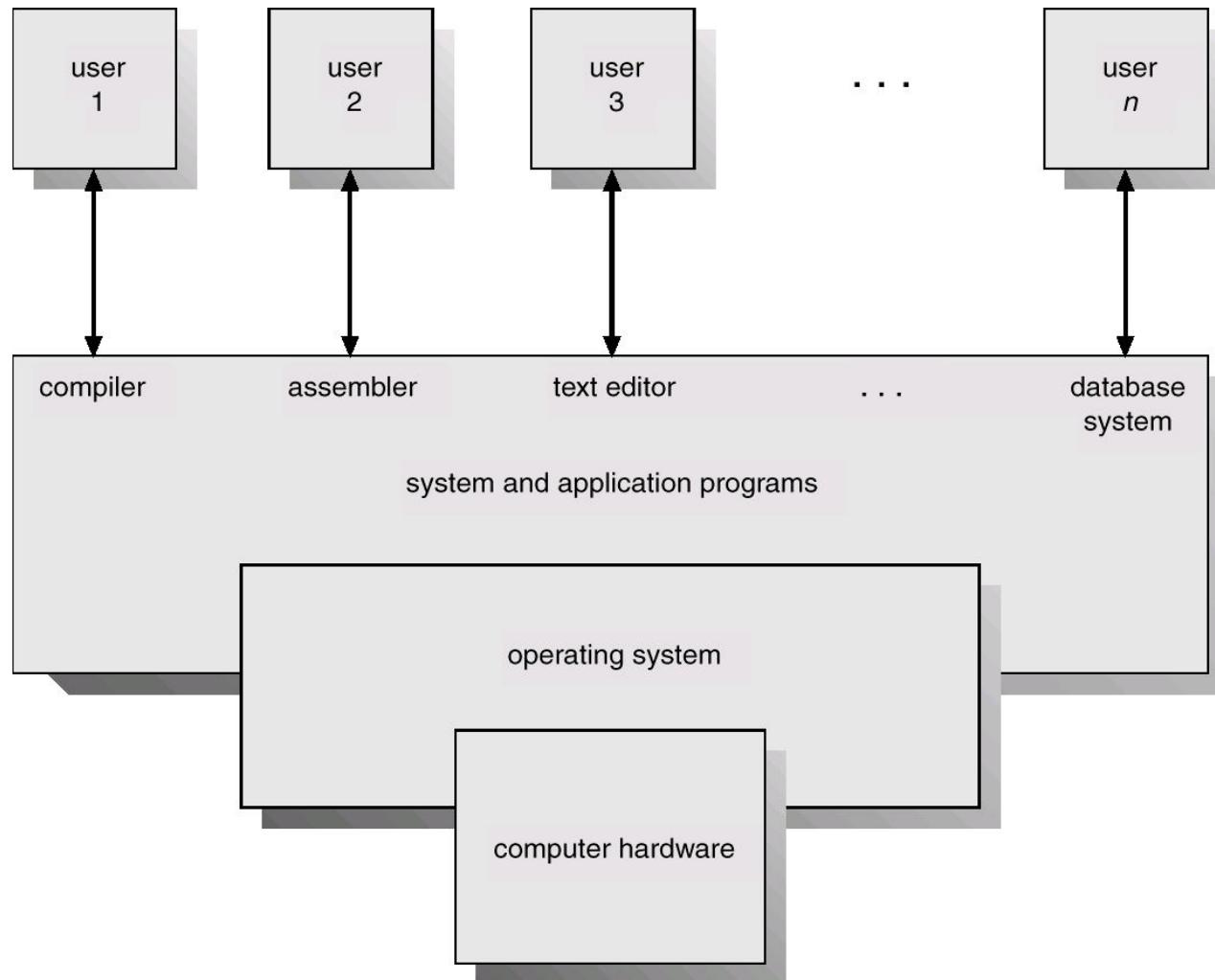


Operating Systems

Introduction to
Operating System (OS)

Components of a Computer System



Components of a Computer System

Contd...

- Hardware – provides basic computing resources like CPU, memory, I/O devices.
- Operating system – OS controls and coordinates the use of the hardware among the various application programs for different users.
- Applications programs – They define the ways in which the system resources are used to solve the computing problems of the users.
- Users (people, machines, other computers).

What is OS?

- Operating System is a software, which makes a computer to actually work.
- It is the software that enables all the programs we use.
- The OS organizes and controls the hardware.
- OS acts as an interface between the application programs and the machine hardware.
- Examples: Windows, Linux, Unix and Mac OS, etc.,

USER View of OS

- Ease to use
- Efficient resource utilization

SYSTEM view of OS

- OS is a **resource allocator**

Manages all resources (CPU time, memory space, storage space, I/O and so on)

Decides between conflicting requests for efficient and fair resource use

- OS is a **control program**

Controls execution of programs to prevent errors and improper use of the computer

Evolution of OS

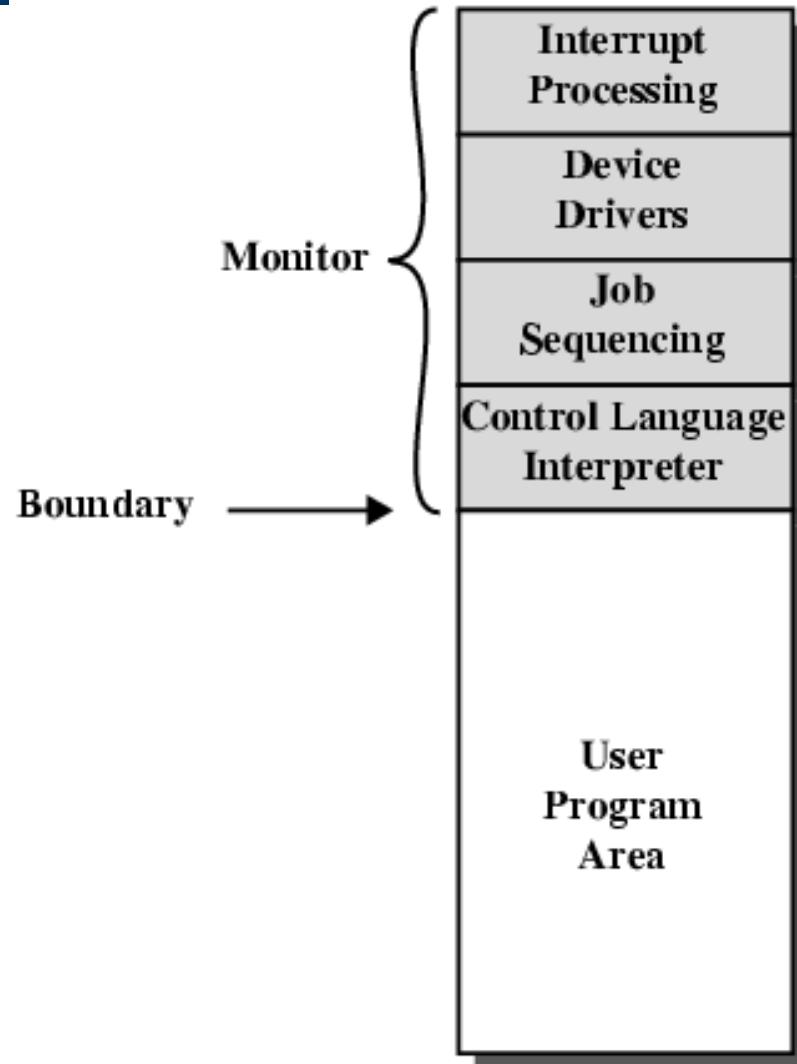
- Batch Processing
- Multiprogramming
- Timesharing

Batch Processing

- Early computers were extremely expensive
 - Important to maximize processor utilization
- Monitor
 - Software that controls the sequence of events
 - Batch jobs together
 - Program returns control to monitor when finished

Batch Processing (Contd...)

- The interface to the monitor was accomplished through Job Control Language (JCL).
- For example, a JCL request could be to run the compiler for a particular programming language, then to link and load the program, then to run the user program.



Batch Processing (Contd...)

Hardware features:

- Memory protection: protect the memory area containing the monitor from the Utilities
- Timer: prevents a job from monopolizing the system

Problems:

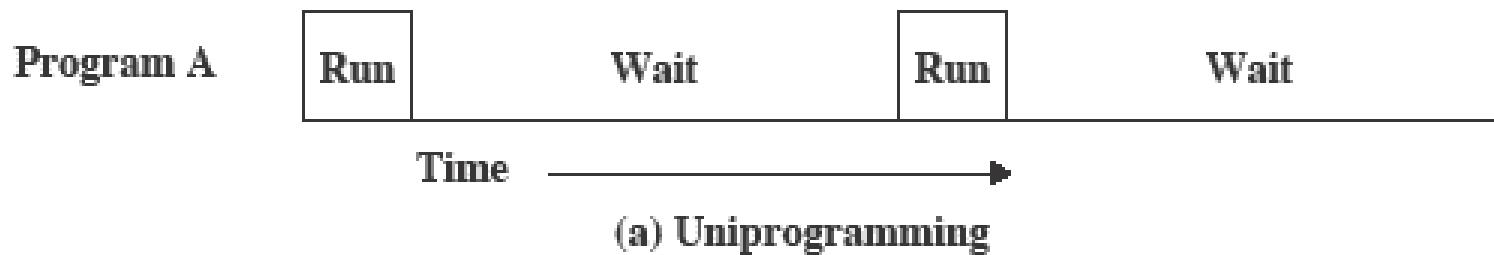
- Bad utilization of CPU time - the processor stays idle while I/O devices are in use.

Multiprogramming

- Multiprogramming is a technique to execute number of programs simultaneously by a single processor.
- Number of processes reside in main memory at a time.
- The OS picks and begins to execute one of the jobs in the main memory.
- If any I/O wait happened in a process, then CPU switches from that job to another job.
- Hence CPU is not idle at any time.

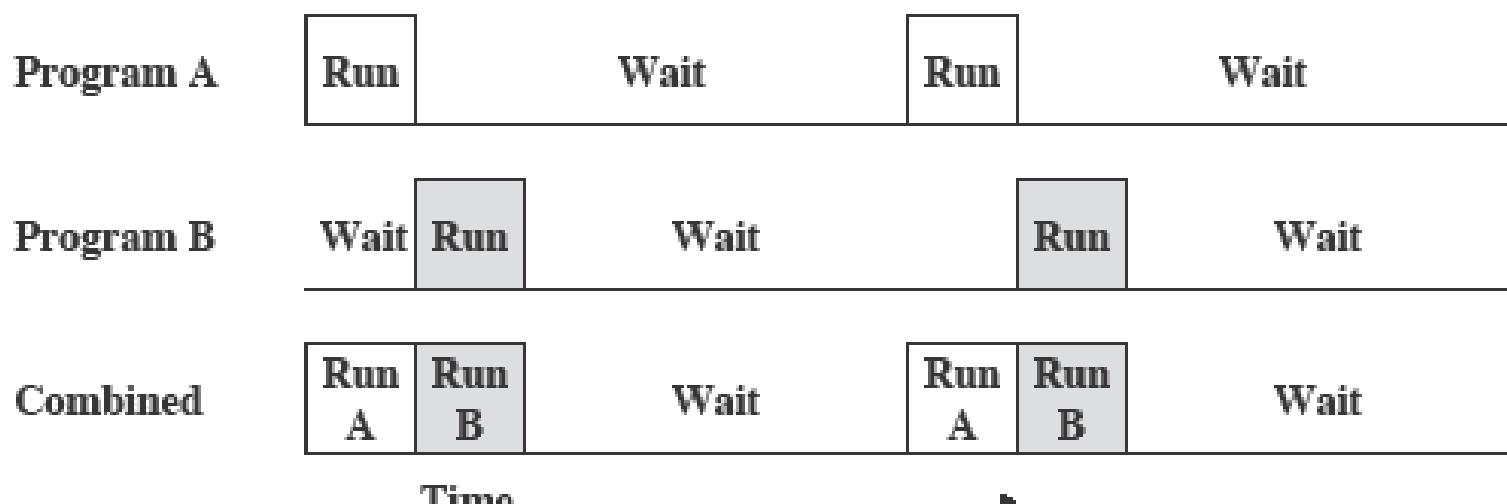
Multiprogramming (Contd...)

- Processor must wait for I/O instruction to complete before preceding



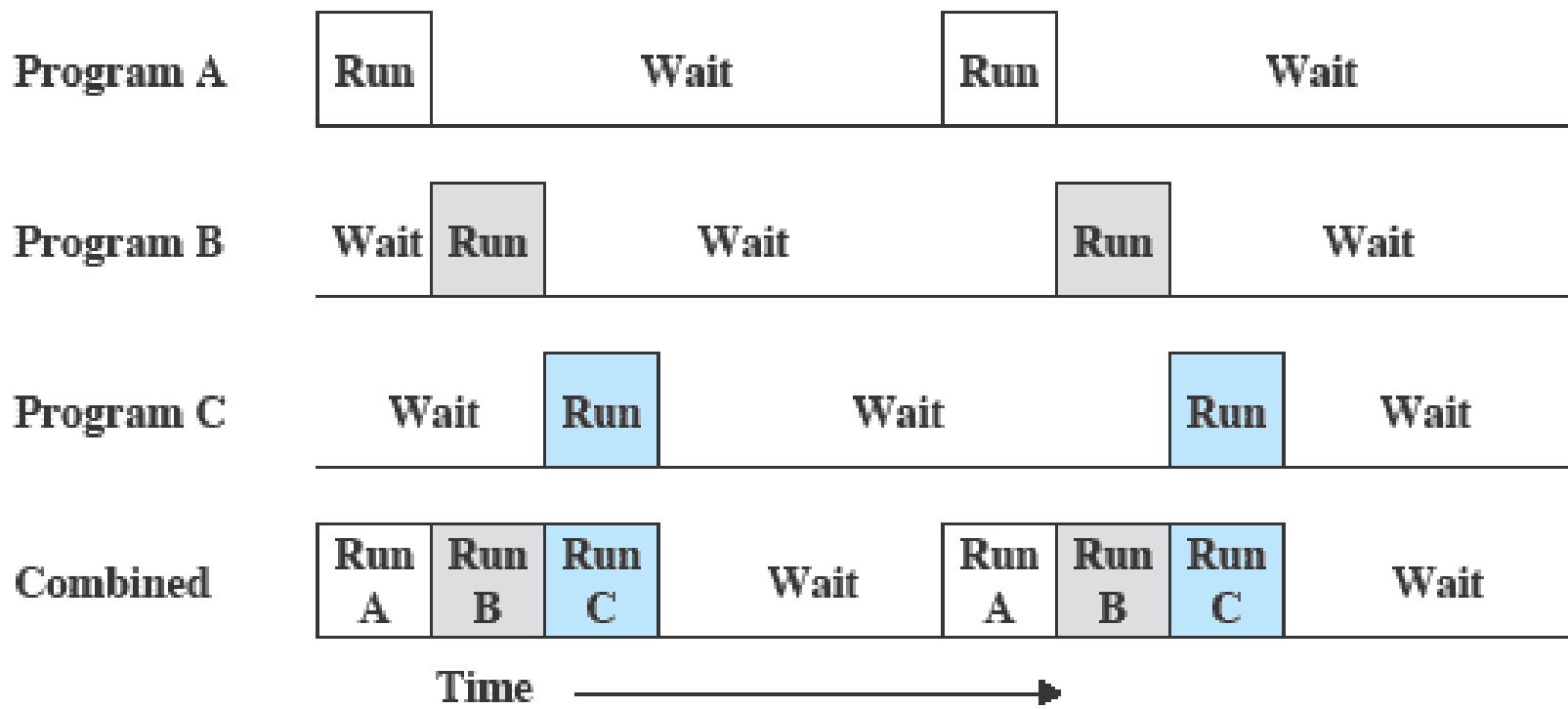
Multiprogramming (Contd...)

- When one job needs to wait for I/O, the processor can switch to the other job



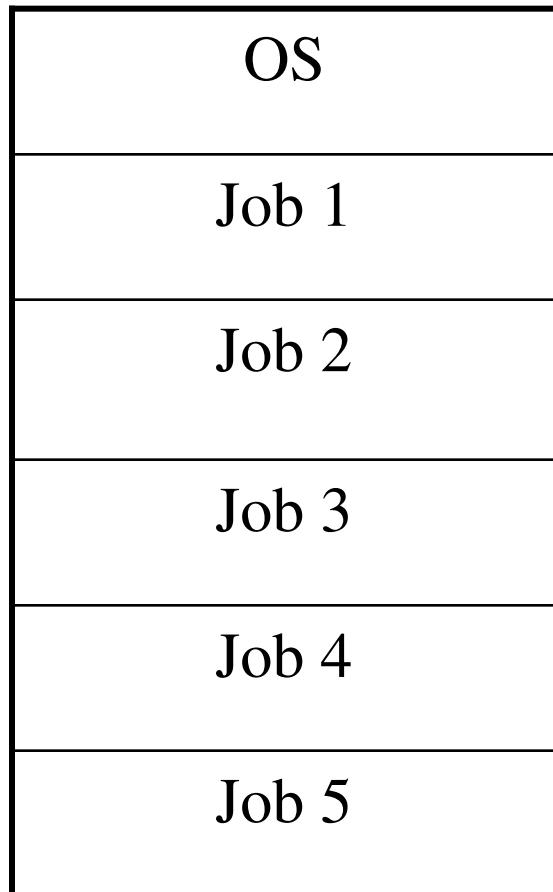
(b) Multiprogramming with two programs

Multiprogramming (Contd...)



(c) Multiprogramming with three programs

Multiprogramming (Contd...)



Advantages

Efficient memory utilization

Throughput increases

CPU is never idle, so performance increases.

Time Sharing Systems

- Multiprogramming systems : several programs use the computer system
- Time-sharing systems : several (human) users use the computer system interactively
- Characteristics:
 - Using multiprogramming to handle multiple interactive jobs
 - Processor's time is shared among multiple users
 - Multiple users simultaneously access the system through terminals
-

Time Sharing Systems (Contd...)

- Time sharing, or multitasking, is a logical extension of multiprogramming
- Multiple jobs are executed by switching the CPU between them
- In this, the CPU time is shared by different processes, so it is called as “Time sharing Systems”.
- Time slice is defined by the OS, for sharing CPU time between processes.
- Examples: Multics, Unix, etc.,

Time Sharing Systems (Contd...)

	Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Types of OS

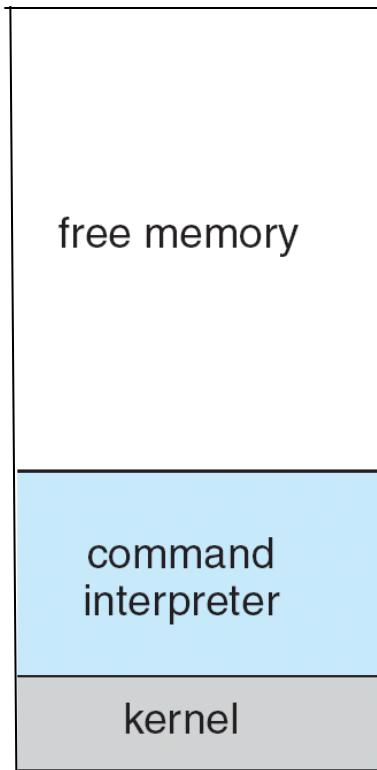
Operating System can also be classified as

- Single User Systems
- Multi User Systems

Single User Systems

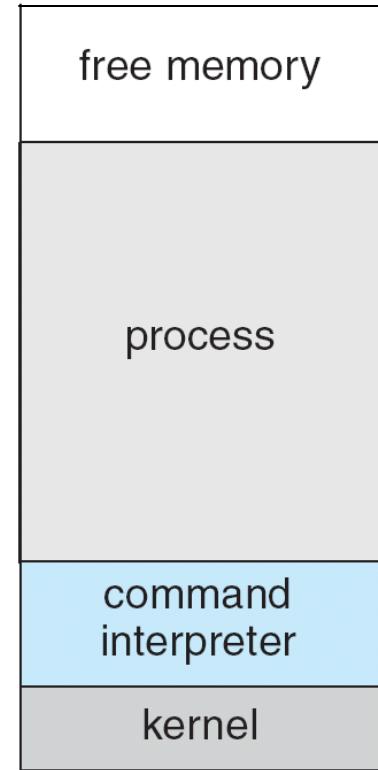
- Provides a platform for only one user at a time.
- They are popularly associated with Desk Top operating system which run on standalone systems where no user accounts are required.
- Example: DOS

MS-DOS execution



(a)

(a) At system startup



(b)

(b) running a program

Multi-User Systems

- Provides regulated access for a number of users by maintaining a database of known users.
- Refers to computer systems that support two or more simultaneous users.
- Another term for *multi-user* is *time sharing*.
- Ex: All mainframes and are multi-user systems.
- Example: Unix

Operating System Services

The following are the services provided by the operating system.

- User Interface
- Program Execution
- I/O Operations
- File-system manipulation
- Communications

Operating System Services (Contd...)

- Error Detection
- Resource Allocation
- Accounting
- Protection

User Interface

- User interface - Almost all operating systems have a user interface (UI)
- A **user interface** controls how to enter data and instructions and how information is displayed on the screen
 - With a graphical user interface (GUI), user interacts with menus and visual images
 - With a **command-line interface**, a user uses the keyboard to enter data and instructions



```
bash-2.05b$ ping -q -c1 en.wikipedia.org
PING rr.chtpa.wikimedia.org (207.142.131.247) 56(84) bytes of data.

--- rr.chtpa.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 112.076/112.076/112.076/0.000 ms
bash-2.05b$ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1          /mnt/usbkey
/dev/sda2          /mnt/ipod
bash-2.05b$ date
Wed May 25 11:36:56 PDT
bash-2.05b$ lsmod
Module           Size  Used by
joydev           8256   0
ipw2200          175112   0
ieee80211         44228   1 ipw2200
ieee80211_crypt    4872   2 ipw2200,ieee80211
e1000            84468   0
bash-2.05b$
```

command entered by user

Program Execution

- Operating system must be able to load a program into memory and to run that program.
- The program must be able to end its execution, either normally or abnormally.

I/O Operations

- A running program may require I/O, which may involve a file or an I/O device

File-System Manipulation

- Programs need to read and write files and directories
- The operating system handles the process of creating, deleting and searching files
- Also list file Information, permission management

Communication

- Every process needs to exchange information with another process
- Communications may be implemented via **shared memory** or by the technique of **message passing**
- Communications occurs in two ways
 - Within processes executing on the **same computer**
 - Between processes executing on **different computers**, tied together by a network

Error Detection

- Errors can happen at any time in the system
- Errors can occur in the CPU, memory, I/O Devices, and in the 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

Resource Allocation

- Multiple jobs run at the same time so resources must be allocated to each one of them
- Many different types of resources are managed by the operating system
- Some (such as CPU cycles, main memory, and file storage) have special allocation code, whereas others (such as I/O devices) have general request and release code
- The operating systems have various routines for different jobs in the system, for example CPU scheduling

Accounting

- Need to keep track of number and type of resources allocated to processes/users
- This tracking of user information can be used to find out the usage statistics.
- These usage statistics are useful for the researchers who wish to re-configure the system to improve computing services.

Protection

- Protection and Security is an important issue in multi-user computer systems.
- **Protection** involves in ensuring that all access to system resources is controlled.
- **Security** of the system requires user authentication, extends to defend external I/O devices from invalid access attempts
- Access to the resources must be authenticated with a password.

Operating Systems

Operating System
Structure

System Components

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

Process Management

- A *process* is a program in execution.
- A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.
- The operating system is responsible for the following activities in connection with process management.
 - Process creation and deletion.
 - process suspension and resumption.
 - Provision of mechanisms for:
 - process synchronization
 - process communication

Memory Management

- Memory is a large array of words or bytes, each with its own address.
- It is a repository of quickly accessible data shared by the CPU and I/O devices.
- Main memory is a volatile storage device. It loses its contents in the case of system failure.
- The operating system is responsible for the following activities in connections with memory management:
 - Keep track of which parts of memory are currently being used and by whom.
 - Decide which processes to load when memory space becomes available.
 - Allocate and deallocate memory space as needed.

I/O Management

- Much of the OS kernel is concerned with I/O.
- The OS provides a standard interface between programs (user or system) and devices.
- **Device drivers** are the processes responsible for each device type.
 - A driver encapsulates device-specific knowledge, e.g., for device initiation and control, interrupt handling, and errors.
- There may be a process for each device, or even for each I/O request, depending on the particular OS.

Secondary Storage Management

- Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory
- Most modern computer systems use disks as the principal on-line storage medium, for both programs and data.
- The operating system is responsible for the following activities in connection with disk management:
 - Free space management
 - Storage allocation
 - Disk scheduling

File Management

- A **file** is a collection of related information defined by its creator.
- Commonly, files represent programs (both source and object forms) and data.
- The operating system is responsible for the following activities in connections with file management:
 - File creation and deletion.
 - Directory creation and deletion.
 - Support of primitives for manipulating files and directories.
 - Mapping files onto secondary storage.
 - File backup on stable (nonvolatile) storage media.

Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
 - distinguish between authorized and unauthorized usage.
 - specify the controls to be imposed.
 - provide a means of enforcement.
- protection mechanisms help to detect errors as well as to prevent malicious destruction

Command Interpreter

- Many commands are given to the operating system by control statements which deal with:
 - process creation and management
 - I/O handling
 - secondary-storage management
 - main-memory management
 - file-system access
 - protection
 - networking

What is a kernel?

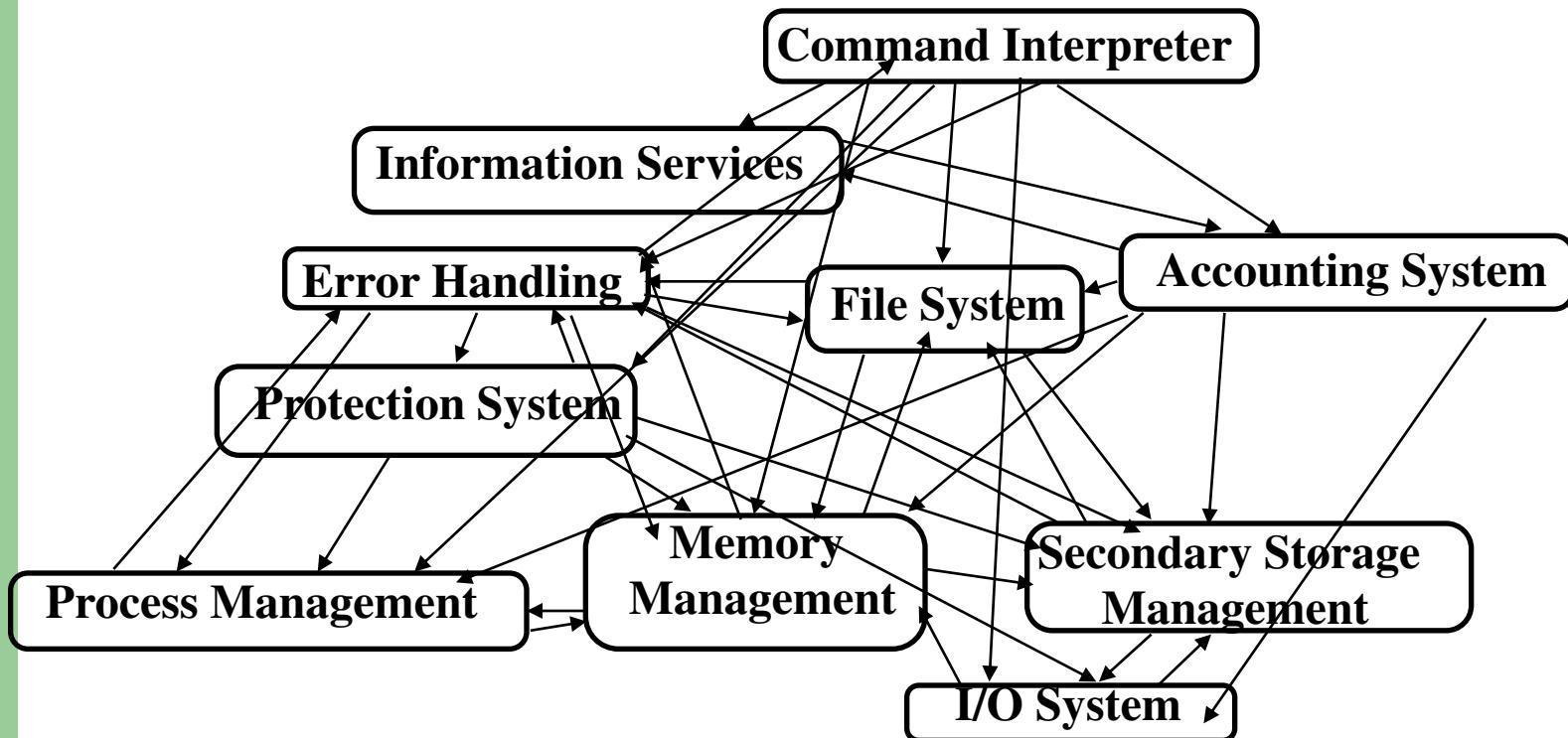
- Operating system provides an hardware abstraction layer
- Since OS provides resource sharing → every access to the hardware from user programs should be done through the OS
- To enforce this in a secure way, and prevent malicious or buggy applications, a protection is needed at the hardware level

What is a kernel? (Contd...)

- Hardware must provide at least two execution levels:
 - **Kernel mode:** the application has access to all the instructions and every piece of hardware.
 - **User mode:** the application is restricted and cannot execute some instructions, and is denied access to some hardware (like some area of the main memory)
- Two areas are defined at application level:
 - **Kernel space:** Code running in the kernel mode is said to be inside the kernel space
 - **User space:** Every other programs, running in user mode, is said to be in user space

OS Structure

The OS (*a simplified view*)



Hardware

Operating Systems Structures

Structure/Organization/Layout of OSs

- Monolithic (one unstructured program)
- Layered
- Microkernel
- Virtual Machines

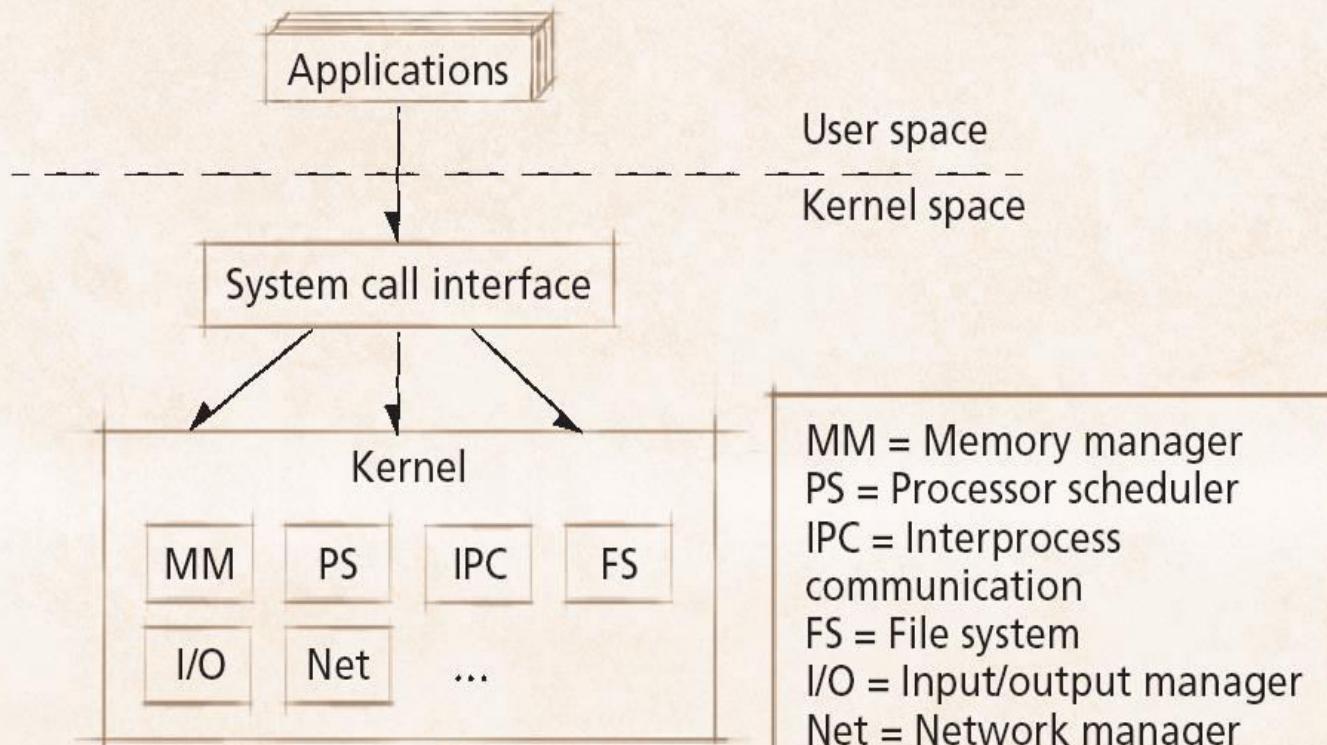
Monolithic OS

- OS is simply a collection of functions
- One big program including everything (system calls, system programs, every managers, device drivers, etc)
- Entire OS resides in main memory
- Ex: MS-DOS, Multics, Unix, BSD, Linux

Monolithic OS (Contd...)

- Most work is done via system calls
- These are interfaces, that access some subsystem within the kernel such as disk operations
- Essentially calls are made within programs and a request is passed through the system call

Monolithic OS (Contd...)



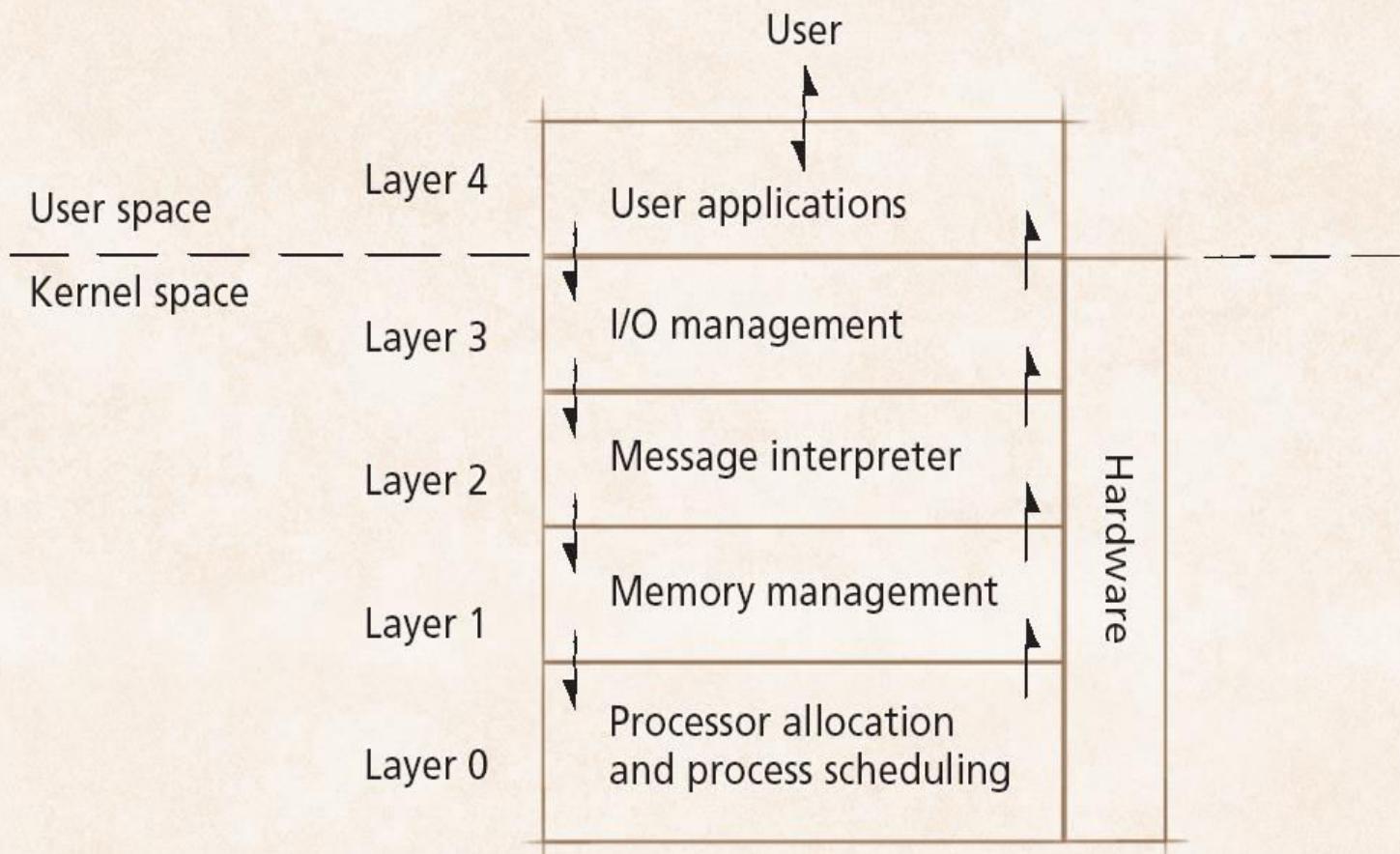
Monolithic OS (Contd...)

- Pros
 - Every component contained in kernel
 - direct communication among all components
 - All OS services operate in kernel space
 - high performance
- Cons
 - Larger size makes it hard to maintain
 - Complexity is more
 - Dependencies between system component
 - not easily extended/modified: new devices, emerging technologies not easily added

Layered Approach

- The operating system is divided into a number of layers (levels), providing different functionalities
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- A layer can only use the services provided by layer in a level below
- System calls might pass through many layers before completion

Layered Approach (Contd...)



Layered Approach (Contd...)

- Pros:
 - Information hiding between layers
 - Increased security and protection
 - Easy to debug, test, and modify OS
- Cons:
 - If one layer stops working, entire system will stop
 - Mapping overhead between layers
 - Difficult to categorize into layers

Microkernel OS

- Move as much functionality as possible from the *kernel* into *user* space.
- Only a few essential functions in the kernel
 - primitive memory management (**address space**)
 - I/O and interrupt management
 - Inter-Process Communication (**IPC**)
 - basic scheduling
- Other OS services are provided by processes running in user mode (**vertical servers**):
 - device drivers, file system, virtual memory...

Microkernel OS (Contd...)

- Communication takes place between user modules using message passing.
- More flexibility, extensibility, portability and reliability.
- But performance overhead caused by replacing service calls with message exchanges between processes.

Benefits of a Microkernel

- Extensibility/Reliability
 - easier to extend a microkernel
 - easier to port the operating system to new architectures
 - more reliable (less code is running in kernel mode)
 - more secure
 - small microkernel can be rigorously tested.
- Portability
 - changes needed to port the system to a new processor is done in the microkernel, not in the other services.

Virtual Machines

- A Virtual Machine (VM) takes the layered and microkernel approach to its logical conclusion.
- It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface identical to the underlying bare hardware.
- The operating system host creates the illusion that a process has its own processor and (virtual memory).
- Each guest provided with a (virtual) copy of underlying computer.

Virtual Machines (Contd...)

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

Operating Systems

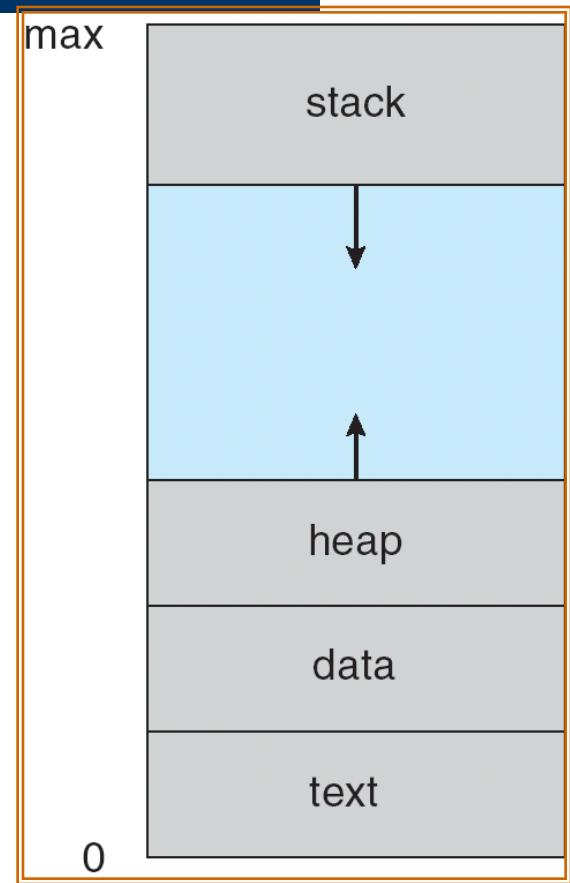
Processes and Threads

Process

- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
 - program counter
 - stack
 - data section

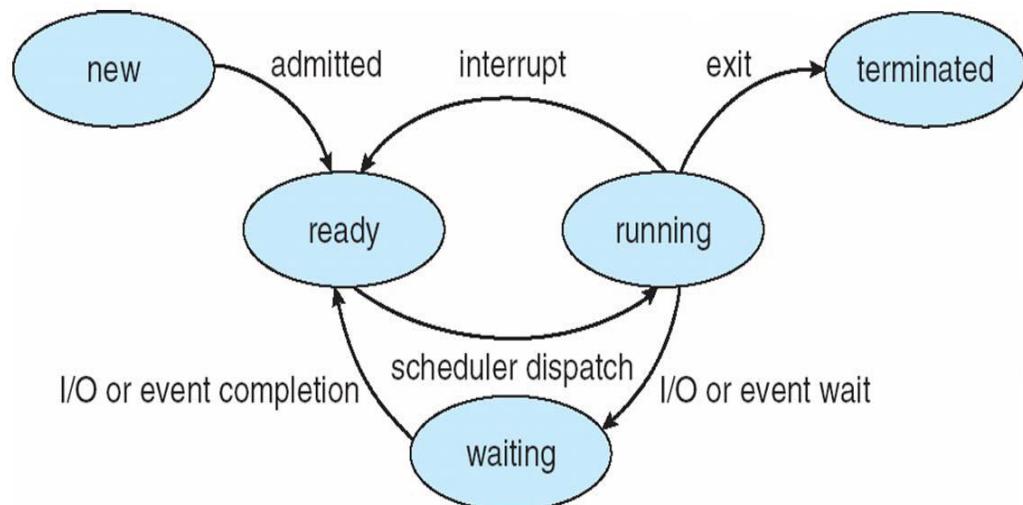
Process in Memory

- A process has its own address space consisting of
 - **Text region**
 - Stores the code that the processor executes
 - **Data region**
 - Contains global variables
 - **Stack region**
 - Stores local variables for active procedure calls, function parameters, return addresses
 - **Heap**
 - Contains memory dynamically allocated during run time



Process State

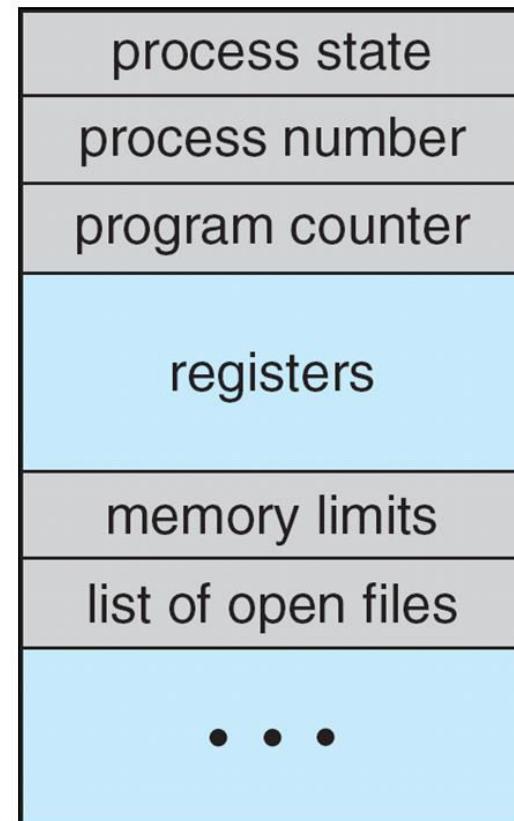
- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished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



Process Control Block (PCB)

Information associated with each process

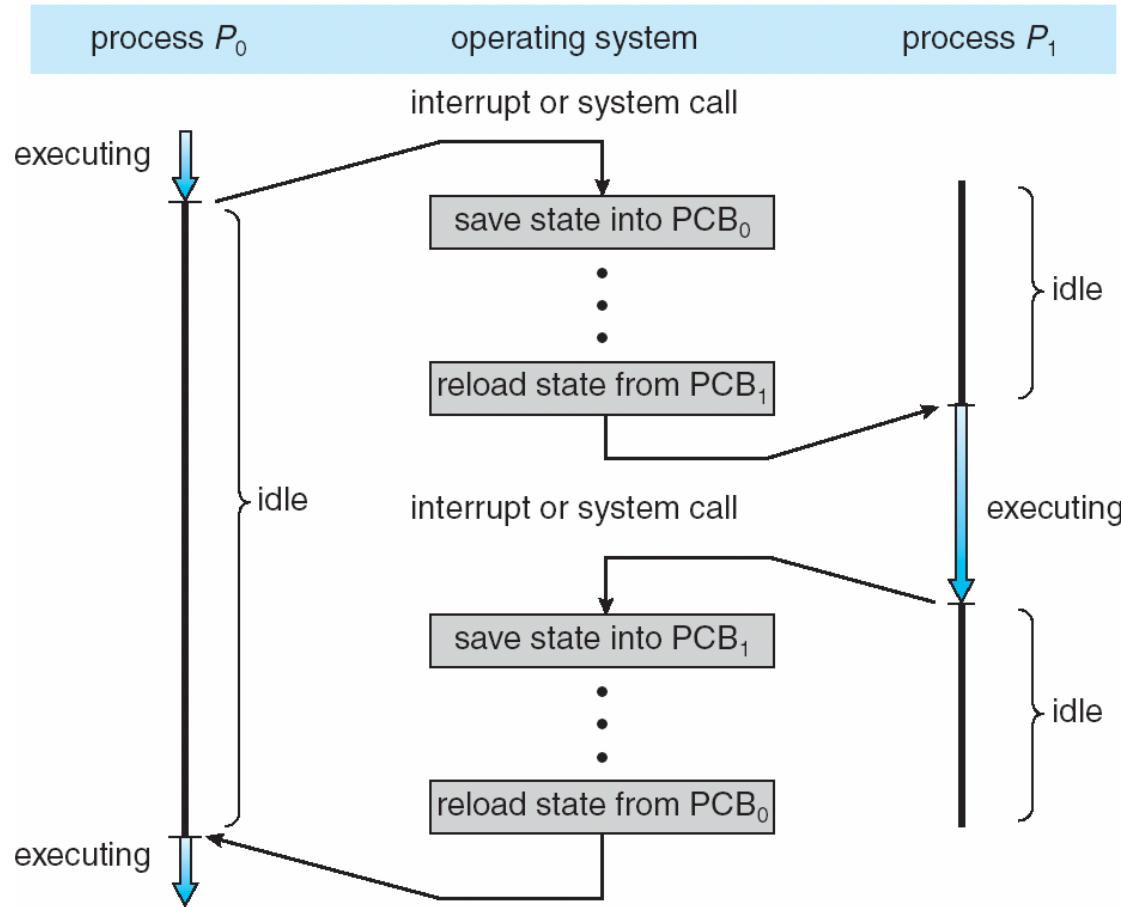
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



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 → longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

