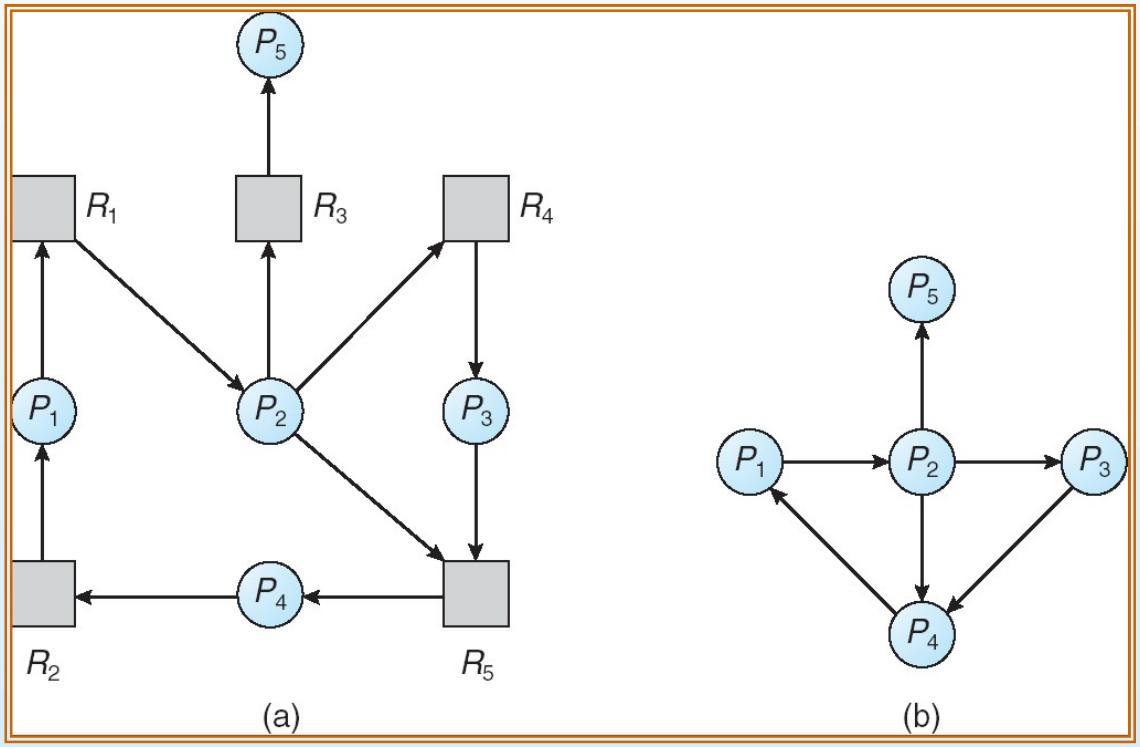
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- ***Available***: A vector of length *m* indicates the number of available resources of each type.
- ***Allocation***: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- ***Request***: An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

   (a) *Work* = *Available*

   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

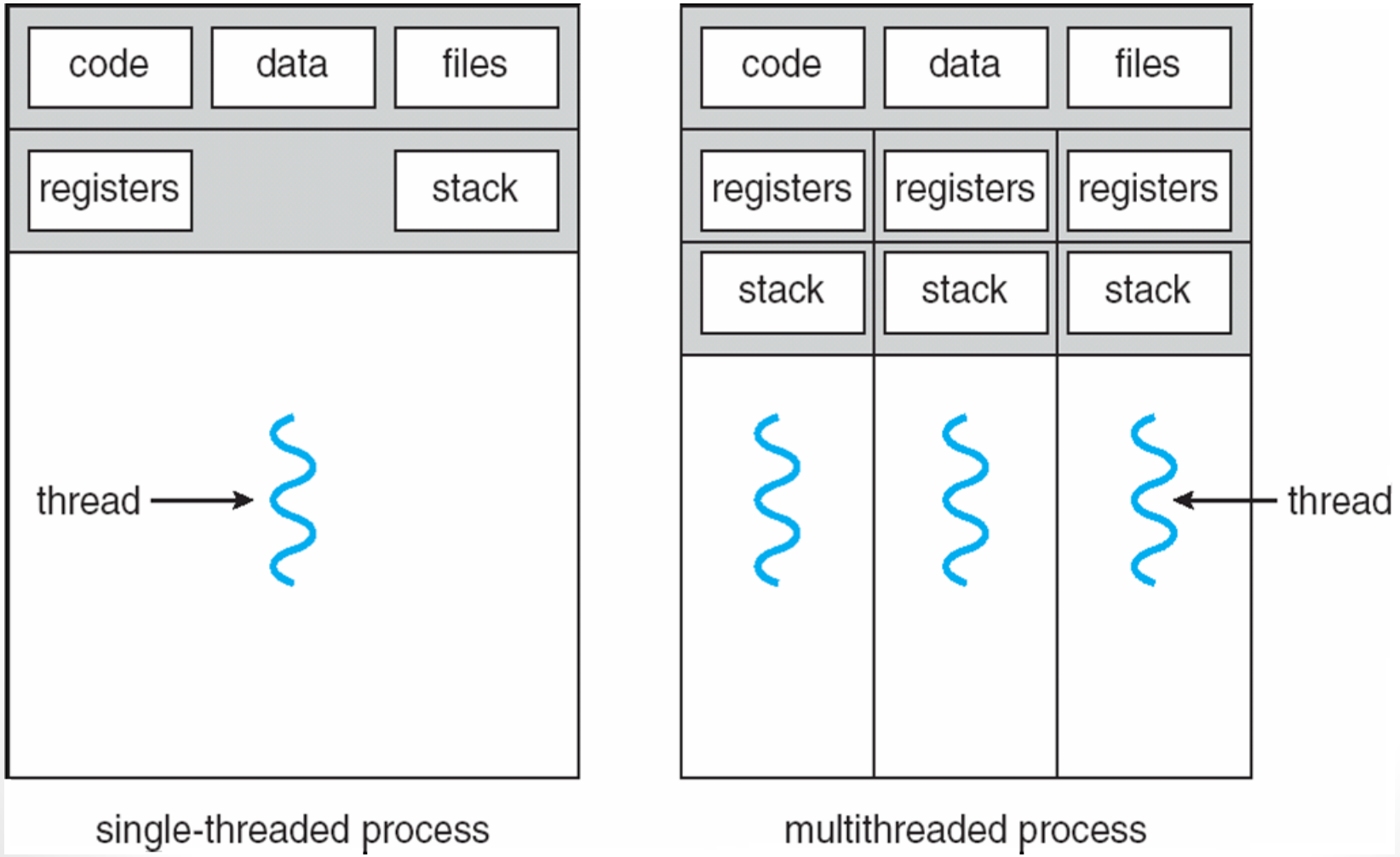
Multithreaded programming

D. Geraldine Bessie Amali
Assistant Professor
SCOPE

Definition

- A thread is a basic unit of CPU utilization.
- It comprises of a thread Id, a program counter , register set and a stack.
- It shares the code , data and other resources with other threads belonging to the same process.
- Multithreading-The ability of an OS to support multiple, concurrent paths of execution within a single process