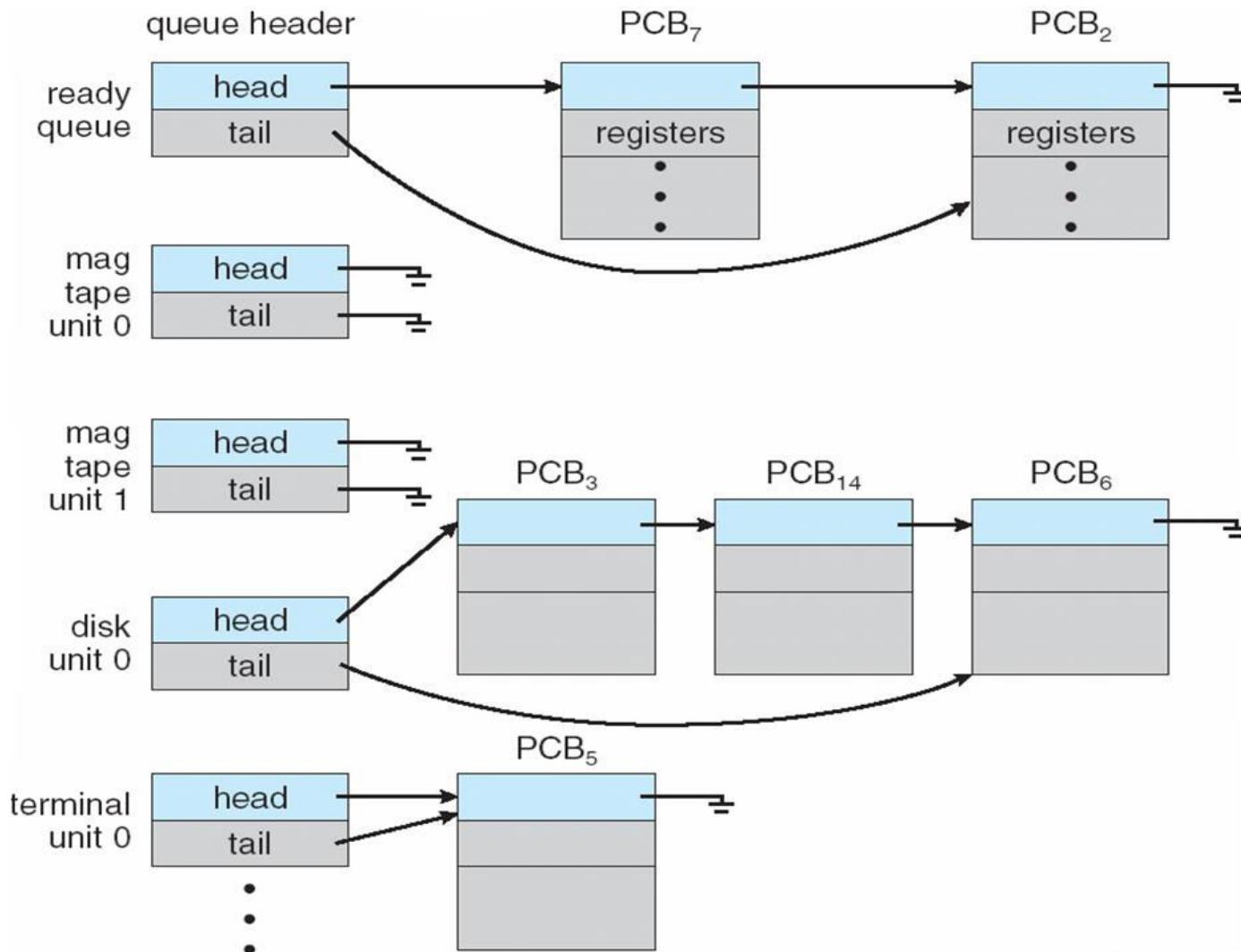
CPU Switch From Process to Process



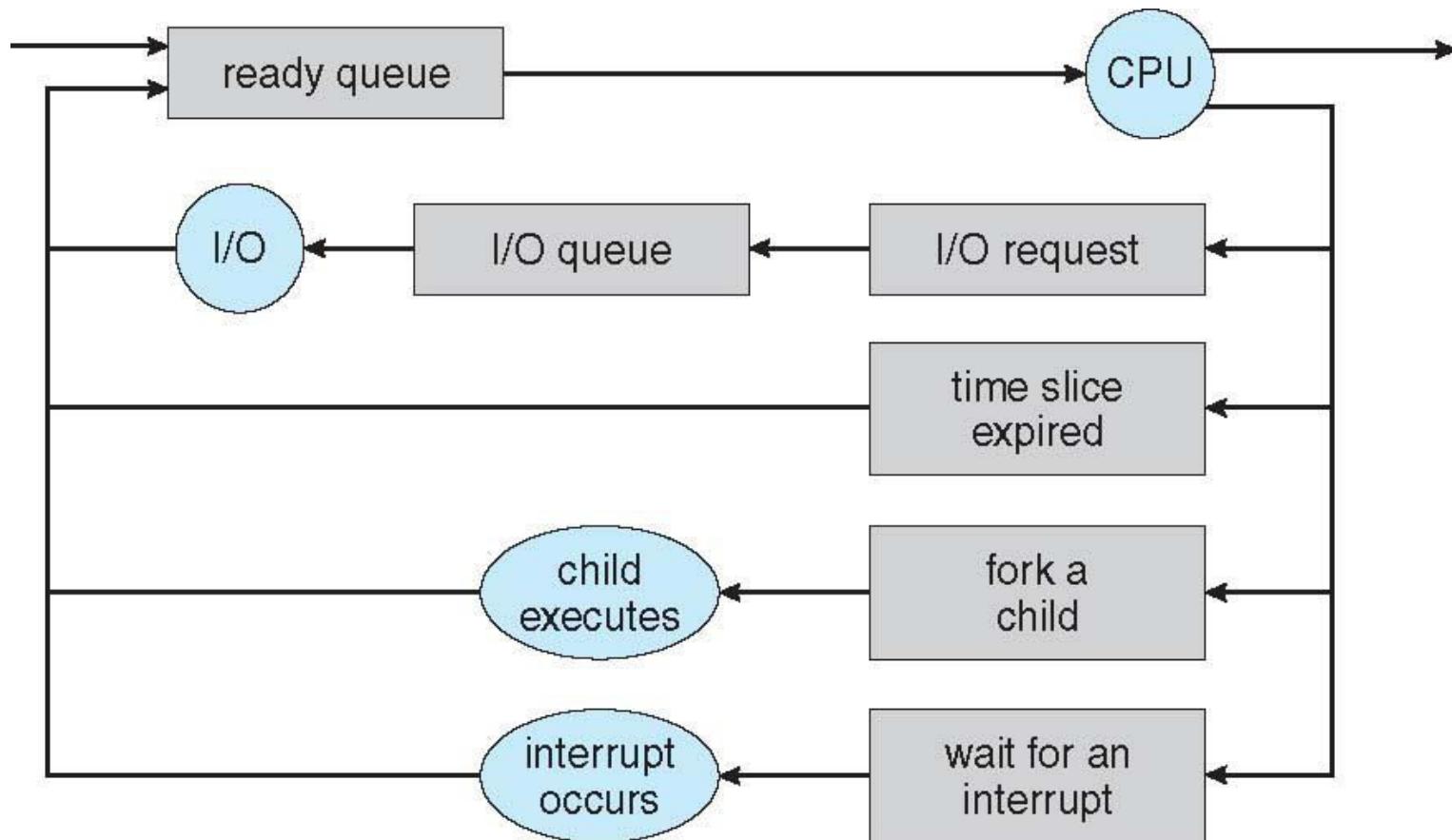
Process Scheduling

- 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



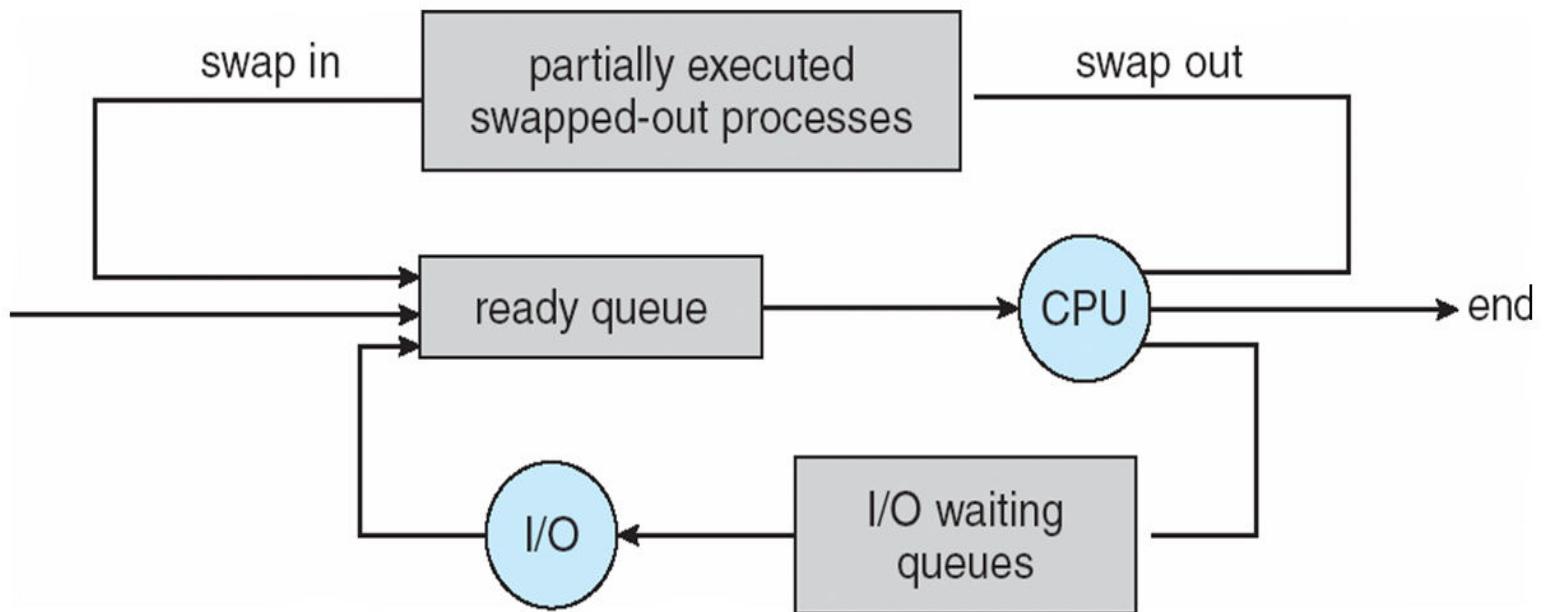
Schedulers

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

Schedulers (Cont.)

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

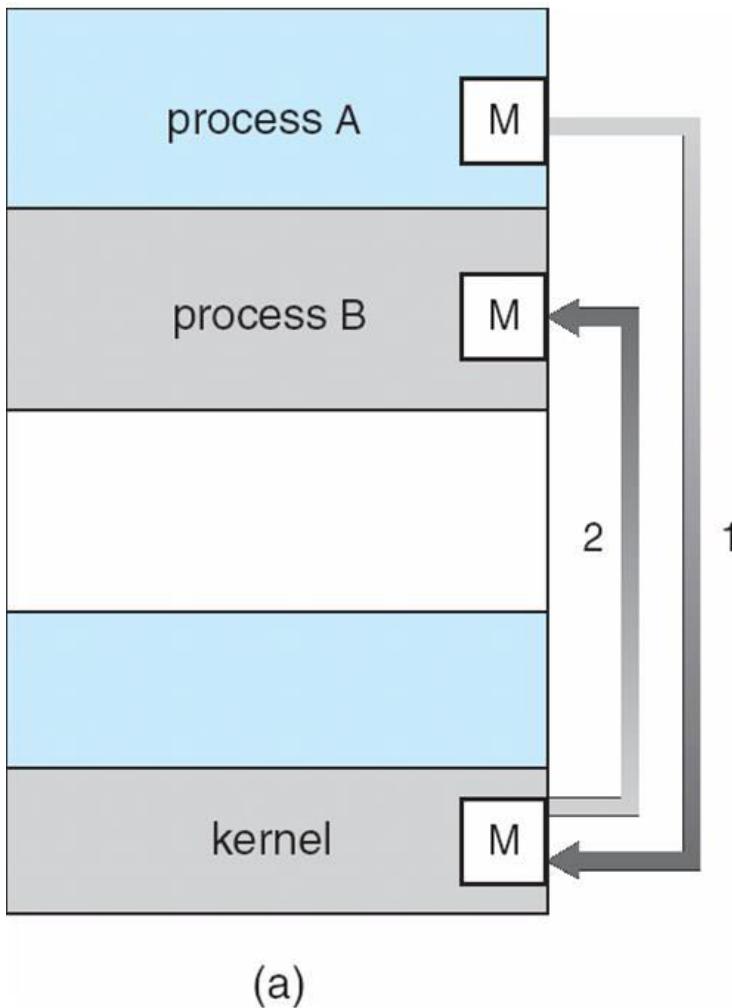
Addition of Medium Term Scheduling



Interprocess Communication

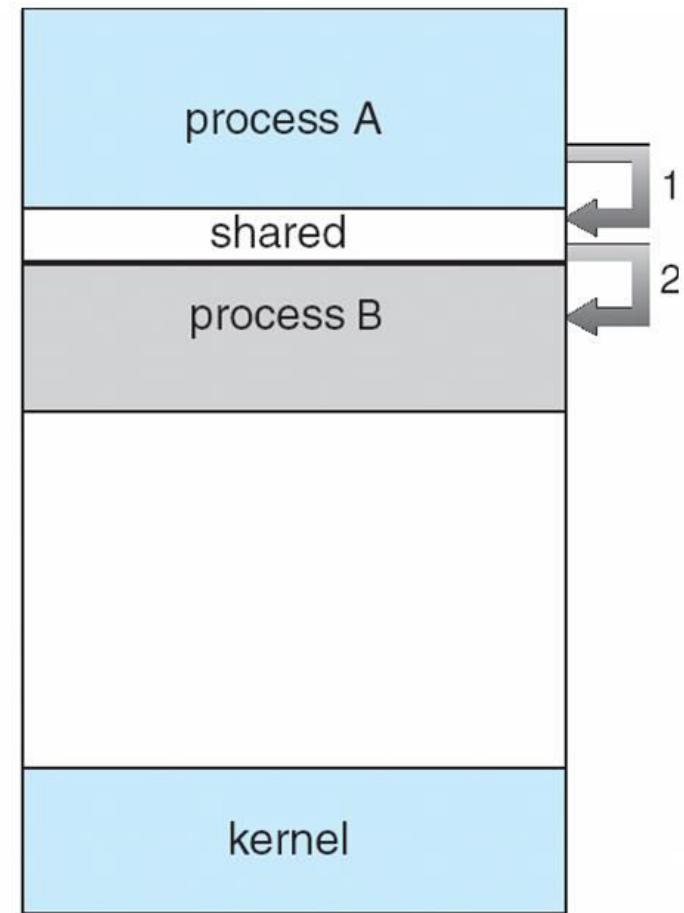
- Processes within a system may be **independent** or **cooperating**
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes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

Communications Models



15

(a)



(b)

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 – Producer

```
while (true) {  
    /* Produce an item */  
    while ((in = (in + 1) % BUFFER SIZE ) == out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```

Bounded Buffer – Consumer

```
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;  
}
```

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send(*message*)** – message size fixed or variable
 - **receive(*message*)**
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

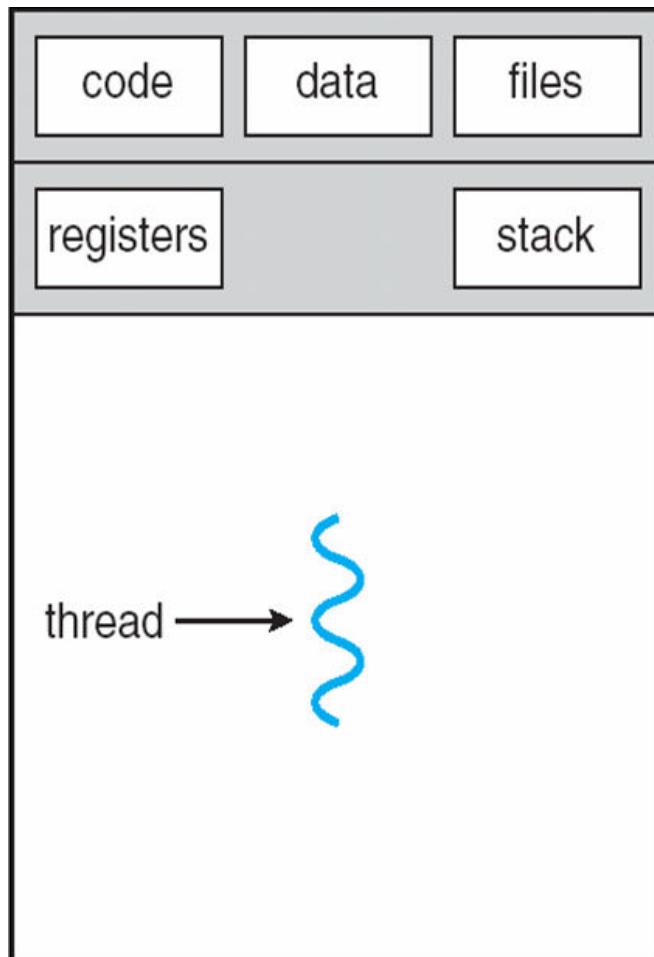
Threads

- A thread is a flow of execution through the process code, with its own
 - program counter,
 - system registers and
 - stack.
- A thread is also called a **light weight process**.
- Threads provide a way to improve application performance through parallelism
- Threads represent a software approach to improving performance of operating system by reducing the overhead
thread is equivalent to a classical process.

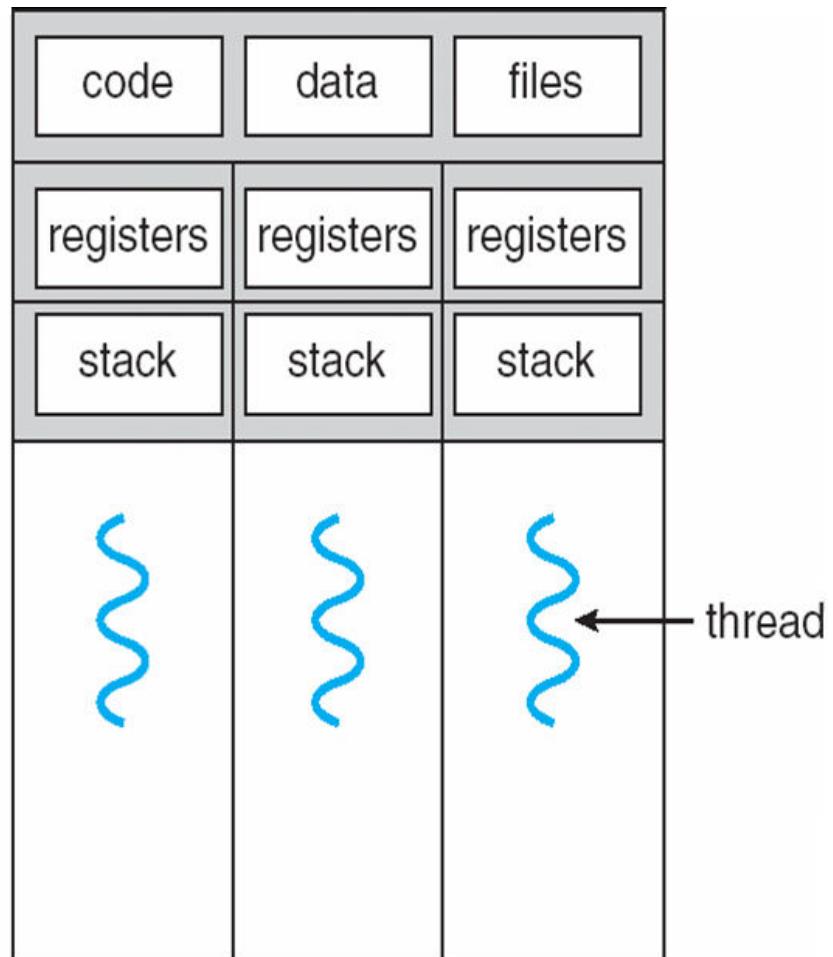
Threads (Contd...)

- Each thread belongs to exactly one process
- No thread can exist outside a process
- Each thread represents a separate flow of control
- Threads have been successfully used in implementing network servers and web server
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Single and Multithreaded Process



single-threaded process



multithreaded process

Difference between Process and Thread

Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.

Difference between Process and Thread

If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Thread minimize context switching time
- Use of threads provides concurrency within a process
- Efficient communication
- Economy- It is more economical to create and context switch threads
- Utilization of multiprocessor architectures to a greater scale and efficiency

Types of Thread

- Threads are implemented in following two ways
- **User Level Threads** -- User managed threads
- **Kernel Level Threads** -- Operating System managed threads acting on kernel, an operating system core.

User Level Threads

- In this case, application manages thread management kernel is not aware of the existence of threads.
- The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- The application begins with a single thread and begins running in that thread.

User Level Threads

- Advantages
 - Thread switching does not require Kernel mode privileges.
 - User level thread can run on any operating system.
 - Scheduling can be application specific in the user level thread.
 - User level threads are fast to create and manage.
- Disadvantages
 - In a typical operating system, most system calls are blocking.
 - Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

- In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Kernel Level Threads

- Advantages
 - Kernel can simultaneously schedule multiple threads from the same process or multiple processes.
 - If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
 - Kernel routines themselves can be multithreaded.
- Disadvantages
 - Kernel threads are generally slower to create and manage than user threads.
 - Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

- Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types
 - Many to many relationship.
 - Many to one relationship.
 - One to one relationship.

Multithreading Models

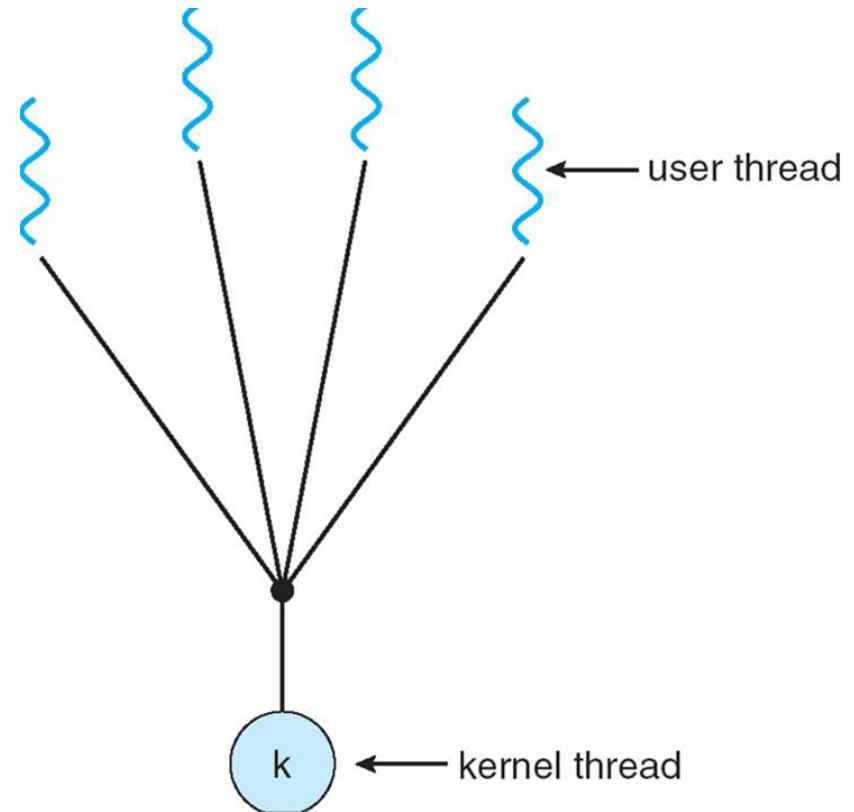
- Many-to-One: map **many user-level** threads to **one kernel** thread
- One-to-One: map each user-level thread to a kernel thread
- Many-to-Many: multiplexes many user-level threads to a smaller or equal number of kernel threads

Many-to-One Model

- Many to one model maps many user level threads to one Kernel level thread
- Thread management is done in user space
- When thread makes a blocking system call, the entire process will be blocked
- Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.
- If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.

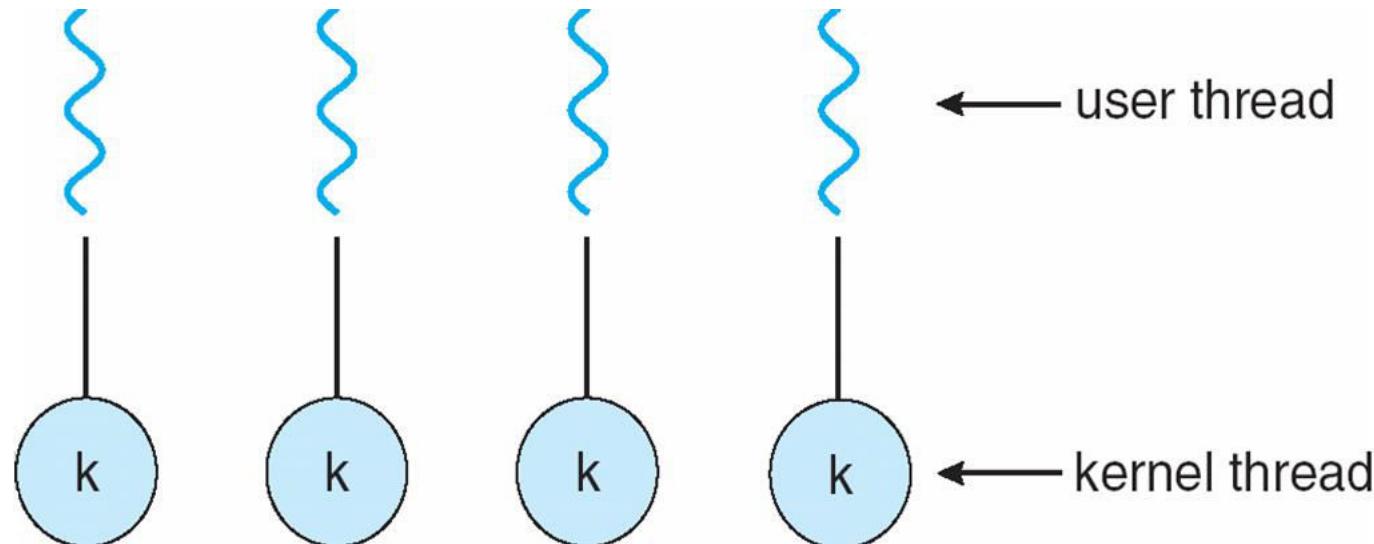
Many-to-One Model

- Many user-level threads mapped to single kernel thread
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads



One-to-One

- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

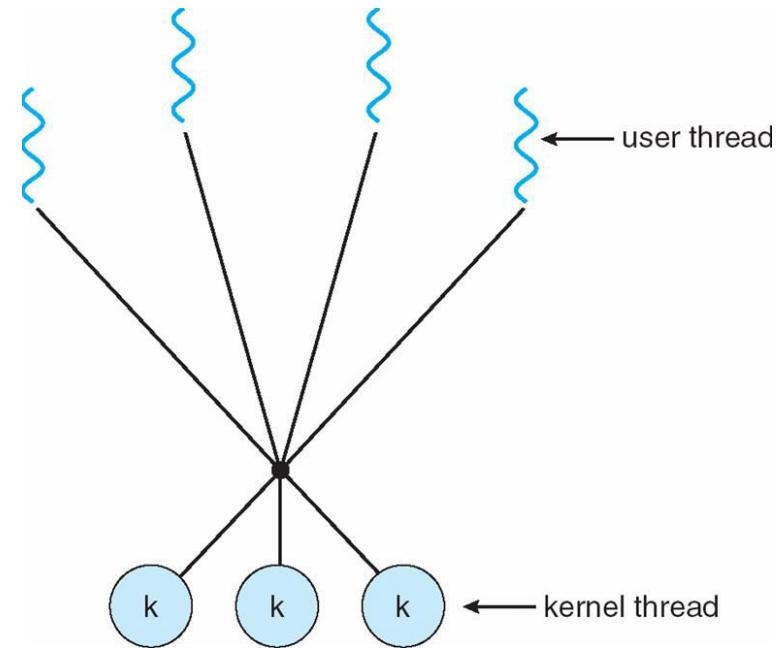


One-to-One

- There is one to one relationship of user level thread to the kernel level thread
- This model provides more concurrency than the many to one model
- It also another thread to run when a thread makes a blocking system call
- It support multiple thread to execute in parallel on microprocessors
- Disadvantage of this model is that creating user thread requires the corresponding Kernel thread
- OS/2, windows NT and windows 2000 use one to one relationship model.

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

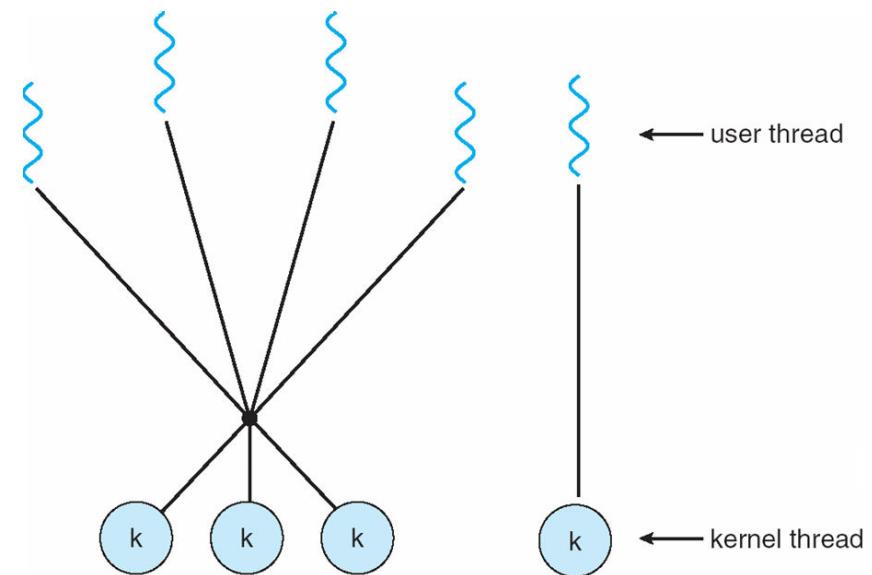


Many-to-Many Model

- In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers
- The number of Kernel threads may be specific to either a particular application or a particular machine.
- In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor.

Two-level Model

- Similar to 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



Difference between User Level & Kernel Level Thread

User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User level thread is generic and can run on any operating system.	Kernel level thread is specific to the operating system.
Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

User-Level Threads

- Thread management done by user-level threads library
- Examples
 - POSIX *Pthreads*
 - Mach *C-threads*
 - Solaris *threads*
 - Java threads

User-Level Threads

- Thread library entirely executed in user mode
- Cheap to manage threads
 - Create: setup a stack
 - Destroy: free up memory
- Context switch requires few instructions
 - Just save CPU registers
 - Done based on program logic
- A blocking system call blocks all peer threads

Kernel-Level Threads

- Kernel is aware of and schedules threads
- A blocking system call, will not block all peer threads
- Expensive to manage threads
- Expensive context switch
- Kernel Intervention

Kernel Threads

- Supported by the Kernel
- Examples: newer versions of
 - Windows
 - UNIX
 - Linux

Linux Threads

- Linux refers to them as tasks rather than *threads*.
- Thread creation is done through `clone()` system call.
- Unlike `fork()`, `clone()` allows a child task to share the address space of the parent task (process)

Pthreads

- 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.
- POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
- Common in UNIX operating systems.

LWP Advantages

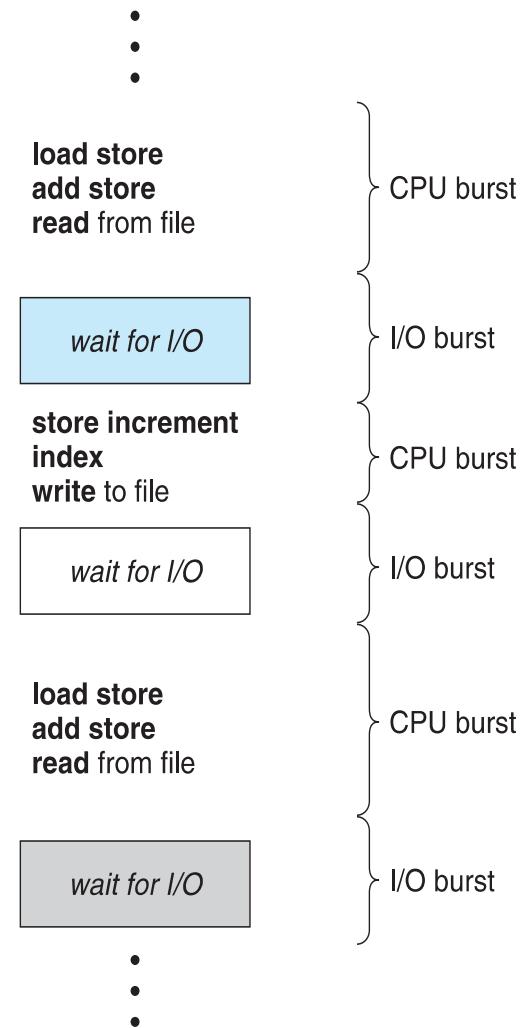
- Cheap user-level thread management
- A blocking system call will not suspend the whole process
- LWPs are transparent to the application
- LWPs can be easily mapped to different CPUs

Operating Systems

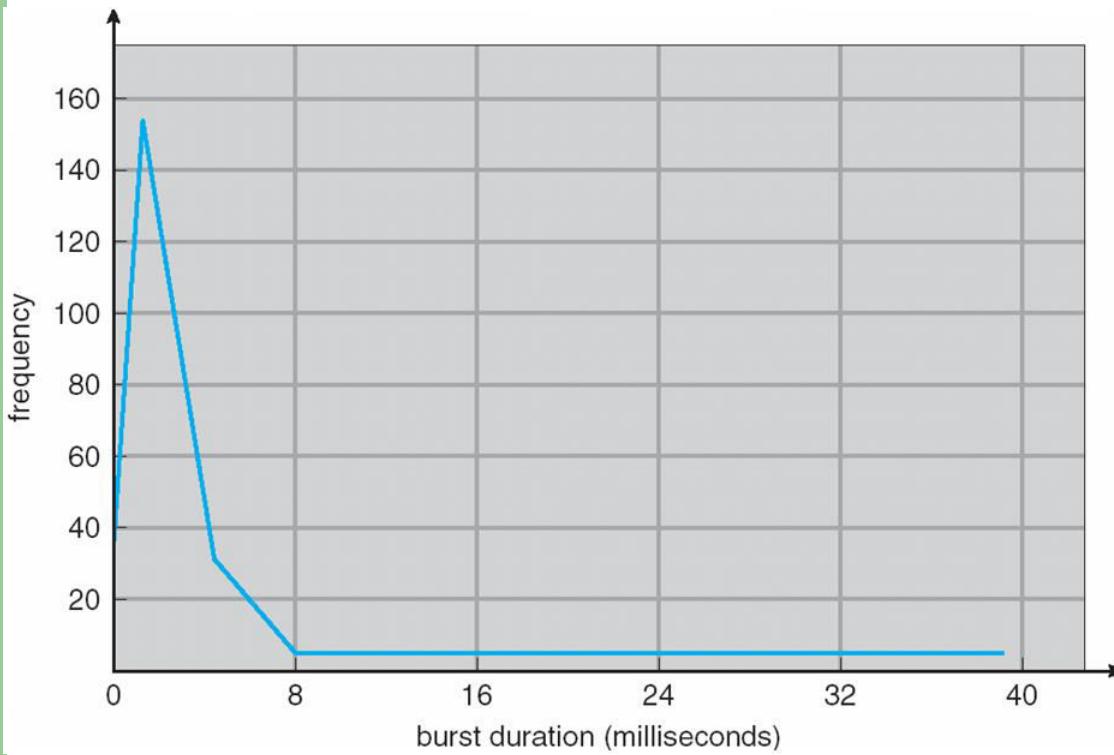
Processes Scheduling

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** and then **CPU burst** and then **I/O burst** and finally terminates with a **CPU burst**.
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



- An I/O bound program typically has many short CPU bursts whereas a CPU bound program might have few long CPU bursts.
- This distribution can be important in the selection of an appropriate CPU scheduling algorithm.
- CPU bursts vary from process to process, and from computer to computer, but an extensive study shows frequency patterns similar to that shown in Figure.
- With a large no. of short CPU bursts and a small no. of long CPU bursts

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
 - i. Switches from running to waiting state (I/O request)
 - ii. Switches from running to ready state (interrupt)
 - iii. Switches from waiting to ready (completion of I/O)
 - iv. Terminates
- Scheduling under i and iv is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

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

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process (from time of submission to time of completion)
- **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

- Maximum CPU utilization
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time

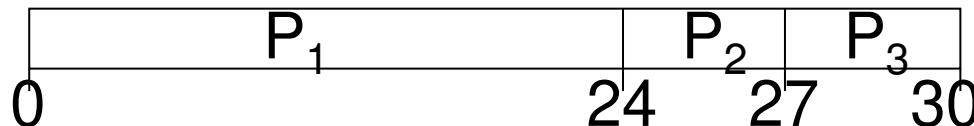
First-Come, First-Served (FCFS) Scheduling

- Runs the processes in the order they arrive at the short-term scheduler, i.e. processes are executed on first come, first serve basis.
- Removes a process from the processor only if it blocks (i.e., goes into the Wait state) or terminates
- Wonderful for long processes when they finally get on
- Terrible for short processes if they are behind a long process
- Easy to understand and implement.
- Poor in performance as average wait time is high.

First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
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 $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

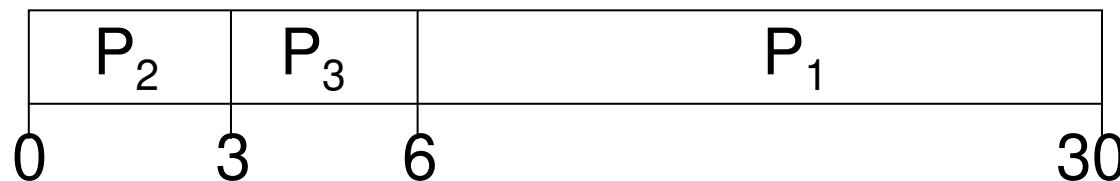
- Average waiting time: $(0 + 24 + 27)/3 = 17$

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 (one CPU-bound and more I/O-bound implies lower CPU and I/O utilization).

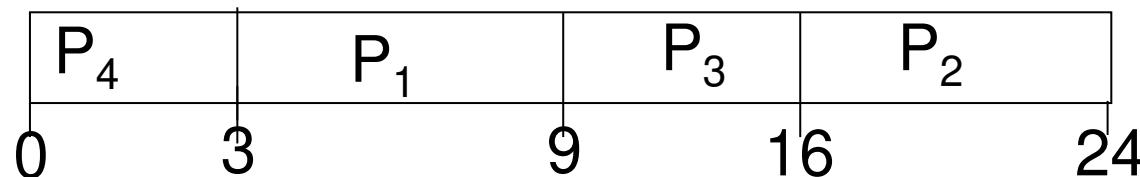
Shortest-Job-First (SJF) Scheduling

- 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
- Impossible to implement

Example of SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart

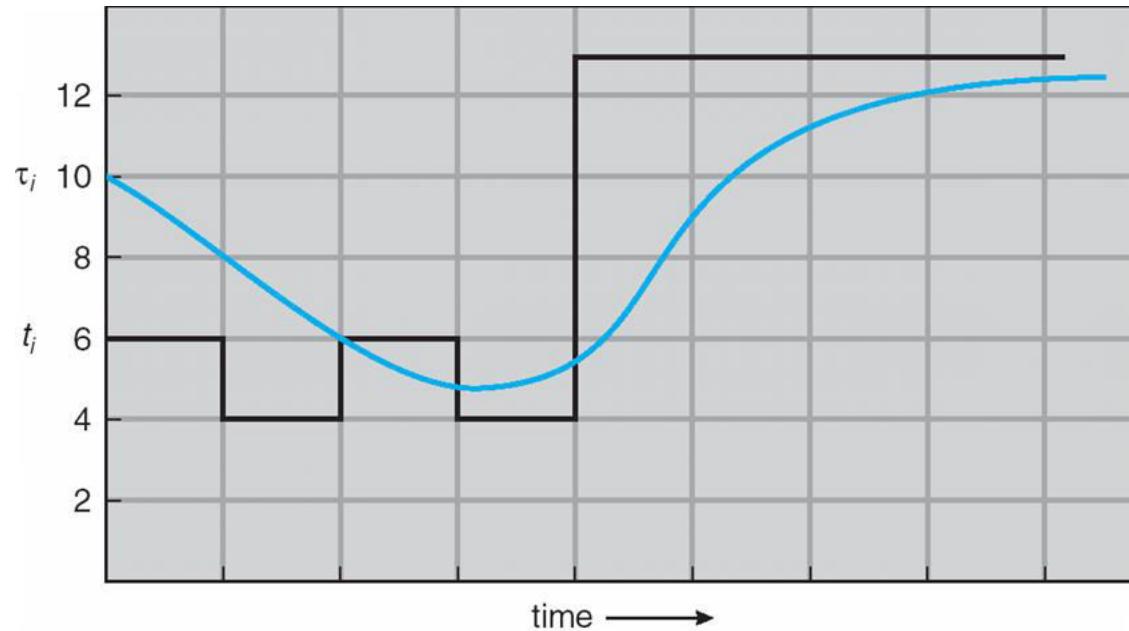


- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Commonly, α set to $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

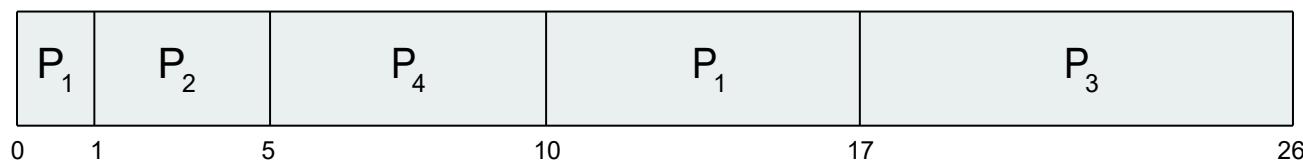
- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:
$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

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

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
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

- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first serve basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priority Scheduling

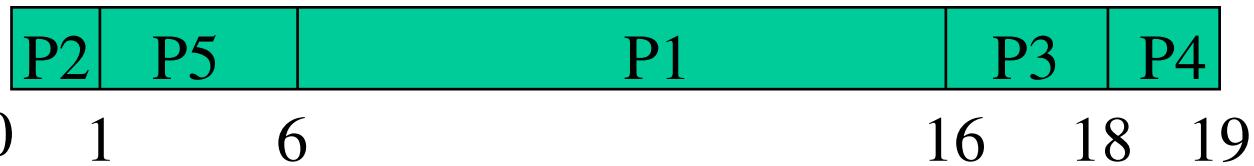
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive
 - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem \equiv **Starvation**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

Example

- Process Burst Time Priority

P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

- Gantt Chart:



- Average waiting time: 8.2 ms

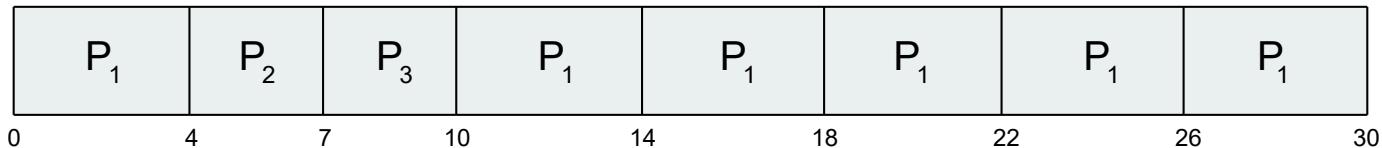
Round Robin (RR)

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

Example of RR with Time Quantum = 4

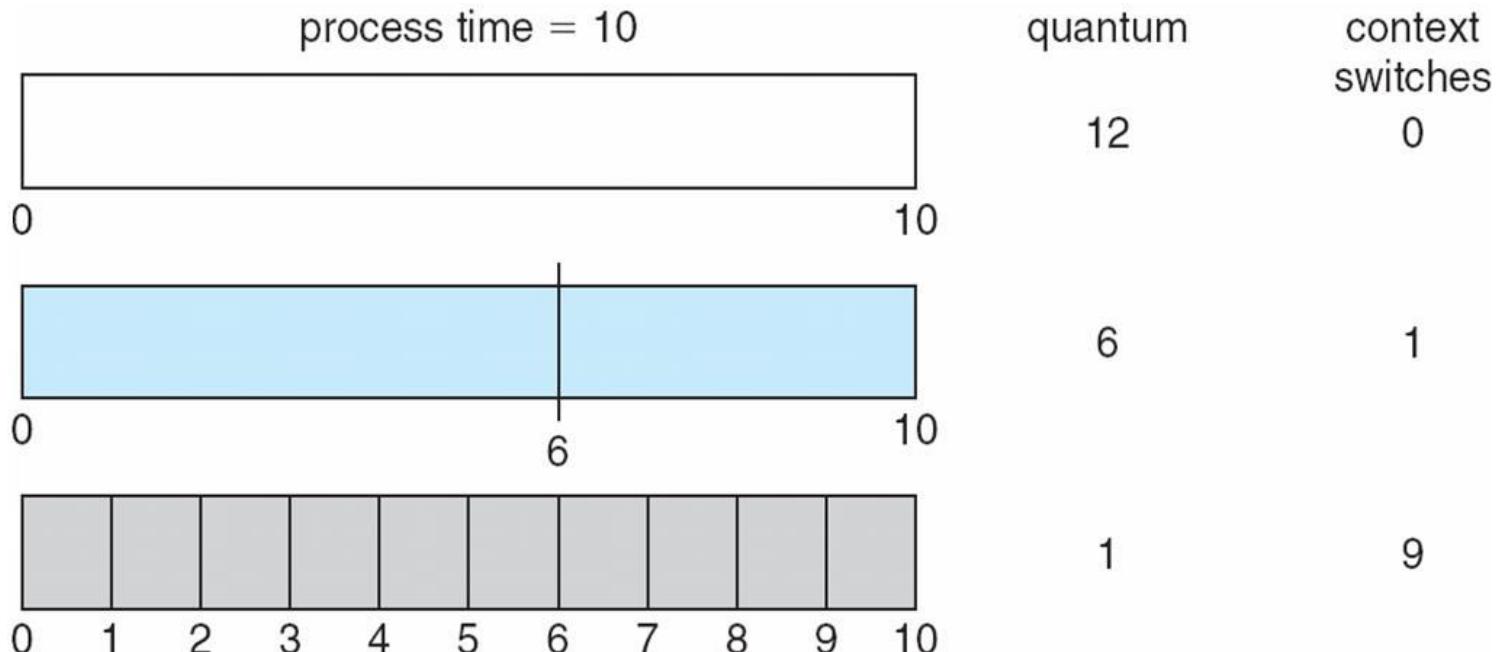
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

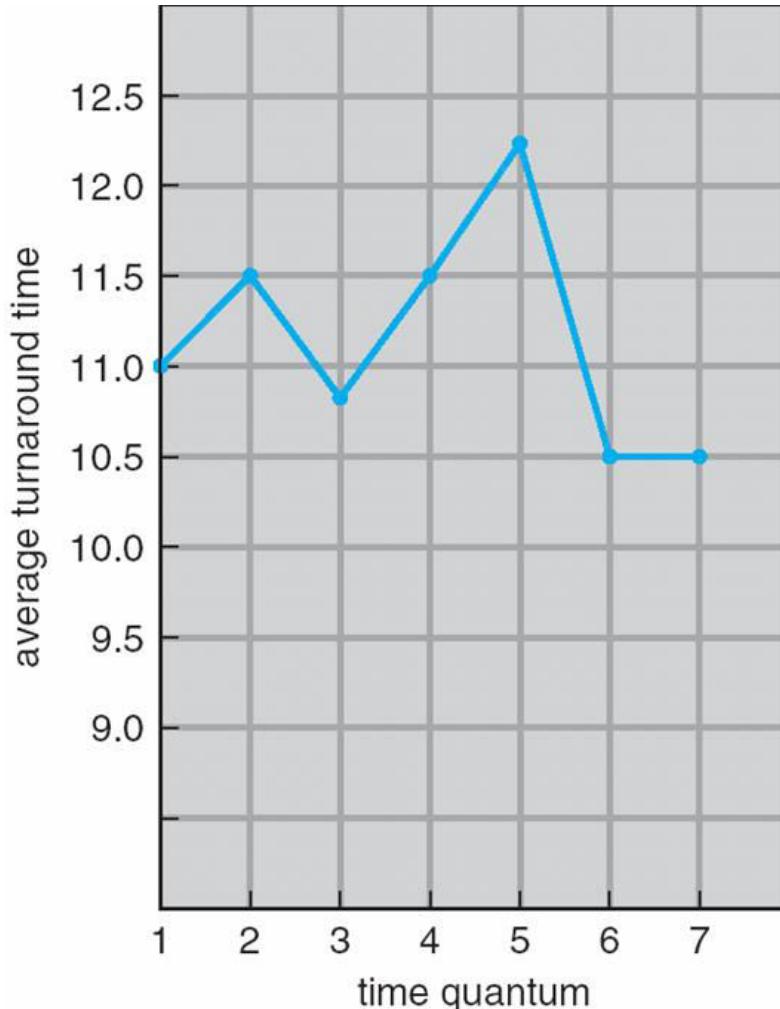


- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



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

Scheduling Policies

Round Robin: very common base policy.

- Run each process for its time slice (scheduling quantum)
 - After each time slice, move the running thread to the back of the queue.
 - Selecting a time slice:
 - Too large - waiting time suffers, degenerates to FCFS if processes are never preempted.
 - Too small - throughput suffers because too much time is spent context switching.
 - Balance the two by selecting a time slice where context switching is roughly 1% of the time slice.
 - A typical time slice today is between 10-100 milliseconds, with a context switch time of 0.1 to 1 millisecond.
 - Max Linux time slice is 3,200ms, Why?
- 24
- Is round robin more fair than FCFS? A)Yes B)No

Properties of RR

- Advantages: simple, low overhead, works for interactive systems
- Disadvantages: if quantum is too small, too much time wasted in context switching; if too large (i.e. longer than mean CPU burst), approaches FCFS.
- Typical value: 20 – 40 msec
- **Rule of thumb:** Choose quantum so that large majority (80 – 90%) of jobs finish CPU burst in one quantum
- RR makes the assumption that all processes are equally important

Deterministic Modelling

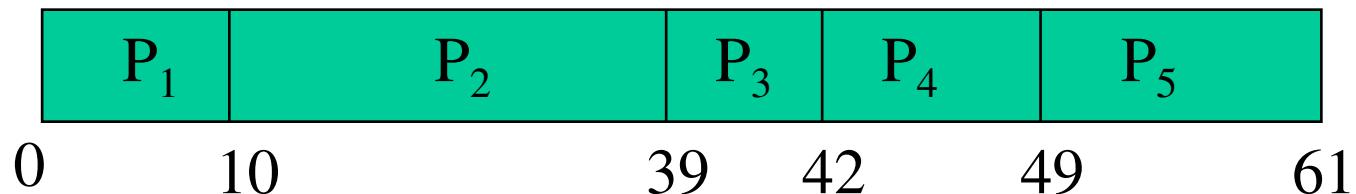
- Take a given workload and calculate the performance of each scheduling algorithm:
 - FCFS, SJF, and RR (quantum = 10 ms)

Example

- At time 0.
- | <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P ₁ | 10 |
| P ₂ | 29 |
| P ₃ | 3 |
| P ₄ | 7 |
| P ₅ | 12 |

Gantt Charts and Times

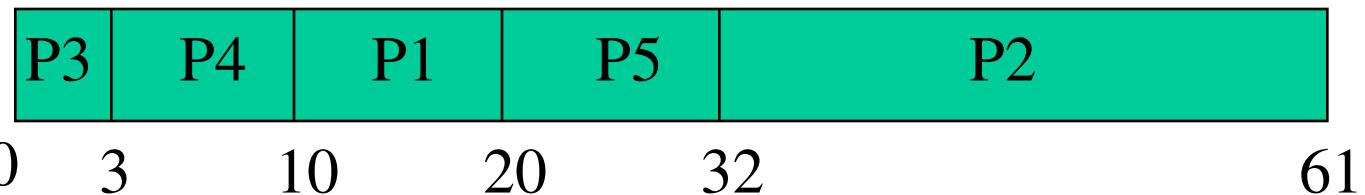
- 1. FCFS:



- Average waiting time:

$$\begin{aligned}(0 + 10 + 39 + 42 + 49)/5 \\ = 28 \text{ ms}\end{aligned}$$

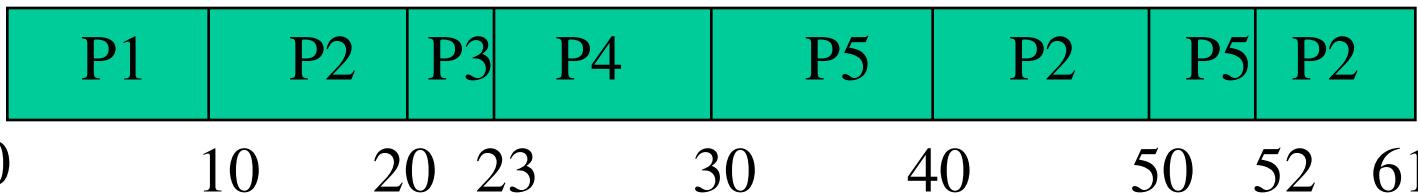
- 2. Non-preemptive SJF:



- Average waiting time:

$$\begin{aligned}(10 + 32 + 0 + 3 + 20)/5 \\ = 13 \text{ ms}\end{aligned}$$

- 3. RR:

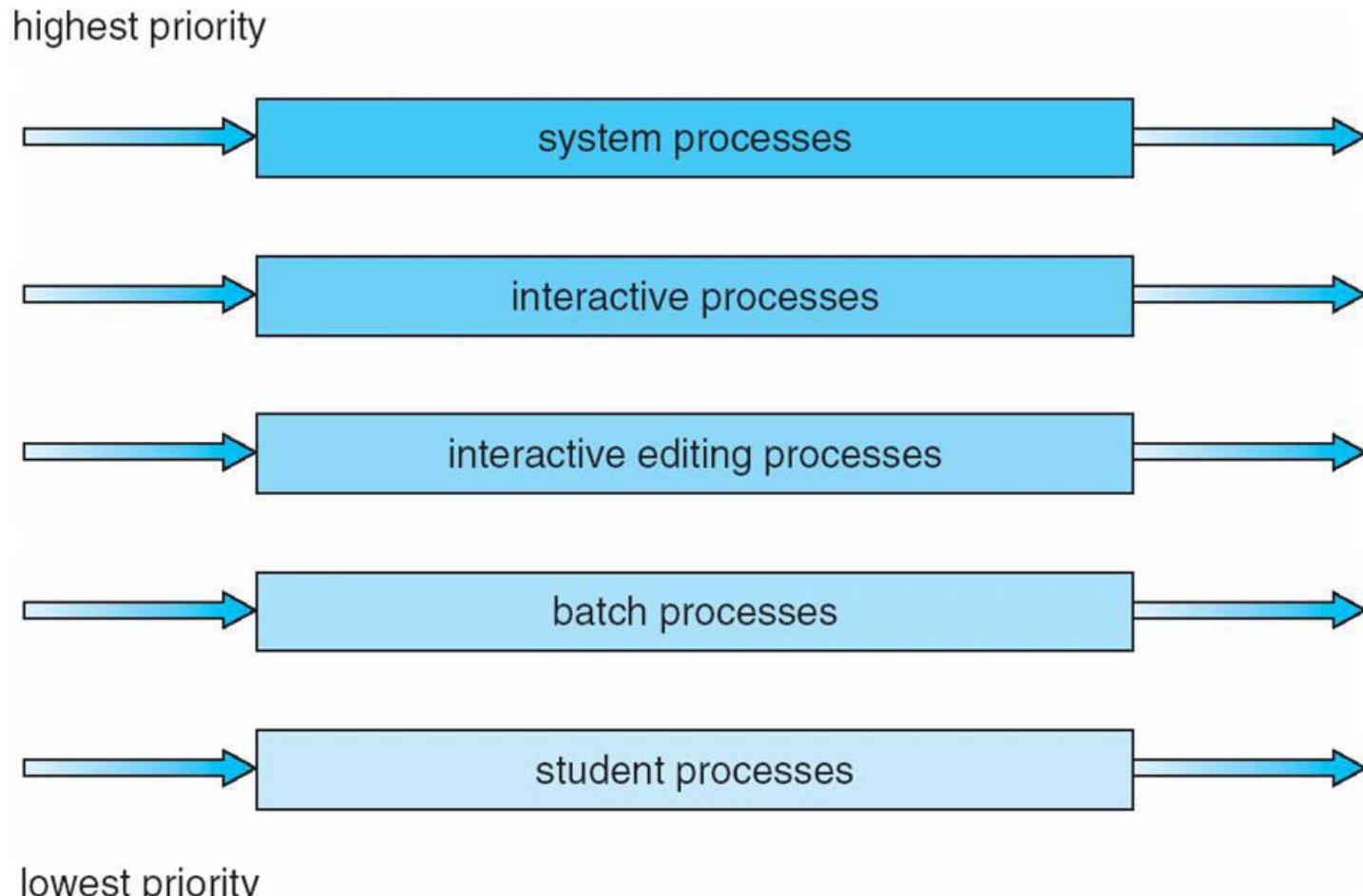


- Average waiting time:
$$(0 + 32 + 20 + 23 + 40)/5 = 23 \text{ ms}$$

Multilevel Queue

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

Multilevel Queue Scheduling

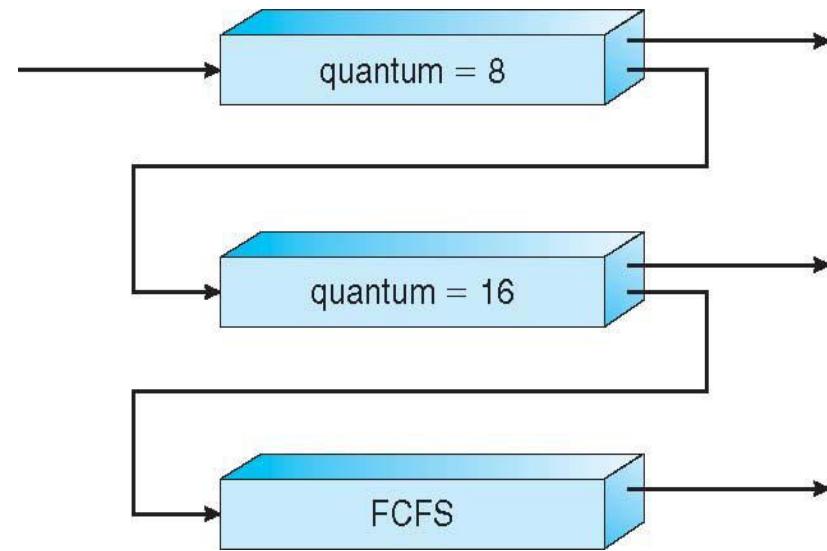


Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

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



Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Operating Systems

Processes
Synchronization

Background

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
 - 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 **counter** that keeps track of the number of full buffers.
 - Initially, **counter** 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.

Process Synchronization

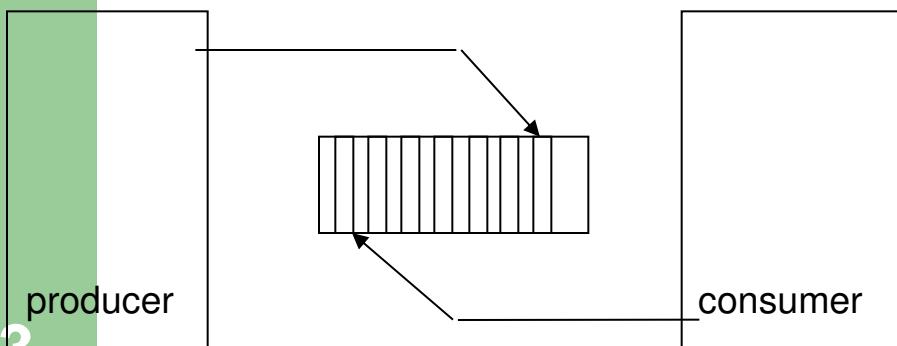
The Producer Consumer Problem

A **producer** process "produces" information "consumed" by a **consumer** process.