Single and Multithreaded Processes



Single thread Vs multi thread

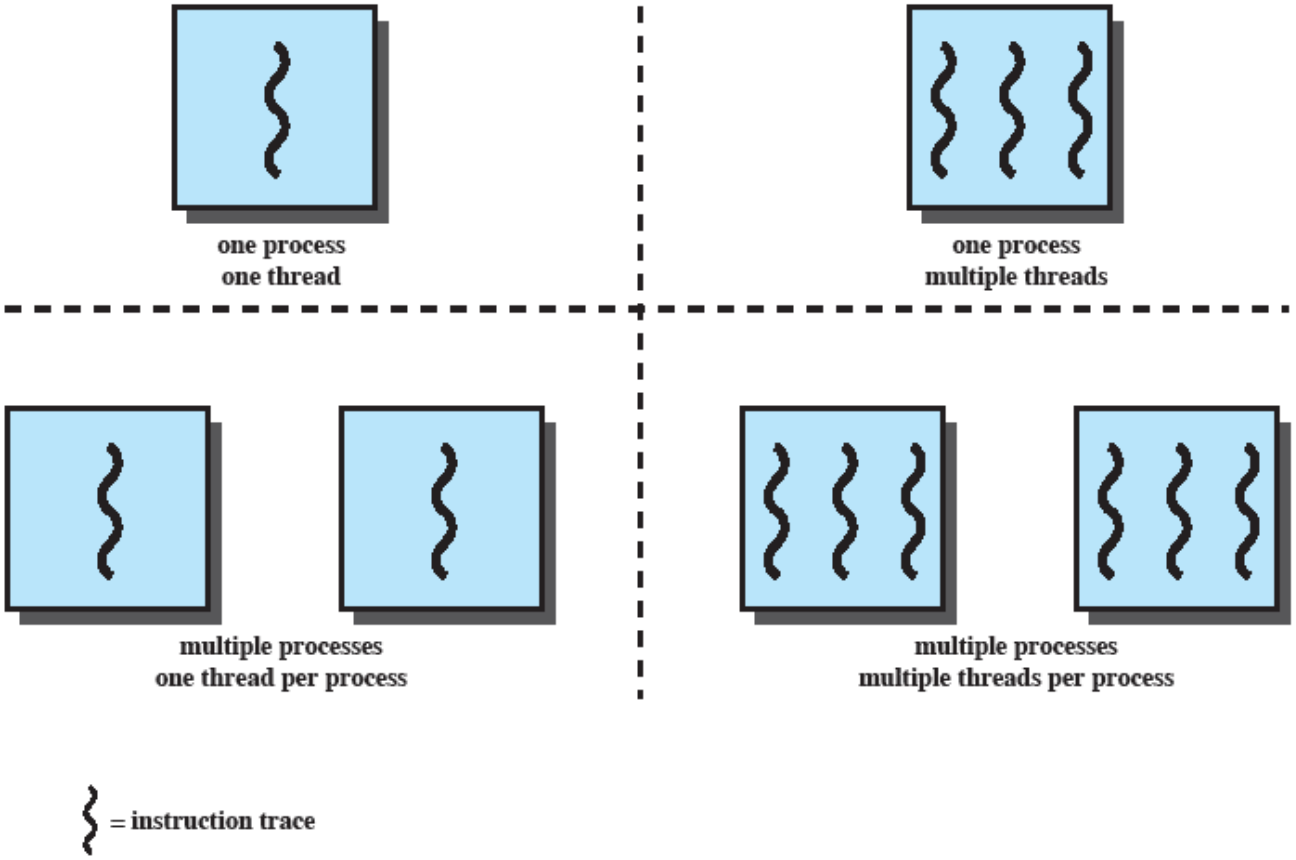


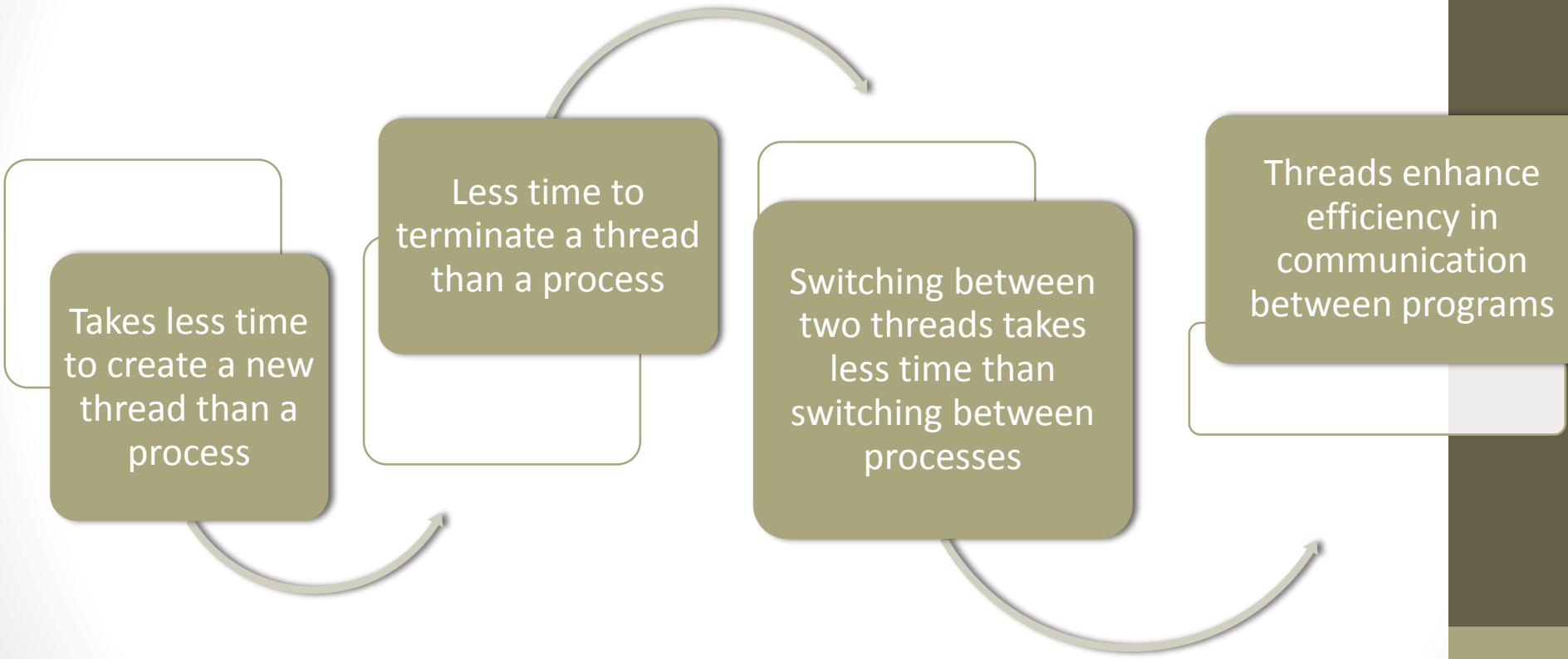
Figure 4.1 Threads and Processes [ANDE97]

Each thread has:

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability


Takes less time to create a new thread than a process
Less time to terminate a thread than a process
Switching between two threads takes less time than switching between processes
Threads enhance efficiency in communication between programs

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- **Most of the state information dealing with execution is maintained in thread-level data structures**
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

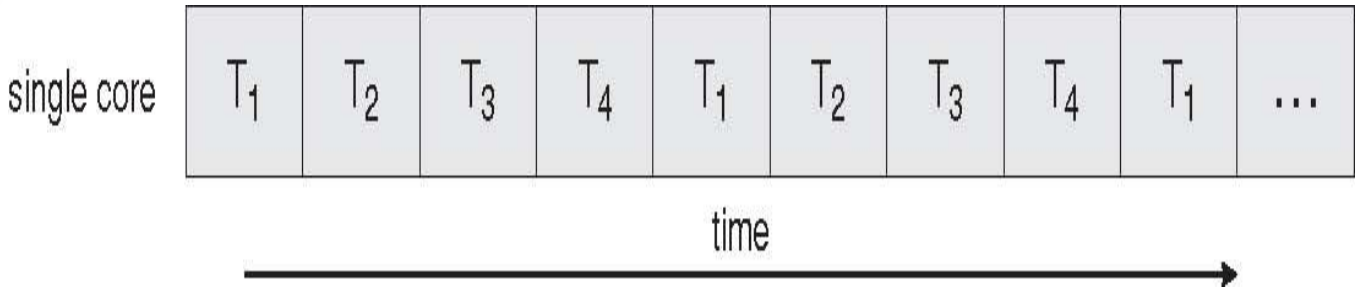
Thread operations associated with a change in thread state are:

- Spawn (create)
- Block
- Unblock
- Finish

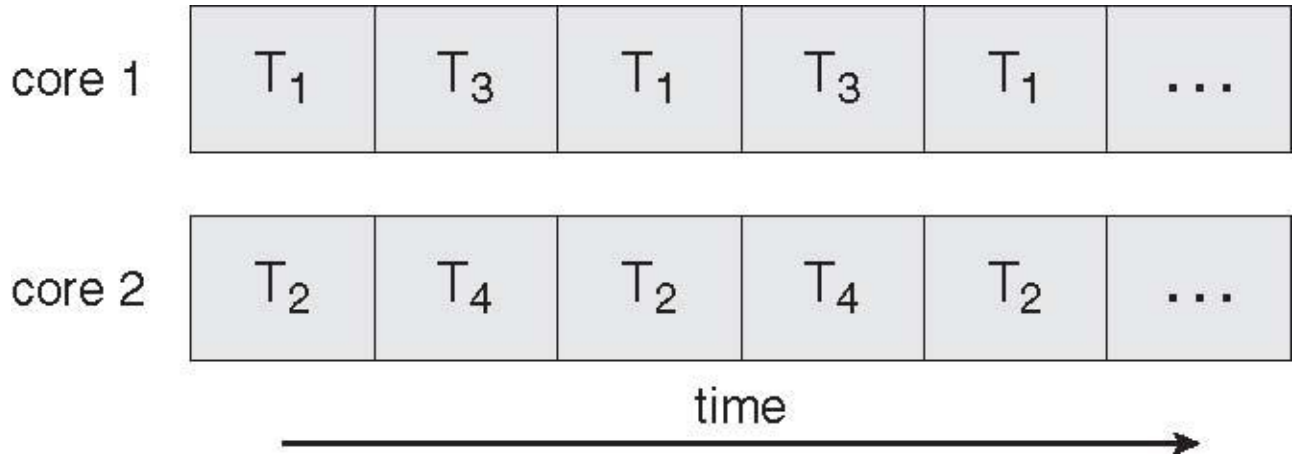
Thread Execution

- A key issue with threads is whether or not they can be scheduled independently of the process to which they belong.
- Or, is it possible to block one thread in a process without blocking the entire process?
  - If not, then much of the flexibility of threads is lost.