```
item nextProduced;           PRODUCER  
  
while (TRUE) {  
    while (counter == BUFFER_SIZE);  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

```
#define BUFFER_SIZE 10  
typedef struct {  
    DATA data;  
} item;  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;  
int counter = 0;
```

```
item nextConsumed;           CONSUMER  
  
while (TRUE) {  
    while (counter == 0);  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```



Race Condition

- A **race condition** is where the outcome of the execution depends on the particular order in which the processes access the shared data.
- **counter++** could be implemented as
- **counter--** could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

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

Critical Section Problem

- Consider system of n processes $\{ P_0, P_1, \dots, P_{n-1} \}$
- Each process has a **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc.
 - When one process is in its critical section, no other may be executing in its critical section
- ***Critical-section problem*** is to design a protocol to solve this
- Each process must ask permission to enter its critical section in **entry section**, may follow critical section with **exit section**, the remaining code is in its **remainder section**

Critical Section

- General structure of process P_i is

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

- **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

Solution to the critical section problem must satisfy the following three requirements:

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 only the processes not in their remainder section can participate in the selection of the process that will enter its critical section next; the selection 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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

Solution to Critical-Section Problem

- Two approaches for handling critical sections in OS, depending on if kernel is preemptive or non-preemptive
 - **Preemptive** – allows preemption of process when running in kernel mode
 - Specially difficult in multiprocessor architectures, but it makes the system more responsive
 - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - Essentially free of race conditions on kernel data structures in kernel mode, since only one active process in the kernel at a time

Types of solutions

Software

- algorithms

Hardware

- rely on some special machine instructions

Operating System supported solutions

- provide some functions and data structures to the programmer to implement a solution

Software solutions: Algorithm 1

Process P_i :

```
do {  
    while (turn != i);  
        critical section  
    turn=j  
        remainder section  
} while (true);
```

An execution view of Algorithm 1

Process P_0 :

```
do {  
    while (turn != 0);  
        critical section  
    turn=1;  
        remainder section  
} while (true);
```

Process P_1 :

```
do {  
    while (turn != 1);  
        critical section  
    turn=0;  
        remainder section  
} while (true);
```

Software solutions: Algorithm 1 (Contd.)

- The shared variable ***turn*** is initialized (to 0 or 1) before executing any P_i
- P_i 's critical section is executed iff **turn = i**
- P_i is ***busy waiting*** if P_j is in CS: mutual exclusion is satisfied
- Progress requirement is not satisfied since it requires **strict alternation** of CSs
- If a process requires its CS more often than the other, it cannot get it.

Algorithm 2 (Contd...)

Process P_i :

```
do {  
    flag[i] = true;  
    while (flag[j]);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```

Process P_0 :

```
do {  
    flag[0] = true;  
    while (flag[1]);  
        critical section  
    flag[0] = false;  
        remainder section  
} while (true);
```

Process P_1 :

```
do {  
    flag[1] = true;  
    while (flag[0]);  
        critical section  
    flag[1] = false;  
        remainder section  
} while (true);
```

- Keep 1 Bool variable for each process: flag[0] and flag[1]
- P_i signals that it is ready to enter its CS by: $\text{flag}[i]:=\text{true}$
- Mutual Exclusion is satisfied but not the progress requirement
- If we have the sequence:

T0: $\text{flag}[0]:=\text{true}$

T1: $\text{flag}[1]:=\text{true}$

Both process will wait forever to enter their CS: we have a ***deadlock***

Peterson's Solution

- Good algorithmic description of solving the problem (but no guarantees for modern architectures)
- Solution restricted to two processes in alternate execution (critical section and remainder section)
- Assume that the **load** and **store** instructions are **atomic**; i.e., cannot be interrupted
- The two processes share two variables:
 - **int turn**
 - **boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter its critical section
- The **flag** array is used to indicate if a process is ready to enter its critical section
flag[i] == true implies that process P_i is ready to enter its critical section

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
        flag[i] = false;
    remainder section
} while (true);
```

- Provable that
 - Mutual exclusion is preserved
 - Progress requirement is satisfied
 - Bounded-waiting requirement is met

Algorithm for Process P_i

- Initialization: flag[0]=flag[1]:=false turn:= 0 or 1
- Willingness to enter CS specified by flag[i]=true
- If both processes attempt to enter their CS simultaneously, only one turn value will last
- Exit section: specifies that P_i is unwilling to enter CS

Execution view of Algorithm 3

Process P_0 :

```
do {  
    flag[0] = true;  
    turn = 1;  
    while (flag[1] && turn == 1);  
        critical section  
    flag[0] = false;  
        remainder section  
} while (true);
```

Process P_1 :

```
do {  
    flag[1] = true;  
    turn = 0;  
    while (flag[0] && turn == 0);  
        critical section  
    flag[1] = false;  
        remainder section  
} while (true);
```

Proof of correctness

Mutual exclusion is preserved since:

- P_0 and P_1 are both in CS only if $\text{flag}[0] = \text{flag}[1] = \text{true}$ and only if $\text{turn} = i$ for each P_i (impossible)

The progress and bounded waiting requirements are satisfied:

- P_i cannot enter CS only if stuck in `while()` with condition $\text{flag}[j] = \text{true}$ and $\text{turn} = j$.
- If P_j is not ready to enter CS then $\text{flag}[j] = \text{false}$ and P_i can then enter its CS
- If P_j has set $\text{flag}[j]=\text{true}$ and is in its `while()`, then either $\text{turn}=i$ or $\text{turn}=j$
- If $\text{turn}=i$, then P_i enters CS. If $\text{turn}=j$ then P_j enters CS but will then reset $\text{flag}[j]=\text{false}$ on exit: allowing P_i to enter CS
- but if P_j has time to reset $\text{flag}[j]=\text{true}$, it must also set $\text{turn}=i$
- since P_i does not change value of turn while stuck in `while()`, P_i will enter CS after at most one CS entry by P_j (bounded waiting)

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- 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-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution to Critical-Section Problem Using Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Solution using test_and_set()

- Two **test_and_set()** cannot be executed simultaneously
- Shared boolean variable **lock**, initialized to FALSE
- Solution:

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

```
do {
    while (test_and_set(&lock))
        /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

compare_and_swap Instruction

- Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

Solution using compare_and_swap

- Shared boolean variable **lock** initialized to FALSE
- Each process has a local boolean variable **key**
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0) //value, expected, new  
        ; /* do nothing */  
        /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock); //only first lock==false will set key=false
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n; //look for the next P[j] waiting: bound-waiting req.
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false; //wakeup only one process P[j] without releasing lock
    /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest synchronization is done with mutex lock
- Protect critical regions by first **acquire()** a lock (then all other processes attempting to get the lock are blocked), and then **release()** it
 - Boolean variable indicating if lock is available or not

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}  
  
release() {  
    available = true;  
}  
  
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Mutex Locks

- Calls to `acquire()` and `release()` must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - All other processes trying to get the lock must continuously loop
 - This lock is therefore called a **spinlock**
 - Very wasteful of CPU cycles
 - Might still be more efficient than (costly) context switches for shorter wait times

Semaphore

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

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Then equivalent to a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

P₁:

```
S1;  
signal(synch); //sync++ has added one resource
```

P₂:

```
wait(synch); //executed only when synch is > 0  
S2;
```

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` codes are placed in the critical 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

- 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 in the appropriate waiting queue
 - wakeup** – remove one of the processes in the waiting queue and place it in the ready queue

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;  
  
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem

Bounded-Buffer Problem

Shared data structures:

- 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

The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next_produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

The structure of the consumer process

```
do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next_consumed */  
    ...  
} while (true);
```

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 a ***single*** writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Variations to the problem:
 - ***First variation*** – no reader kept waiting, unless writer has permission to use shared object
 - ***Second variation*** – once writer is ready, it performs a write as soon as possible
 - Both may have starvation, leading to even more variations
 - first: readers keep coming in while writers are never treated
 - second: writers keep coming in while readers are never treated
 - Problem is solved on some systems by kernel providing **reader-writer locks**

Readers-Writers Problem

Shared data

- Data set
- Semaphore **rw_mutex** initialized to 1
- Semaphore **mutex** initialized to 1
- Integer **read_count** initialized to 0

The structure of a writer process

```
do {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
} while (true);
```

The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```

Dining-Philosophers Problem

- Five philosophers sit around a table with five chopsticks and rice bowl to eat.
- Philosophers think for a while and they want to eat for a while.
- To eat a philosopher requires two chopsticks, one from the left and one from right.
- Assume a philosopher can only pick up one chopstick at a time.
- After eating, chopsticks are placed down and philosopher goes back to thinking.



Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopstick[ (i + 1) % 5] );  
        // eat  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
        // think  
} while (TRUE);
```

What is the problem with the above?

Dining-Philosophers Problem

- First attempt: take left fork, then take right fork
 - Wrong! Results in deadlock.
- Second attempt: take left fork, check to see if right is available, if not put left one down.
 - Still has race condition and can lead to starvation.
- Change so wait a random time.
- Will work usually, but want a solution that will always work, guaranteed.

Dining-Philosophers Problem

- Idea #1: Make the two wait operations occur together as an atomic unit (and the two signal operations)
 - Use another binary semaphore for this
- Idea #2: Does it help if one of the neighbors picks their left chopstick first and the other picks their right chopstick first?
 - Odd philosopher picks left fork followed by right
 - Even philosopher does vice versa
- What is the most # philosophers that can eat simultaneously?

Dining philosophers: textbook solution code

```
#define n 5
#define left (i-1) % n
#define right (i+1) % n
#define thinking 0
#define hungry 1
#define eating 2
int state[n];
semaphore mutex = 1;
semaphore s[n]={0, 0, 0, 0, 0}; // per each philosopher

void philosopher(int i) {
    while(true) {
        think();
        pick_sticks(i);
        eat();
        put_sticks(i);
    }
}
```

Dining philosophers: solution code Π

```
void pick_sticks(int i) {  
    wait(mutex);  
    state[i] = hungry;  
    test(i); /* try getting 2 forks */  
    signal(mutex);  
    wait(s[i]); /* block if no forks acquired */  
}
```

```
void put_sticks(int i) {  
    wait(mutex);  
    state[i] = thinking;  
    test(left);  
    test(right);  
    signal(mutex);  
}
```

```
void test(int i) {  
    if(state[i] == hungry && state[left] != eating  
        && state[right] != eating) {  
        state[i] = eating;  
        signal(s[i]);  
    }  
}
```

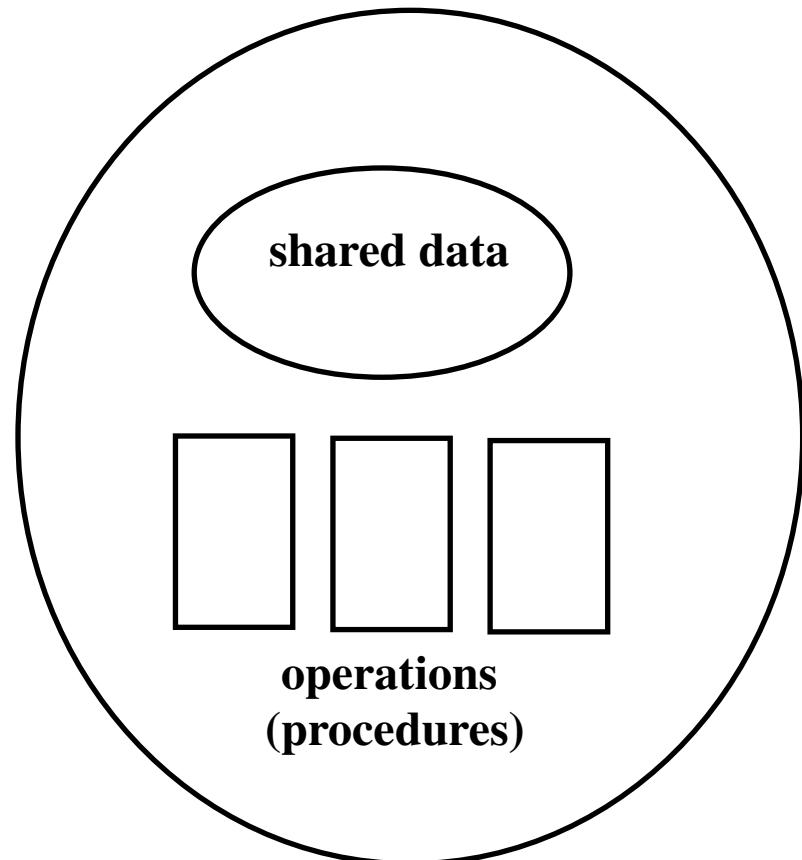
Is the algorithm deadlock-free? What about starvation?

Synchronization with Semaphores

- Semaphores can be used to solve any of the traditional synchronization problems, but suffer from several problems:
 1. semaphores are essentially shared global variables
 2. there is no connection between the semaphore and the data being controlled by the semaphore
 3. Use same mechanism for scheduling and synchronization.
 4. they can be accessed from anywhere in the code (**global**)
 5. there is no control or guarantee of proper usage
- So, semaphores are sometimes hard to use and prone to bugs.
- One solution is to provide programming language support for synchronization.

Monitors

- A monitor is a programming language construct that controls access to shared data
- A monitor is a module that encapsulates
 - Shared data structures
 - Procedures that operate on shared data structures
 - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access



Monitors

- A monitor is a set of multiple routines which are protected by a mutual exclusion **lock**.
- None of the routines in the monitor can be executed by a process until that process acquires the lock.
- This means that only ONE process can execute within the monitor at a time.
- Any other processes must wait for the process that's currently executing to give up control of the lock.
- However, a process can actually suspend itself inside a monitor and then wait for an event to occur.
- If this happens, then another process is given the opportunity to enter the monitor.
- The process that was suspended will eventually be notified that the event it was waiting for has now occurred, which means it can wake up and reacquire the lock.

Monitor Facilities

- **A monitor guarantees mutual exclusion**
 - only one process can be executing within the monitor at any instant
 - semaphore implicitly associated with monitor.
 - if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
 - More restrictive than semaphores
 - easier to use most of the time
- **Once in the monitor, a process may discover that it cannot continue, and may wish to sleep. Or it may wish to allow a waiting process to continue.**
 - *Condition Variables* provide synchronization within the monitor so that processes can wait or signal others to continue.

Condition Variables

- To allow a process to wait within the monitor, a **condition** variable must be declared, as
condition x, y;
- Condition variable can only be used with the operations **wait()** and **signal()**.
 - The operation
x.wait();
means that the process invoking this operation is suspended until another process invokes
x.signal();
 - The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal()** operation has no effect.
- Wait() and signal() operations of the monitors are not the same as semaphore wait() and signal() operations!

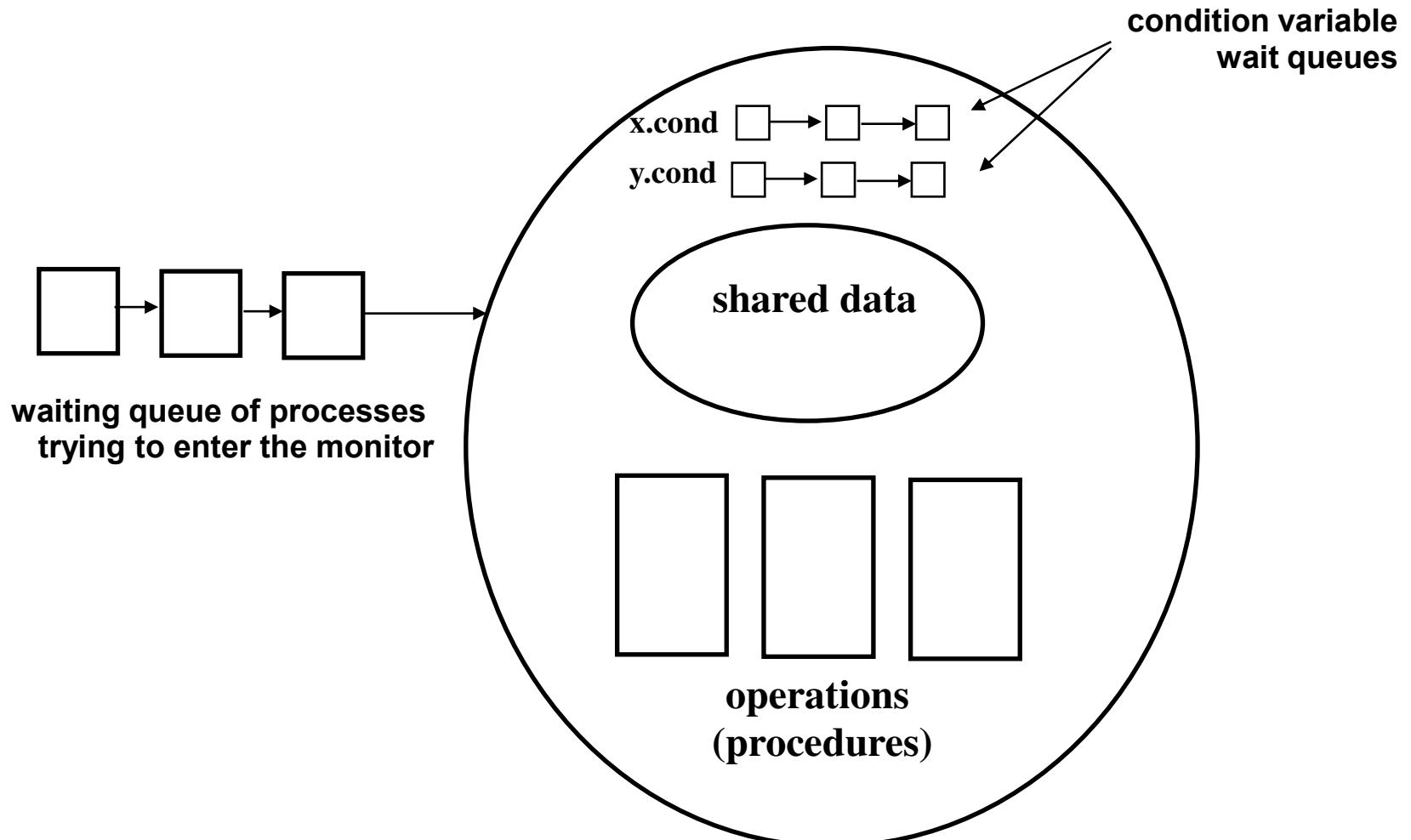
Condition Variables

- Three operations on condition variables
 - Condition.wait(c)
 - release monitor lock, wait for someone to signal condition
 - Condition.signal(c)
 - wakeup 1 waiting thread
 - Condition.broadcast(c)
 - wakeup all waiting threads

Basic Ideas

- the monitor is controlled by a lock; only 1 process can enter the monitor at a time; others are queued
- condition variables provide a way to wait; when a process blocks on a condition variable, it gives up the lock.
- a process *signals* when a **resource** or **condition** has become available; this causes a waiting process to resume *immediately*.
- The lock is automatically passed to the waiter; the original process blocks.

Monitors Have Several Associated Queues



Monitor with Condition Variables

- When a process P “signals” to wake up the process Q that was waiting on a condition, potentially both of them can be active.
- However, monitor rules require that at most one process can be active within the monitor. Who will go first?
 - **Signal-and-wait:** P waits until Q leaves the monitor (or, until Q waits for another condition).
 - **Signal-and-continue:** Q waits until P leaves the monitor (or, until P waits for another condition).
 - **Signal-and-leave:** P has to leave the monitor after signaling

The design decision is different for different programming languages

Solution to Dining Philosophers

```
monitor dp {  
    enum {thinking, hungry, eating} state[5];  
    condition self[5];  
}  
initialization_code() {  
    for (int i = 0; i < 5; i++)  
        state[i] = thinking;  
}
```

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

```
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.)

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

dp.pickup(i);

...

eat

...

dp.putdown(i);

- No deadlocks, but starvation is possible

Differences between Monitors and Semaphores

- Both Monitors and Semaphores are used for the same purpose – process synchronization.
- But, monitors are simpler to use than semaphores because they handle **all of the details of lock acquisition and release**.
- An application using semaphores has to release any locks a process has acquired when the application terminates – this must be done by the application itself.
- If the application does not do this, then any other process that needs the shared resource will not be able to proceed.
- Another difference when using semaphores is that every routine accessing a shared resource has to explicitly acquire a lock before using the resource.
- This can be easily forgotten when coding the routines dealing with multithreading . Monitors, unlike semaphores, automatically acquire the necessary locks.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Each external function F will be replaced by

```
wait(mutex);
...
body of  $F$ ;
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable x , we have:

```
semaphore x_sem; // (initially = 0)  
int x_count = 0;
```

The operation **x.wait()** can be implemented as:

```
x_count++;  
if (next_count > 0)  
    signal(next);  
else  
    signal(mutex);  
wait(x_sem);  
x_count--;
```

The operation **x.signal()** can be implemented as:

```
if (x_count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```

Resuming Processes within a Monitor

- If several processes queued on condition **x**, and **x.signal()** executed, which one should be resumed?
- First-come, first-served (FCFS) frequently not adequate
- **conditional-wait** construct of the form **x.wait(c)**
 - where **c** is **priority number**
 - Process with lowest number (highest priority) is scheduled next
 - Example of ResourceAllocator, next

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator {  
    boolean busy;  
    condition x;  
  
    void acquire(int time) {  
        if (busy)  
            x.wait(time); //time: maximum time it will keep resource  
        busy = TRUE;  
    }  
  