Multicore programming



Concurrent execution on a single core system

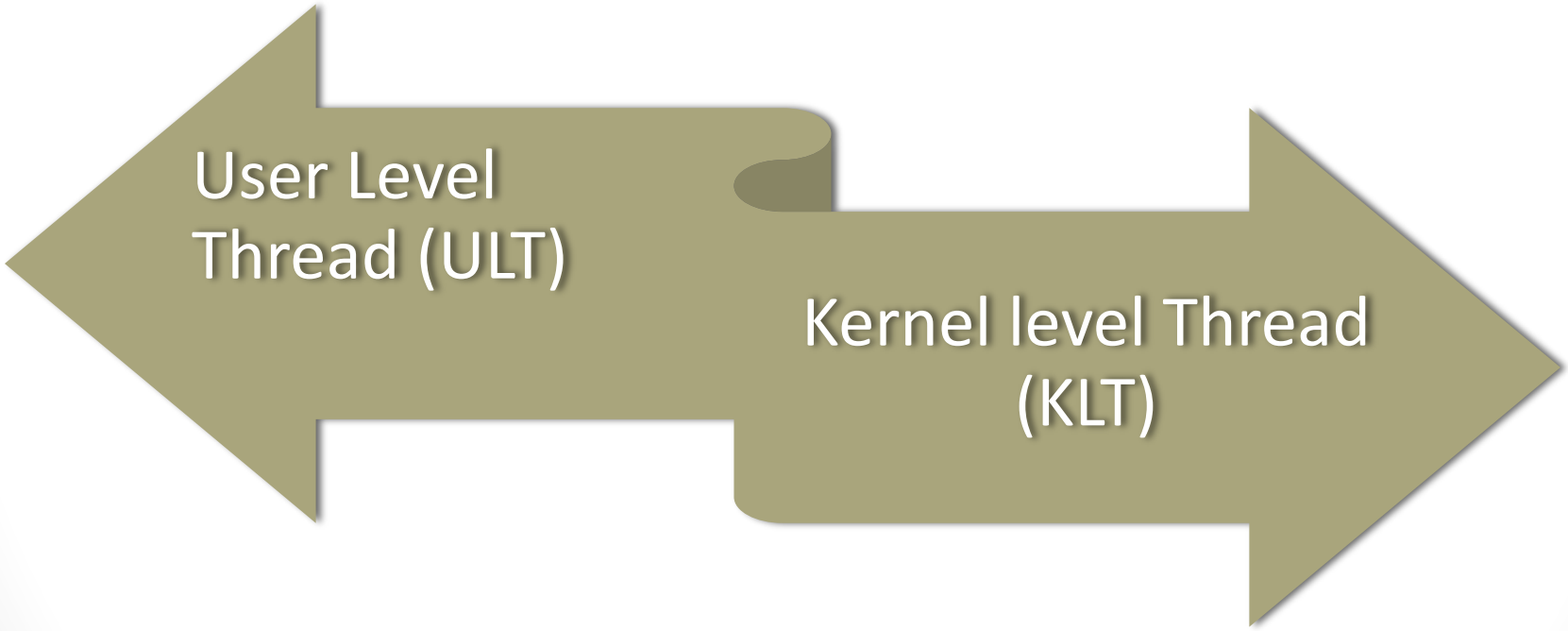


Parallel execution on a multi core system

Challenges in programming for multi core systems

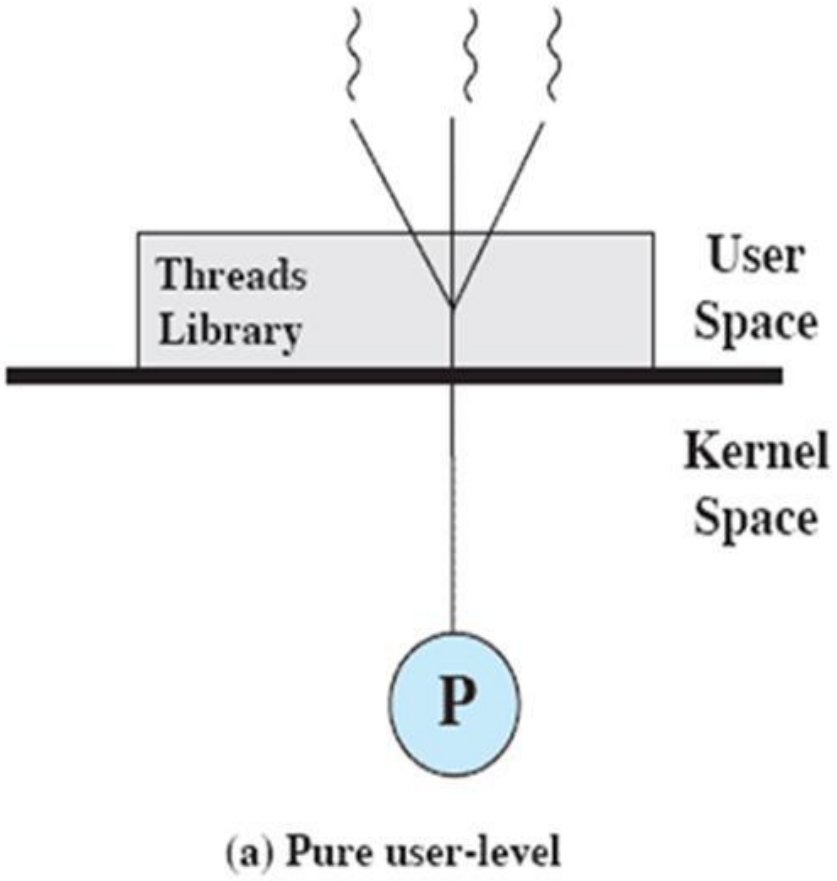
- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

Types of Threads



User-Level Threads (ULTs)

- Thread management is done by the application
- The kernel is not aware of the existence of threads

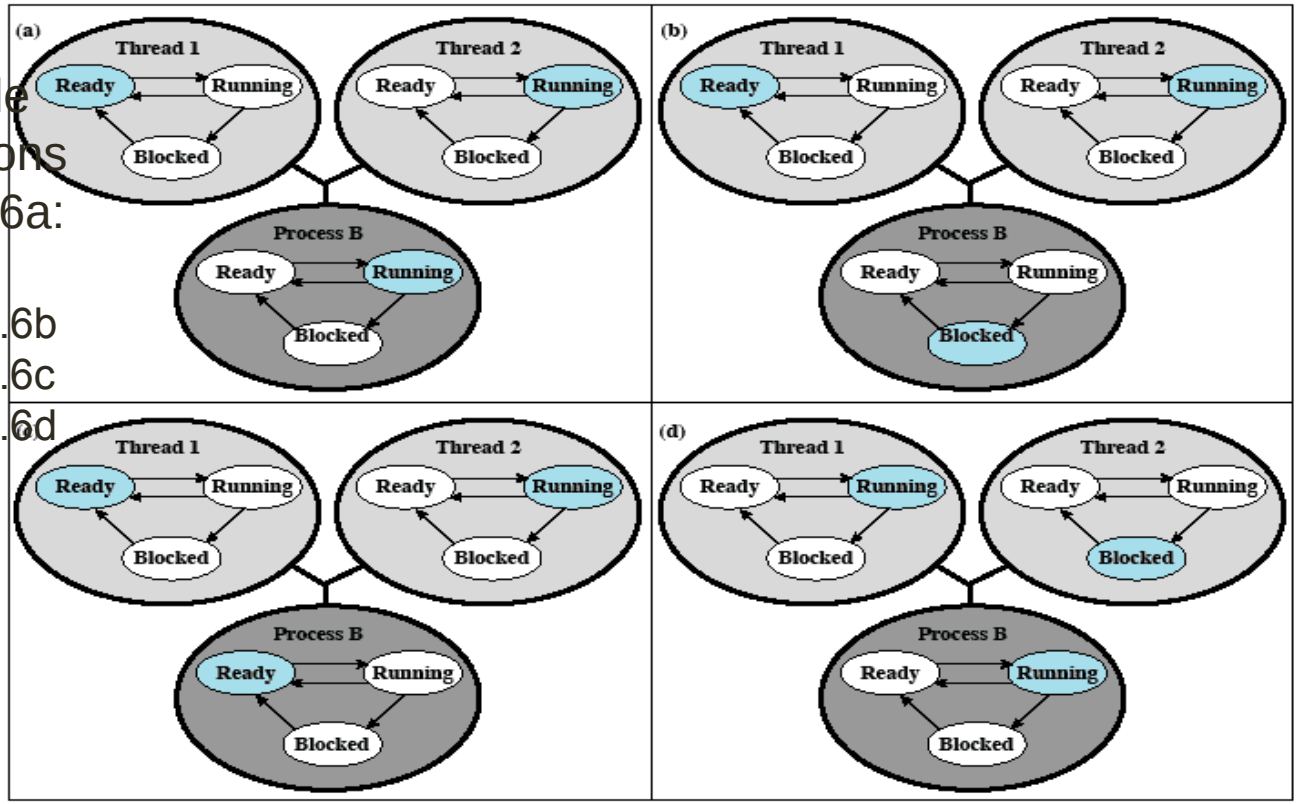


(a) Pure user-level

Relationships Between ULT States and Process States

Possible transitions from 4.6a:

4.6a→4.6b
4.6a→4.6c
4.6a→4.6d

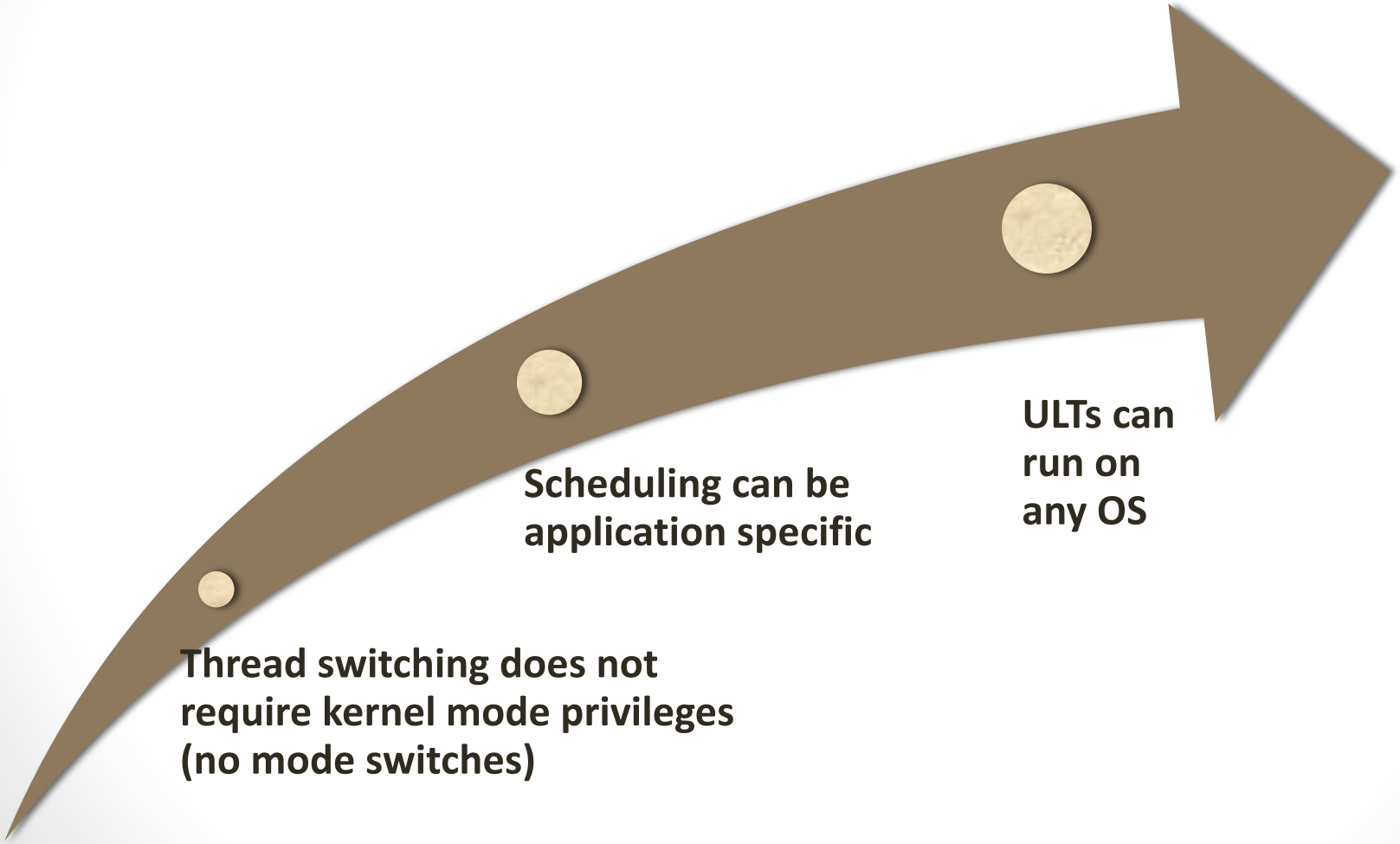


Colored state is current state

Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

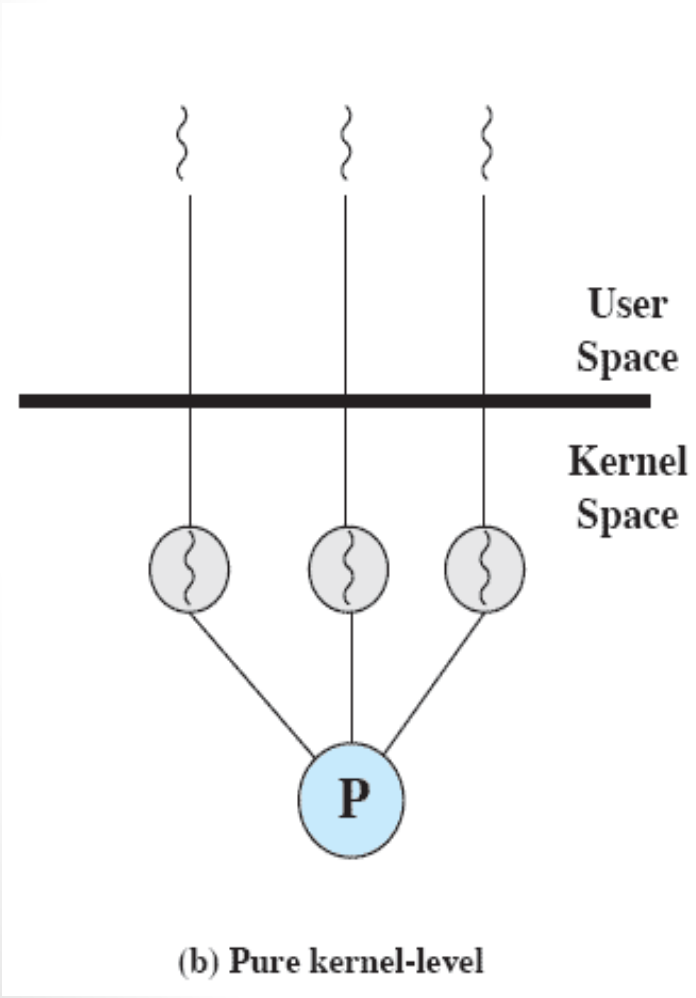
Advantages of ULTs



Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel (could call them K*M*T)
  - no thread management is done by the application
  - Windows is an example of this approach

Advantages of KLTs

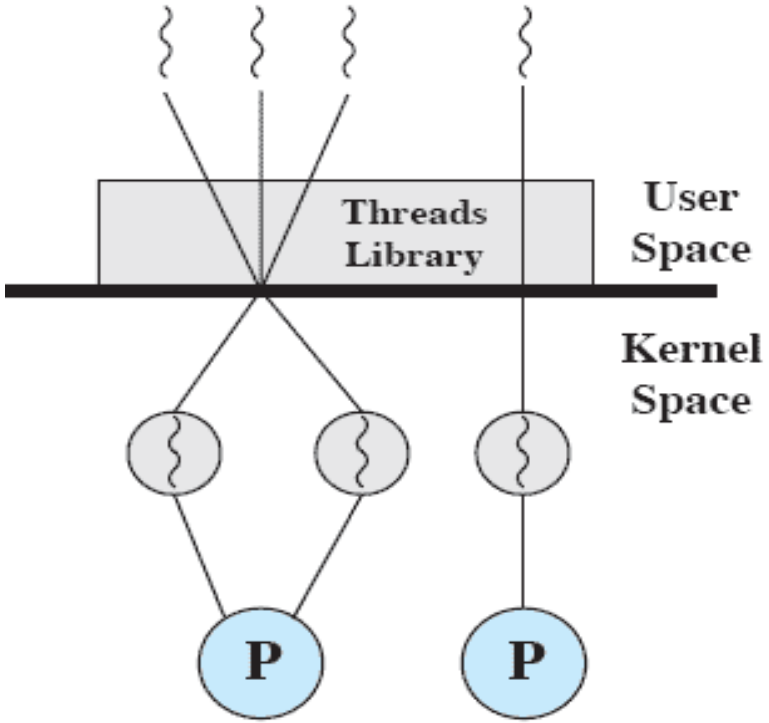
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process

Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example

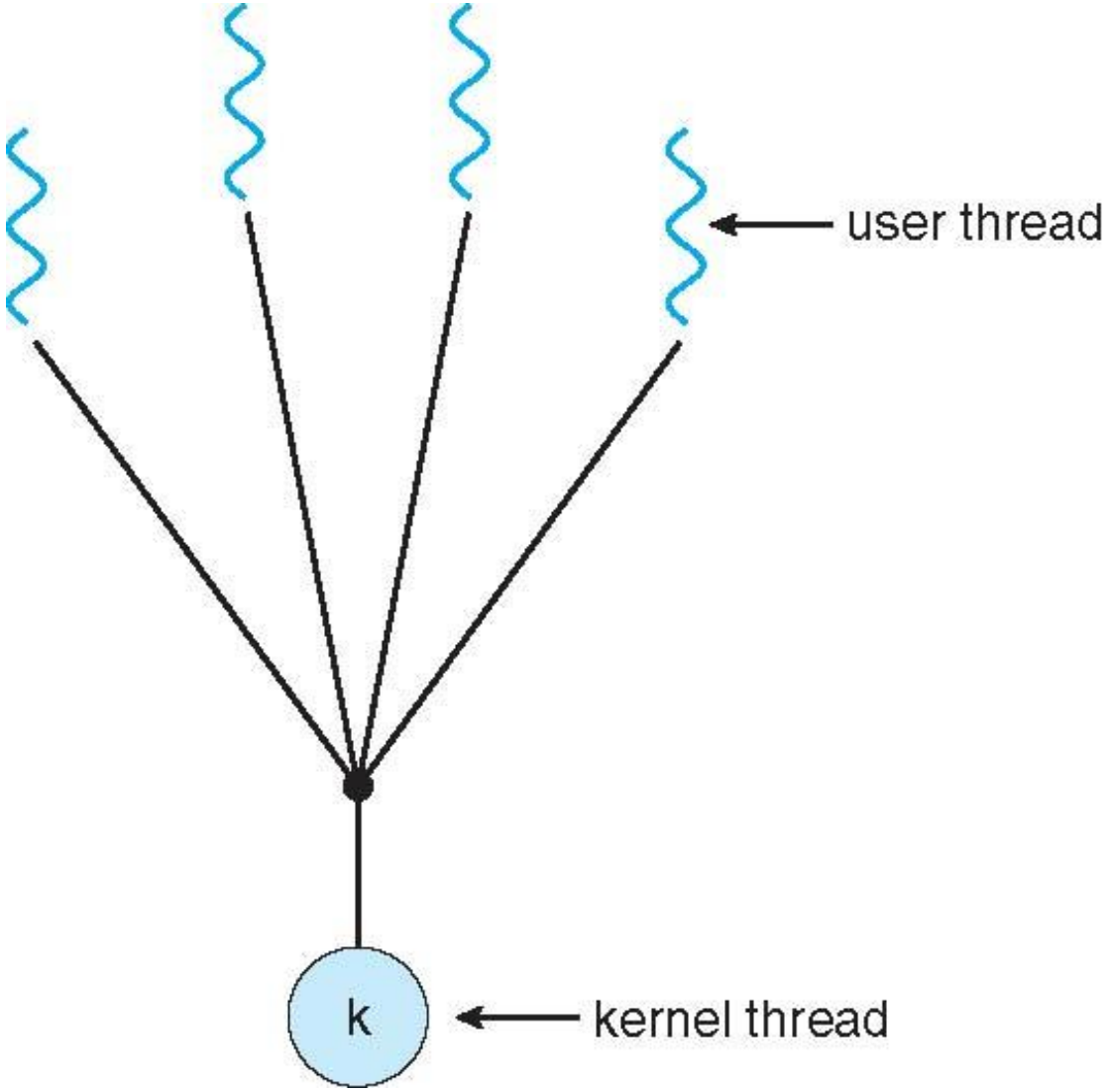


(c) Combined

Multithreading Models

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

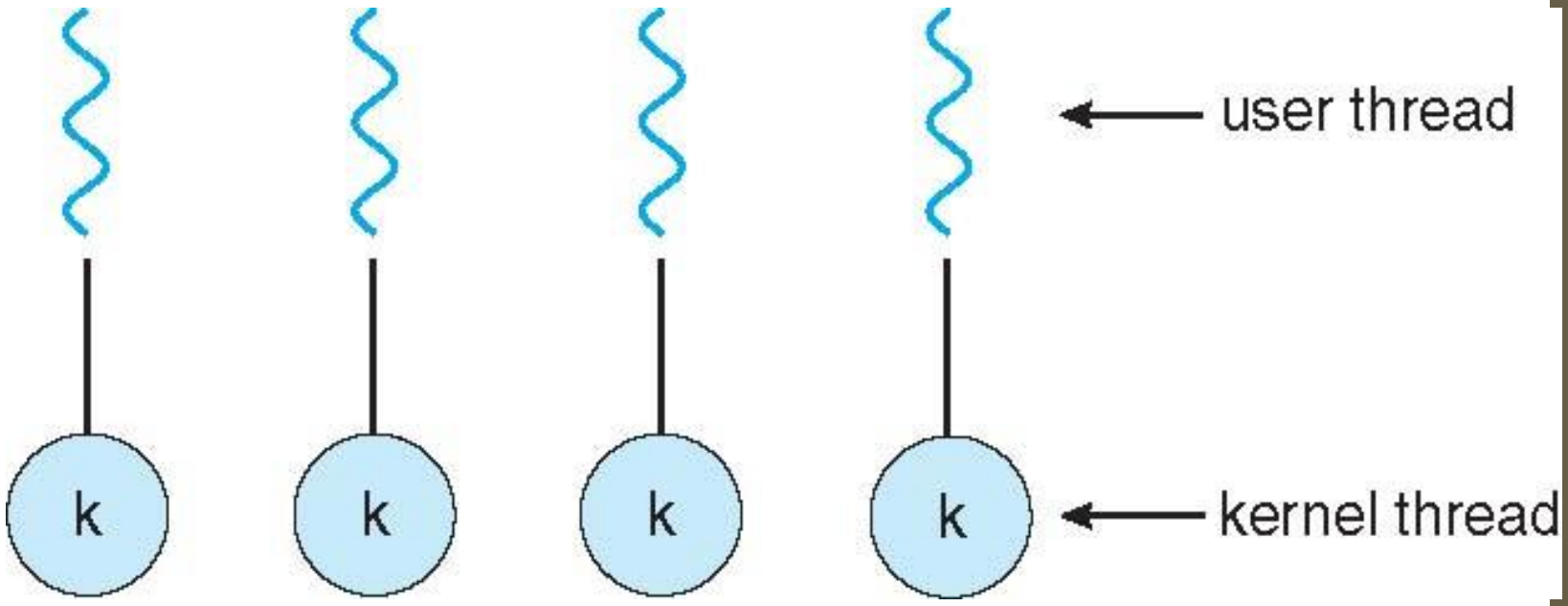
Many-to-One Model



Many-to-One Model

- Many user-level threads mapped to single kernel thread
- User-level threads can be concurrent without being parallel, thread switching incurs low overhead, and blocking of a user-level thread leads to blocking of all threads in the process.
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