    void release() {  
        busy = FALSE;  
        x.signal();  
    }  
  
    initialization_code() {  
        busy = FALSE;  
    }  
}
```

Synchronization Examples

- Windows XP
- Solaris
- Linux
- Pthreads

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land (outside the kernel) which may act as mutex locks, semaphores, events, and timers
 - **Event** acts much like a condition variable (notify thread(s) when condition occurs)
 - **Timer** notifies one or more threads when specified amount of time has expired
 - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutex locks** for efficiency when protecting data from short code segments
 - Starts as a standard semaphore spin-lock
 - If lock is held, and by a thread running on another CPU, spins
 - If lock is held by non-run-state thread, block and sleep, waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
 - Turnstiles are per-lock-holding-thread, not per-object

Linux Synchronization

- Linux:
 - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
 - Version 2.6 and later, fully preemptive
- Linux provides:
 - semaphores
 - spinlocks
 - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

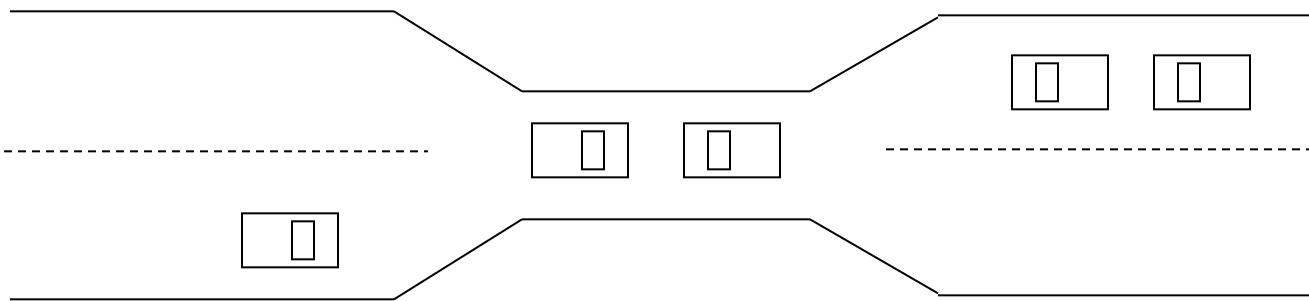
Operating Systems

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 $P_0 \quad P_1$
wait (A); wait(B)
wait (B); wait(A)

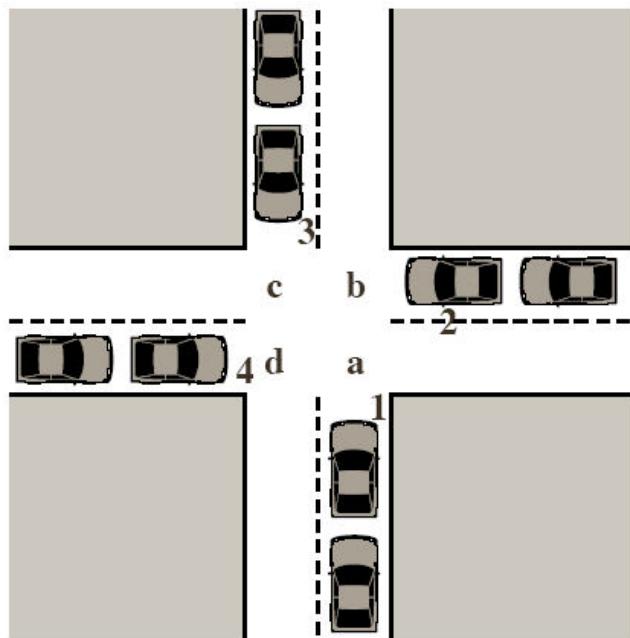
Bridge Crossing Example



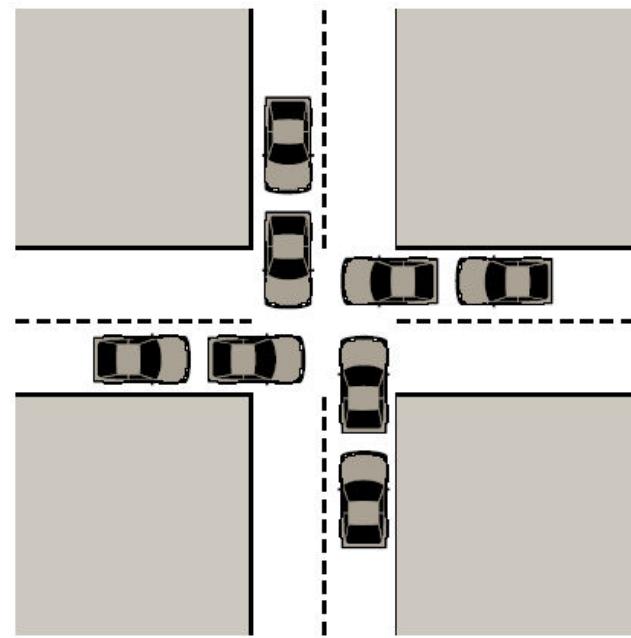
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

Deadlock Principles

- A deadlock is a permanent blocking of a set of threads/processes
 - ✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other



(a) Deadlock possible



(b) Deadlock

System Model

- Resource types R_1, R_2, \dots, 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 $\{P_0, P_1, \dots, P_n\}$ 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 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

Resource-Allocation Graph

A visual (mathematical) way to determine if a deadlock has, or may occur.

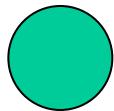
$G = (V, E)$ The graph contains nodes and edges.

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

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$: An arrow from the process to resource indicates the process is requesting the resource
- **assignment edge** – directed edge $R_j \rightarrow P_i$: An arrow from resource to process shows an instance of the resource has been allocated to the process

Resource-Allocation Graph (Cont.)

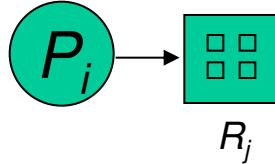
- Process



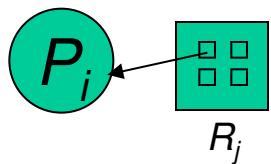
- Resource Type with 4 instances



- P_i requests instance of R_j

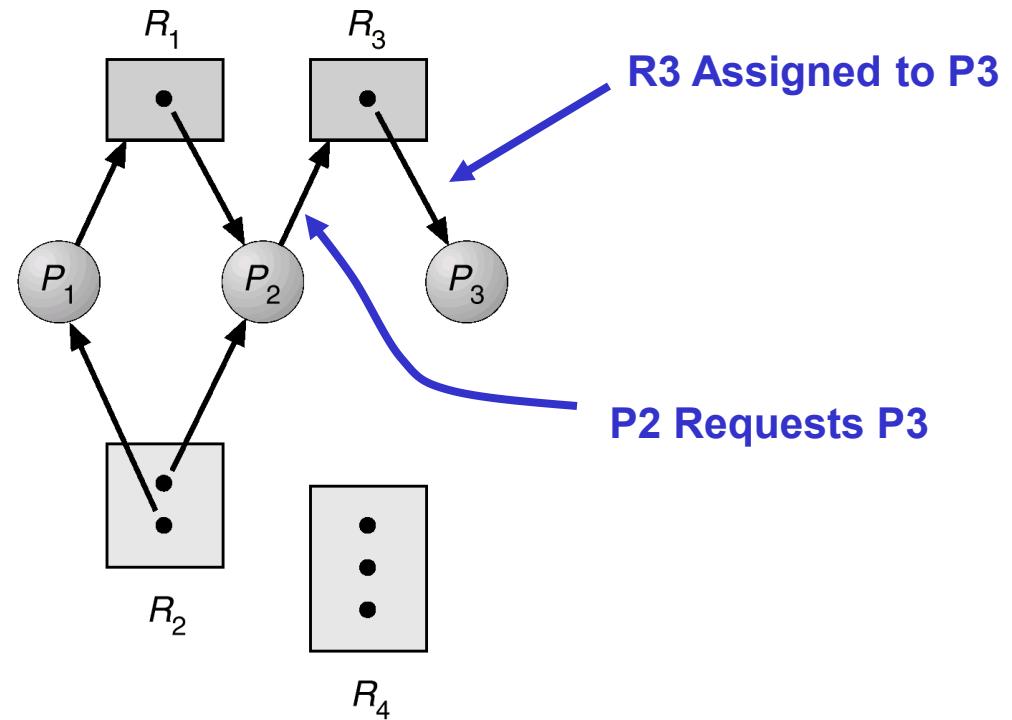


- P_i is holding an instance of R_j



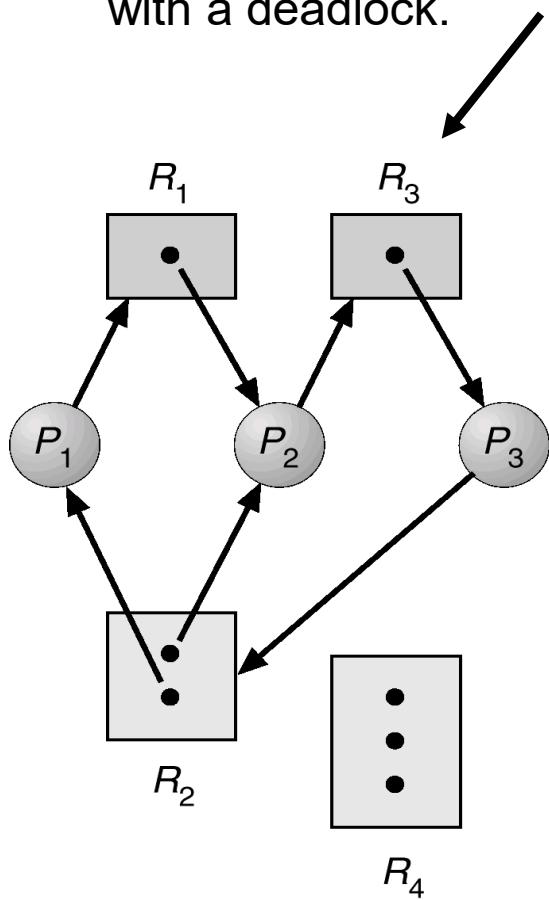
Example of a Resource Allocation Graph

Resource allocation graph

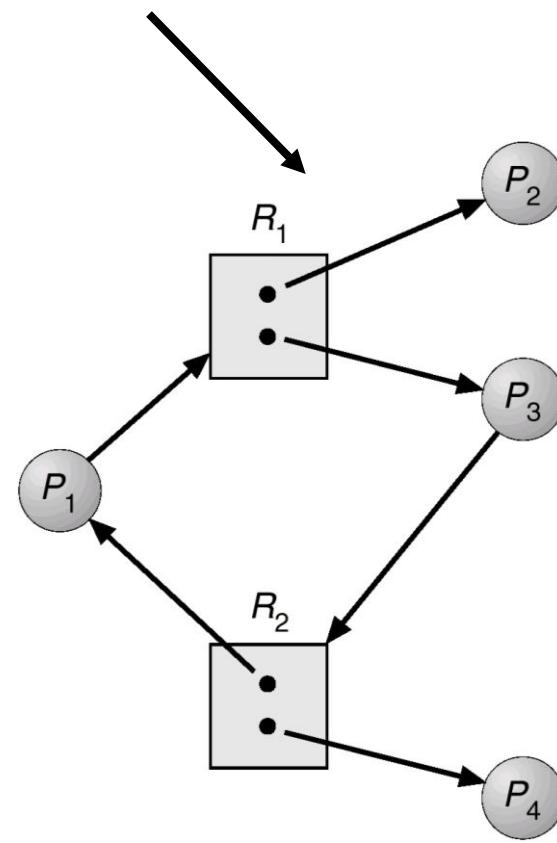


Resource Allocation Graph

Resource allocation graph
with a deadlock.



Resource allocation 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

How To Handle Deadlocks – General Strategies

There are three methods:

- Ignore Deadlocks:

- Ensure deadlock **never** occurs using either

Prevention Prevent any one of the 4 conditions from happening.

Avoidance Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations..

- **Allow** deadlock to happen. This requires using both:

Detection Know a deadlock has occurred.

Recovery Regain the resources.

Deadlock Prevention

Do not allow one of the four conditions to occur.

- **Mutual Exclusion**
 - Automatically holds for printers and other non-sharables.
 - Shared entities (read only files) don't need mutual exclusion (and aren't susceptible to deadlock.)
 - Prevention not possible, since some devices are intrinsically non-sharable.
- **Hold and Wait**
 - Collect all resources before execution.
 - must guarantee that whenever a process requests a resource, it does not hold any other resources
 - **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

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

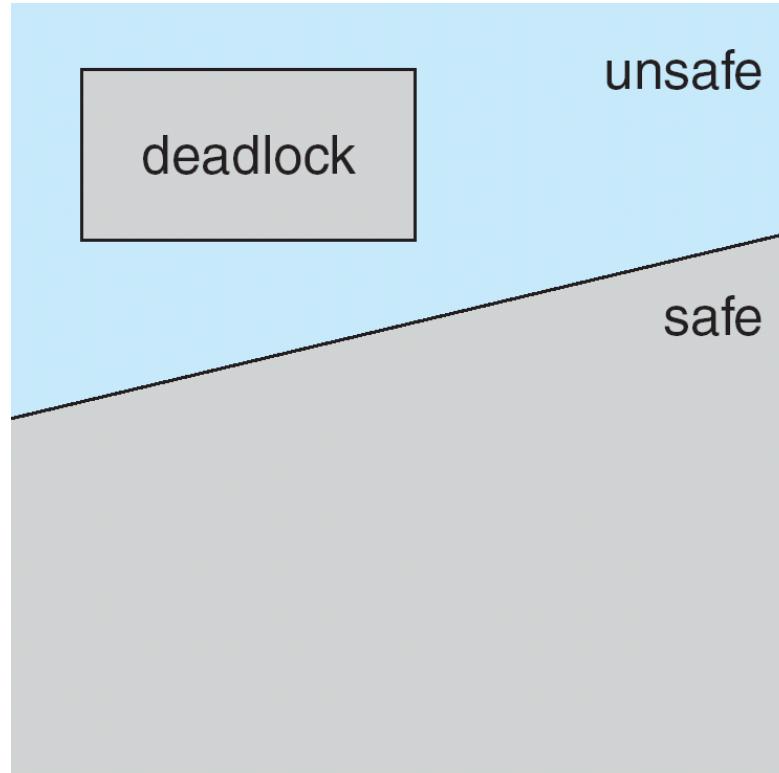
Safe State

- 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 $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in 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 \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



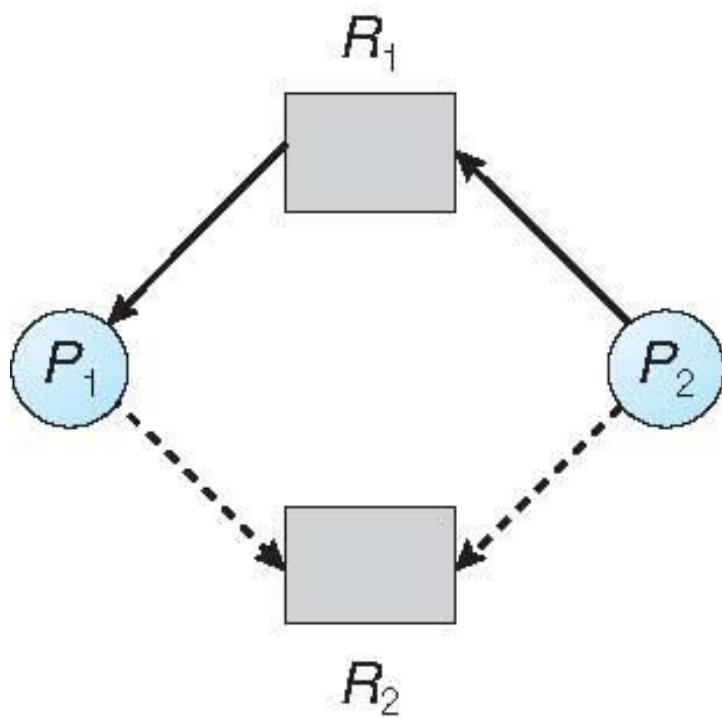
Avoidance algorithms

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

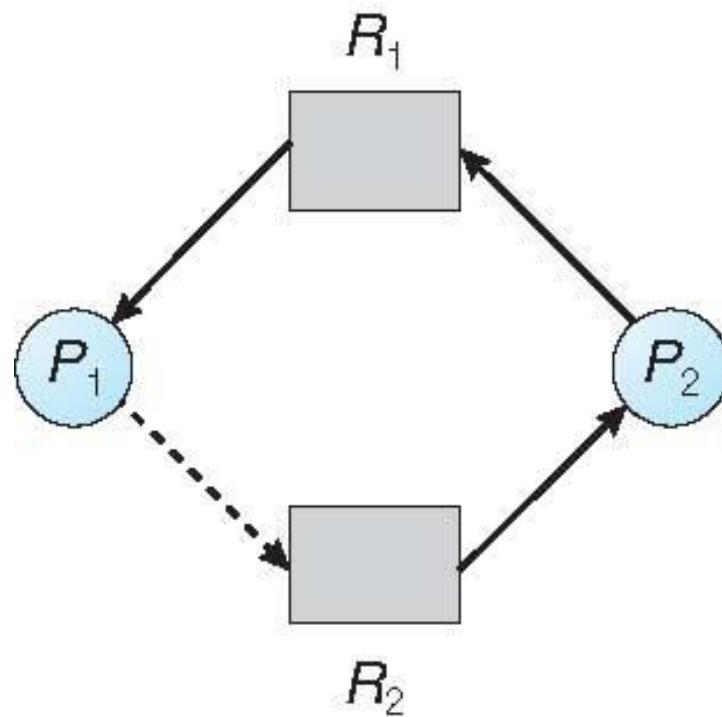
Resource-Allocation Graph Scheme

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

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Safety Algorithm

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

$Work = Available$

$Finish [i] = false$ for $i = 0, 1, \dots, n - 1$

2. Find an 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 P_i
- If unsafe $\Rightarrow P_i$ 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 (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	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*

	<u>Need</u>		
	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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 2	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?

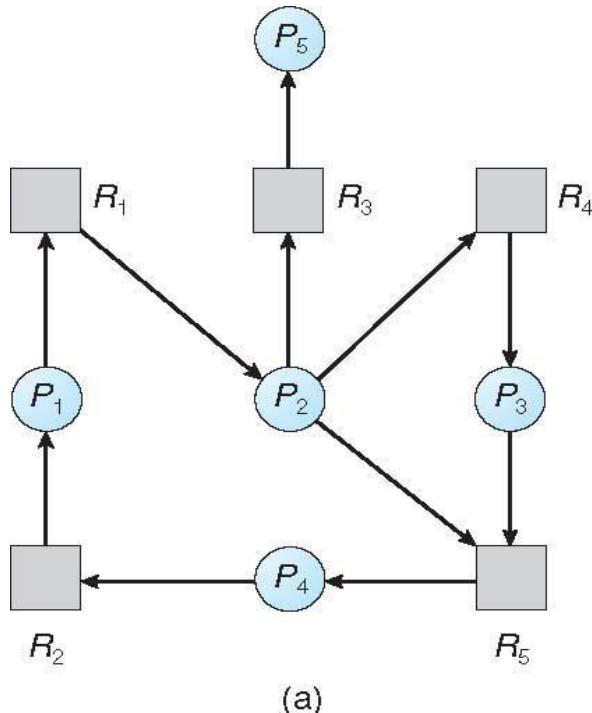
Deadlock Detection

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

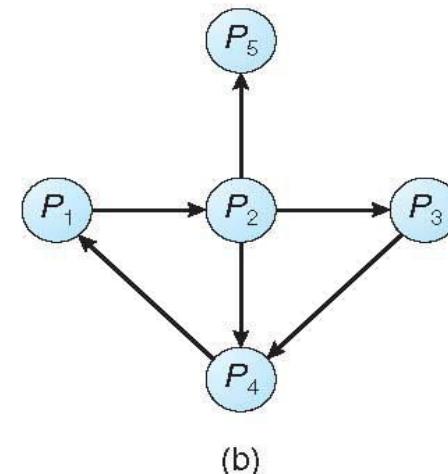
Single Instance of Each Resource Type

- 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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ 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) $\text{Work} = \text{Available}$
 - (b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then
 $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$
2. Find an index i such that both:
 - (a) $\text{Finish}[i] == \text{false}$
 - (b) $\text{Request}_i \leq \text{Work}$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 $Finish[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == \text{false}$, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ 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 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ 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	2
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
- Problem: starvation – same process may always be picked as victim, include number of rollback in cost factor

What is a Livelock?

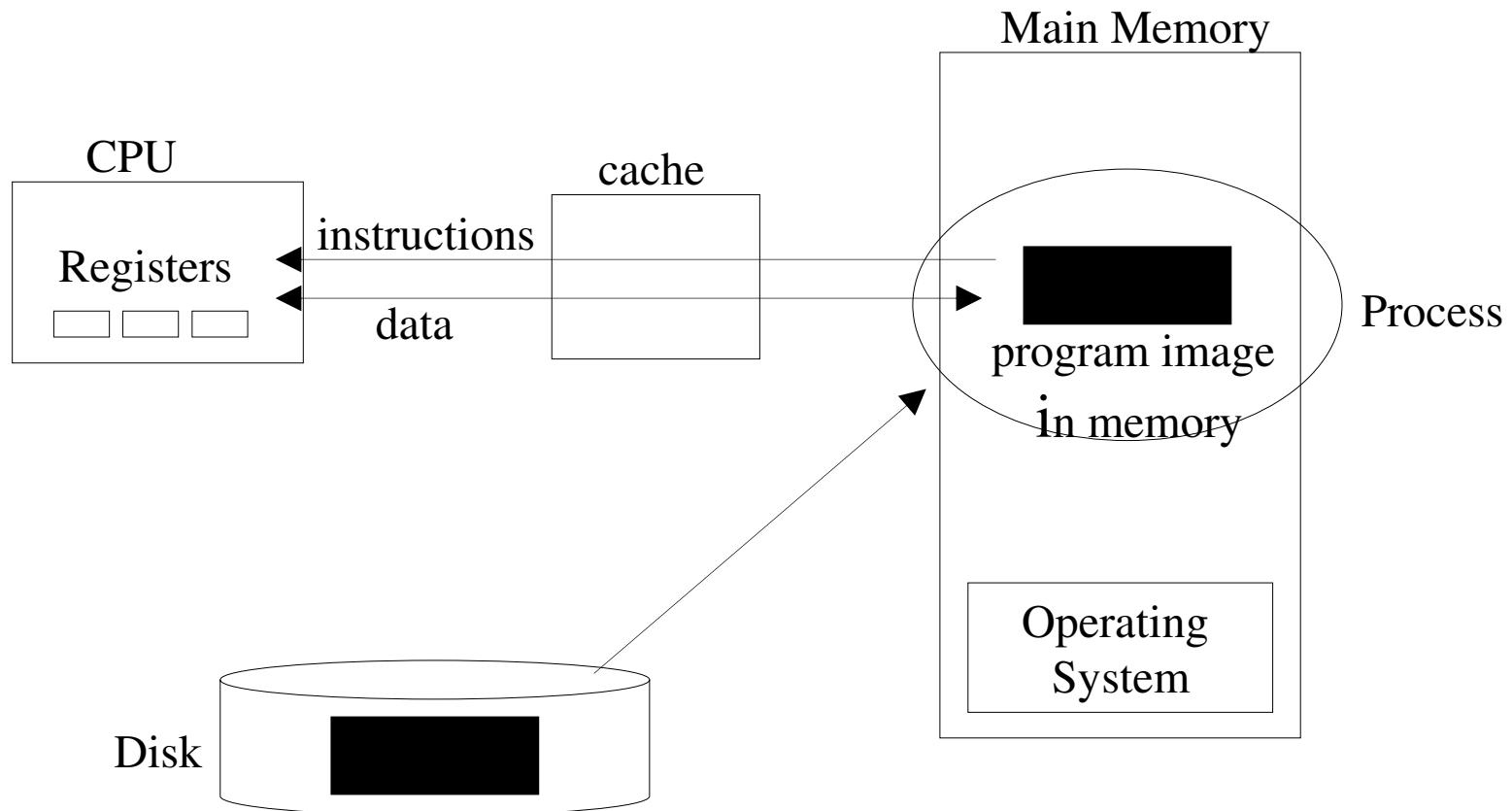
- There is a variant of deadlock called **livelock**.
- This is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
- This is similar to deadlock in that no progress is made but differs in that **neither process is blocked nor waiting for anything**.
- A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

Operating Systems

Memory Management

Alok Kumar Jagadev

Background



Memory Management

- In most memory management schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes
- Memory management is the process of
 - allocating primary memory to user programs
 - reclaiming that memory when it is no longer needed
 - protecting each user's memory area from other user programs; i.e., ensuring that each program only references memory locations that have been allocated to it.

Memory Management

- In order to manage memory effectively the OS must have
 - Memory allocation policies
 - Methods to track the status of memory locations (**free** or **allocated**)
 - Policies for preempting memory from one process to allocate to another

Memory Management Requirements

- Memory management is intended to satisfy the following requirements:
 - Relocation
 - Protection
 - Sharing
 - Logical organization
 - Physical organization

Requirements: Relocation

- Relocation is the process of adjusting program addresses to match the actual physical addresses where the program resides when it executes
- Why is relocation needed?
 - programmer cannot know where the program will be placed in memory when it is executed
 - a process may be (often) relocated in main memory due to swapping (maximize the CPU utilization)
 - swapping enables the OS to have a larger pool of ready-to-execute processes
 - memory references in code (for both instructions and data) must be translated to actual physical memory address

Requirements: Protection

- Processes should not be able to reference memory locations in another process without permission
- impossible to check addresses at compile time in programs since the program could be relocated
- Memory references generated by a process must be checked at run time

Requirements: Sharing

- must allow several processes to access a common portion of main memory without compromising protection
- cooperating processes may need to share access to the same data structure
- better to allow each process to access the same copy of the program rather than have their own separate copy

Requirements: Logical Organization

- Main memory is organized as a linear (1-D) address space consisting of a sequence of bytes or words.
- users write programs in modules with different characteristics
 - instruction modules are **execute-only**
 - data modules are either **read-only** or **read/write**
 - some modules are **private** others are **public**
- To effectively deal with user programs, the OS and hardware should support a basic form of module to provide the required protection and sharing

Requirements: Physical Organization

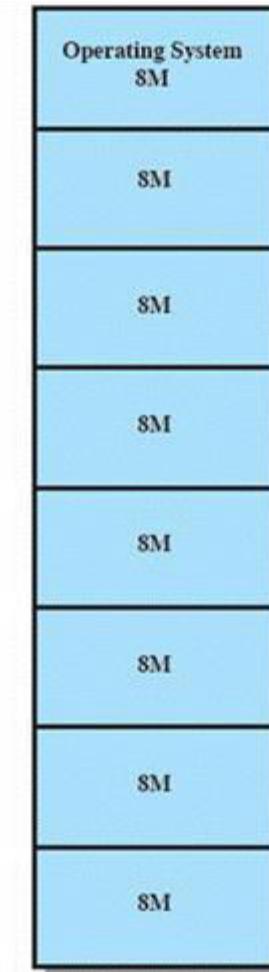
- **secondary memory** is the long term store for programs and data whereas **main memory** holds program and data currently in use
- moving information between these two levels of memory is a major concern of memory management
 - it is highly inefficient to leave this responsibility to the application programmer

Simple Memory Management

- An executing process must be loaded entirely in main memory (if overlays are not used)
- Although the following simple memory management techniques are not used in modern OS, they lay the ground for a proper discussion of virtual memory
 - fixed partitioning
 - dynamic partitioning
 - simple paging
 - simple segmentation

Fixed Partitioning

- **Fixed partitions:** Partition main memory into a set of non overlapping regions called **partitions**
- Partitions can be of **equal** or **unequal** sizes
- Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap out a process if all partitions are full and no process is in the **ready** or **running** state



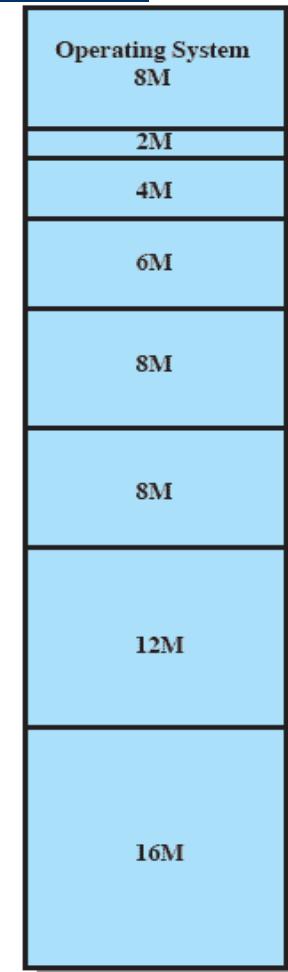
(a) Equal-size partitions

Fixed Partitioning Problems

- a program may be **too large** to fit in a partition. The **programmer must then design the program with overlays**
 - when the module needed is not present the user program must load that module into the program's partition, overlaying whatever program or data are there
- Main memory utilization is inefficient
 - any program, regardless of size, occupies an entire partition
 - ***internal fragmentation***
 - » wasted space due to the block of data loaded being smaller than the partition

Solution – Unequal Size Partitions

- Using unequal size partitions helps lessen the problems
 - programs up to 16M can be accommodated without overlays
 - partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation



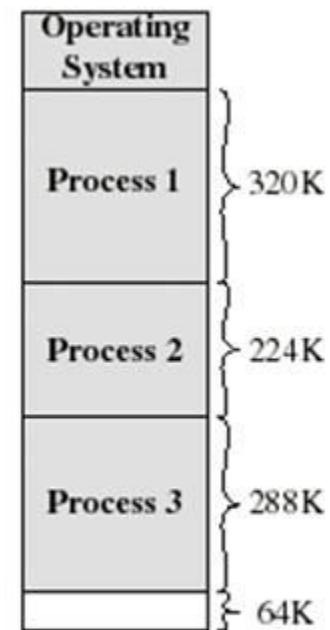
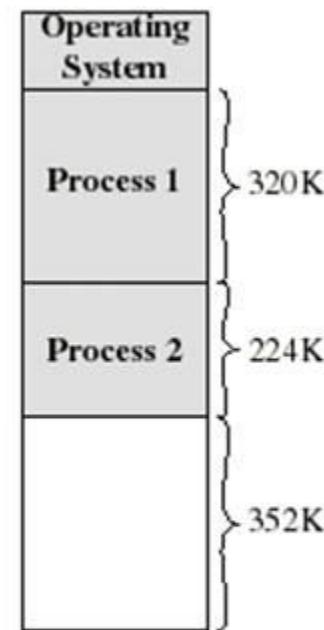
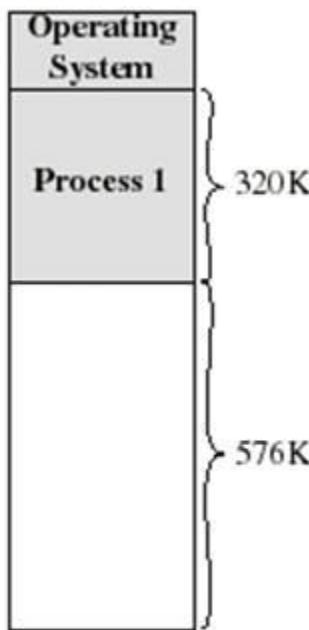
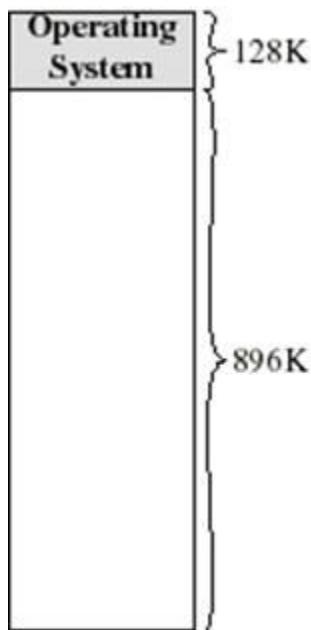
(b) Unequal-size partitions

Dynamic Partitioning

- Partitions are of variable length and number
- Each process is allocated exactly as much memory as it requires
- Eventually holes are formed in main memory. This is called **external fragmentation**
- Must use **compaction** to shift processes so they are contiguous and all free memory is in one block

Dynamic Partitioning Example

- A hole of 64K is left after loading 3 processes: not enough room for another process
- Eventually each process is blocked. The OS swaps out process 2 to bring in process 4



(a)

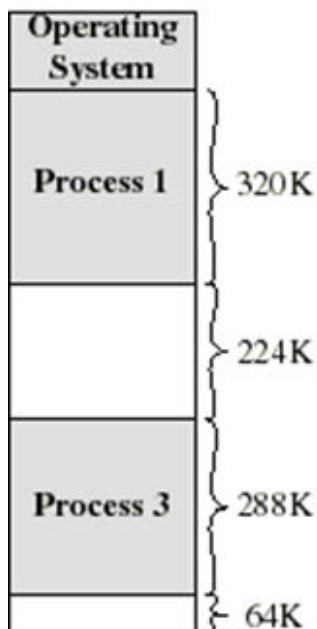
(b)

(c)

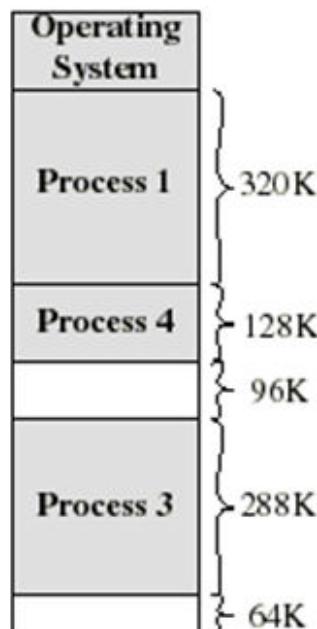
(d)

Dynamic Partitioning: an example

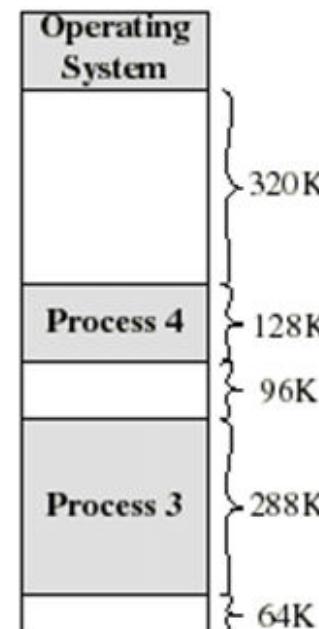
- another hole of 96K is created
- Eventually each process is blocked. The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...
- Compaction would produce a single hole of 256K



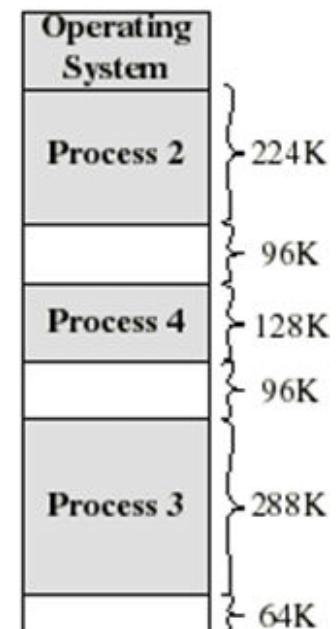
(e)



(f)



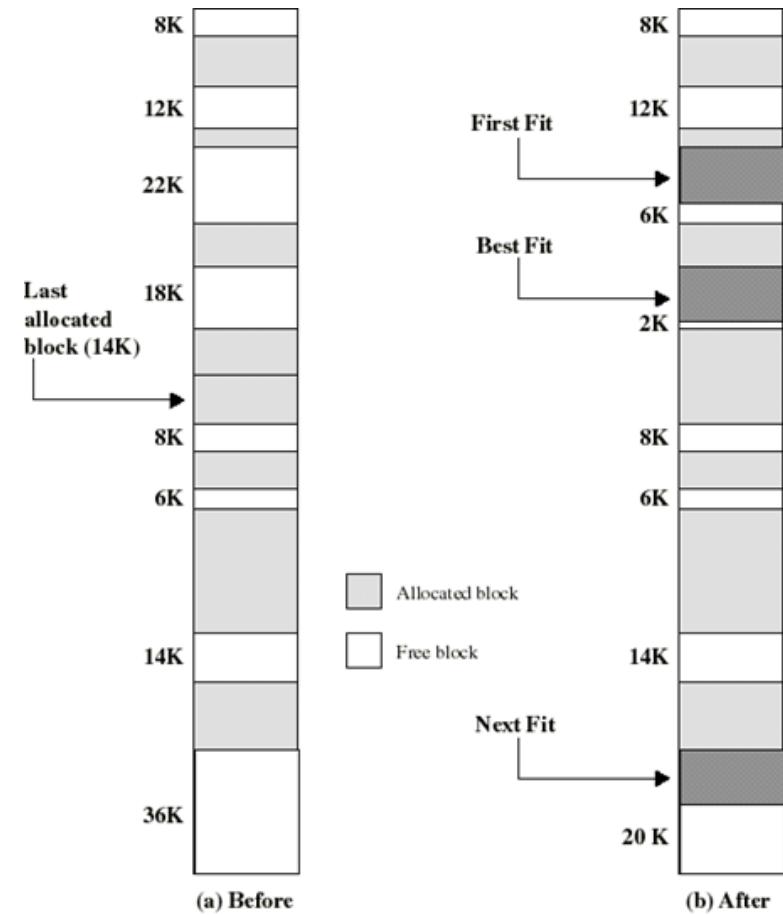
(g)



(h)

Placement Algorithm

- Used to decide which free block to allocate to a process
- Goal: to reduce usage of compaction (time consuming)
- Possible algorithms:
 - Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
 - First-fit:** Allocate the *first* hole that is big enough
 - Worst-fit:** Allocate the *largest* hole; must also search entire list
 - Produces the largest leftover hole



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Address Types

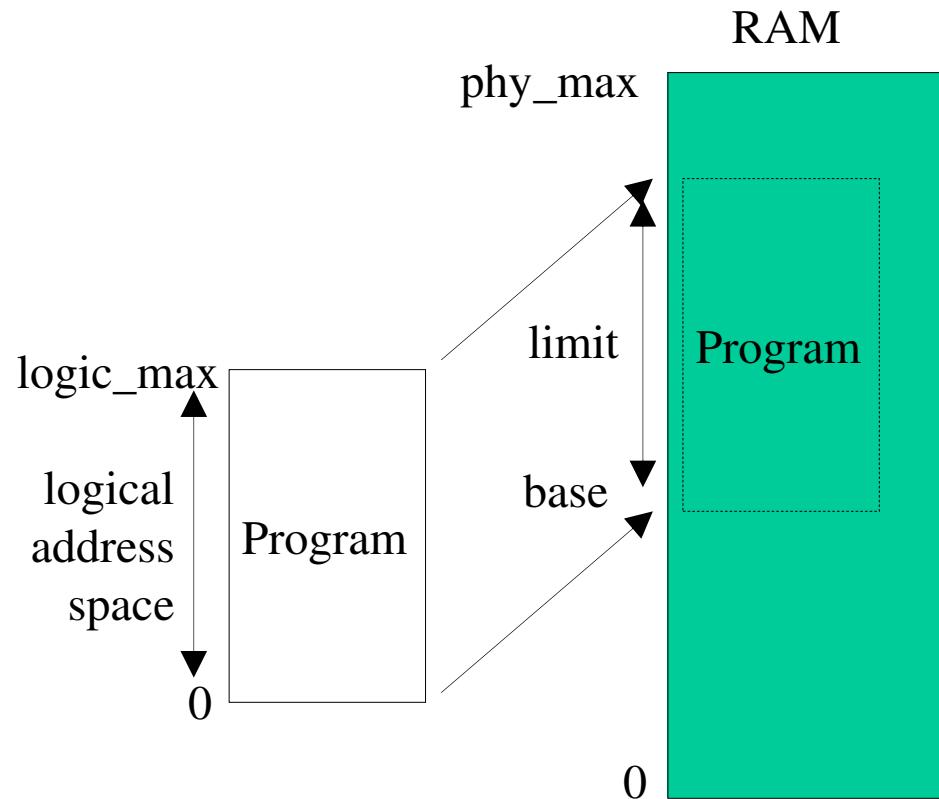
- A **physical address** (**absolute address**) is a physical location in main memory
- A **logical address** is a reference to a memory location independent of the physical structure/organization of memory
- Compilers produce code in which all memory references are logical addresses
- A **relative address** is an example of logical address in which the address is expressed as a location relative to some known point in the program (Ex: the beginning)

Address Translation

- Relative address is the most frequent type of logical address used in pgm modules (ie: executable files)
- Such modules are loaded in main memory with all memory references in relative form
- Physical addresses are calculated
- For **adequate performance**, the translation from relative to physical address must be done by hardware

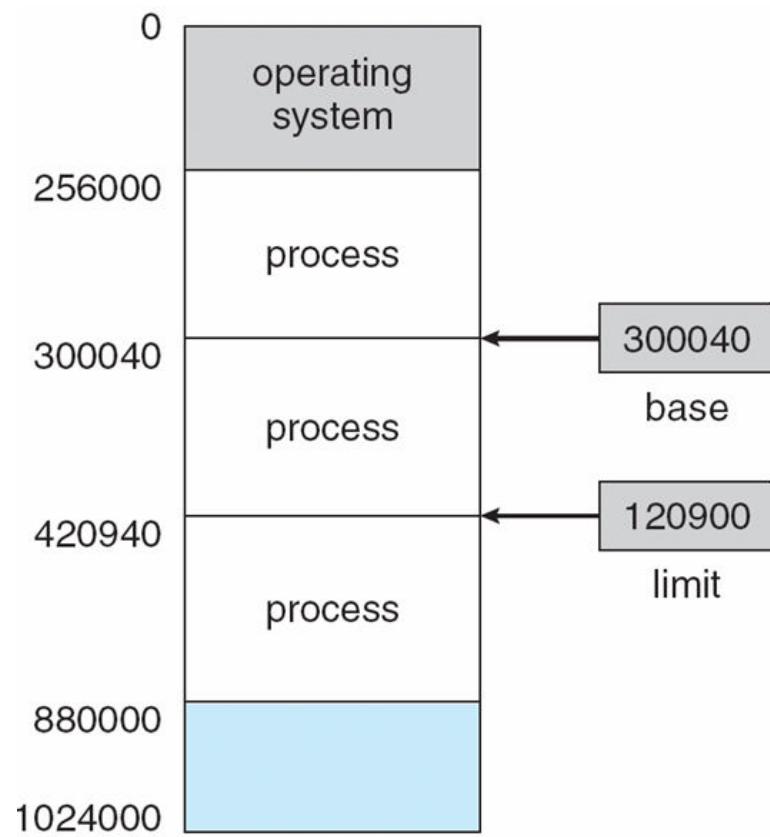
Logical address space concept

- We need logical address space concept, that is different than the physical RAM (main memory) addresses.
- A program uses logical addresses.
- Set of logical addresses used by the program is its logical address space
 - Logical address space can be, for example, $[0, \text{max_address}]$
- Logical address space has to be mapped somewhere in physical memory

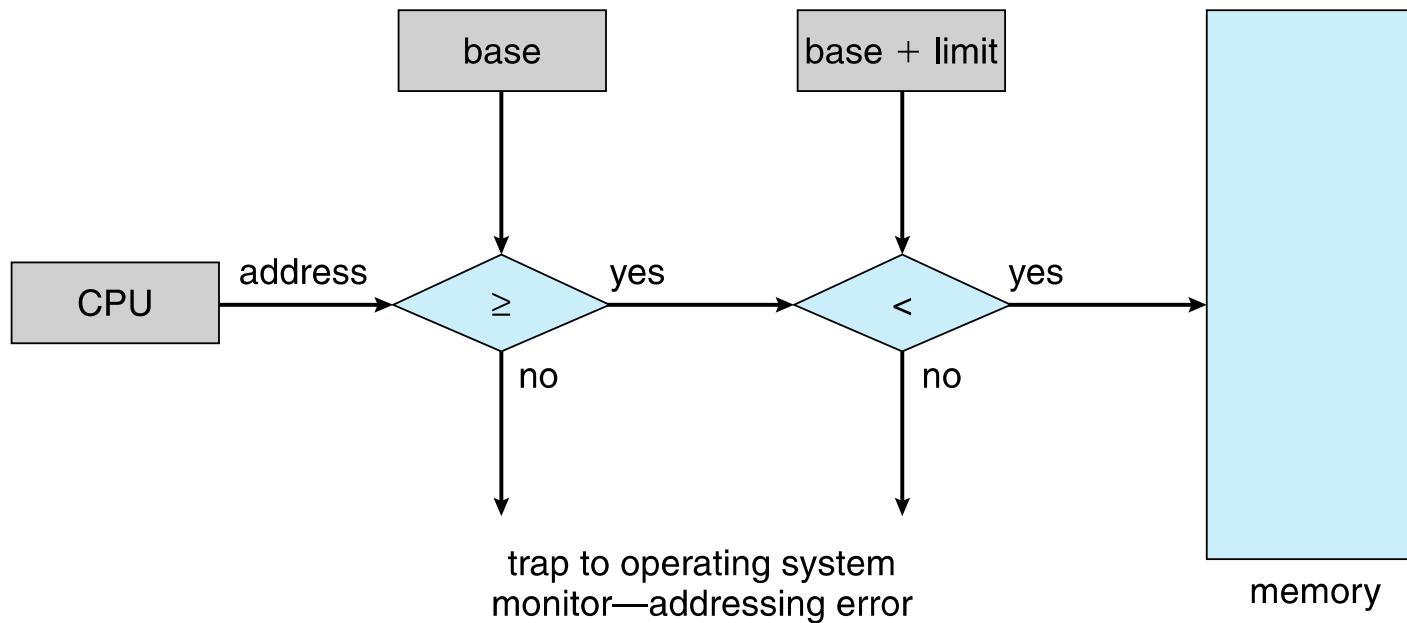


Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space
- Memory management provides protection by using two registers, a base register and a limit register.
- The base register holds the smallest legal physical memory address and the limit register specifies the size of the range.
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

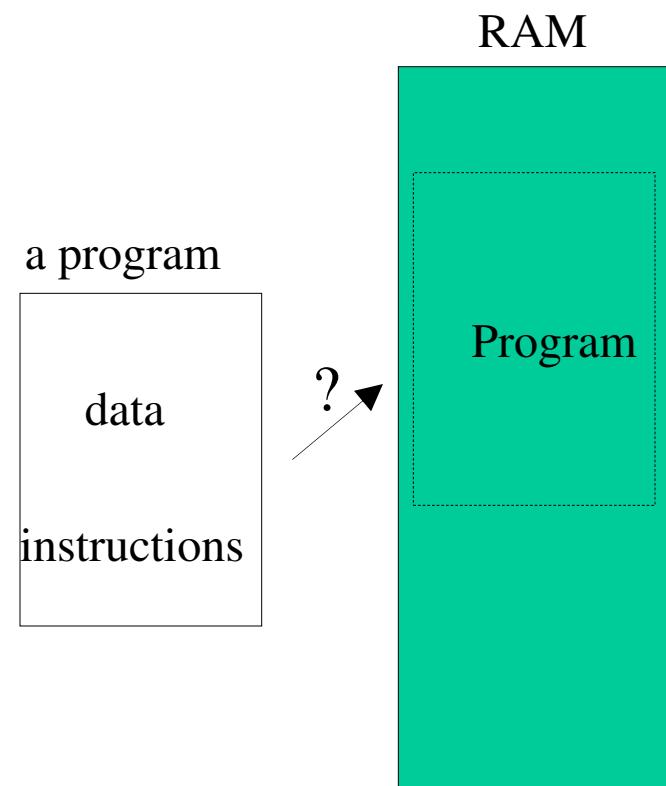


Hardware Address Protection with Base and Limit Registers

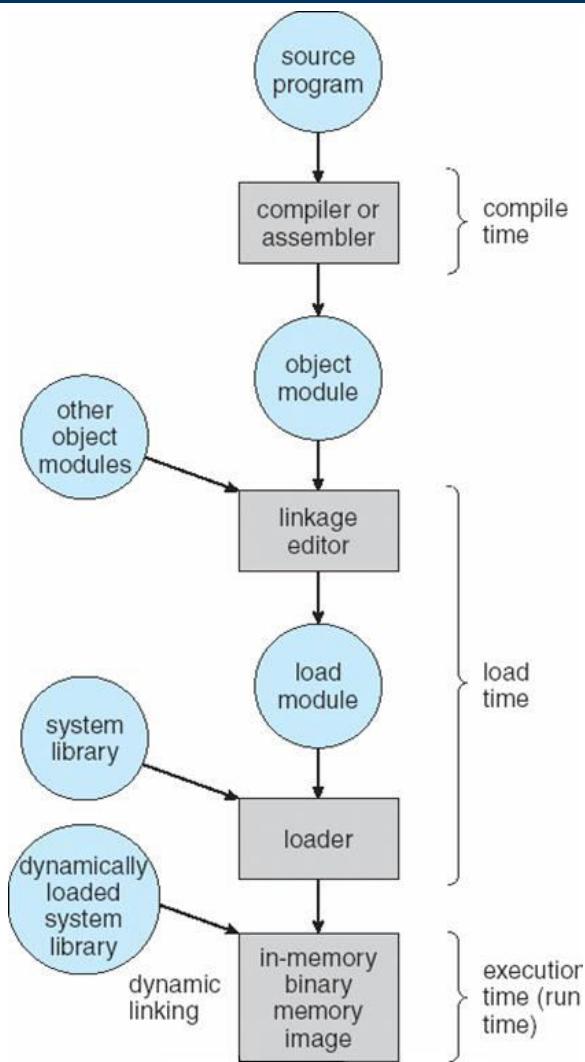


Binding of Instructions and Data to Memory

- Address binding of instructions and data to (physical) memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., **base** and **limit registers**)



Multistep Processing of a User Program



Addresses may be represented in different ways during these steps

Logical vs. Physical Address Space

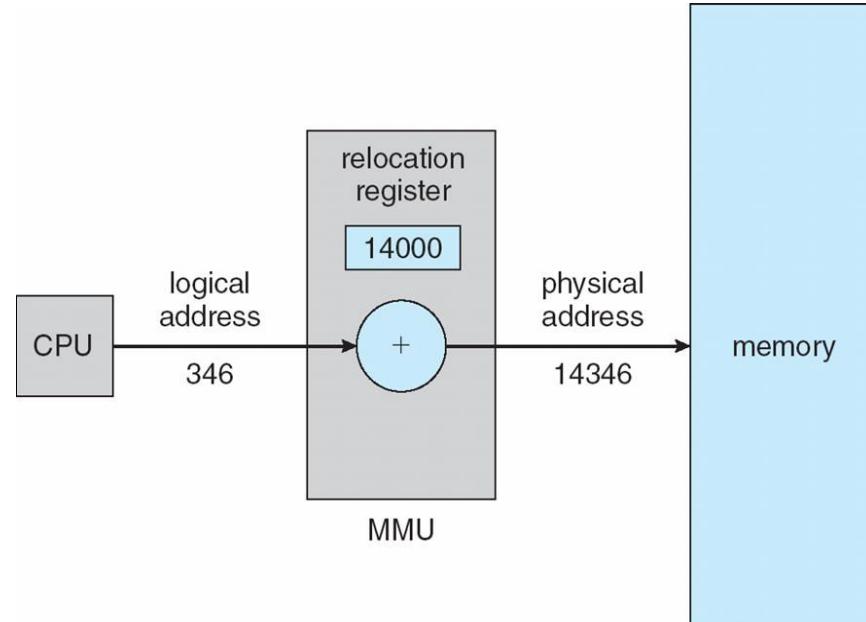
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base register now called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses

Dynamic relocation using a relocation register

- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading



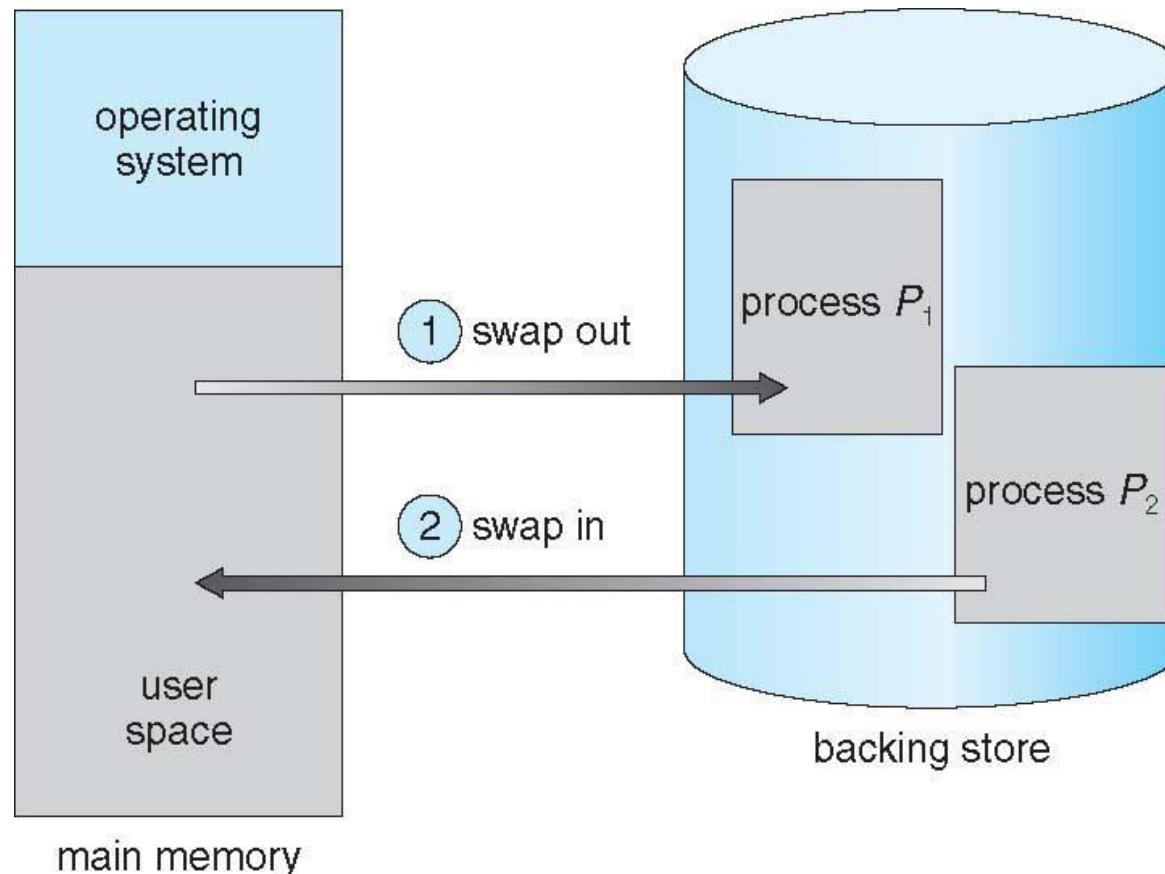
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
 - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Swapping

- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
 - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - Swapping normally disabled
 - Started if more than threshold amount of memory allocated
 - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



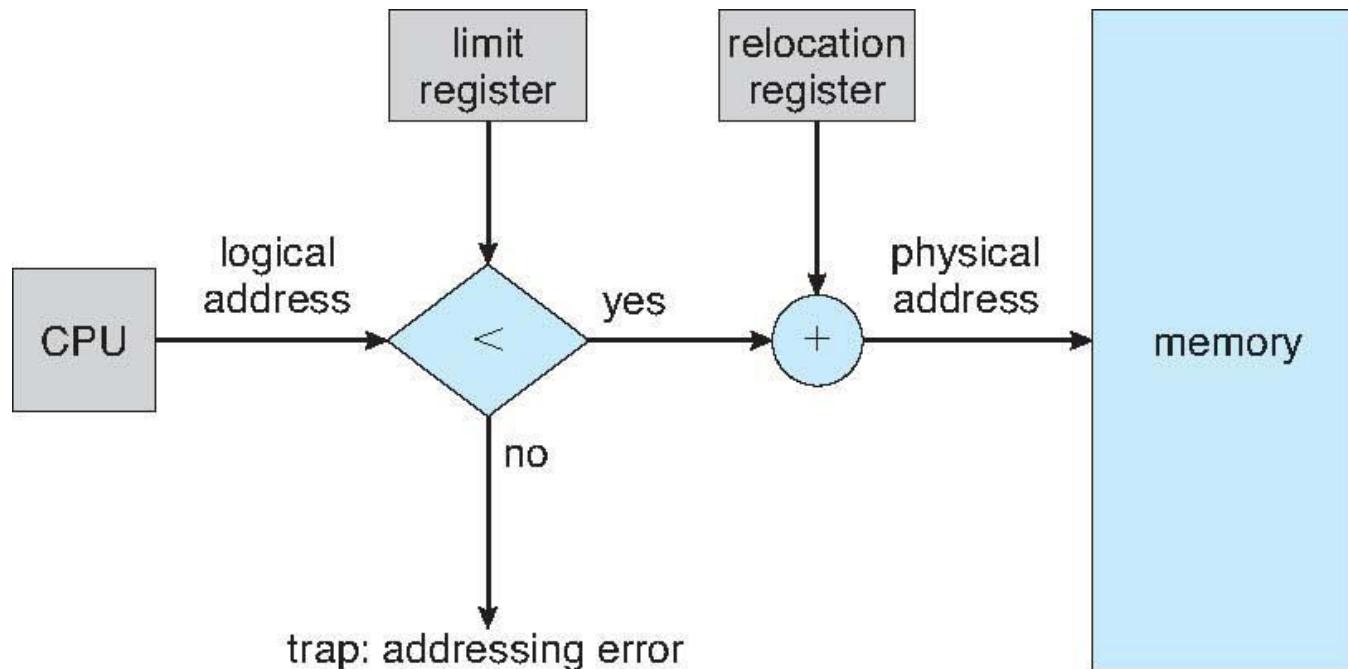
Contiguous Allocation

- Main memory is partitioned usually into two partitions:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - **Base register** contains value of smallest physical address
 - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
 - **MMU** maps logical addresses *dynamically*

Basic Memory Allocation Strategies

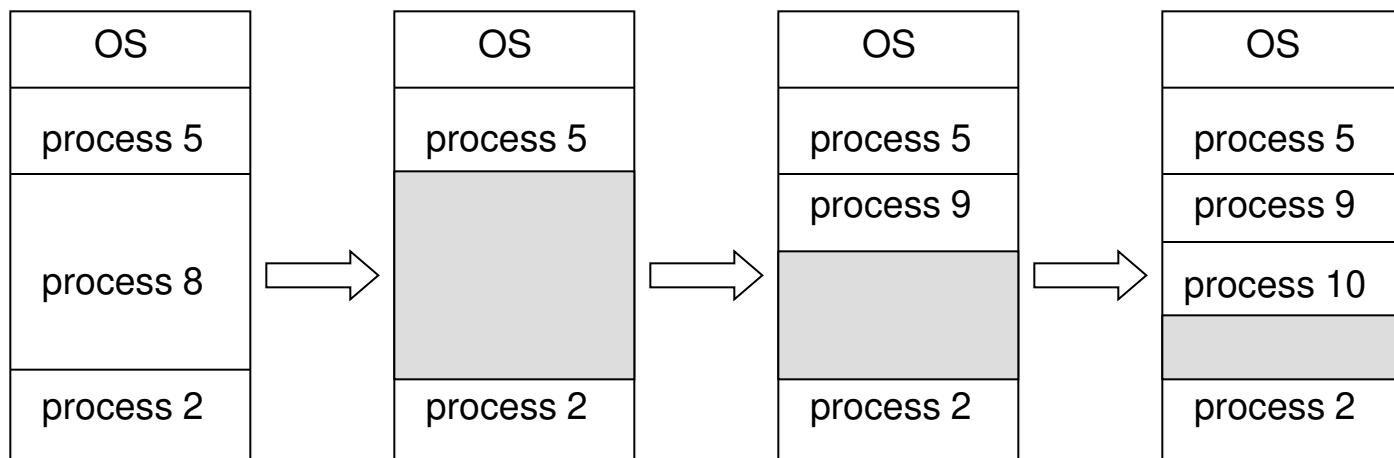
- 3 basic main memory allocation strategies to processes
 - 1) Contiguous allocation
 - 2) Paging
 - 3) Segmentation

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont)

- Multiple-partition allocation
 - Hole – block of available memory; holes of various size are scattered throughout memory
 - When a process arrives, it is allocated memory from a hole large enough to accommodate it
 - Operating system maintains information about:
 - a) allocated partitions
 - b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes

First-fit: Allocate the *first* hole that is big enough •

Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size •

Produces the smallest leftover hole –

Worst-fit: Allocate the *largest* hole; must also search entire list •

Produces the largest leftover hole –

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

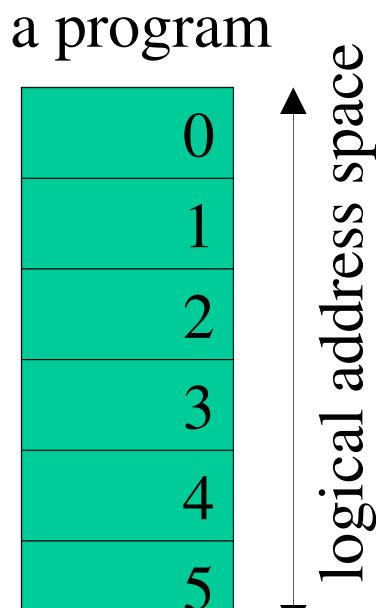
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition (allocation), but not being used
- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is **dynamic**, and is done at **execution time**
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Physical address space will also be **noncontiguous**.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size n pages, need to find n free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

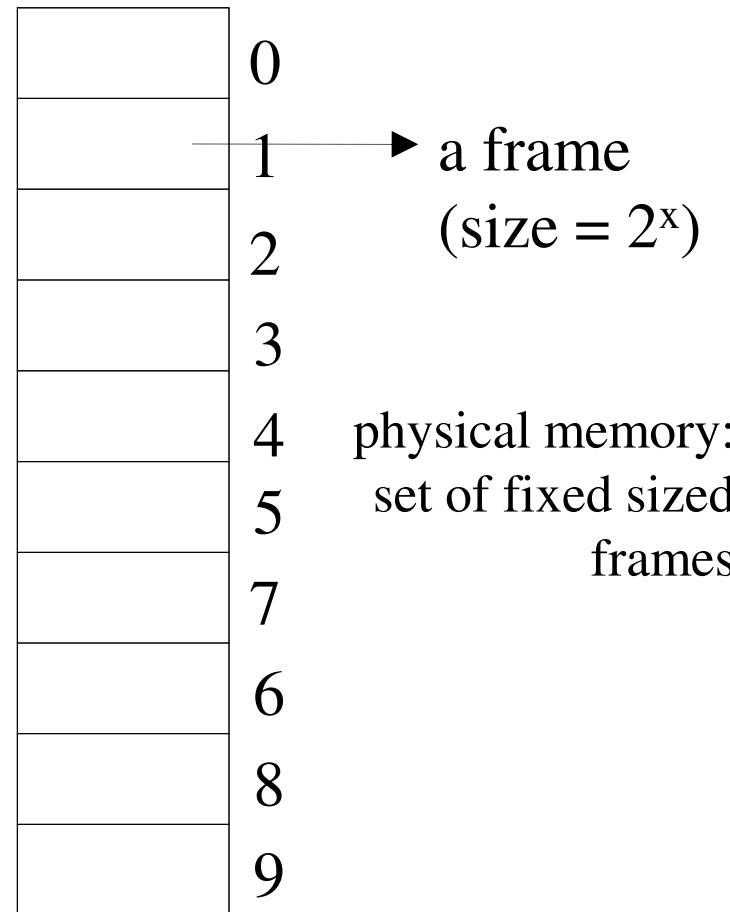
Paging



program: set of pages

Page size = Frame size

RAM (Physical Memory)



Paging

a program

0
1
2
3
4
5

load

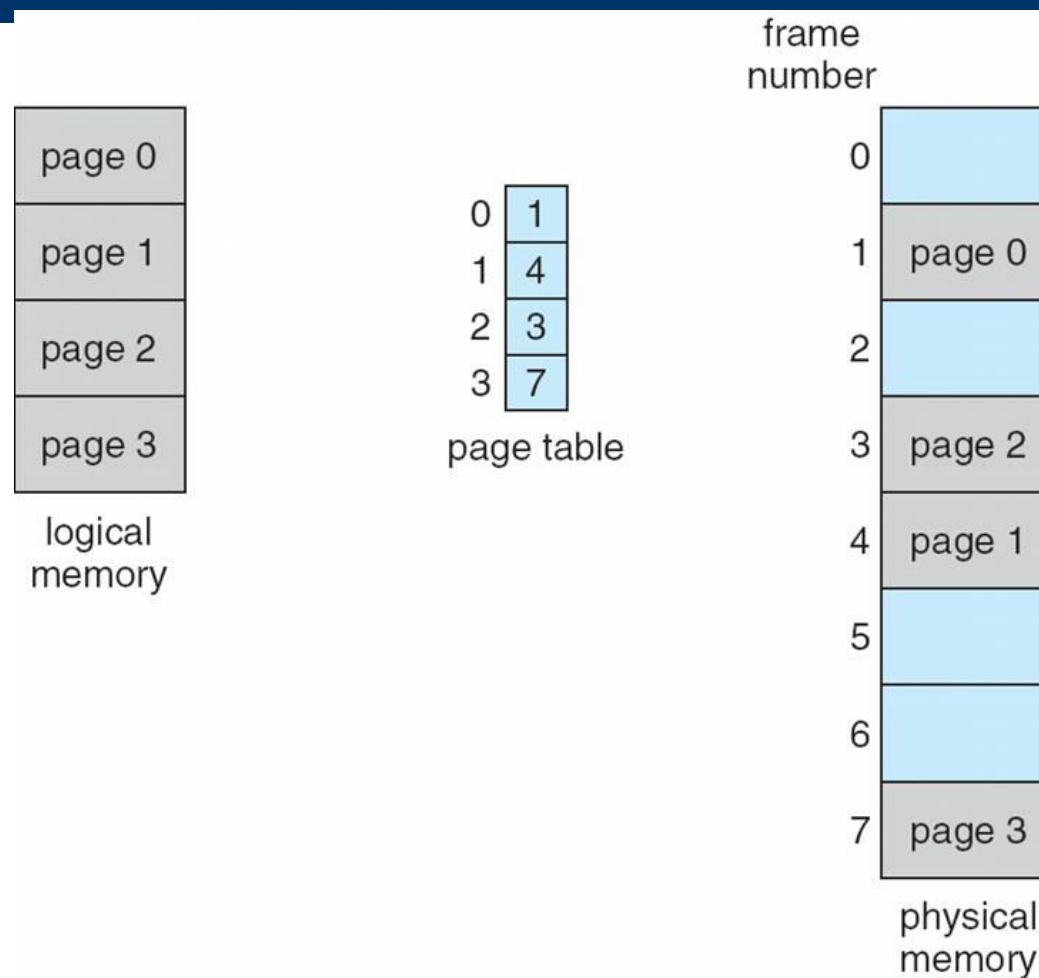
0	mapped_to 1
1	mapped_to 4
2	mapped_to 2
3	mapped_to 7
4	mapped_to 9
5	mapped_to 6

page table

RAM

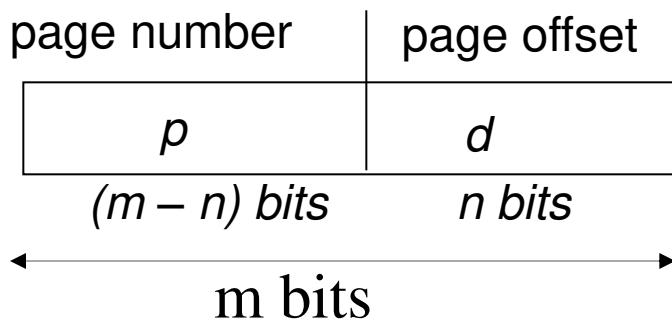
0
1
2
3
4
5
6
7
8
9

Example



Address Translation Scheme

- Assume Logical Addresses are m bits. Then logical address space is 2^m bytes.
- Assume page size is 2^n bytes.
- Logical Address generated by CPU is divided into:
 - **Page number (p)** – used as an index into a *page table* which contains base address of each page in physical memory
 - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



Simple example

Assume m is 3 and n is 2

Logical addresses

000

001

010

011

100

101

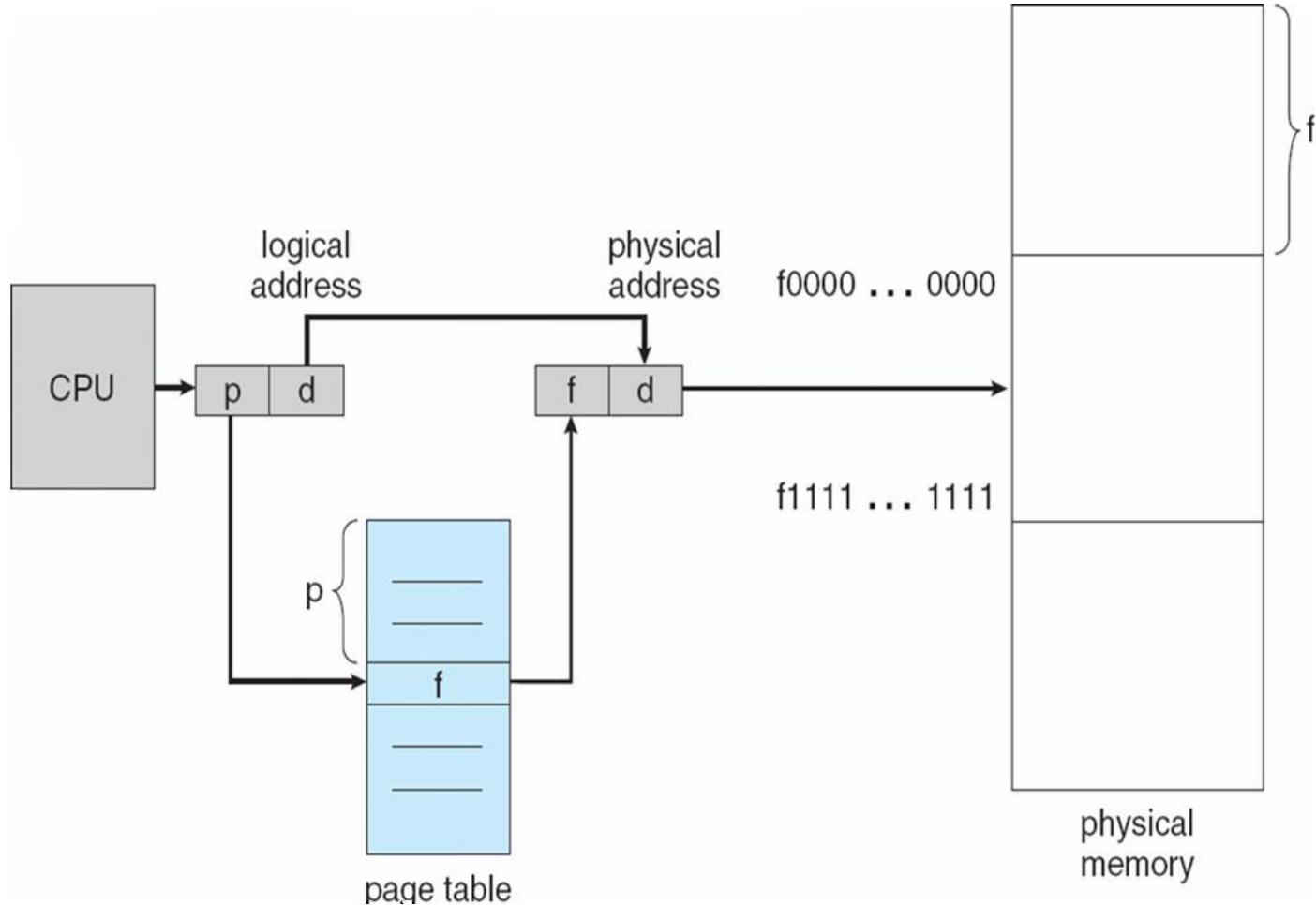
110

111



000	↑
001	page0
010	↓
011	↑
100	page1
101	↓
110	
111	

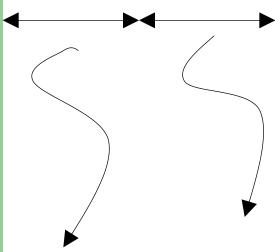
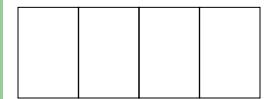
Paging Hardware: address translation



Paging Example

page size = 4 bytes
 $= 2^2$

4 bit logical address



page
number

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

32 byte memory

LA = 5
PA = ?

5 is 0101
PA = 11001

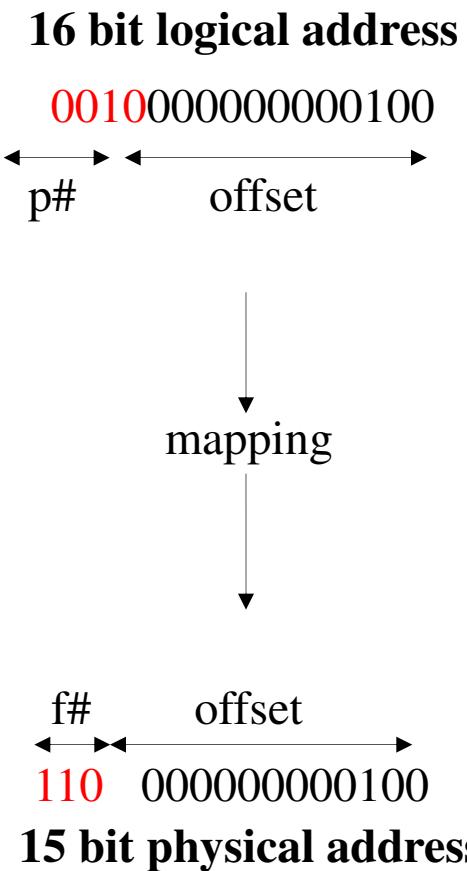
LA = 11
PA = ?

11 is 1011
PA = 00111

LA = 13
PA = ?

13 is 1101
PA = 01001

Address translation example 1



15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

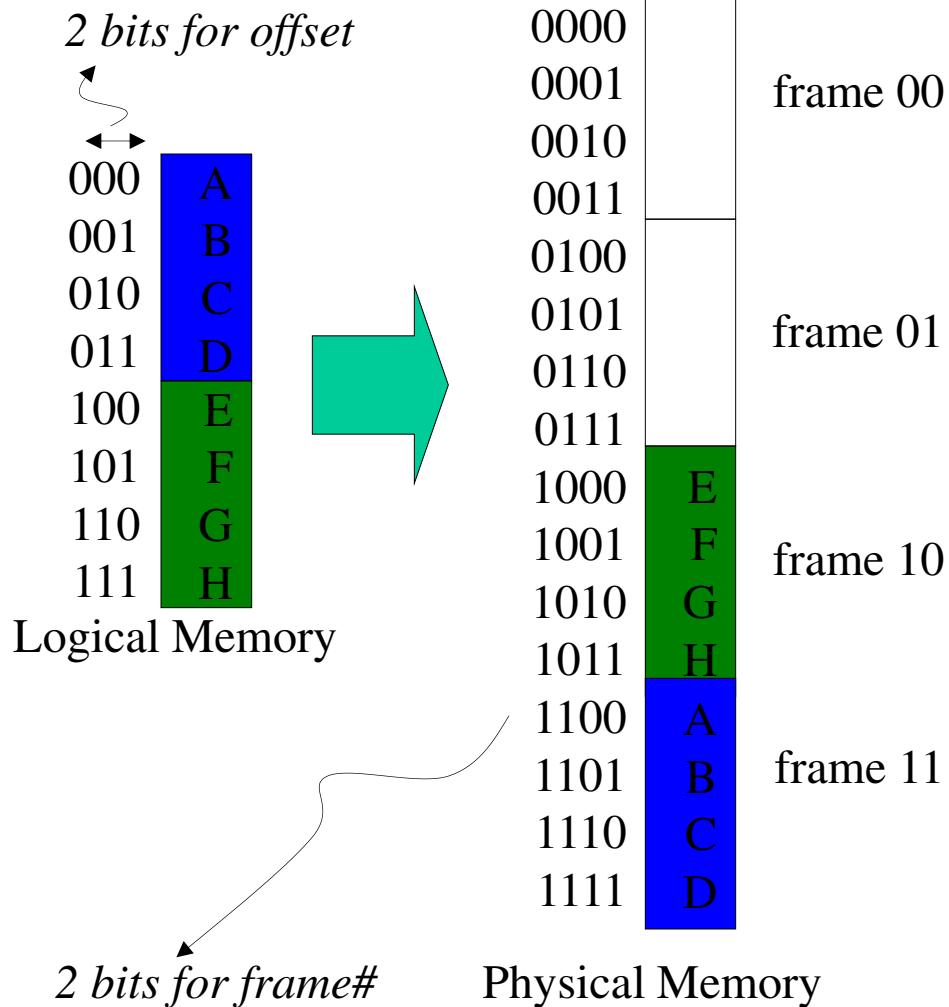
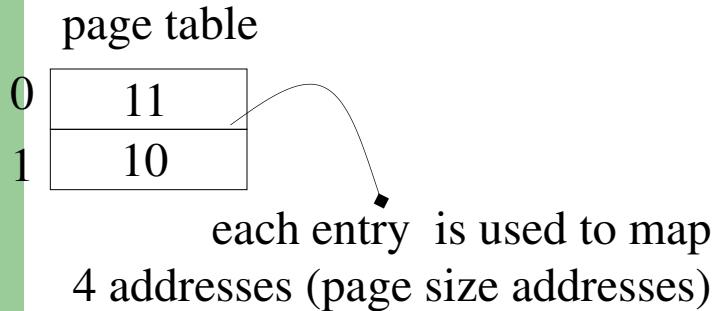
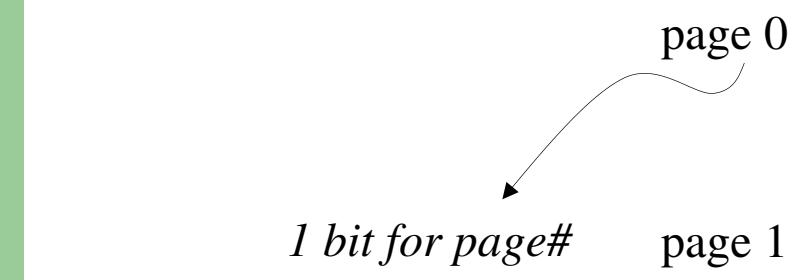
**page size = 4096 bytes
(offset is 12 bits)**

→ frame number
→ valid/invalid bit

page table

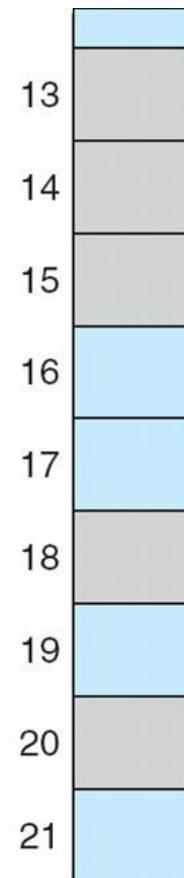
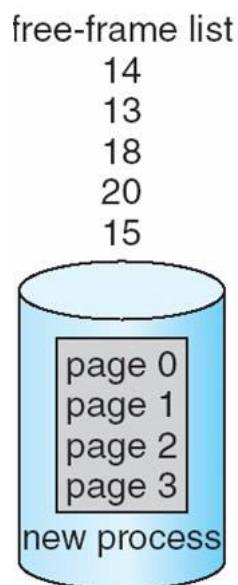
Address translation example 2

$m=3; 2^3 = 8$ logical addresses
 $n=2;$ page size $= 2^2 = 4$



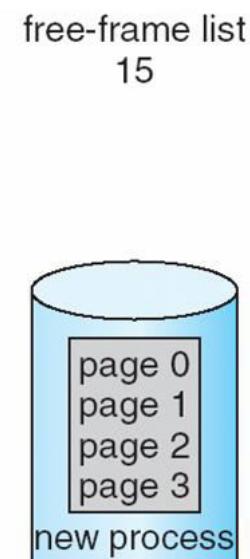
Free Frames

**OS keeps info
about the frames
in its frame table**



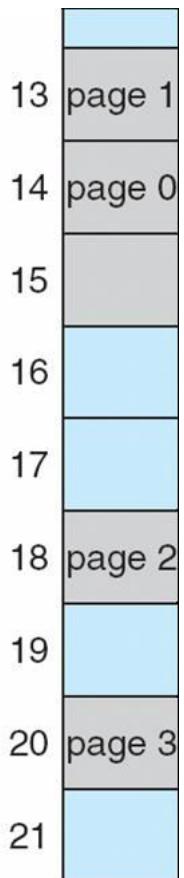
(a)

Before allocation



0	14
1	13
2	18
3	20

new-process page table



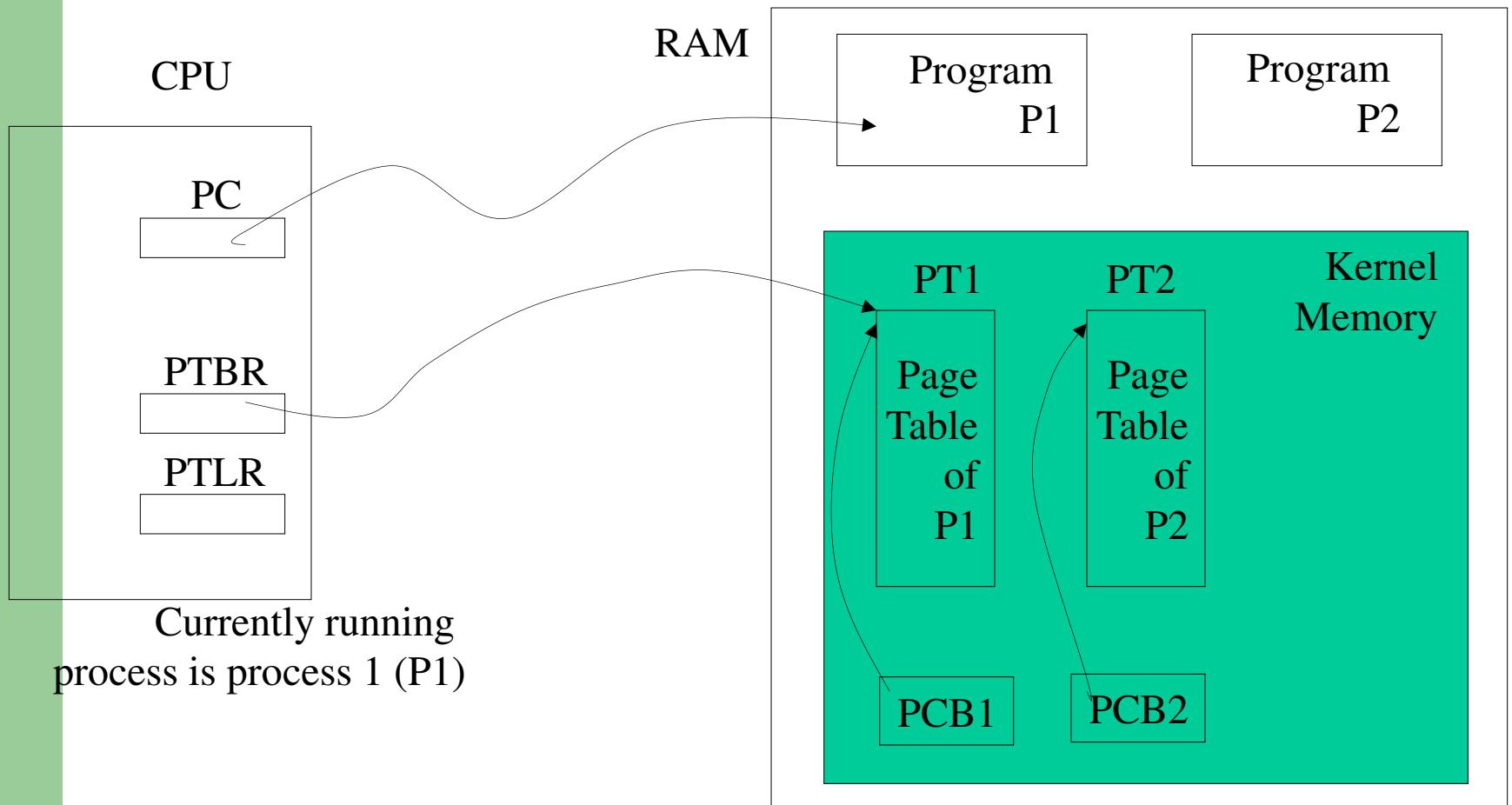
(b)

After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Implementation of Page Table



Associative Memory

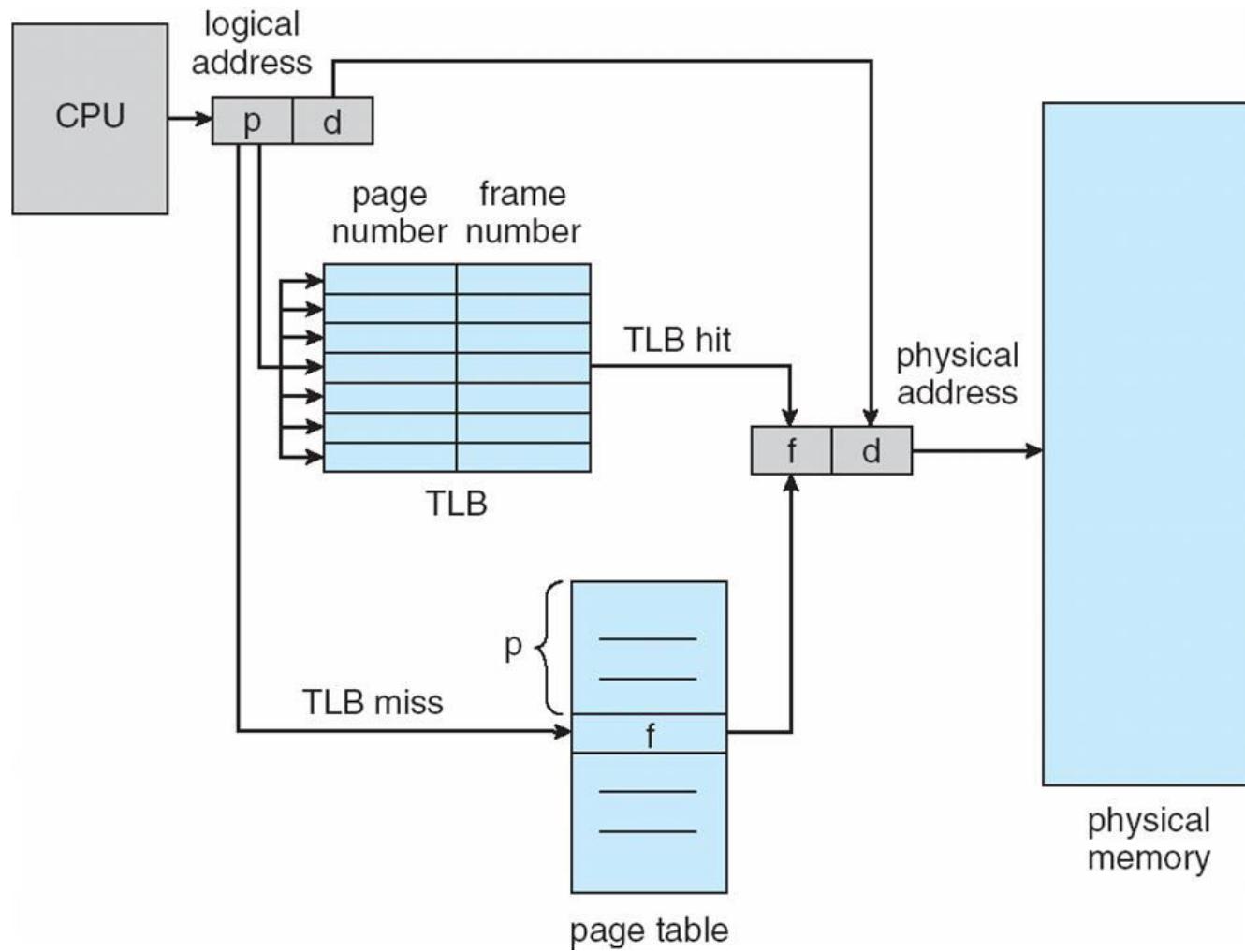
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB

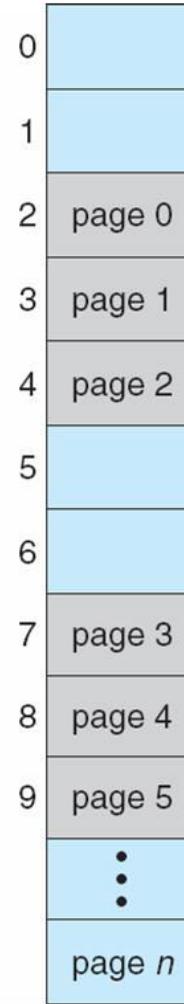
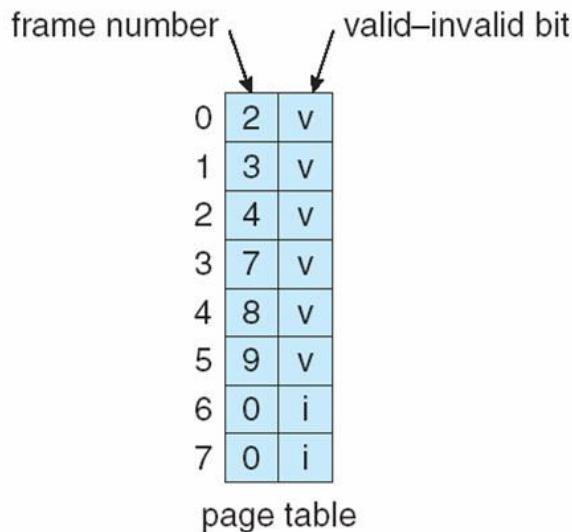


Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
 - “**valid**” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “**invalid**” indicates that the page is not in the process’ logical address space

Valid (v) or Invalid (i) Bit In A Page Table

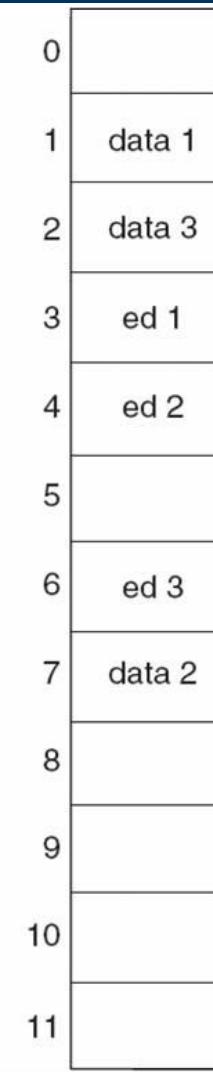
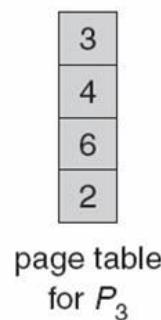
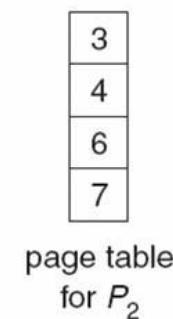
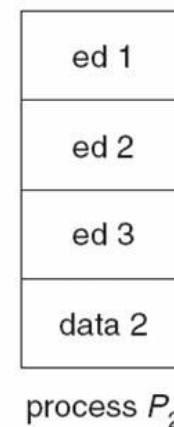
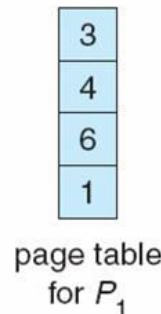
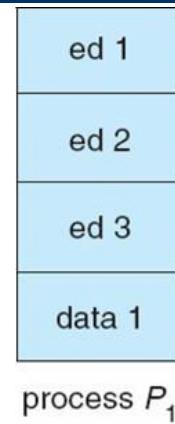
00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	



Shared Pages

- **Shared code**
 - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
 - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
 - Each process keeps a separate copy of the code and data
 - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



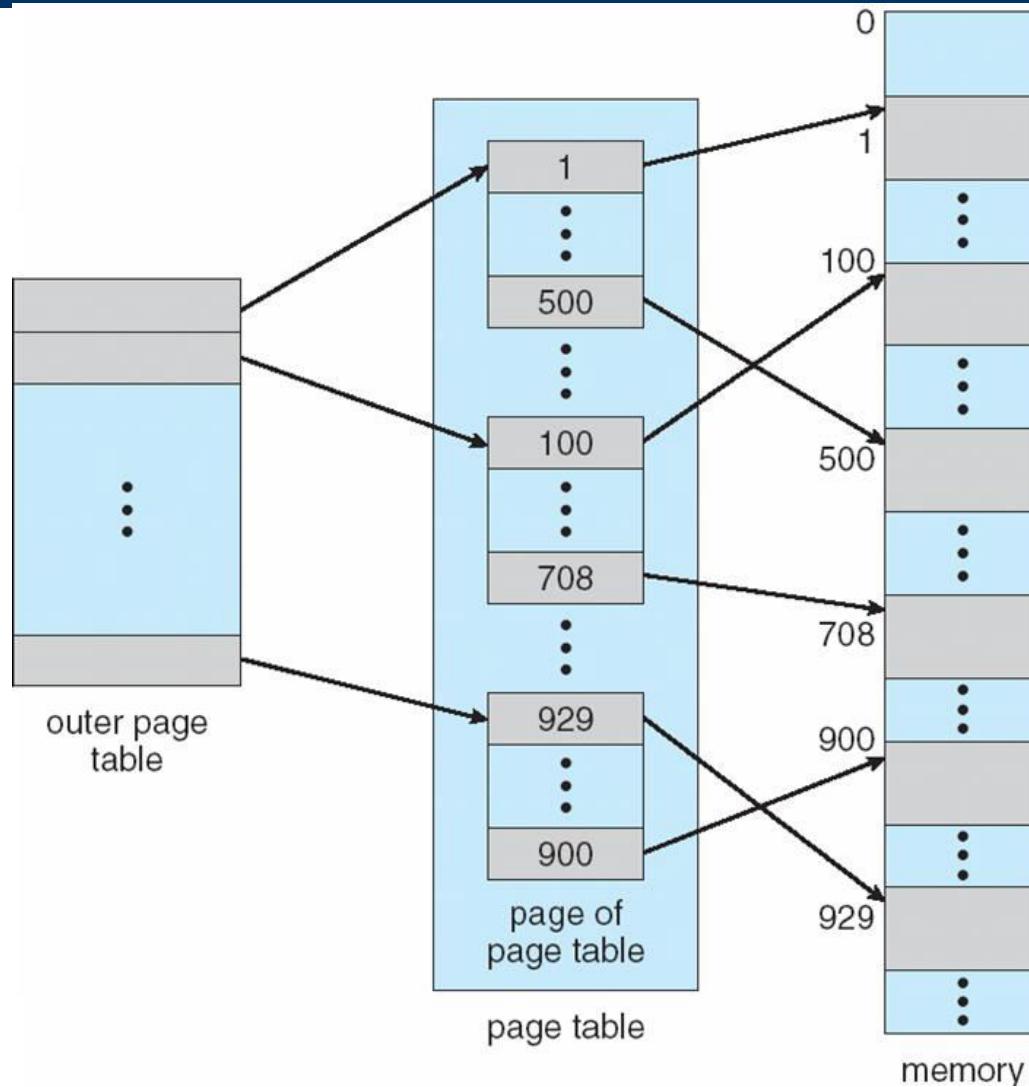
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

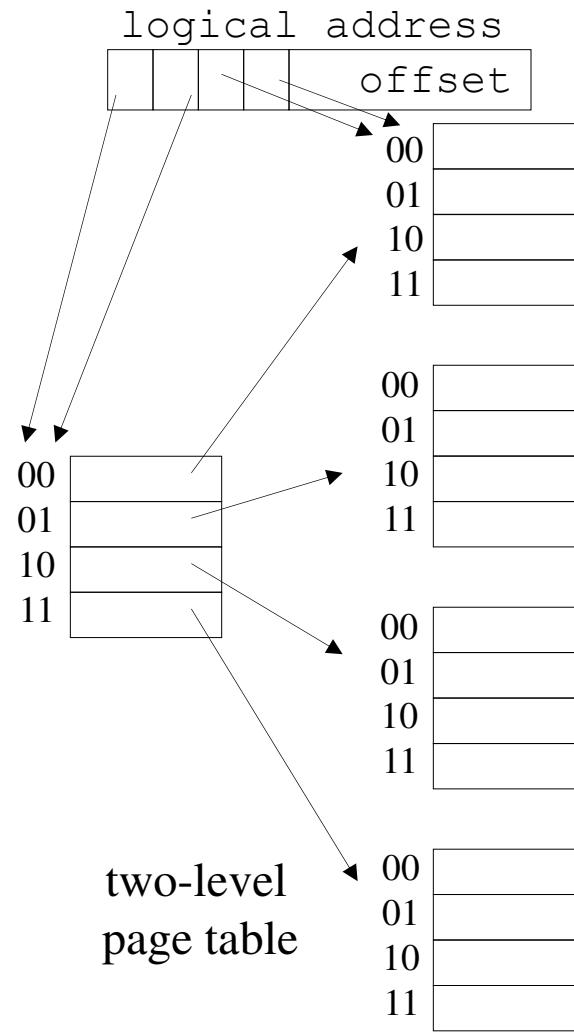
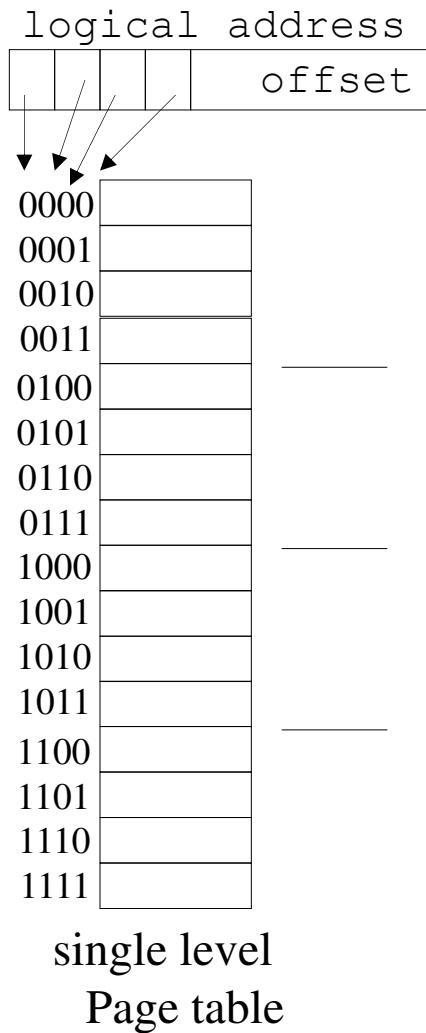
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

Two-Level Page-Table Scheme

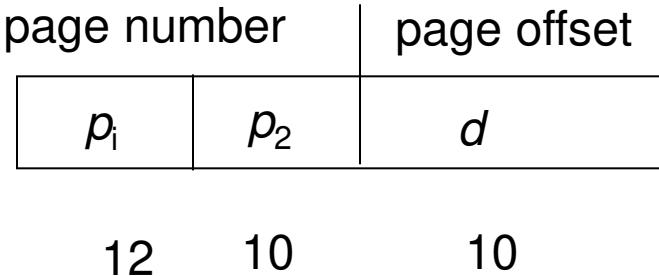


Two-Level Paging Scheme



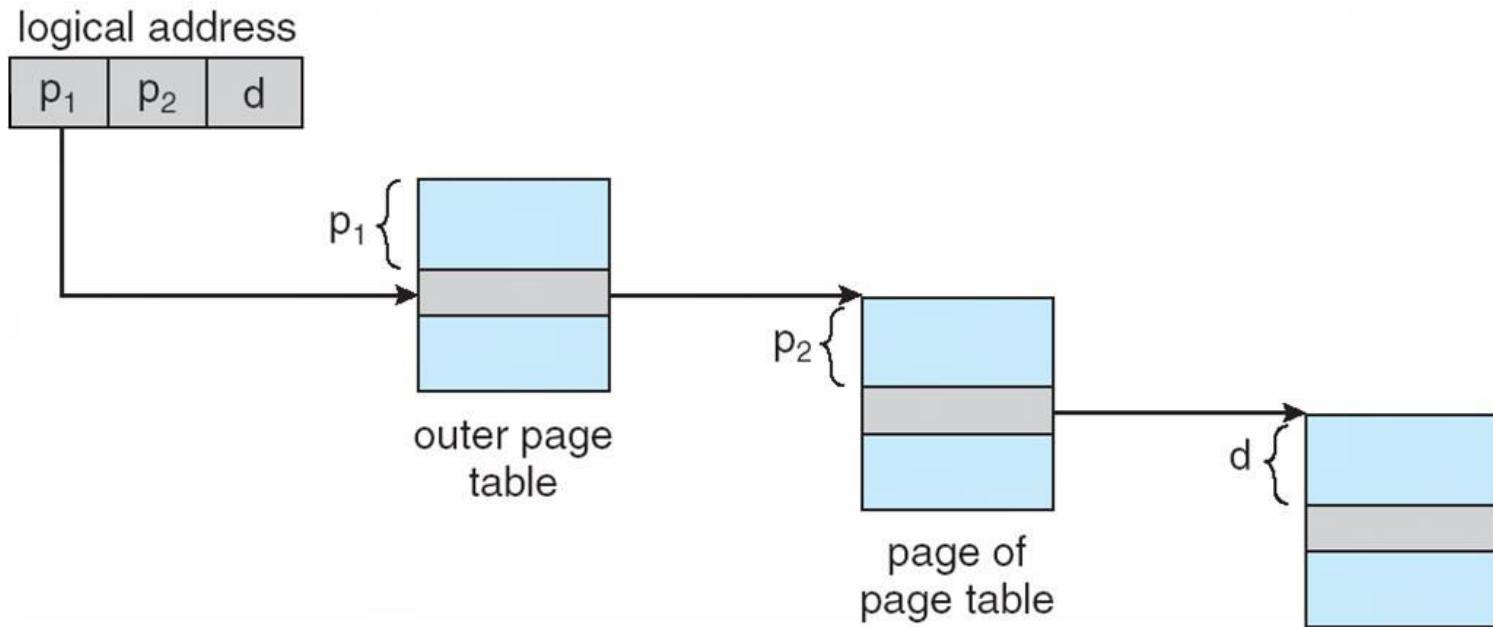
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
 - a page number consisting of 22 bits
 - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
 - a 12-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme



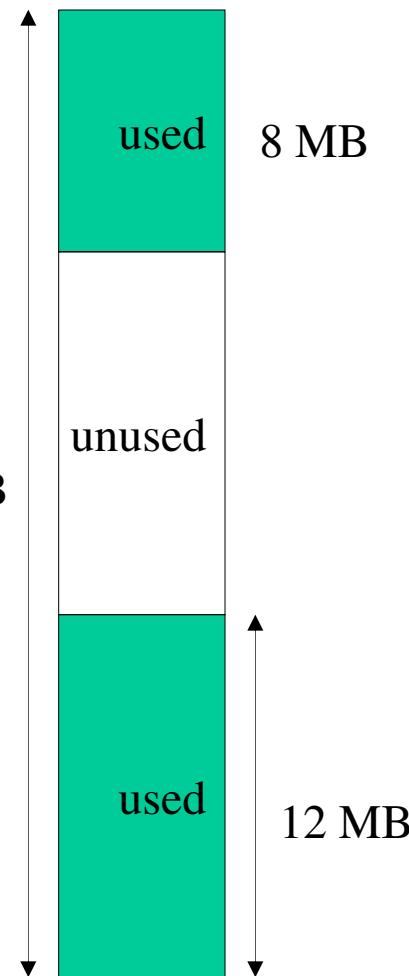
Example: two level page table need

logical address length = 32 bits

pagesize = 4096 bytes

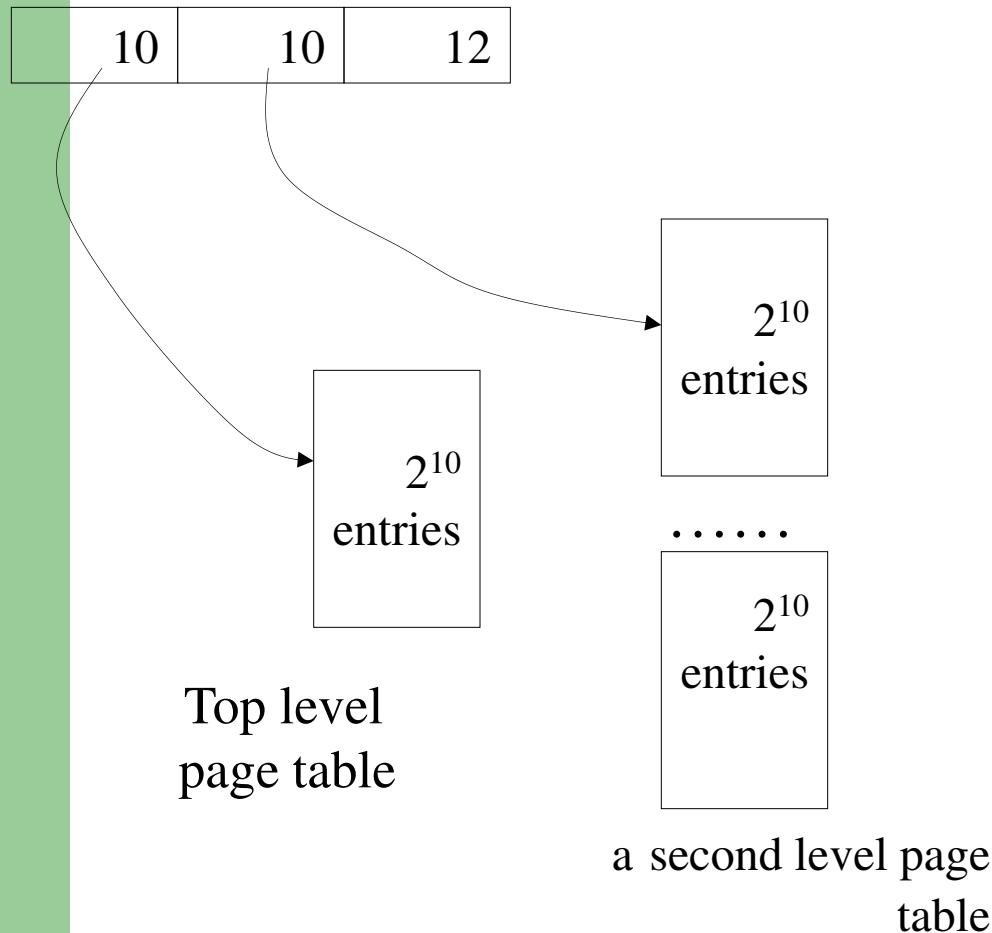
logical address division: 10, 10, 12

2^{32} bytes = 4 GB
logical address space size
(only partially used)



What is total size of two
level page
table if entry size
is 4 bytes?

Example: two level page table need

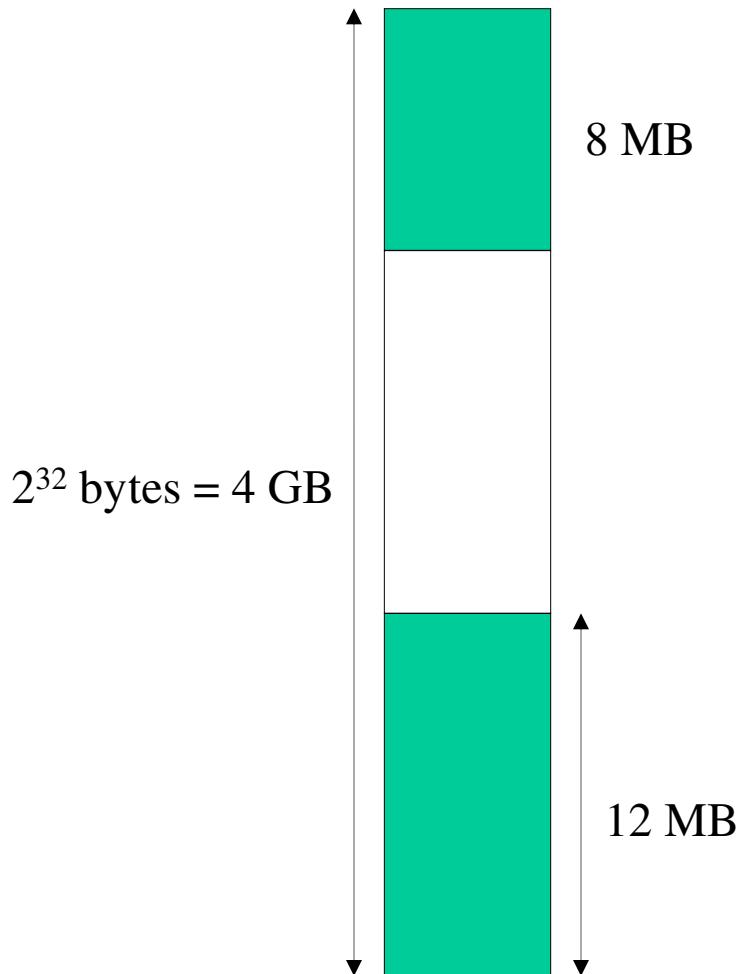


Each entry of a second level page table translates a page# to a frame#; i.e. each entry maps a page which is 4096 bytes

There are 1024 entries in a second level page table