One-to-one Model

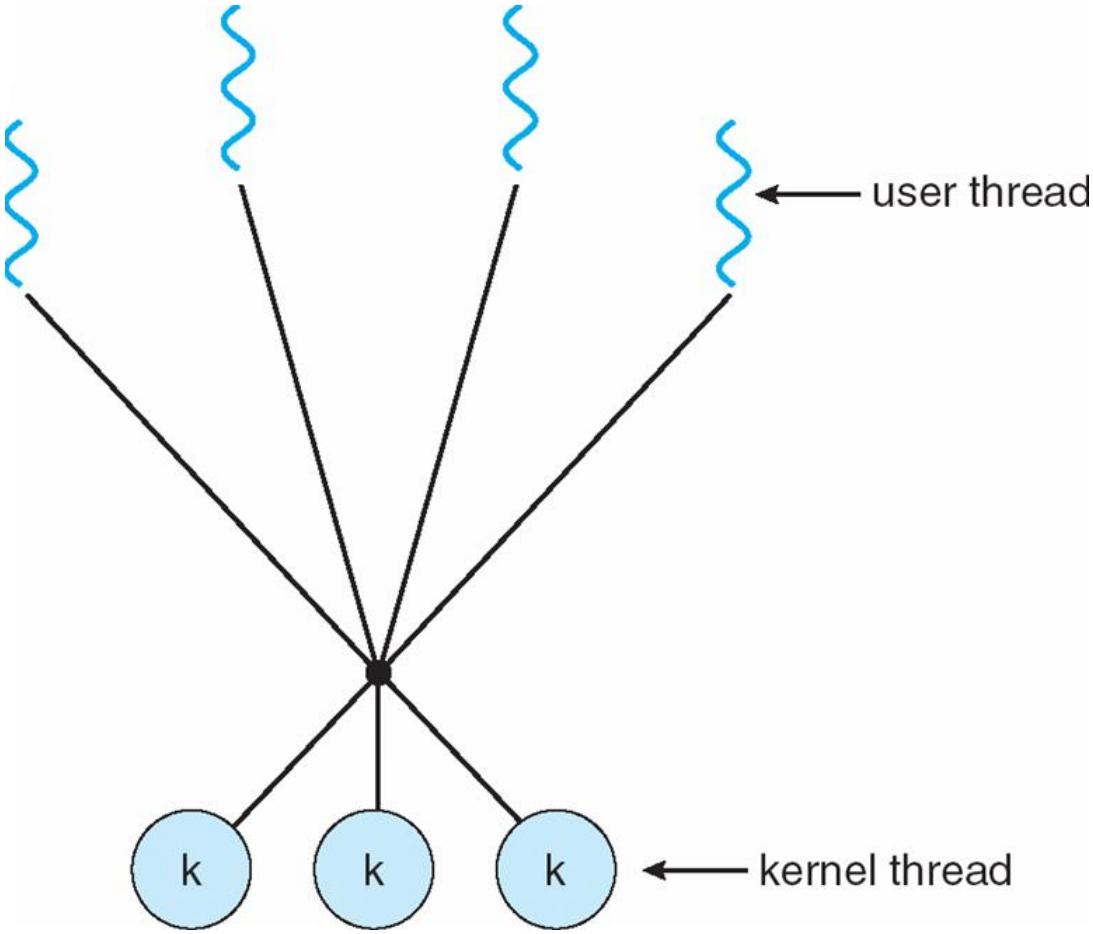


One-to-One

- Each user-level thread maps to kernel thread
- Threads can operate in parallel on different CPUs of a multiprocessor system; however, switching between threads is performed at the kernel level and incurs high overhead.
- Blocking of a user-level thread does not block other user-level threads of the process because they are mapped into different kernel-level threads.

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

Many-to-Many Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads

- It provides parallelism between user-level threads that are mapped into different kernel-level threads at the same time, and provides low overhead of switching.

Examples

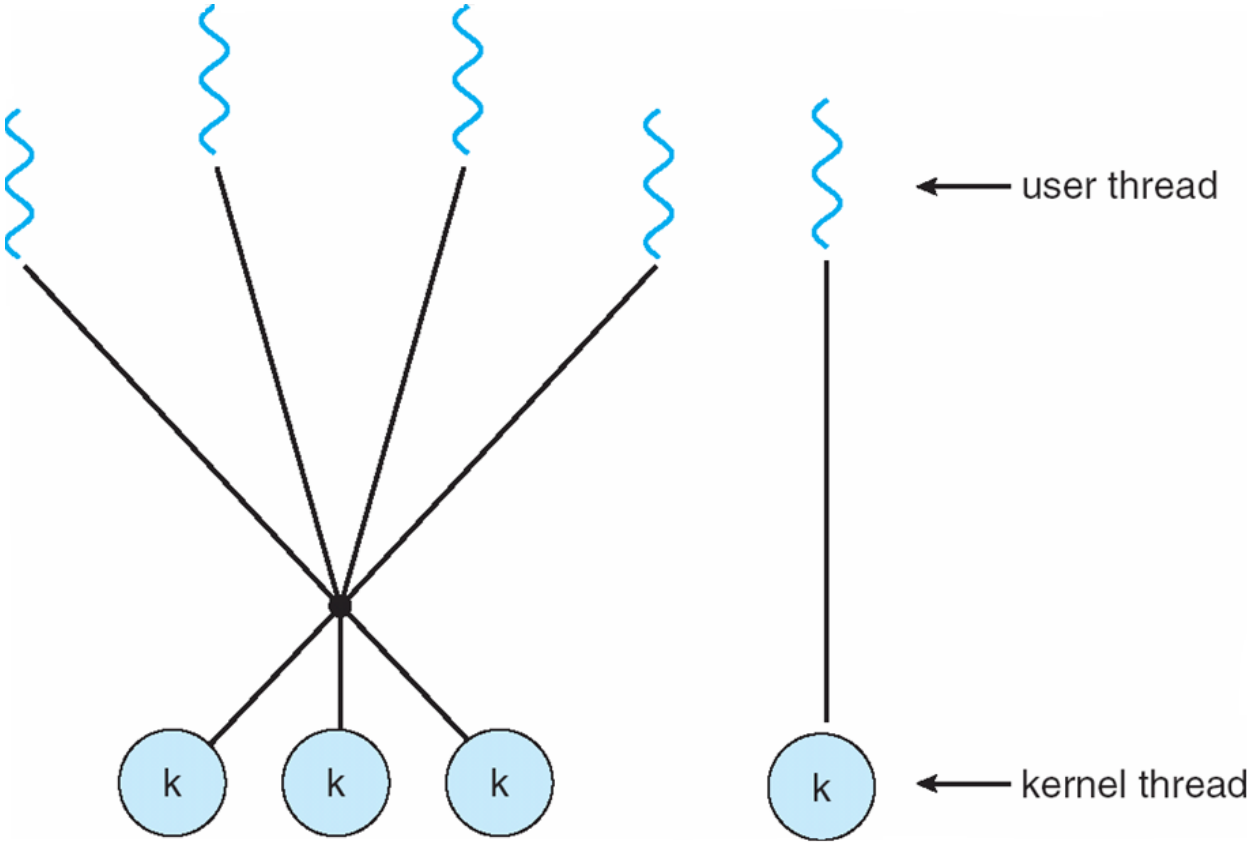
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- The ANSI/IEEE Portable Operating System Interface (POSIX) standard defines the pthreads application program interface for use by C language programs.
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)