Hence, a second level page table can map $2^{10} * 2^{12} = 2^{22} = 4 \text{ MB}$ of logical address space

Example: two level page table need

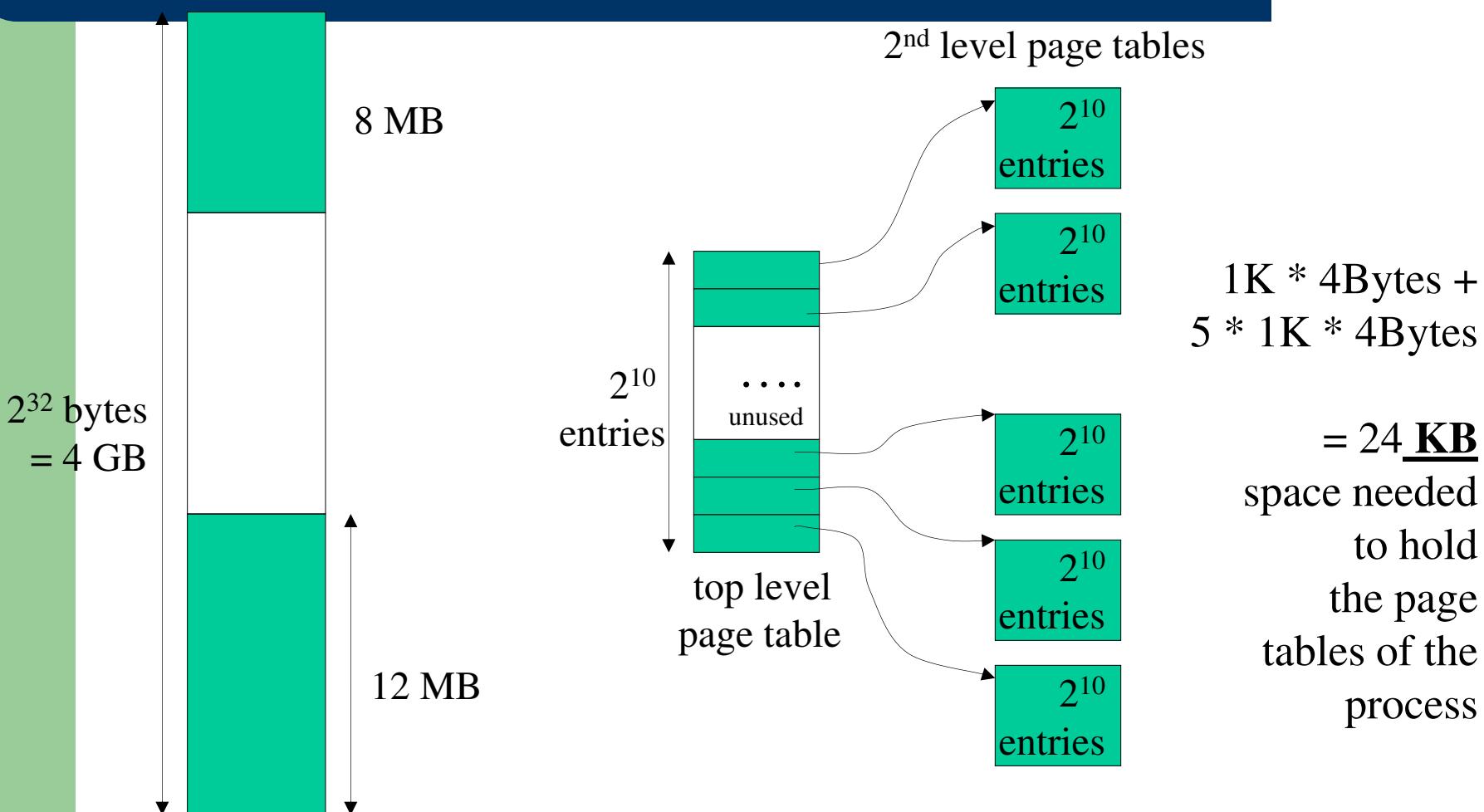


$8 \text{ MB} / 4 \text{ MB} = 2$ second level page tables required to map 8 MB of logical memory

Total = 3 + 2 = 5 second level page tables required

$12 \text{ MB} / 4 \text{ MB} = 3$ second level page tables required to map 12 MB of logical memory

Example: two level page table need



Three-level Paging Scheme

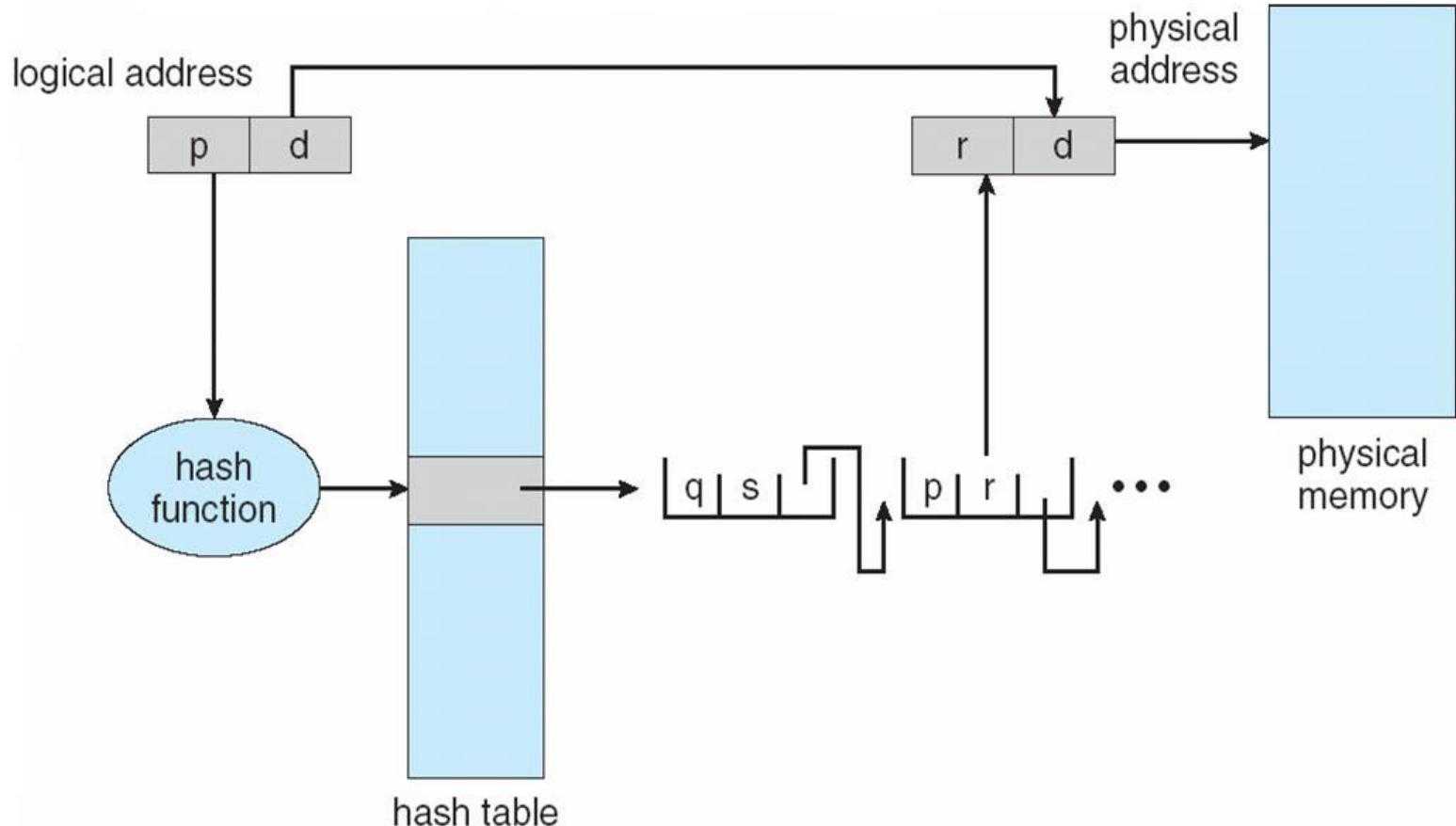
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- A page table entry contains a chain of elements hashing to the same location
 - each element = <a virtual page number, frame number>
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted

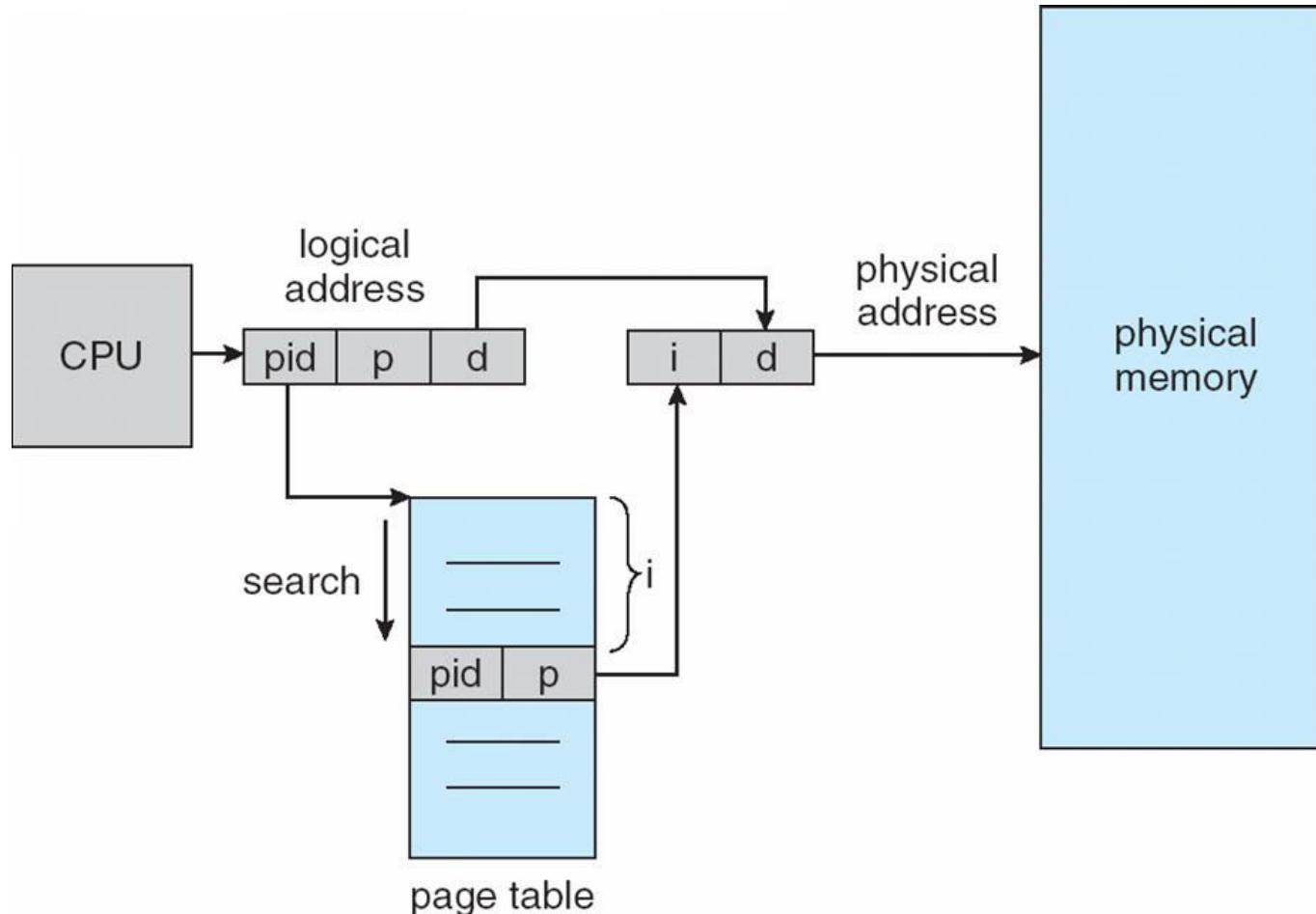
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



Simple Segmentation

- Each program is subdivided into blocks of non-equal size called **segments**
- When a process gets loaded into main memory, its different segments can be located anywhere
- Each segment is fully packed with instructions/data: no internal fragmentation
- There is external fragmentation; it is reduced when using small segments

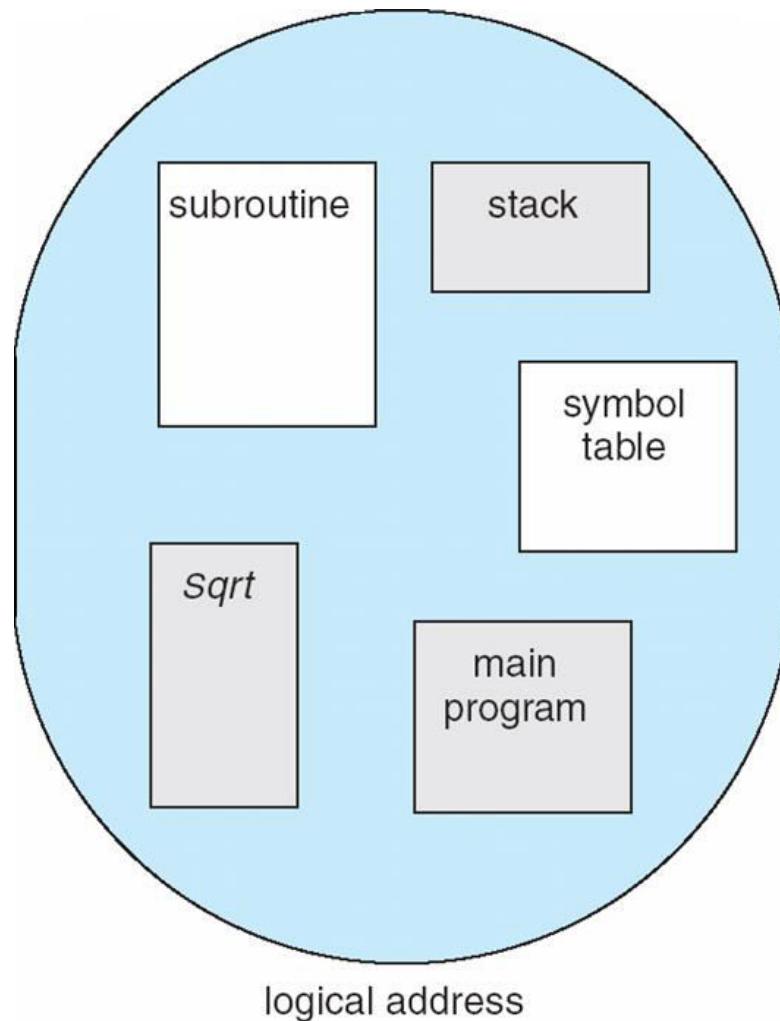
Simple Segmentation

- In contrast with paging, segmentation is visible to the programmer
 - provided as a convenience to organize logically programs (ex: data in one segment, code in another segment)
 - must be aware of segment size limit
- The OS maintains a **segment table** for each process. Each entry contains:
 - the starting physical addresses of that segment.
 - the length of that segment (for protection)

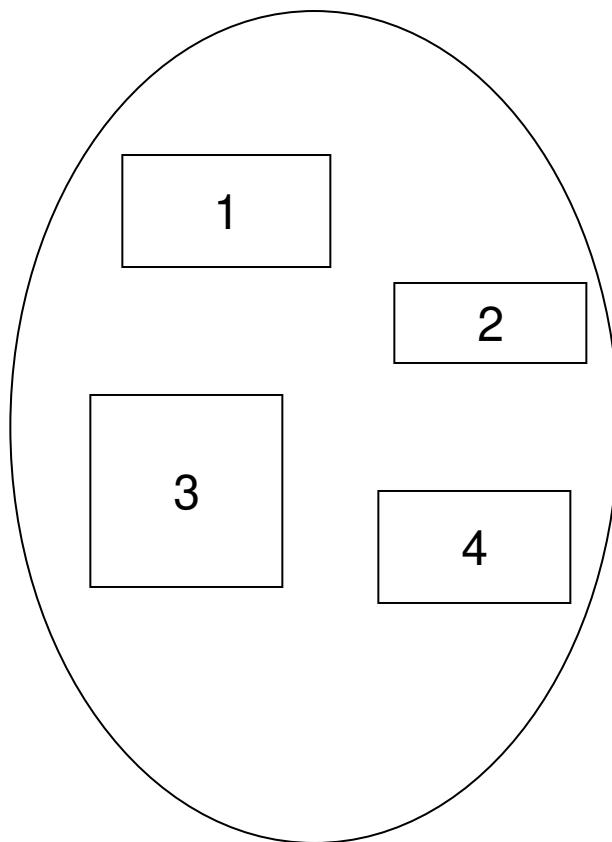
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays

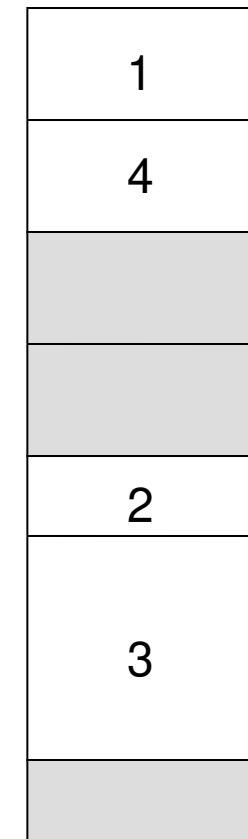
User's View of a Program



Logical View of Segmentation



user space

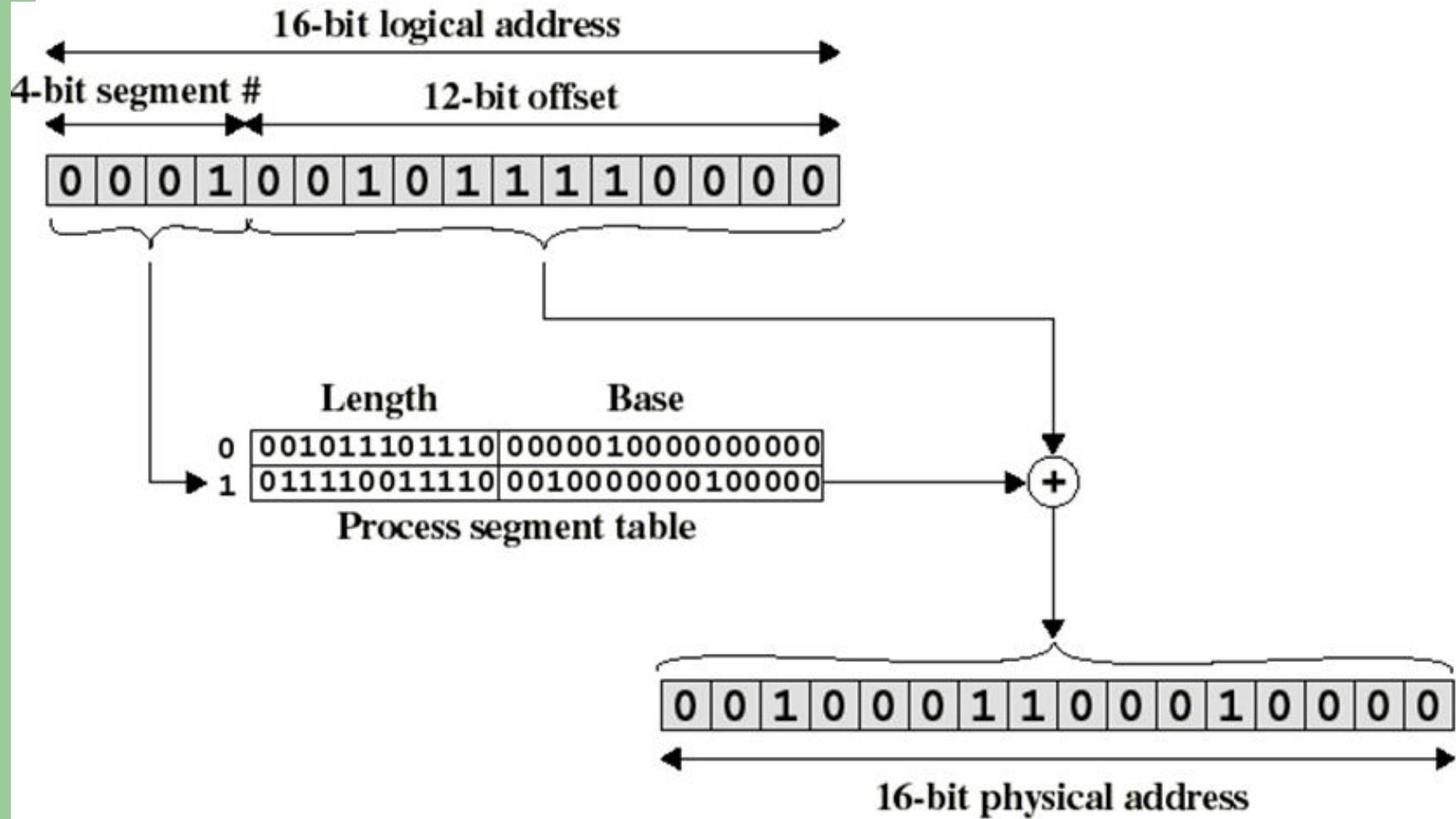


physical memory space

Logical address used in segmentation

- When a process enters the Running state, a CPU register gets loaded with the starting address of the process's segment table.
- Presented with a **logical address (segment number, offset) = (n,m)**, the CPU indexes (with n) the segment table to obtain the starting physical address k and the length l of that segment
- The physical address is obtained by **adding** m to k (in contrast with paging)
 - the hardware also compares the offset m with the length l of that segment to determine if the address is valid

Logical-to-Physical Address Translation in segmentation



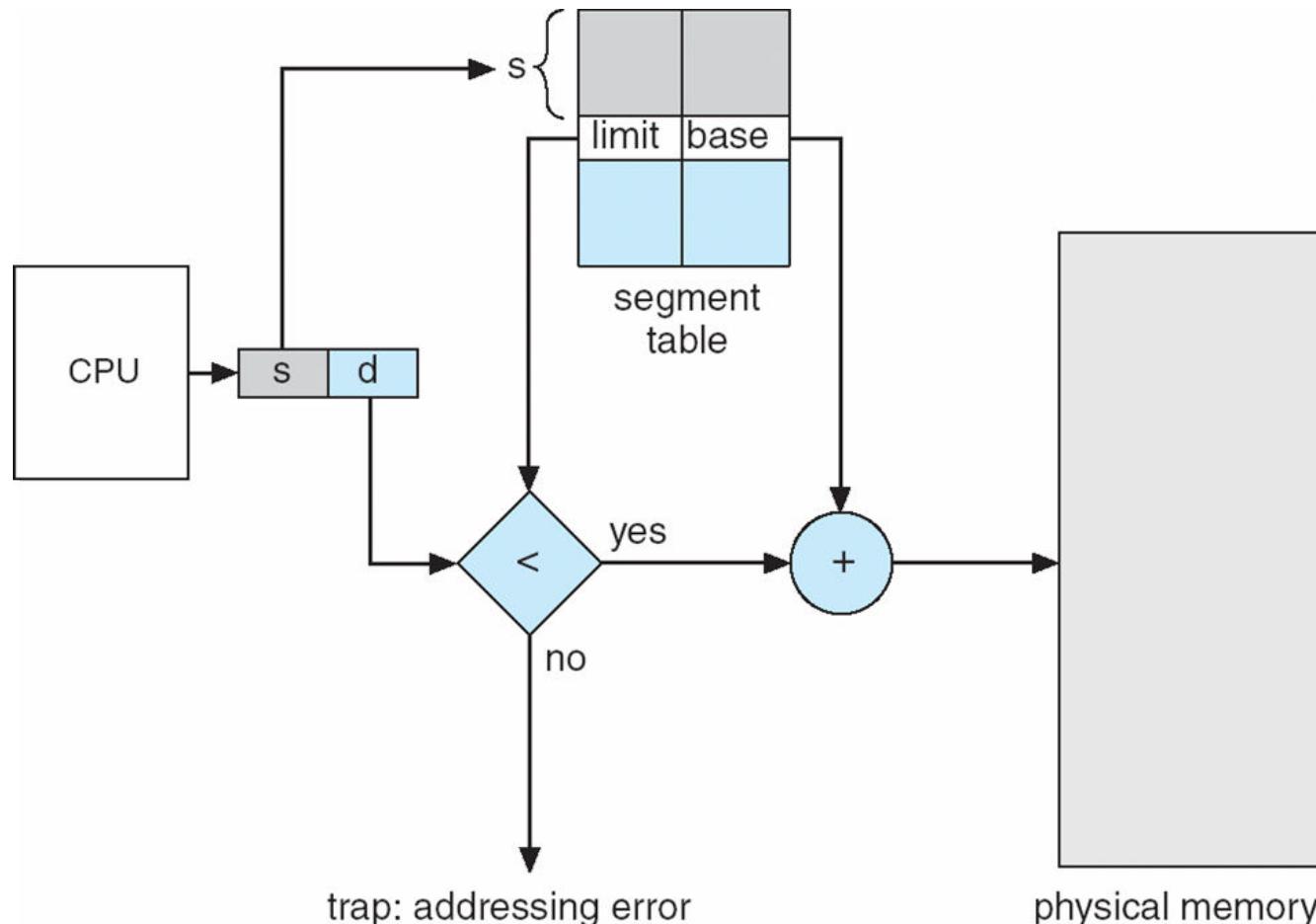
Segmentation Architecture

- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle,$
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - **base** – contains the starting physical address where the segments reside in memory
 - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
segment number **s** is legal if **s < STLR**

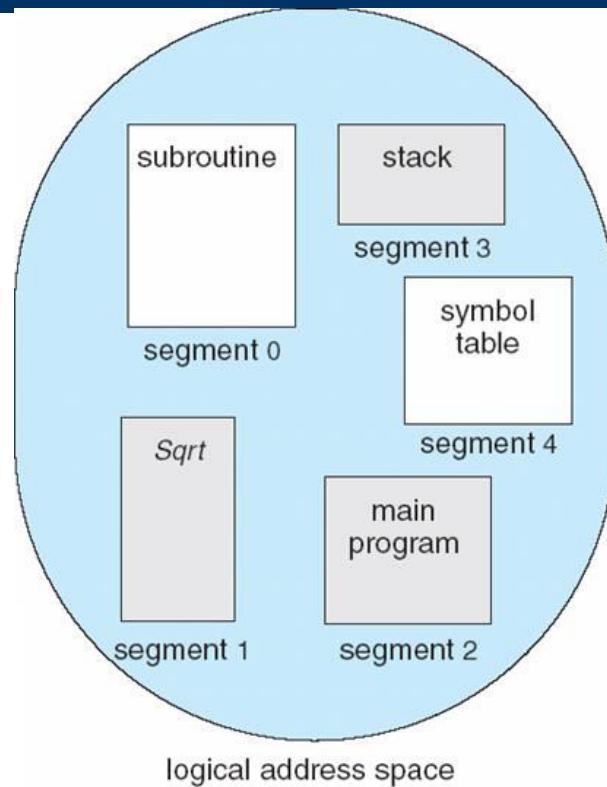
Segmentation Architecture (Cont.)

- Protection
 - With each entry in segment table associate:
 - validation bit = 0 \Rightarrow illegal segment
 - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware

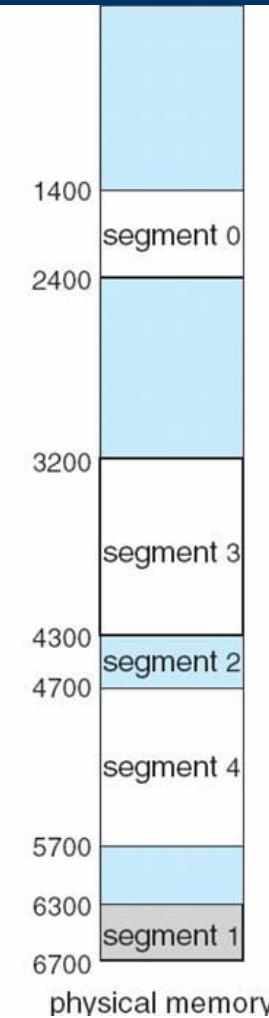


Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Simple segmentation and paging comparison

- Segmentation requires more complicated hardware for address translation
- Segmentation suffers from external fragmentation
- Paging only yield a small internal fragmentation
- Segmentation is visible to the programmer whereas paging is transparent
- Segmentation can be viewed as commodity offered to the programmer to organize logically a program into segments and using different kinds of protection (ex: execute-only for code but read-write for data)
 - for this we need to use protection bits in segment table entries

Operating Systems

Placement Algorithm

Alok Kumar Jagadev

Placement Algorithm

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

Placement Algorithm

First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K – 212K)

426K must wait

Placement Algorithm

Best-fit:

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Placement Algorithm

Worst-fit:

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, best-fit turns out to be the best.

Paging

Assuming a 1 KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

- a. 2375
- b. 19366
- c. 30000
- d. 256
- e. 16385

Paging

Assuming a 1 KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

- a. page = 2; offset = 327
- b. page = 18; offset = 934
- c. page = 29; offset = 304
- d. page = 0; offset = 256
- e. page = 1; offset = 1

Paging

Consider a logical address space of 32 pages with 1024 words per page; mapped onto a physical memory of 16 frames.

- a) How many bits are required in the logical address?

- b) How many bits are required in the physical address?

Paging

- a) $2^5 = 32 + 2^{10} = 1024 = 15$ bits.
- b) $2^4 = 32 + 2^{10} = 1024 = 14$ bits.

Operating Systems

Virtual Memory

Alok Kumar Jagadev

Background

- Virtual memory is a technique that allows the execution of processes which are not completely available in memory.
- The main visible advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory is the separation of user logical memory from physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

Background

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.

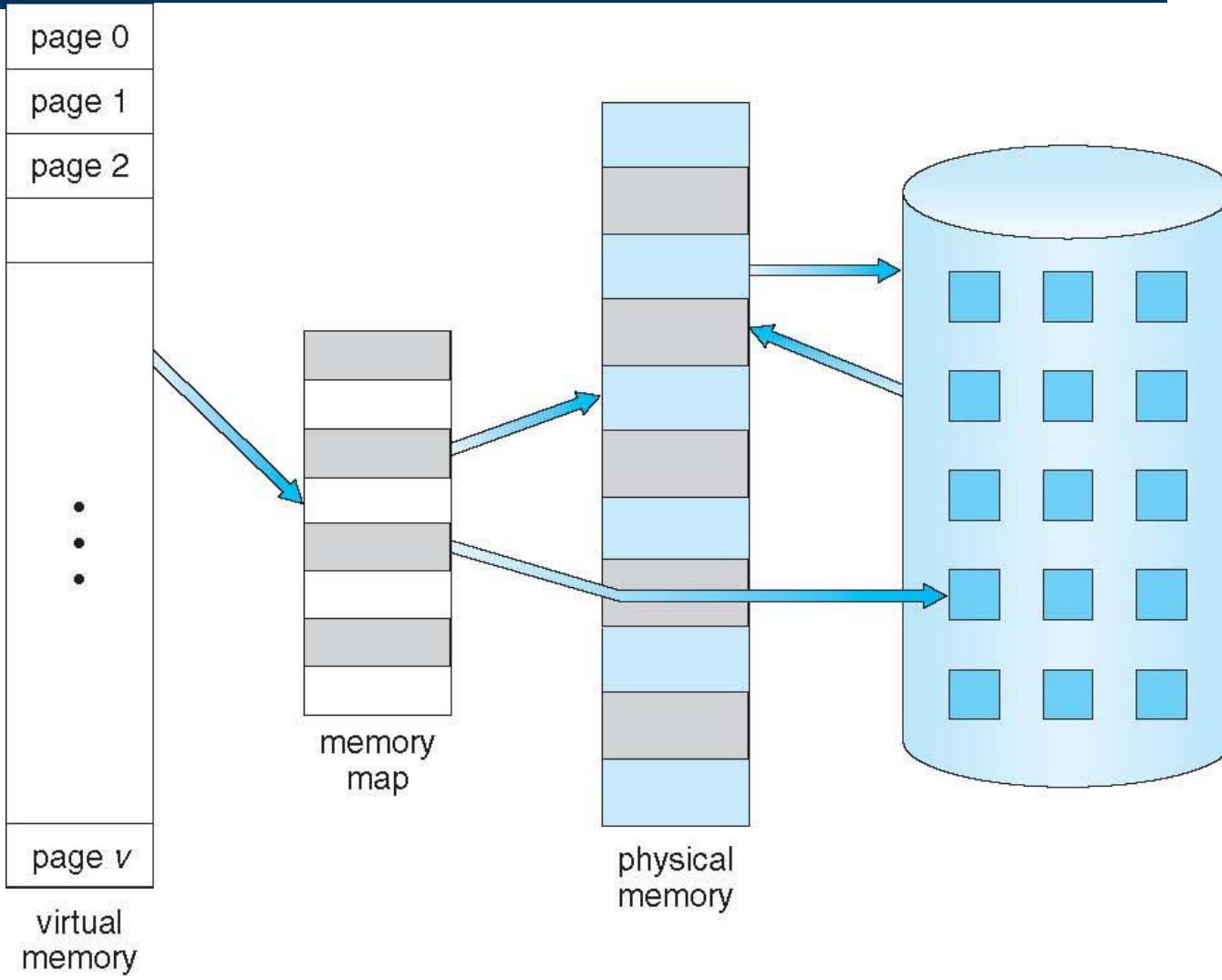
Background

- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

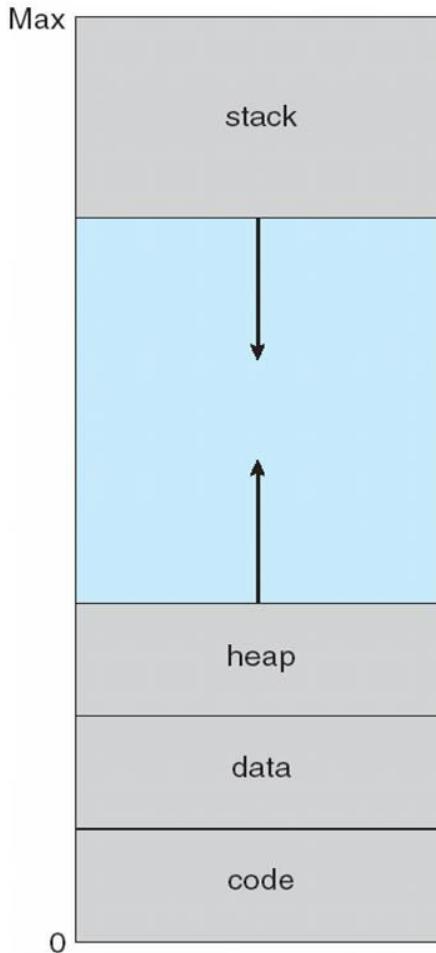
Background

- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

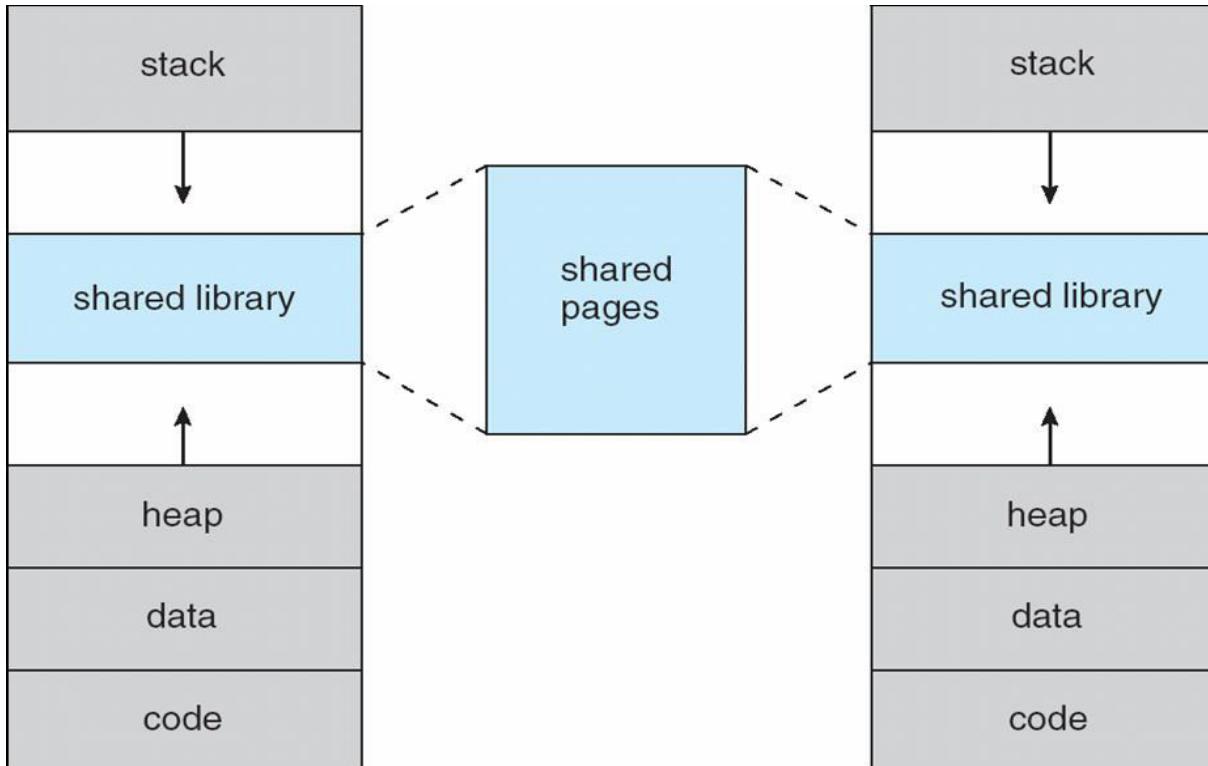


Virtual-address Space



- Refers to the logical view of how a process is stored in memory.
- A process begins at a certain logical address (such as 0) and exists in contiguous memory.
- Physical frames may not be contiguous.

Shared Library Using Virtual Memory



- Stack or heap grow if we wish to dynamically link libraries during program execution.
- System library can be shared by several process through mapping the shared object into a virtual address space.

Demand Paging

- A **demand paging** system is quite similar to a **paging** system with **swapping**.
- When a process is to be executed, swap it into memory. Rather than swapping the entire process into memory, however, a **lazy swapper** called **pager** is used.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into memory.
- Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the **swap time** and the amount of physical **memory needed**.

Demand Paging

- Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme.
- Where valid and invalid pages can be checked by checking the bit.
- Marking a page will have no effect if the process never attempts to access the page.
- While the process executes and accesses pages that are memory resident, execution proceeds normally.

Valid-Invalid Bit

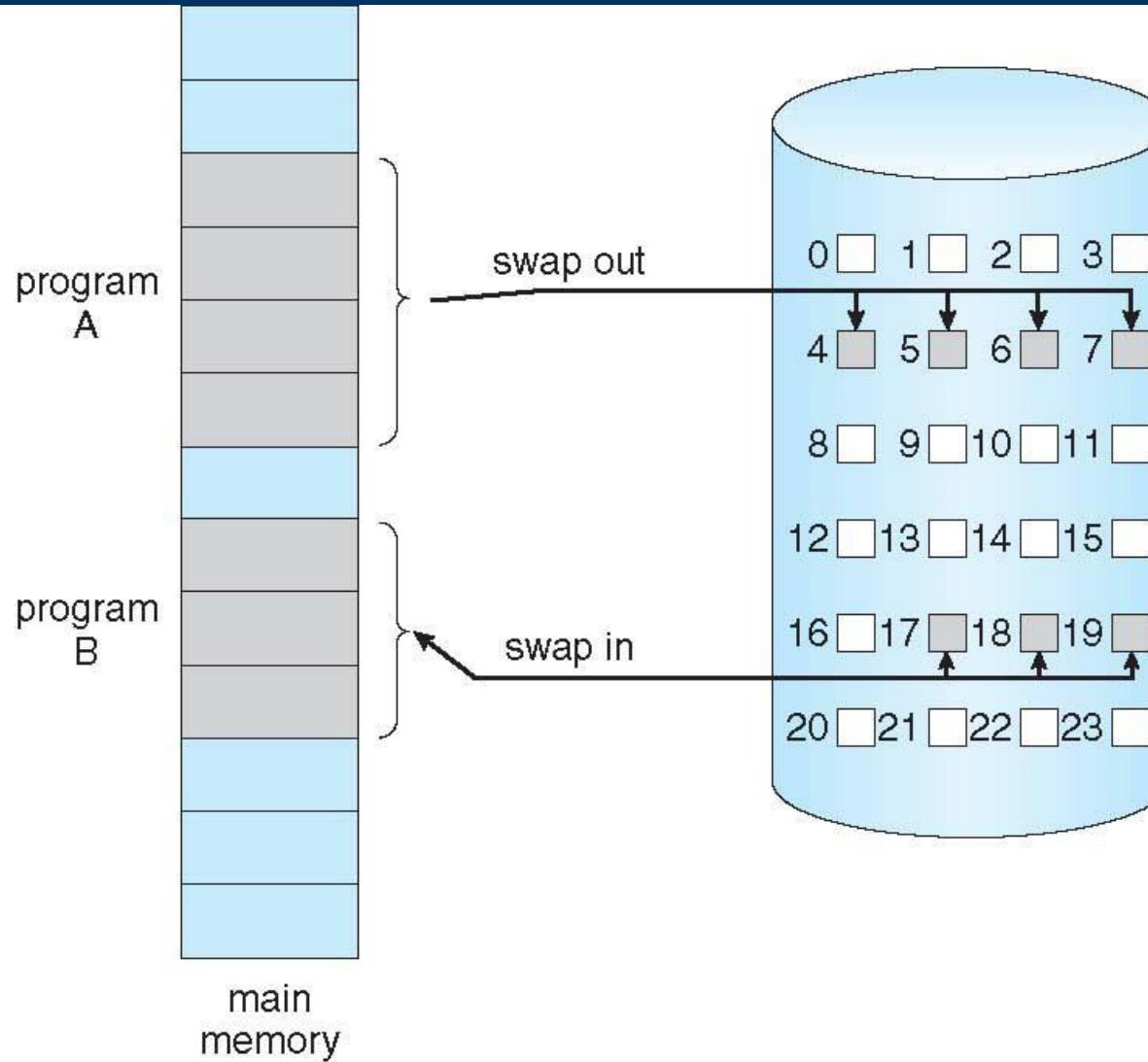
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Transfer of a Paged Memory to Contiguous Disk Space



Steps

Steps	Description
Step 1	Check an internal table for this process, to determine whether the reference was a valid or it was an invalid memory access.
Step 2	If the reference was invalid, terminate the process. If it was valid, but page have not yet brought in, page in the latter.
Step 3	Find a free frame.
Step 4	Schedule a disk operation to read the desired page into the newly allocated frame.
Step 5	When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
Step 6	Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory. Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

Advantages/Disadvantages

Advantages

Following are the advantages of Demand Paging

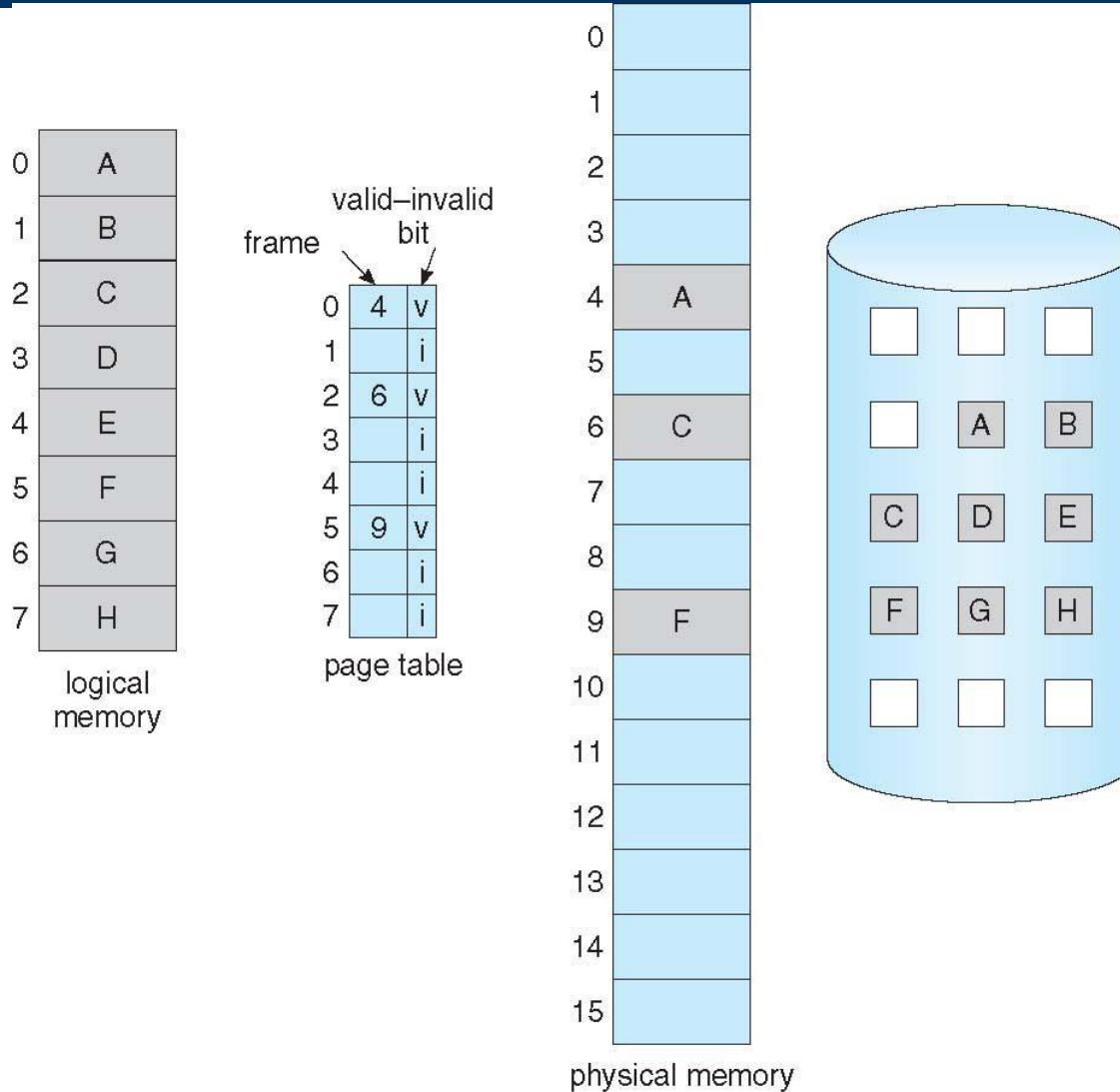
- Large virtual memory.
- More efficient use of memory.
- Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

Disadvantages

Following are the disadvantages of Demand Paging

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of an explicit constraints on a jobs address space size.

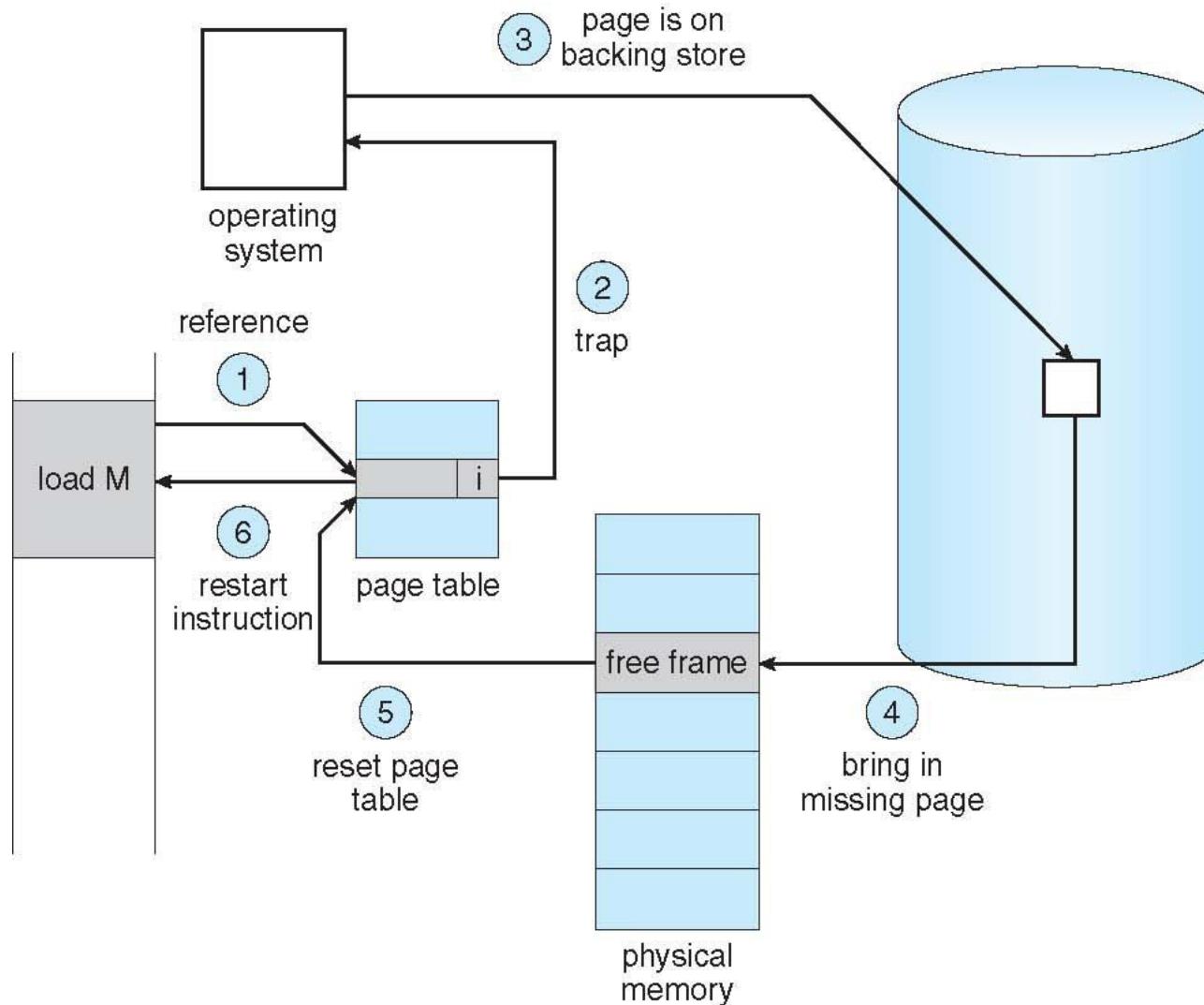
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:
page fault
 1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
 2. Get empty frame
 3. Swap page into frame via scheduled disk operation
 4. Reset tables to indicate page now in memory
Set validation bit = **v**
 5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

Stages in Demand Paging

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 - i. Wait in a queue for this device until the read request is serviced
 - ii. Wait for the device seek and/or latency time
 - iii. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Paging

- If a page is not in physical memory
 - find the page on disk
 - find a free frame
 - bring the page into memory
- What if there is no free frame in memory?

Page Replacement

- Basic idea
 - if there is a free page in memory, use it
 - if not, select a *victim frame*
 - write the victim out to disk
 - read the desired page into the now free frame
 - update page tables
 - restart the process

Page Replacement

- Main objective of a good replacement algorithm is to achieve a **low *page fault rate***
 - insure that heavily used pages stay in memory
 - the replaced page should not be needed for some time
- Secondary objective is to reduce latency of a page fault
 - efficient code
 - replace pages that do not need to be written out

Reference String

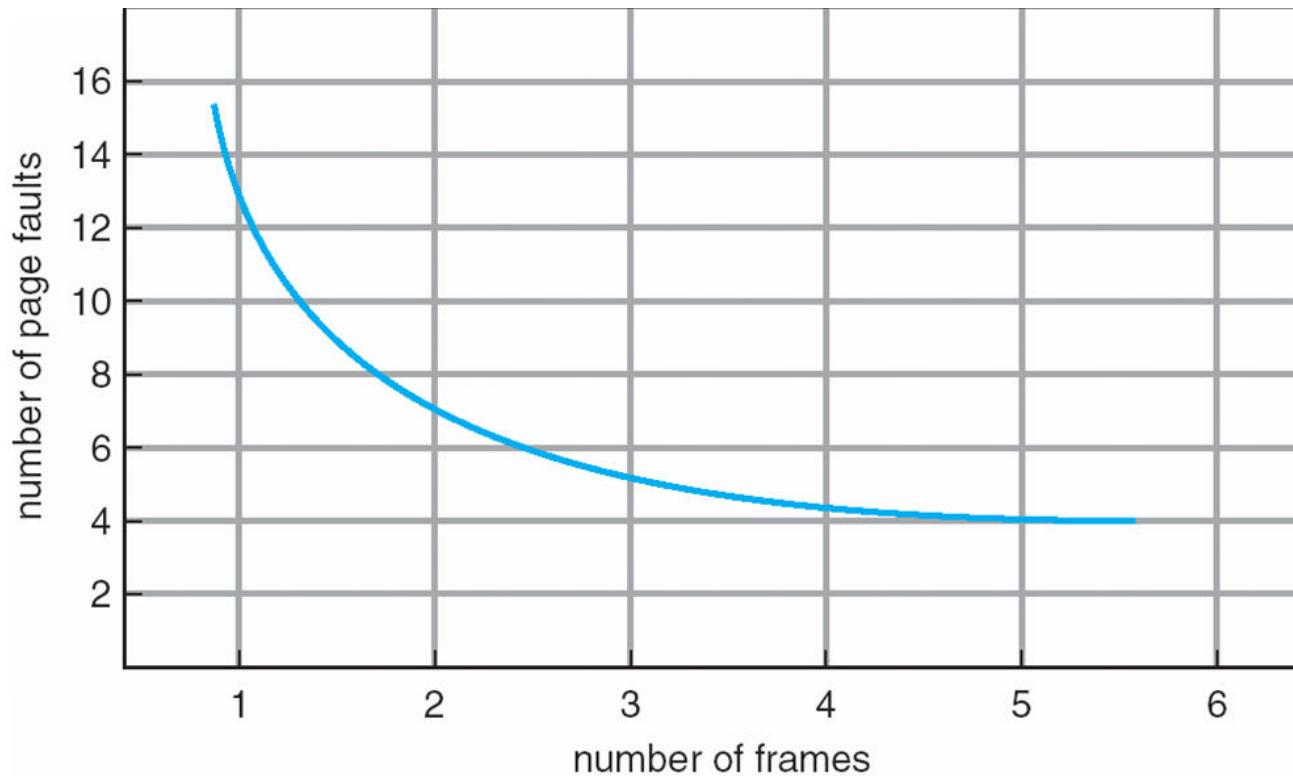
- Reference string is the sequence of pages being referenced
- If user has the following sequence of addresses
 - 123, 215, 600, 1234, 76, 96
- If the page size is 100, then the reference string is
 - 1, 2, 6, 12, 0, 0

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
- The reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In, First-Out (FIFO)

- The oldest page in physical memory is the one selected for replacement
- Very simple to implement
 - keep a list
 - victims are chosen from the tail
 - new pages in are placed at the head

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process):

1	1	4	5
2	2	1	3
3	3	2	4

9 page faults

- 4 frames:

1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

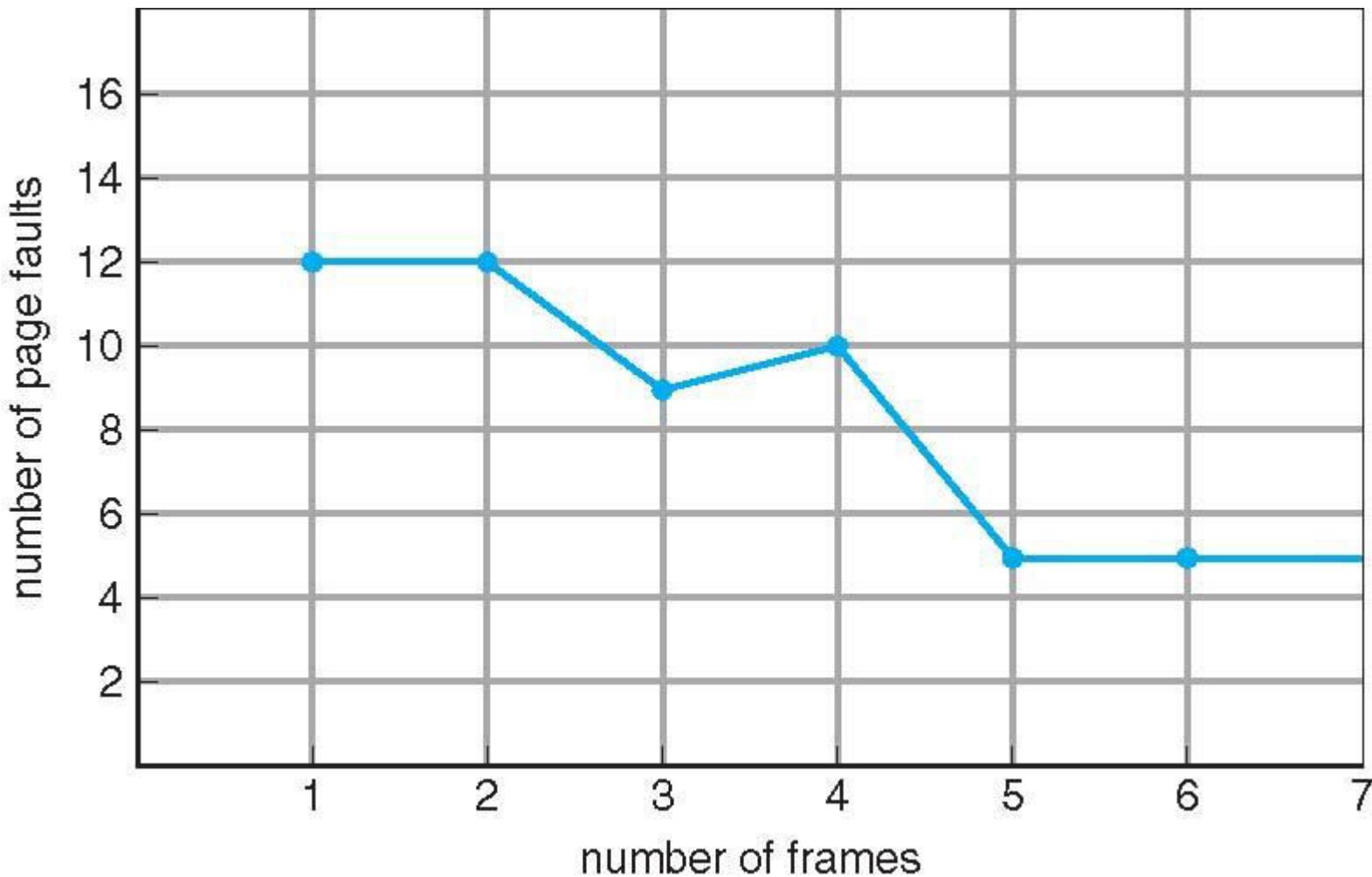
10 page faults

- FIFO Replacement manifests Belady's Anomaly:
 - more frames $\not\Rightarrow$ less page faults

FIFO Issues

- Poor replacement policy
- Evicts the oldest page in the system
 - usually a heavily used variable should be around for a long time
 - FIFO replaces the oldest page - perhaps the one with the heavily used variable
- FIFO does not consider page usage

FIFO Illustrating Belady's Anomaly



Optimal Page Replacement

- Often called Balady's Min
- Basic idea
 - replace the page that will not be referenced for the longest time
- This gives the lowest possible fault rate
- **Impossible to implement**
- Does provide a good measure for other techniques

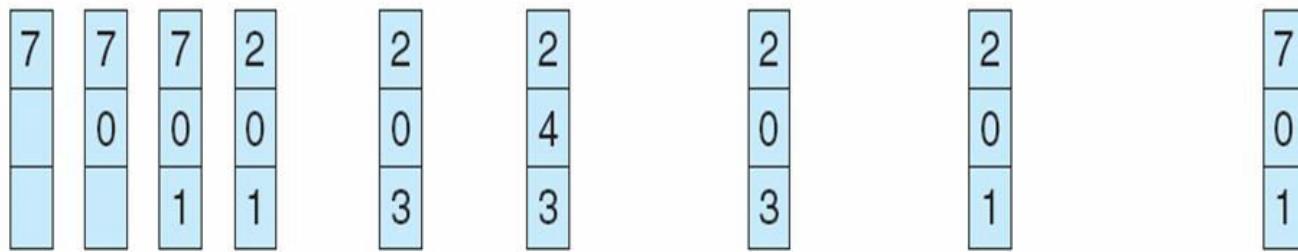
Optimal Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms.
- An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

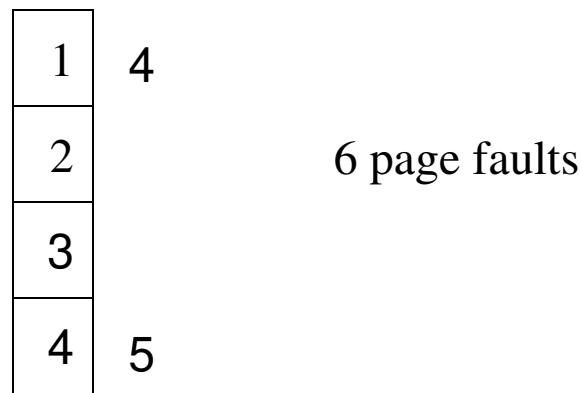


page frames

Optimal Algorithm

- Replace page that will not be used for longest period of time.
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- How do you know this?
- Used for measuring how well your algorithm performs.

Least Recently Used (LRU)

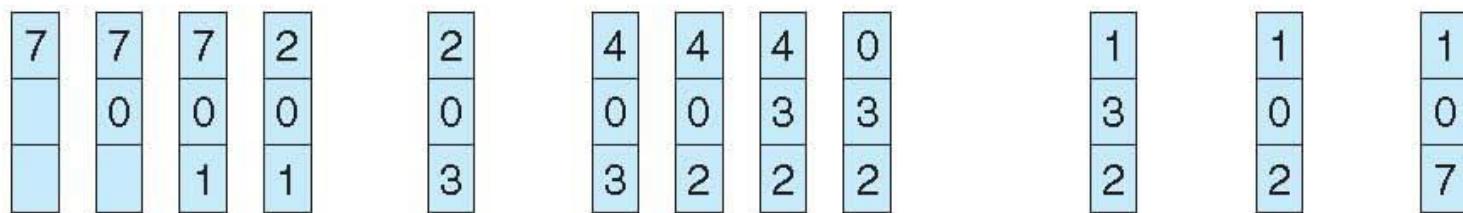
- Basic idea
 - replace the page in memory that has not been accessed for the longest time
- Optimal policy looking back in time
 - as opposed to forward in time
 - fortunately, programs tend to follow similar behavior

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Issues

- How to keep track of last page access?
 - requires special hardware support
- 2 major solutions
 - counters
 - hardware clock “ticks” on every memory reference
 - the page referenced is marked with this “time”
 - the page with the smallest “time” value is replaced
 - stack
 - keep a stack of references
 - on every reference to a page, move it to top of stack
 - page at bottom of stack is next one to be replaced

LRU Issues

- Both techniques just listed require additional hardware
 - remember, memory reference are very common
 - impractical to invoke software on every memory reference
- LRU is not used very often
- Instead, we will try to approximate LRU

Replacement Hardware Support

- Most system will simply provide a *reference bit* in PT for each page
- On a reference to a page, this bit is set to 1
- This bit can be cleared by the OS
- This simple hardware has lead to a variety of algorithms to approximate LRU

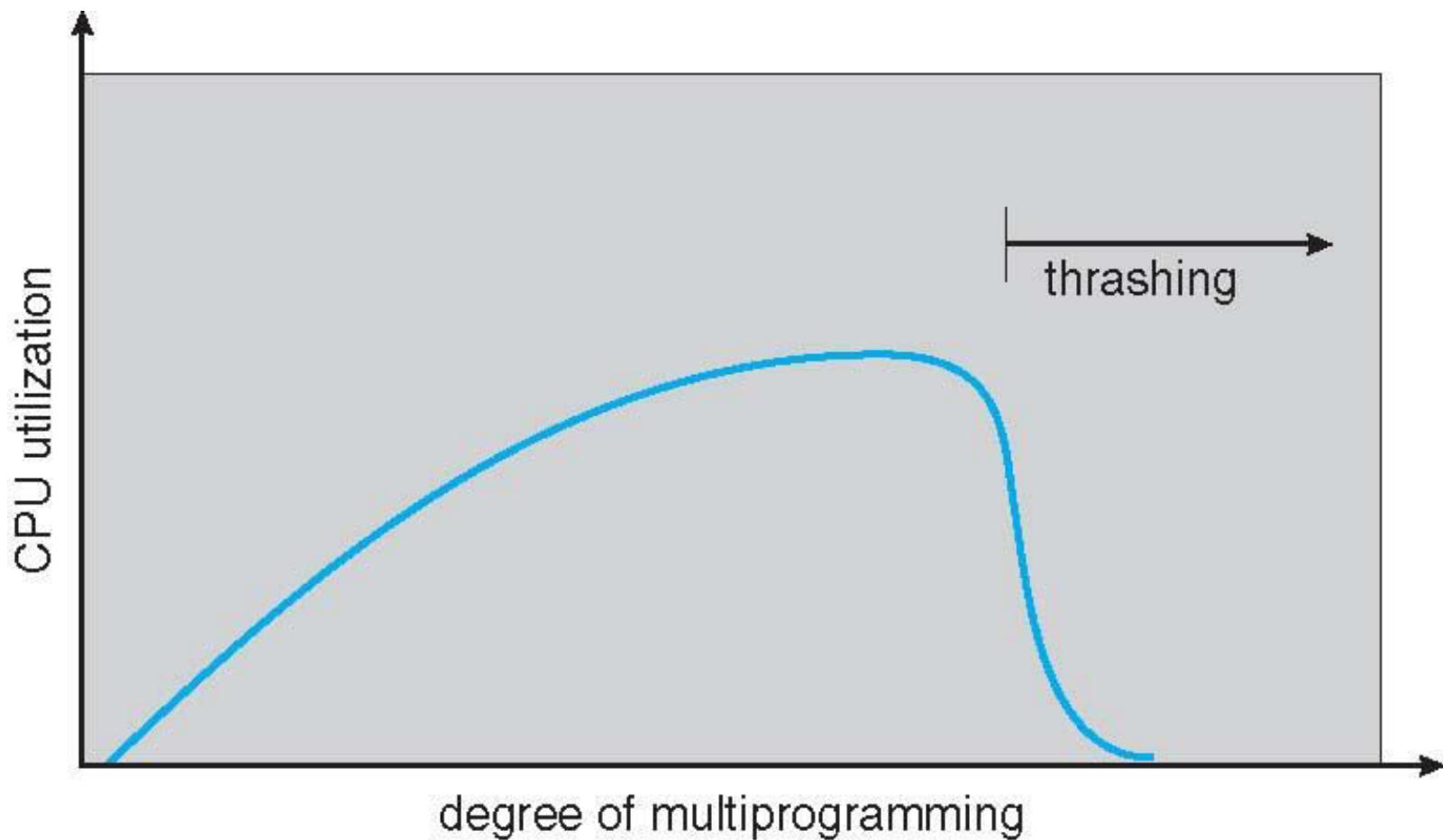
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - low CPU utilization
 - OS thinks it needs increased multiprogramming
 - adds another process to system
- *Thrashing* is when a process is busy swapping pages in and out

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
 - This leads to:
 - Low CPU utilization
 - Operating system thinking that it needs to increase the degree of multiprogramming
 - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Cause of Thrashing

- *Why does paging work?*
 - *Locality model*
 - *process migrates from one locality to another*
 - *localities may overlap*
- *Why does thrashing occur?*
 - *sum of localities > total memory size*
- *How do we fix thrashing?*
 - *Working Set Model*
 - *Page Fault Frequency*

Demand Paging and Thrashing

- Why does demand paging work?

Locality model

- Process migrates from one locality to another
 - Localities may overlap

- Why does thrashing occur?

Σ size of locality > total memory size

- Limit effects by using local or priority page replacement

Operating Systems

File System

Alok Kumar Jagadev

File

- A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks.
- In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

File Structure

- File structure is a structure according to a required format that operating system can understand.
- A file has a certain defined structure according to its type.
 - A **text file** is a sequence of characters organized into lines.
 - A **source file** is a sequence of procedures and functions.
 - An **object file** is a sequence of bytes organized into blocks that are understandable by the machine.
- When operating system defines different file structures, it also contains the code to support these file structure. Unix, MS-DOS support minimum number of file structure.

File Type

- File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc.
- Many operating systems support many types of files.
- Operating system like MS-DOS and UNIX have the following types of files:
- **Ordinary files**
 - These are the files that contain user information.
 - These may have text, databases or executable program.
 - The user can apply various operations on such files like add, modify, delete or even remove the entire file.
- **Directory files**
 - These files contain list of file names and other information related to these files.

File Type

- Special files:
 - These files are also known as **device files**.
 - These files represent physical device like disks, terminals, printers, networks, tape drive etc.

These files are of two types

- **Character special files** - data is handled character by character as in case of terminals or printers.
- **Block special files** - data is handled in blocks as in the case of disks and tapes.

File Access Mechanisms

- File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files
 - Sequential access
 - Direct/Random access
 - Indexed sequential access

Sequential access

- A sequential access is that in which the records are accessed in some sequence i.e the information in the file is processed in order, one record after the other.
- This access method is the most primitive one. Example: Compilers usually access files in this fashion.

File Access Mechanisms

Direct/Random access

- Random access file organization provides, accessing the records directly.
- Each record has its own address on the file by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

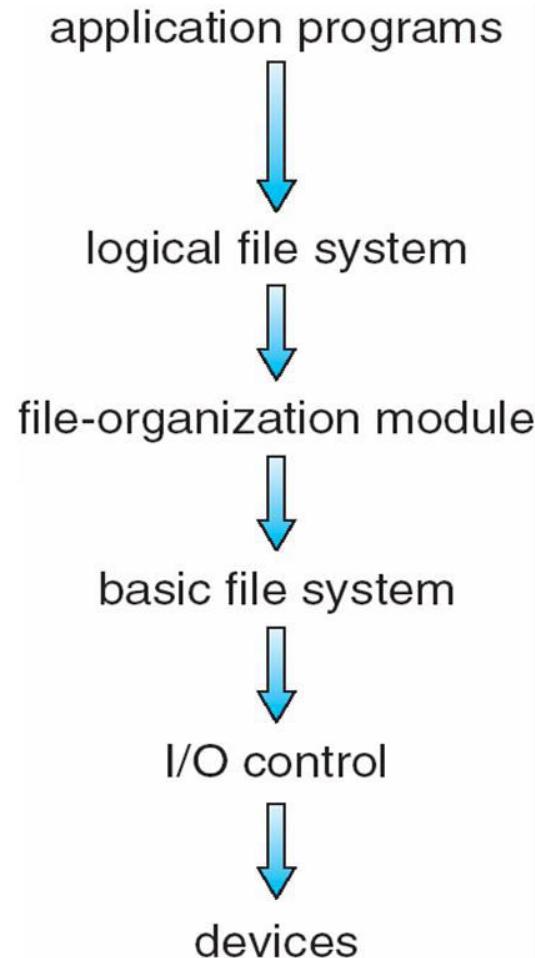
Indexed sequential access

- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.
- Index is searched sequentially and its pointer is used to access the file directly.

File-System Structure

- **File system** resides on secondary storage (disks)
 - Provided user interface to storage, mapping logical to physical
 - Provides efficient and convenient access to disk by allowing data to be stored, located, retrieved easily
- Disk provides in-place rewrite and random access
 - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device

Layered File System



File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
- The **basic file system** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.
 - Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
- The **file organization module** knows about files and their logical blocks, and how they map to physical blocks on the disk.
 - In addition to translating from logical to physical blocks, the file organization module also **maintains the list of free blocks**, and allocates free blocks to files as needed.

File System Layers (Cont.)

- The **logical file system** deals with all of the **metadata** associated with a file (UID, GID, mode, dates, etc), i.e. **everything about the file except the data itself.**
 - This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the metadata as well as block number information for finding the data on the disk.
- Layering is useful for **reducing complexity and redundancy**, but **adds overhead and can decrease performance**
- Translates file name into file number, file handle, location by maintaining file control blocks
 - Logical layers can be implemented by any coding method according to OS designer

File-System Implementation

File systems store several important data structures on the disk:

- A ***boot-control block***, (per volume): the ***boot block*** in UNIX or the ***partition boot sector*** in Windows contains information about how to boot the system of this disk.
 - This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
- A ***volume control block***, (per volume): the ***master file table*** in UNIX or the ***superblock*** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.

File-System Implementation

- A directory structure (per file system), containing file names and pointers to corresponding FCBs.
 - UNIX uses inode numbers, and NTFS uses a *master file table*.
- The **File Control Block, FCB**, (per file) containing details about ownership, size, permissions, dates, etc.
 - UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Partitions and Mounting

- Physical disks are commonly divided into smaller units called **partitions**.
 - They can also be combined into larger units, but that is most commonly done for **RAID installations** and is left for later chapters.
- Partitions can either be used as **raw devices** (**with no structure imposed upon them**), or they can be formatted to hold a filesystem (**i.e. populated with FCBs and initial directory structures as appropriate.**)
- Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system.
- Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.

Partitions and Mounting

- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The *root partition* contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)

Partitions and Mounting

- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

Partitions and Mounting

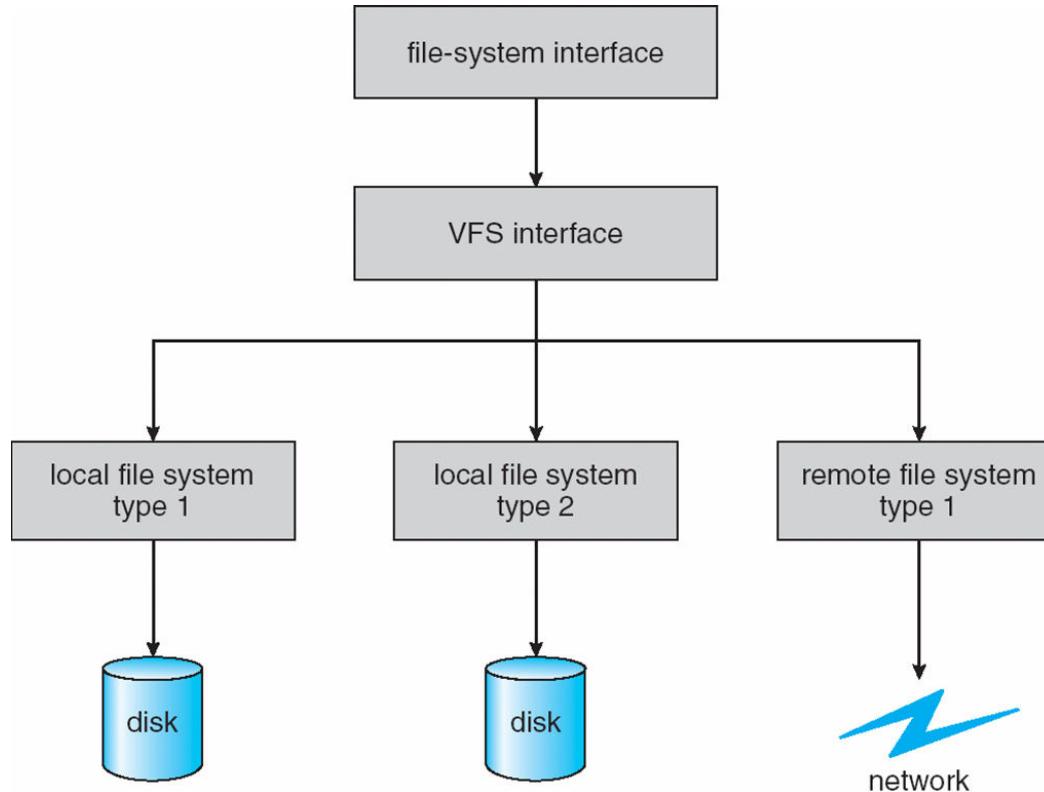
- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
 - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
 - Mounted at boot time
 - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
 - Is all metadata correct?
 - If not, fix it, try again
 - If yes, add to mount table, allow access

Virtual File Systems

- **Virtual File Systems (VFS)** on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
 - Separates file-system generic operations from implementation details
 - Implementation can be one of many file systems types, or network file system
 - Implements **vnodes** which hold inodes or network file details
 - Then dispatches operation to appropriate file system implementation routines

Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system



Virtual File System Implementation

- For example, Linux has four object types:
 - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
 - Every object has a pointer to a function table
 - Function table has addresses of routines to implement that function on that object
 - For example:
 - **int open(...)**—Open a file
 - **int close(...)**—Close an already-open file
 - **ssize_t read(...)**—Read from a file
 - **ssize_t write(...)**—Write to a file
 - **int mmap(...)**—Memory-map a file

Directory Implementation

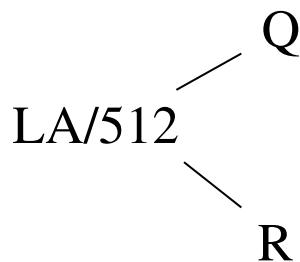
- **Linear list** of file names with pointer to the data blocks
 - Simple to program
 - Time-consuming to execute
 - Linear search time
 - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
 - Decreases directory search time
 - **Collisions** – situations where two file names hash to the same location
 - Only good if entries are fixed size, or use chained-overflow method

Allocation Methods - Contiguous

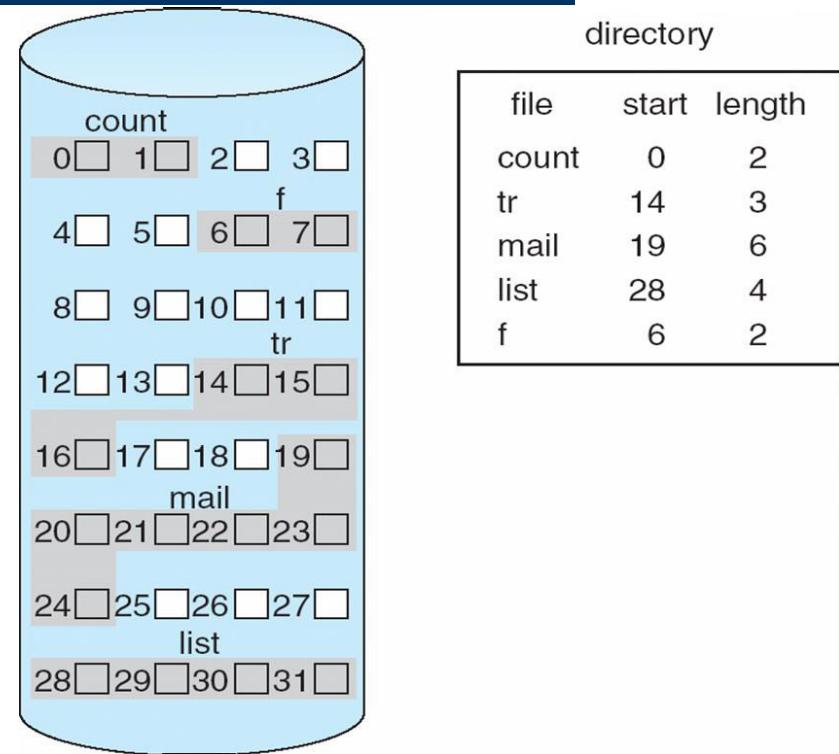
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
 - Best performance in most cases
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line (downtime)** or **on-line**

Contiguous Allocation

- Mapping from logical to physical



Block to be accessed = Q +
starting address
Displacement into block = R



Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
 - Extents are allocated for file allocation
 - A file consists of one or more extents

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
 - File ends at nil pointer
 - No external fragmentation
 - Each block contains pointer to next block
 - No compaction, external fragmentation
 - Free space management system called when new block needed
 - Improve efficiency by clustering blocks into groups but increases internal fragmentation
 - Reliability can be a problem
 - Locating a block can take many I/Os and disk seeks

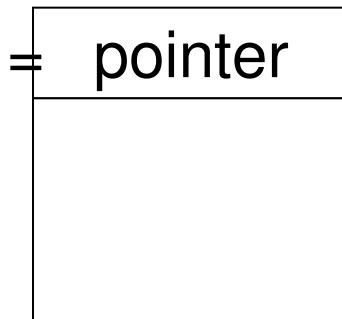
Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation
 - Beginning of volume has table, indexed by block number
 - Much like a linked list, but faster on disk and cacheable
 - New block allocation simple

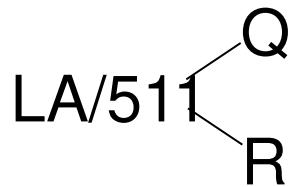
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk

block



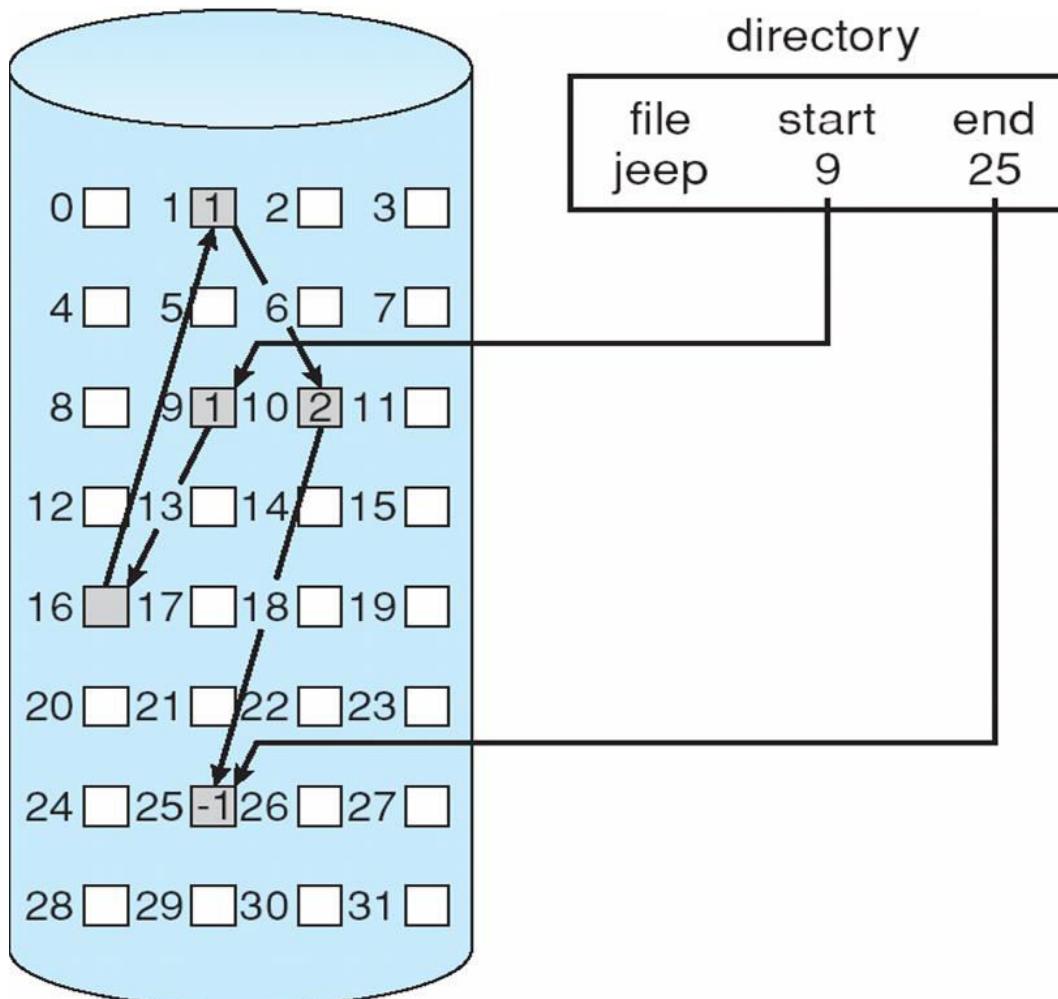
■ Mapping



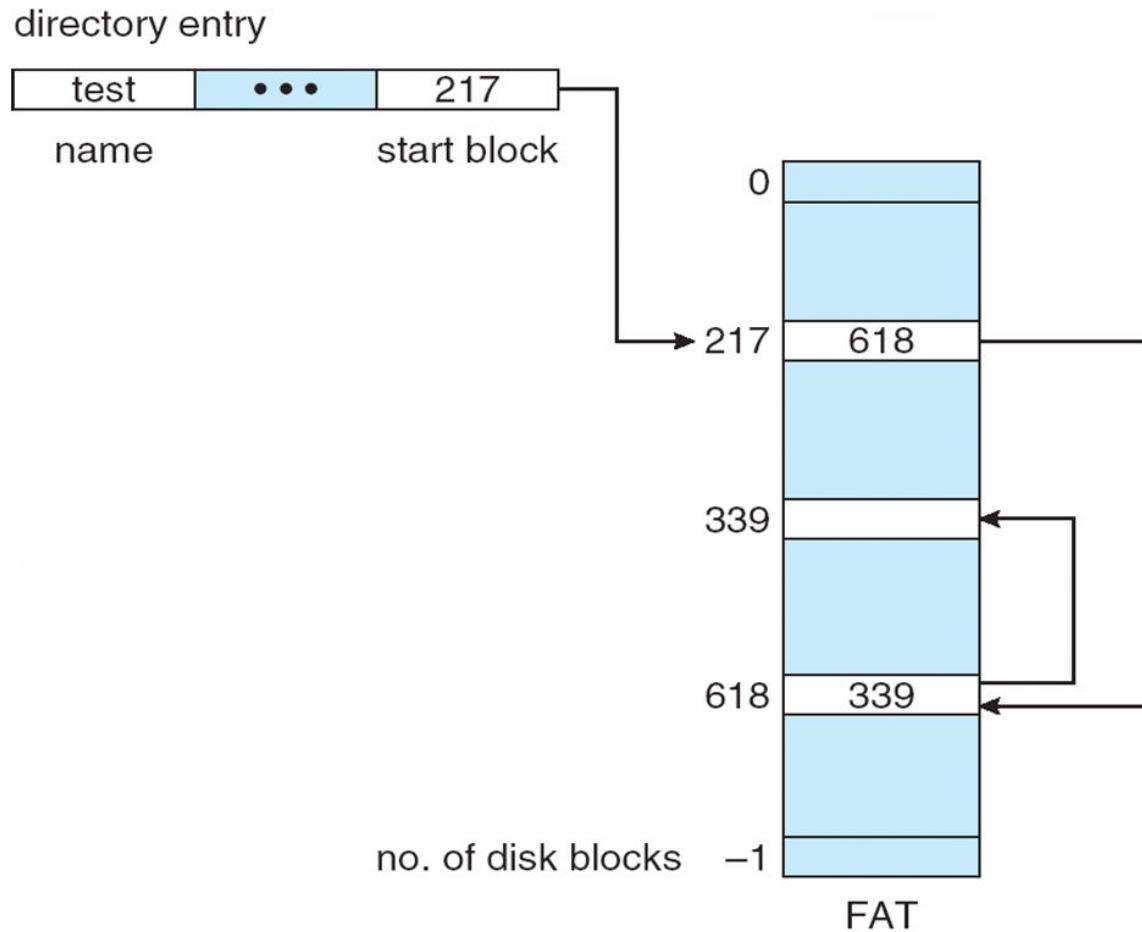
Block to be accessed is the Qth block in the linked chain of blocks
representing the file.

Displacement into block = R + 1

Linked Allocation

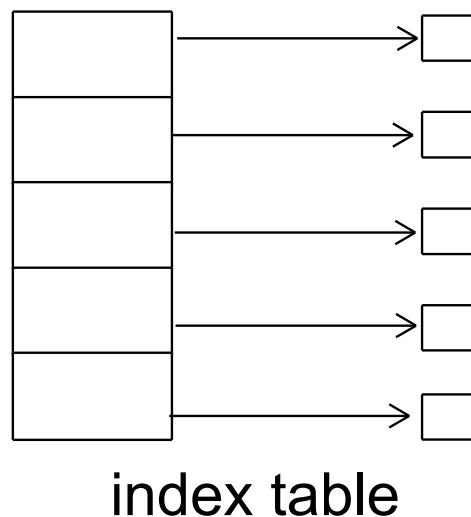


File-Allocation Table

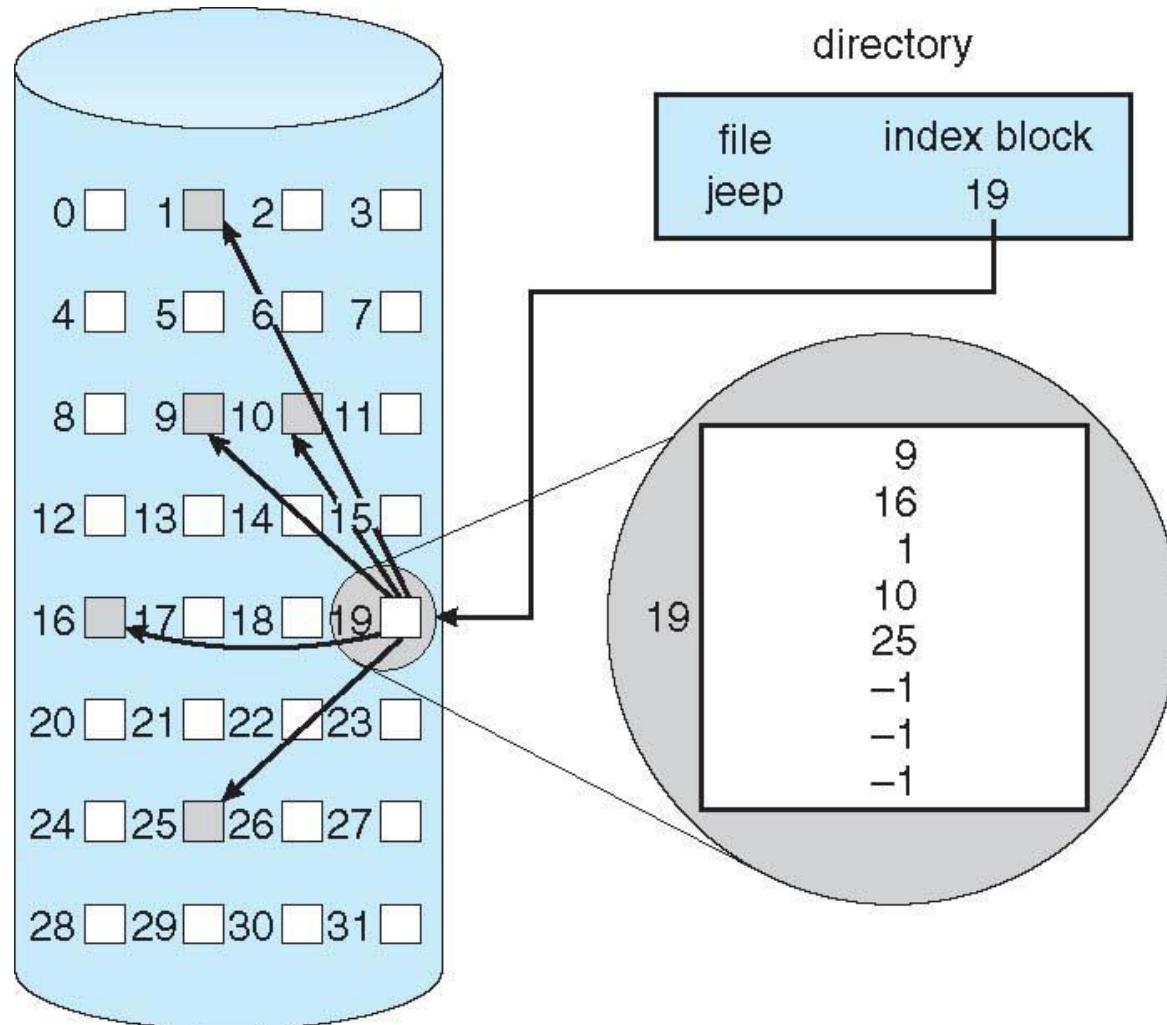


Allocation Methods - Indexed

- **Indexed allocation**
 - Each file has its own **index block**(s) of pointers to its data blocks
- Logical view

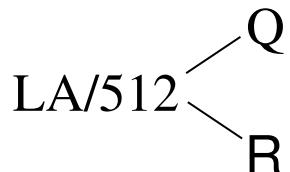


Example of Indexed Allocation



Indexed Allocation (Cont.)

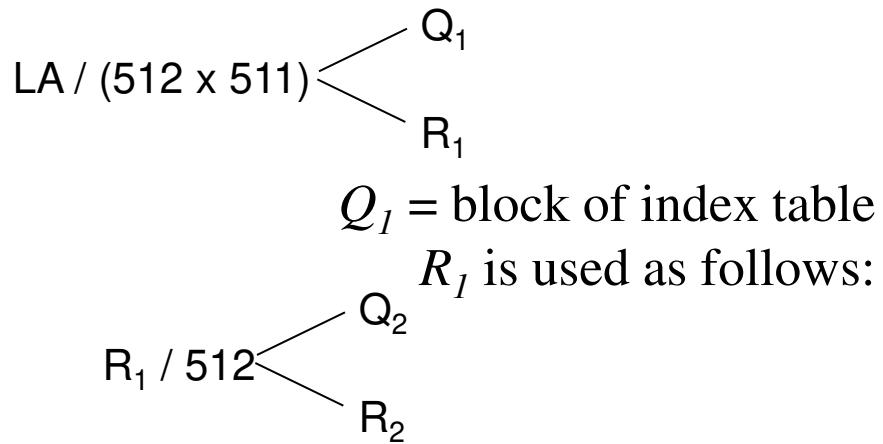
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)

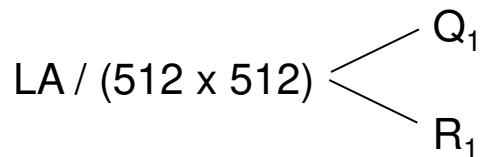


$Q_1 = \text{block of index table}$
 R_1 is used as follows:
 Q_2
 $R_1 / 512$
 R_2

$Q_2 = \text{displacement into block of index table}$
 R_2 displacement into block of file:

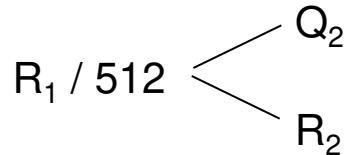
Indexed Allocation – Mapping (Cont.)

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)



Q_1 = displacement into outer-index

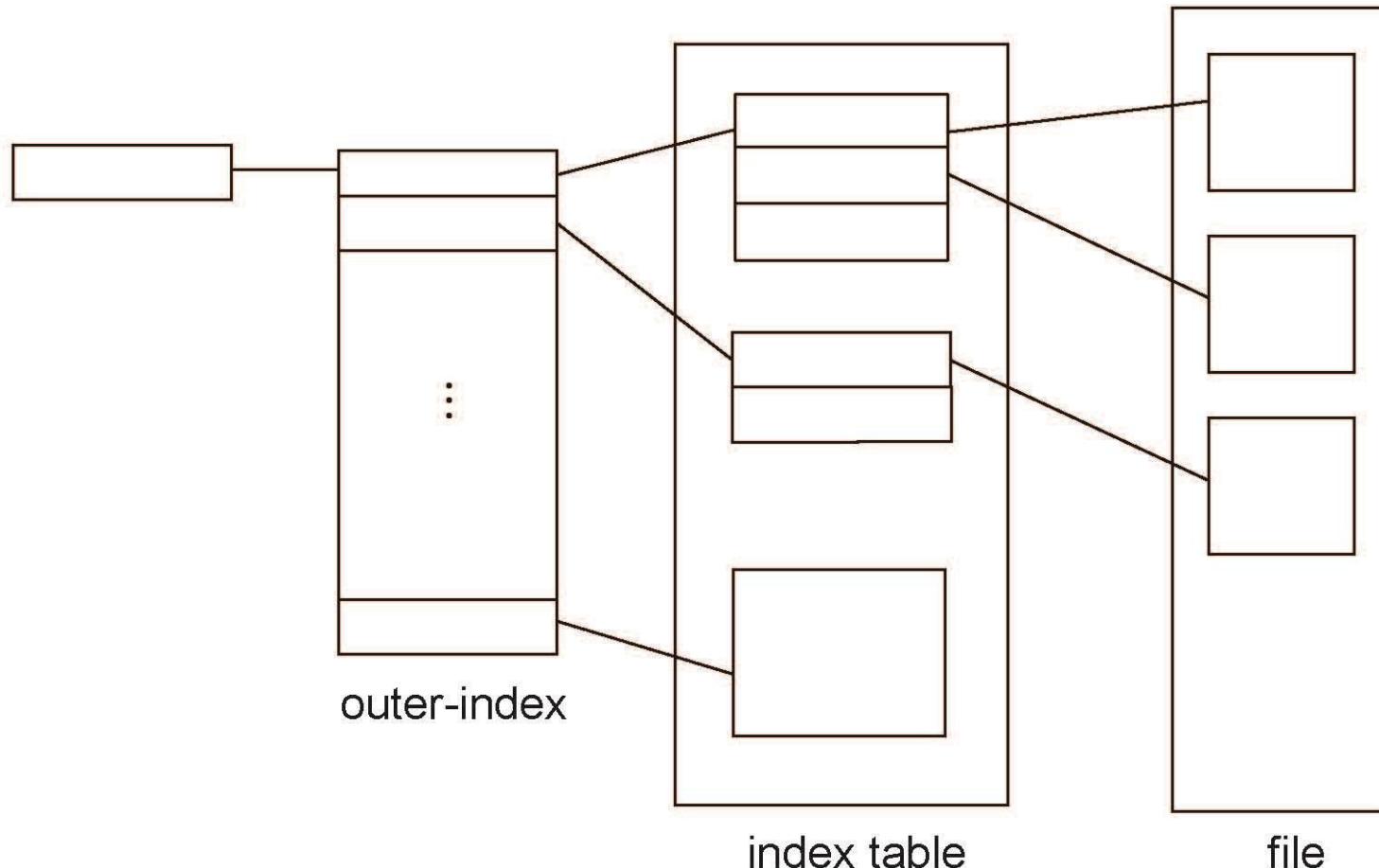
R_1 is used as follows:



Q_2 = displacement into block of index table

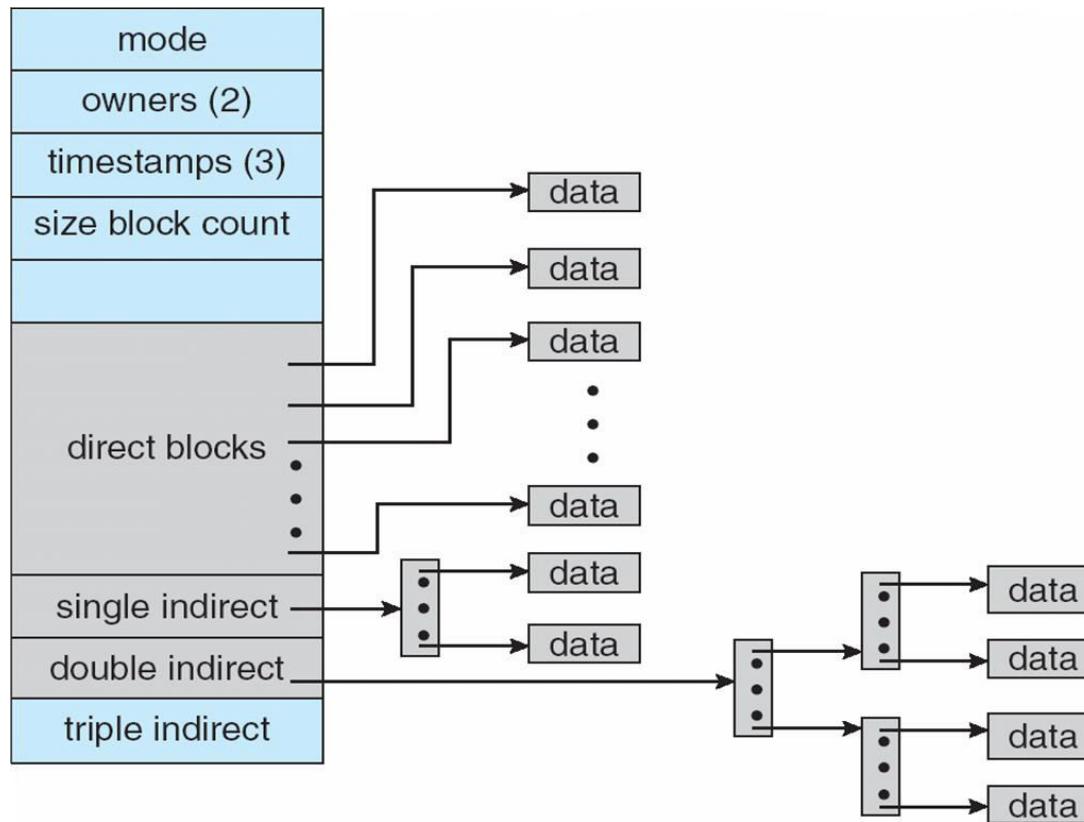
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

Performance

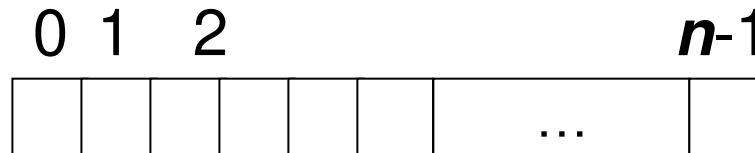
- Best method depends on file access type
 - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
 - Single block access could require 2 index block reads then data block read
 - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- Adding instructions to the execution path to save one disk I/O is reasonable
 - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
 - http://en.wikipedia.org/wiki/Instructions_per_second
 - Typical disk drive at 250 I/Os per second
 - $159,000 \text{ MIPS} / 250 = 630 \text{ million instructions during one disk I/O}$
 - Fast SSD drives provide 60,000 IOPS
 - $159,000 \text{ MIPS} / 60,000 = 2.65 \text{ millions instructions during one disk I/O}$

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (n blocks)



$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{block}[i] \text{ free} \\ 0 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

$$\begin{aligned} & (\text{number of bits per word}) * \\ & (\text{number of 0-value words}) + \\ & \quad \text{offset of first 1 bit} \end{aligned}$$

CPUs have instructions to return offset within word of first “1” bit

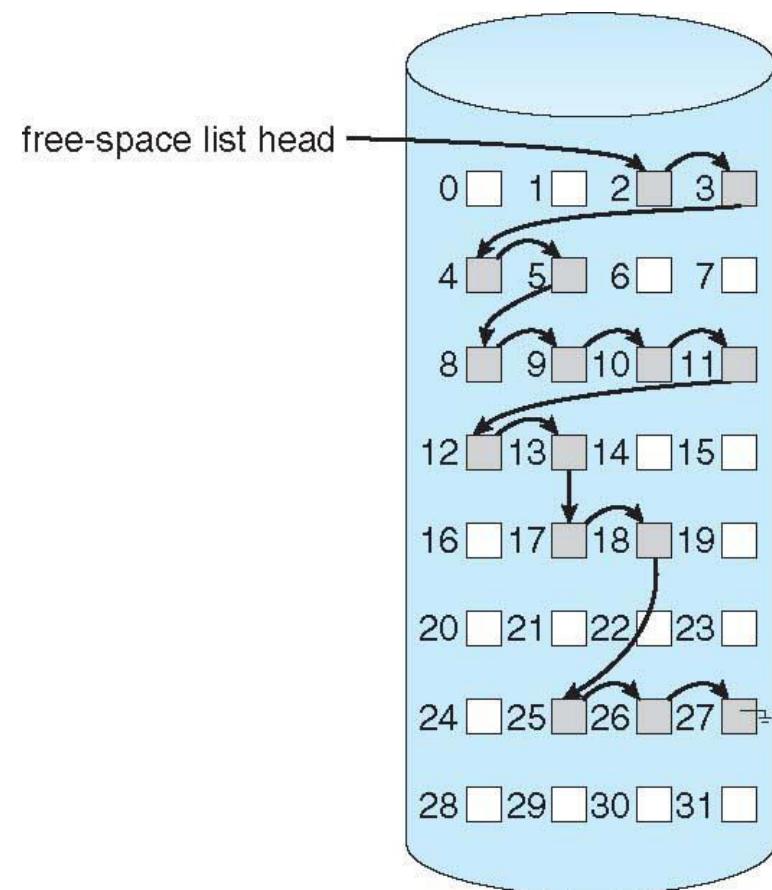
Free-Space Management (Cont.)

- Bit map requires extra space
 - Example:
 - block size = 4KB = 2^{12} bytes
 - disk size = 2^{40} bytes (1 terabyte)
 - $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)
 - if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files

Linked Free Space List on Disk

Linked list (free list)

- Cannot get contiguous space easily
- No waste of space
- No need to traverse the entire list (if # free blocks recorded)



Free-Space Management (Cont.)

- Grouping
 - Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
 - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
 - Keep address of first free block and count of following free blocks
 - Free space list then has entries containing addresses and counts

Free-Space Management (Cont.)

- Space Maps
 - Used in **ZFS**
 - Consider meta-data I/O on very large file systems
 - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
 - Divides device space into **metaslab** units and manages metaslabs
 - Given volume can contain hundreds of metaslabs
 - Each metaslab has associated space map
 - Uses counting algorithm
 - But records to log file rather than file system
 - Log of all block activity, in time order, in counting format
 - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
 - Replay log into that structure
 - Combine contiguous free blocks into single entry

Efficiency and Performance

- Efficiency dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures

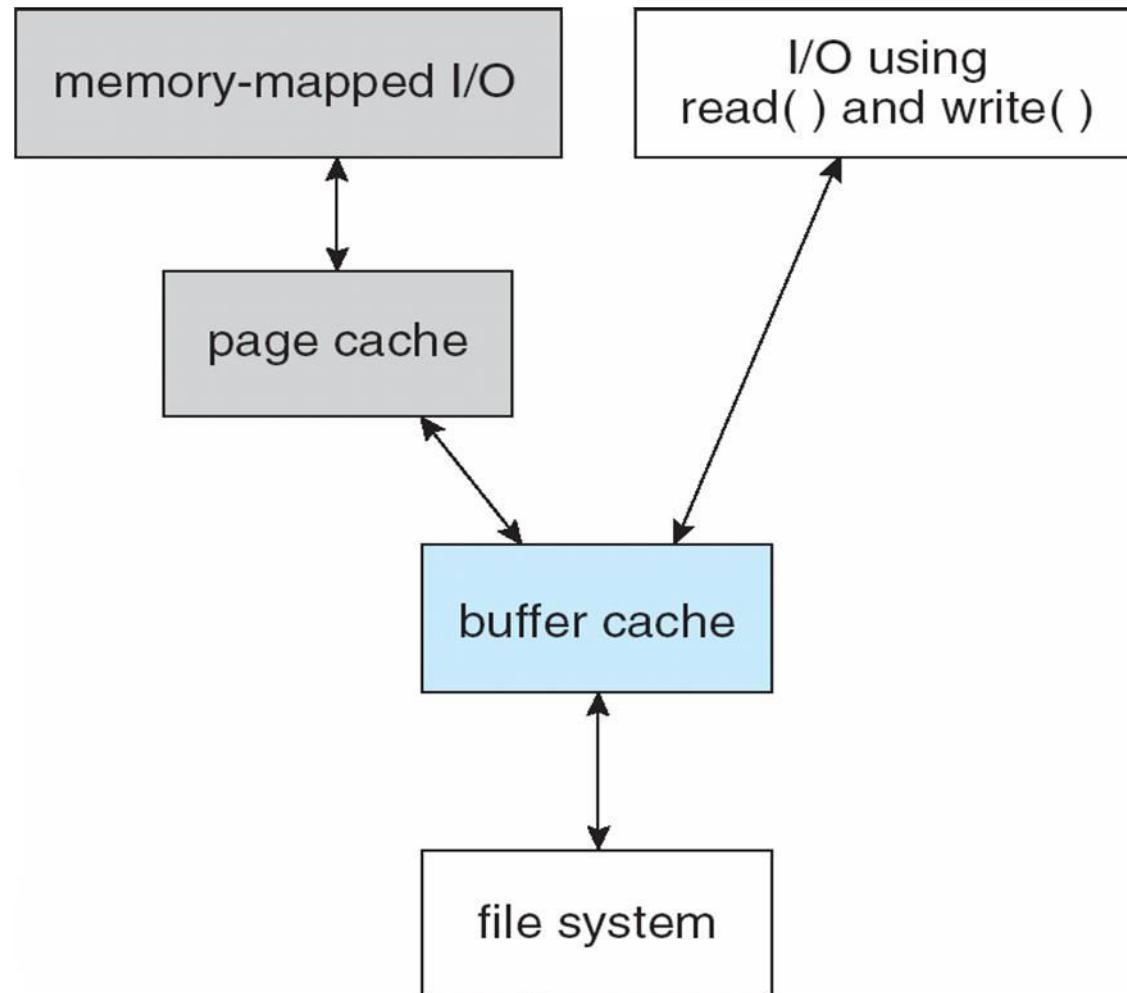
Efficiency and Performance (Cont.)

- Performance
 - Keeping data and metadata close together
 - **Buffer cache** – separate section of main memory for frequently used blocks
 - **Synchronous** writes sometimes requested by apps or needed by OS
 - No buffering / caching – writes must hit disk before acknowledgement
 - **Asynchronous** writes more common, buffer-able, faster
 - **Free-behind** and **read-ahead** – techniques to optimize sequential access
 - Reads frequently slower than writes

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

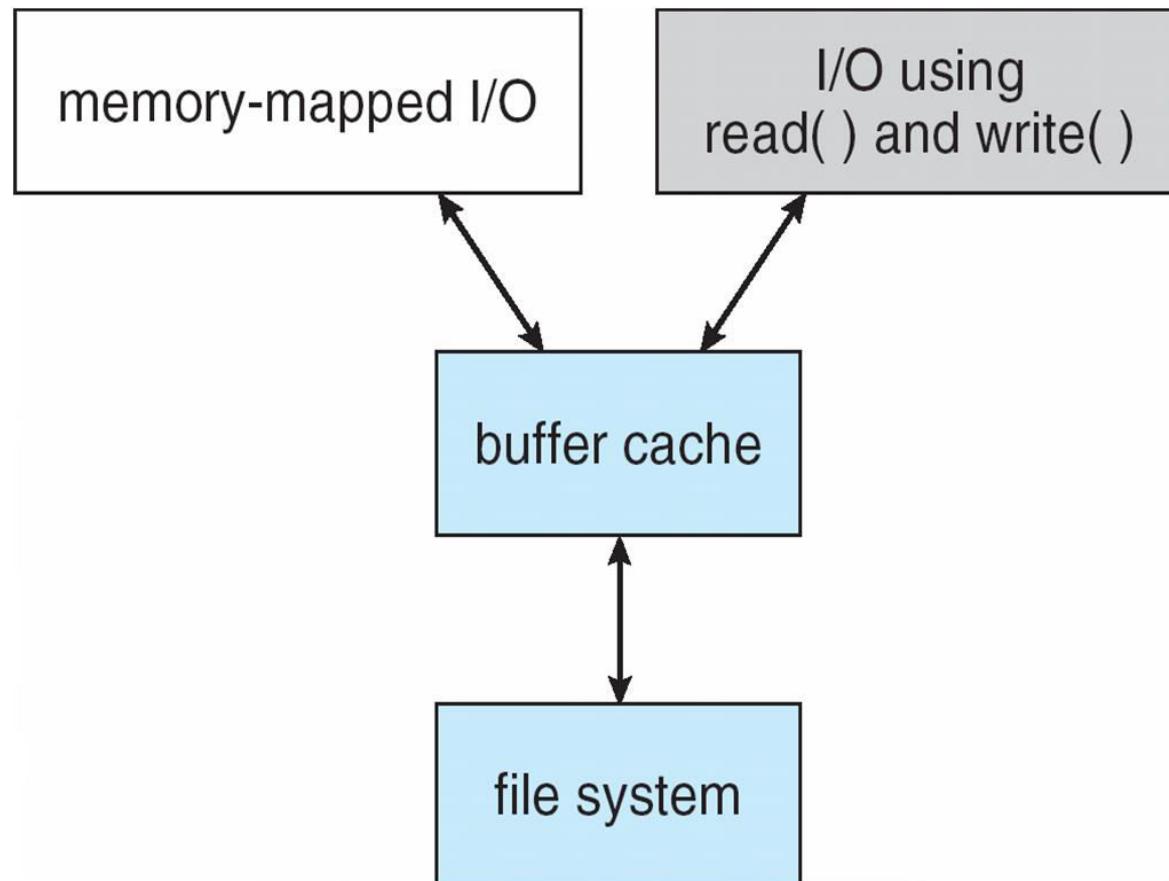
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
 - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
 - A transaction is considered committed once it is written to the log (sequentially)
 - Sometimes to a separate device or section of disk
 - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
 - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet)

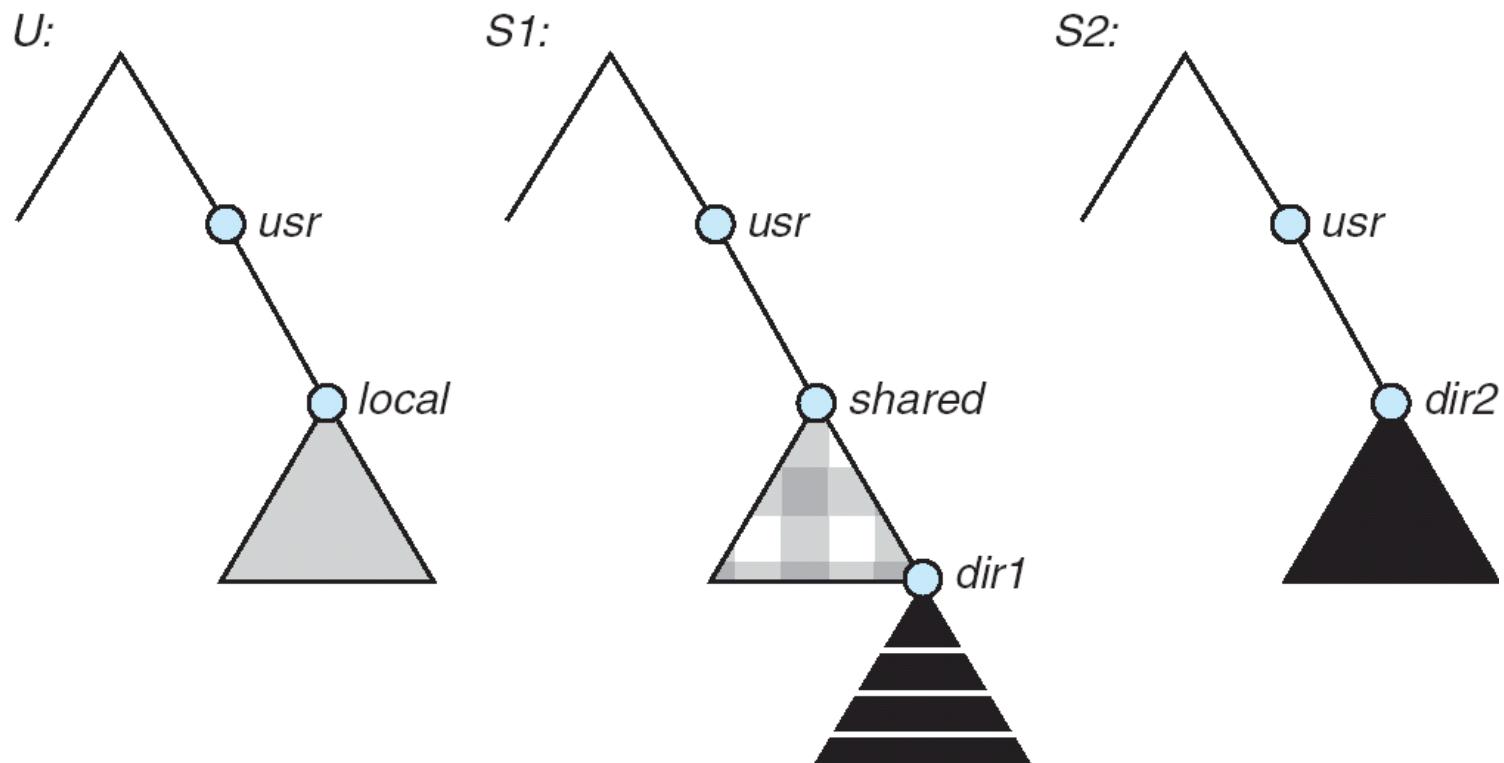
NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
 - A remote directory is mounted over a local file system directory
 - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
 - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
 - Files in the remote directory can then be accessed in a transparent manner
 - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

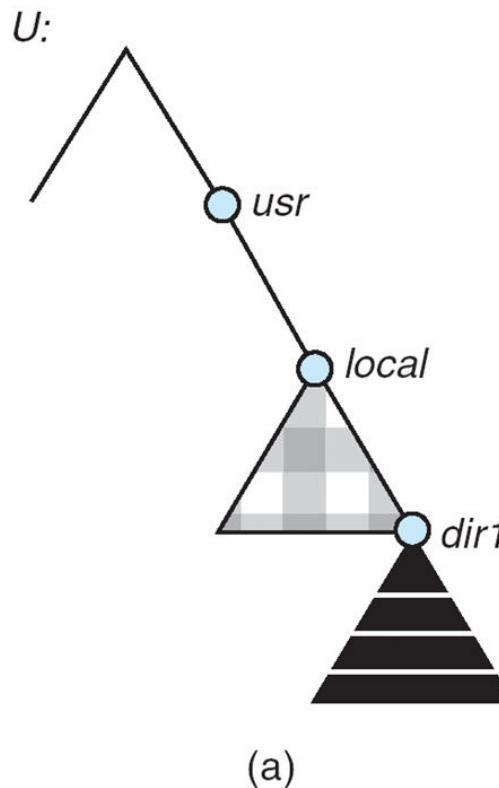
NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

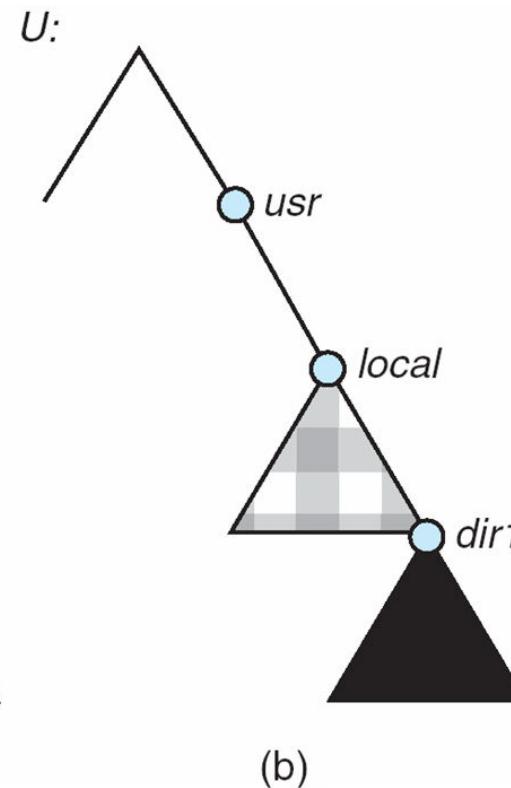
Three Independent File Systems



Mounting in NFS



Mounts



Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
 - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
 - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

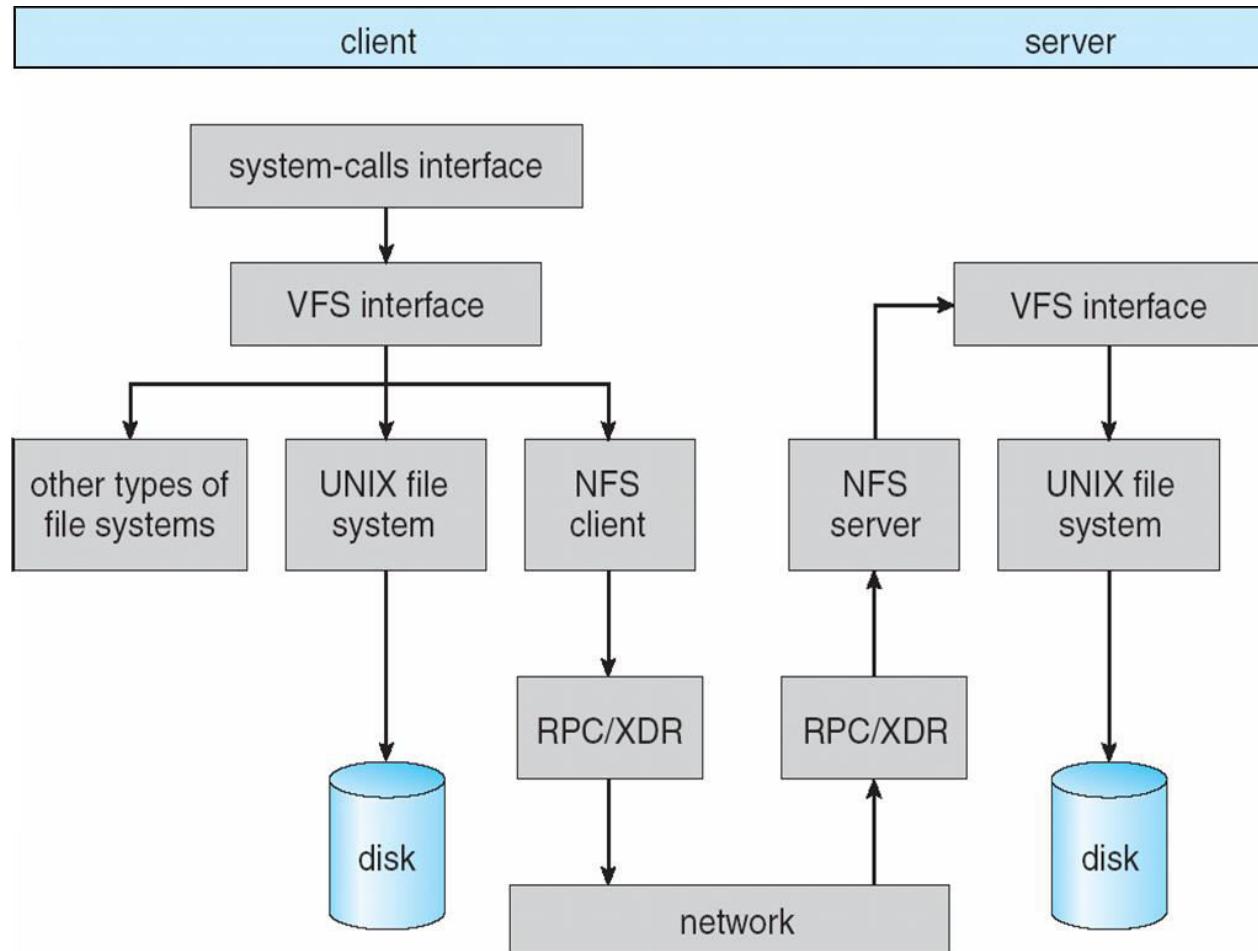
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
 - searching for a file within a directory
 - reading a set of directory entries
 - manipulating links and directories
 - accessing file attributes
 - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
 - The VFS activates file-system-specific operations to handle local requests according to their file-system types
 - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
 - Implements the NFS protocol

Schematic View of NFS Architecture



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names

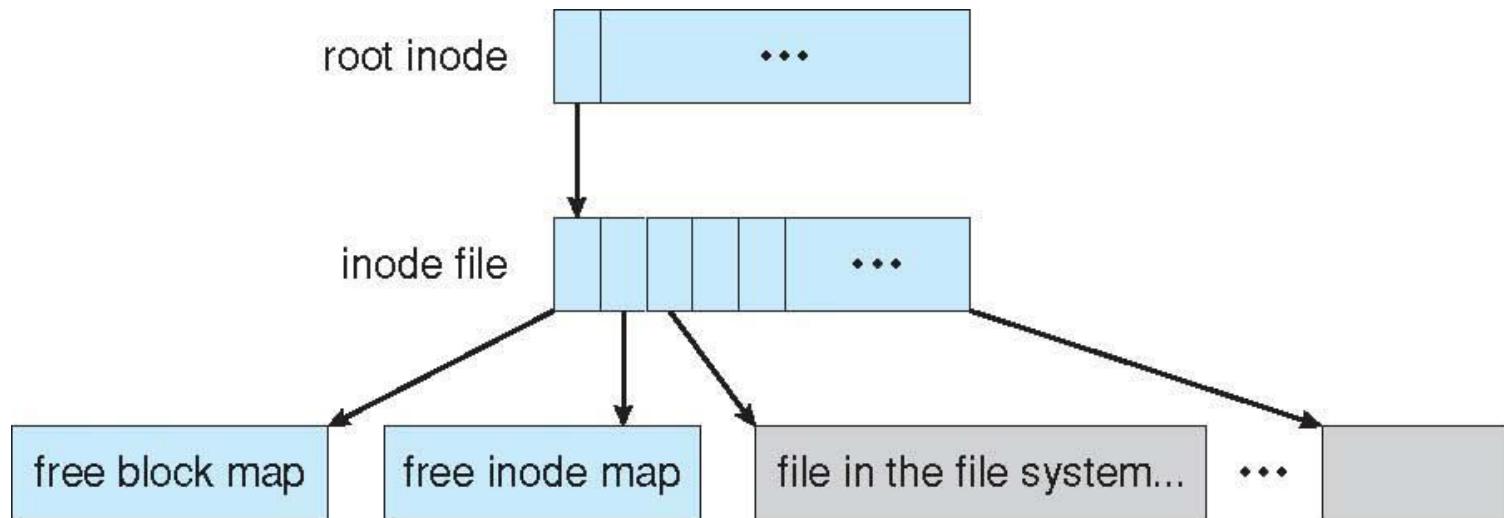
NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
 - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

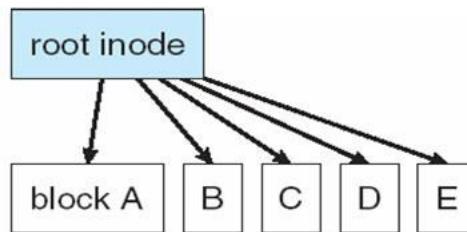
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
 - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

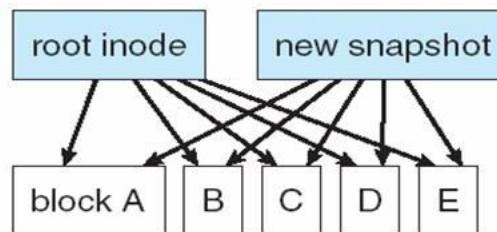
The WAFL File Layout



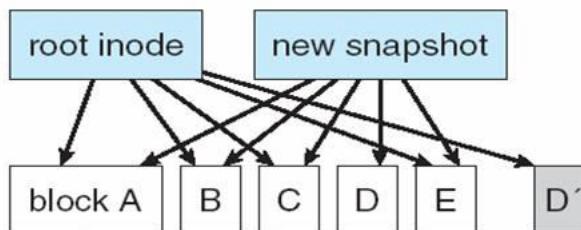
Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.

Operating Systems

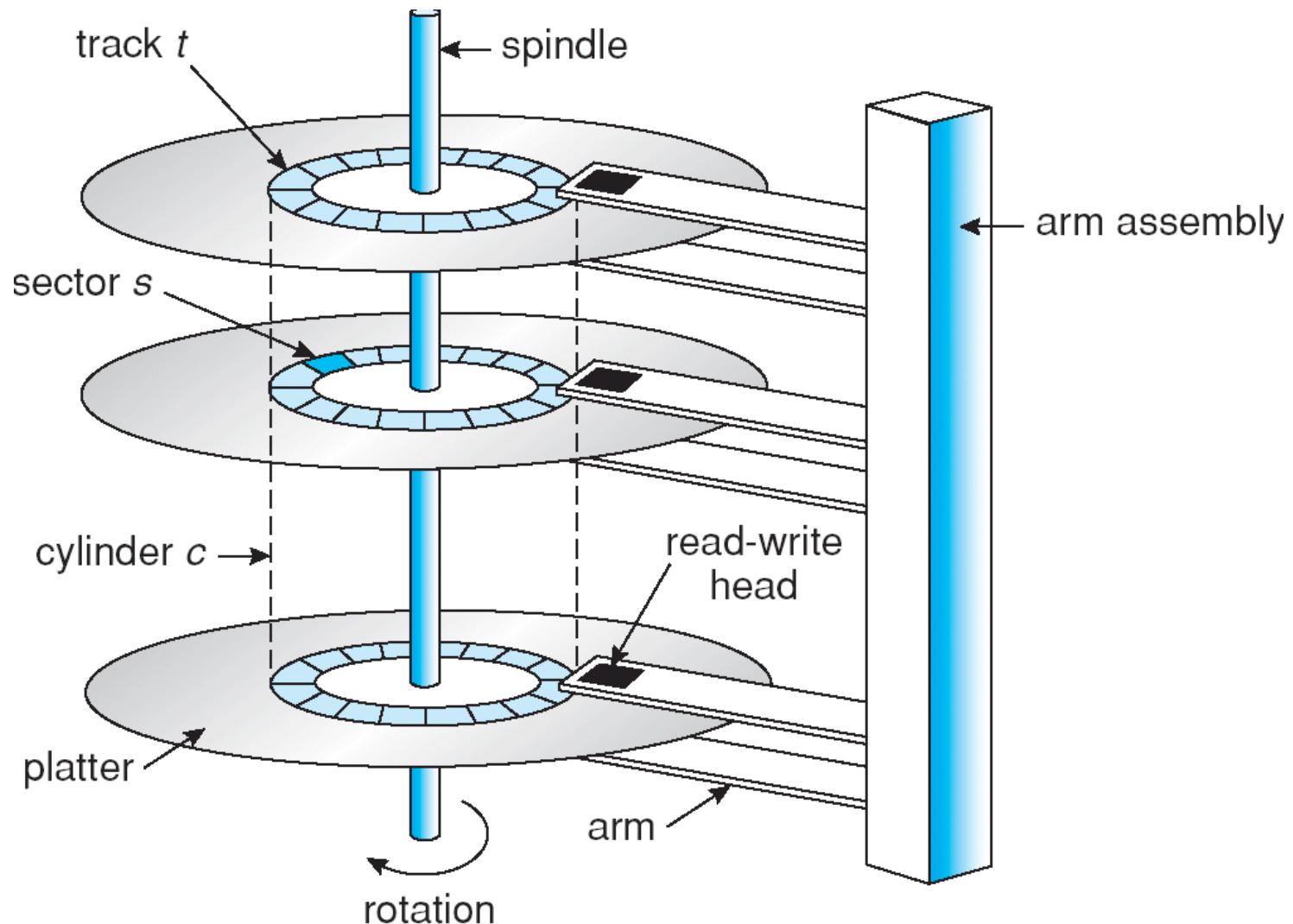
Secondary-Storage
Structure

Alok Kumar Jagadev

Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
 - Drives rotate at 60 to 200 times per second
 - **Transfer rate** is rate at which data flow between drive and computer
 - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
- Disks can be removable
- Drive attached to computer via **I/O bus**
 - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**

Moving-head Disk Mechanism



Overview of Mass Storage Structure (Cont.)

- Magnetic tape
 - Early secondary-storage medium
 - Relatively permanent and holds large quantities of data
 - Access time slow
 - Random access ~1000 times slower than disk
 - Mainly used for backup, storage of infrequently-used data, transfer medium between systems
 - Kept in spool and wound or rewound past read-write head
 - Once data under head, transfer rates comparable to disk
 - 20-200GB typical storage
 - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

Disk Structure

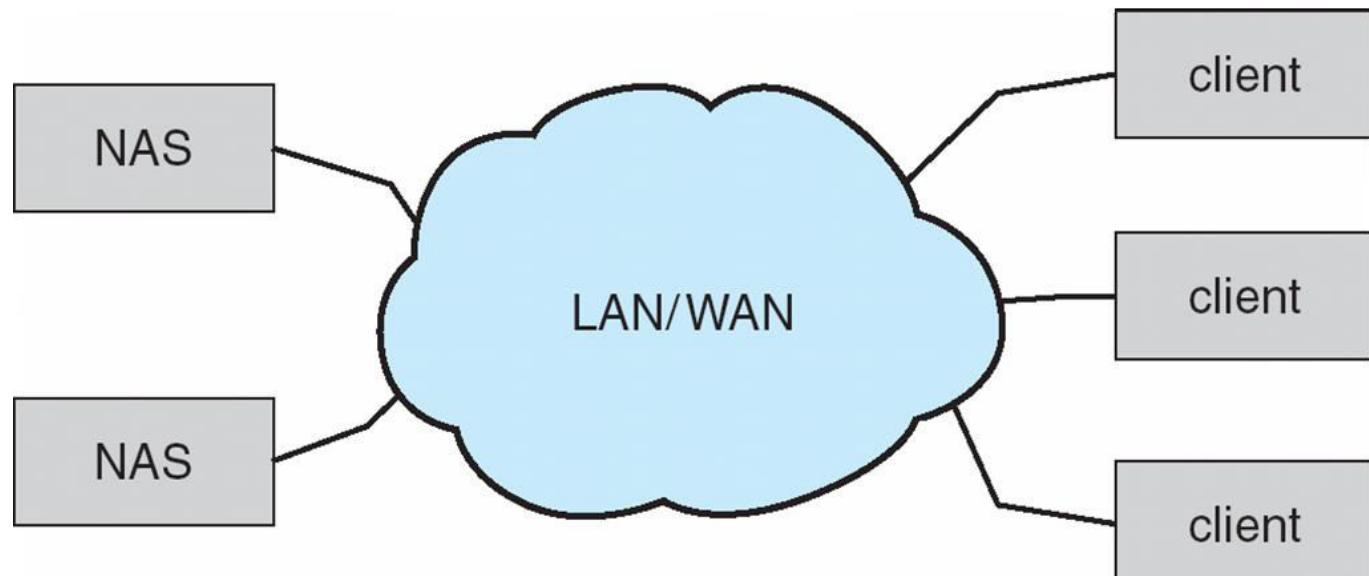
- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the **logical block** is the **smallest unit of transfer**.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
 - **Sector 0** is the first sector of the first track on the outermost cylinder.
 - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Attachment

- Host-attached storage accessed through I/O ports talking to I/O busses
- SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks
 - Each target can have up to 8 **logical units** (disks attached to device controller)
- FC is high-speed serial architecture
 - Can be switched fabric with 24-bit address space – the basis of **storage area networks (SANs)** in which many hosts attach to many storage units
 - Can be **arbitrated loop (FC-AL)** of 126 devices

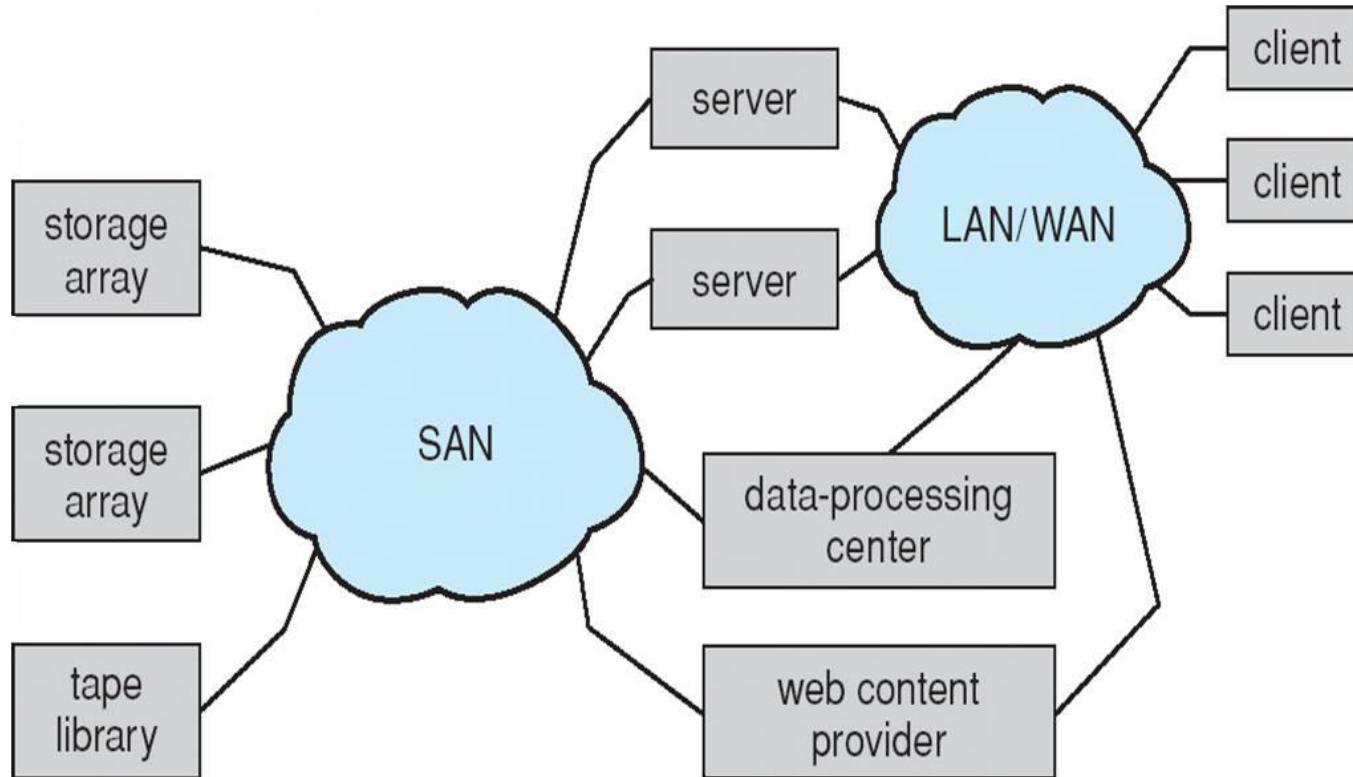
Network-Attached Storage

- Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)
- NFS and CIFS are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage
- New iSCSI protocol uses IP network to carry the SCSI protocol



Storage Area Network

- Common in large storage environments (and becoming more common)
- Multiple hosts attached to multiple storage arrays - flexible



Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
 - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.
 - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time \approx seek distance
- **Disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling (Cont.)

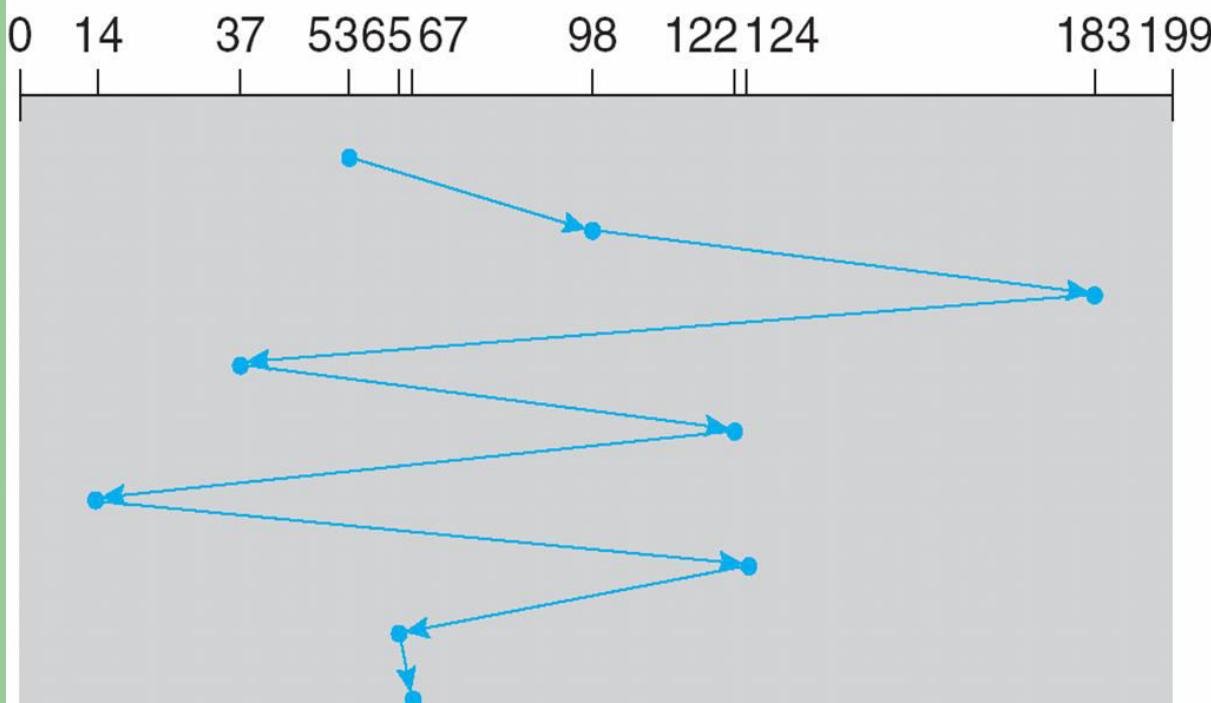
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

FCFS

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



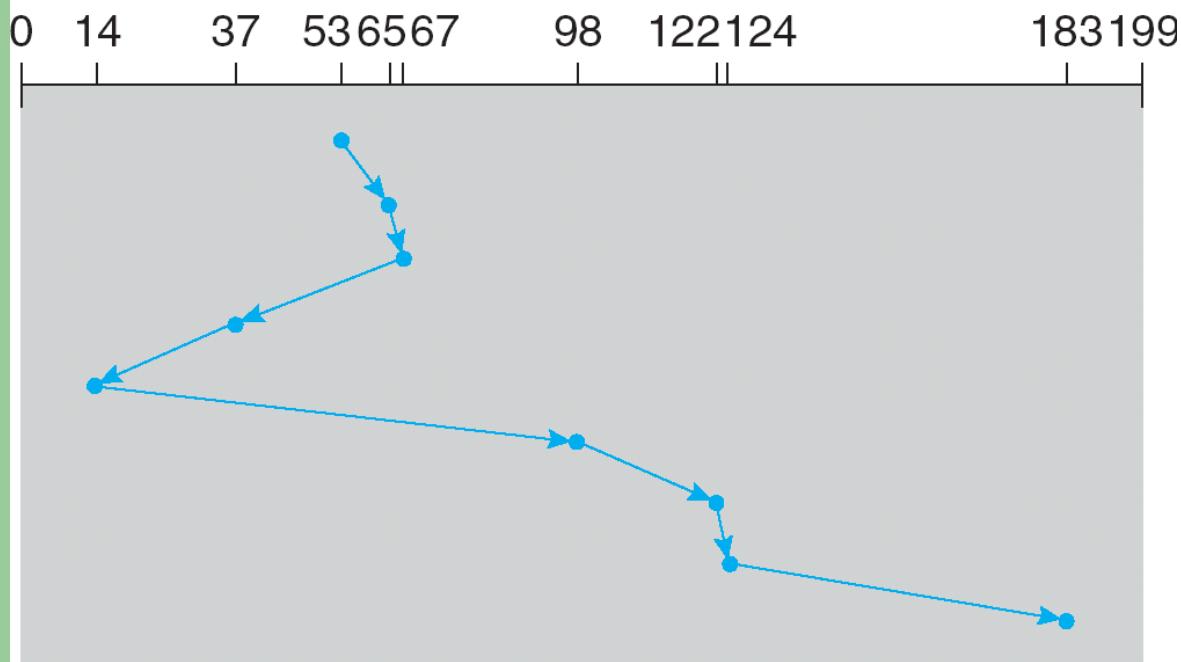
- perform operations in order requested
- no reordering of work queue
- no **starvation**: every request is serviced
- poor performance

Illustration shows total head movement of 640 cylinders

SSTF

queue = 98, 183, 37, 122, 14, 124, 65, 67

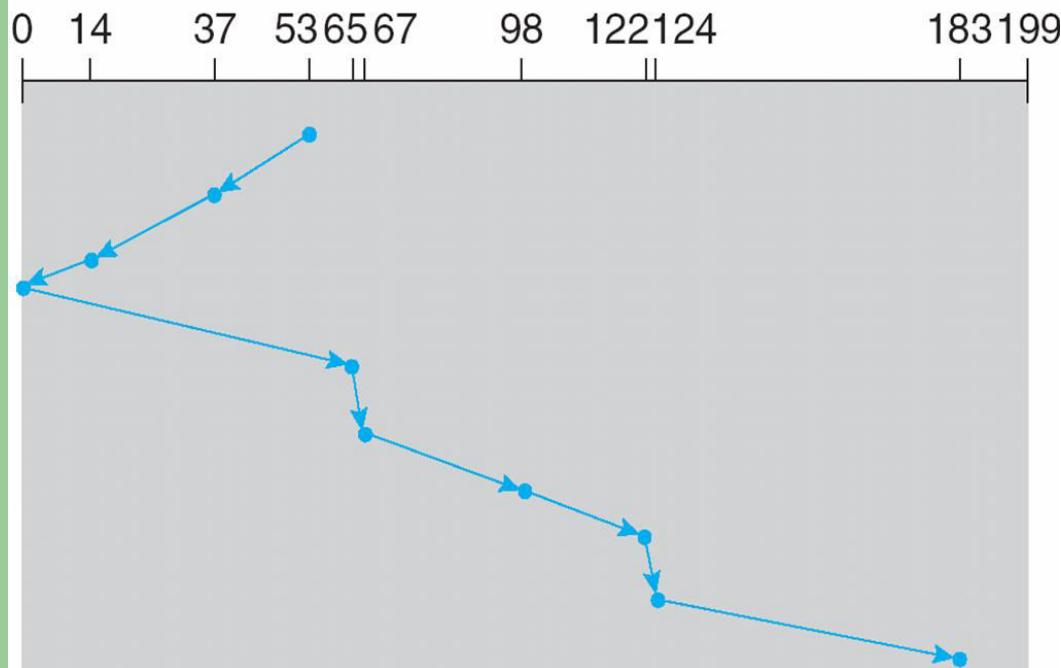
head starts at 53



- Selects the request with the minimum seek time from the current head position.
- **Disadvantages**
 - may cause starvation of some requests.
 - switching directions slows things down
- Illustration shows total head movement of 236 cylinders.

SCAN

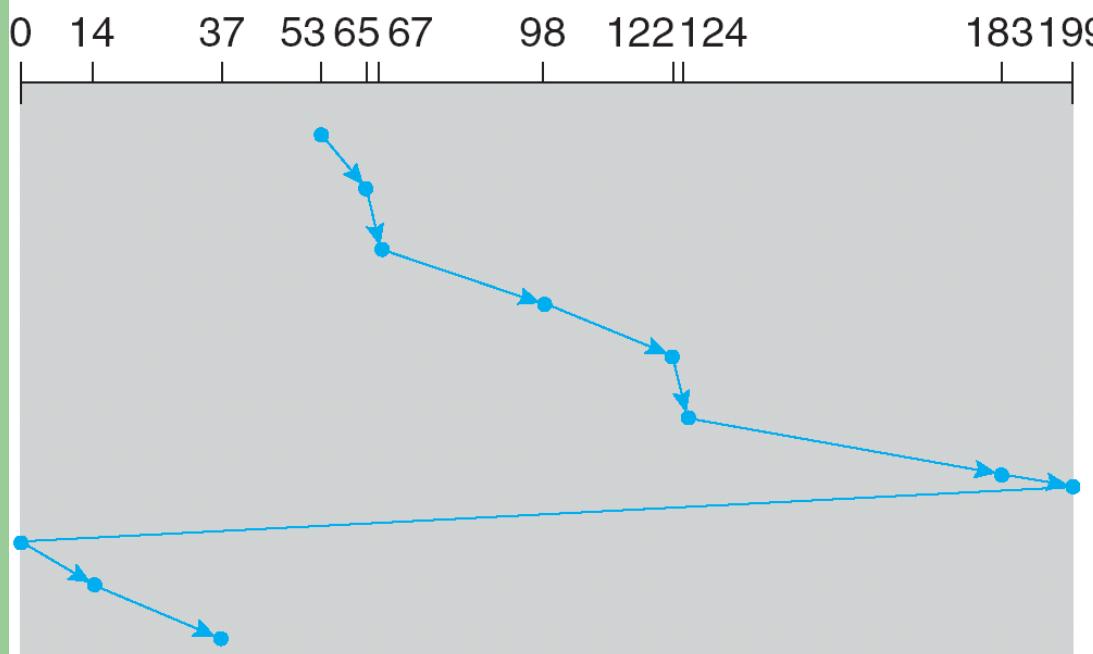
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.

C-SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

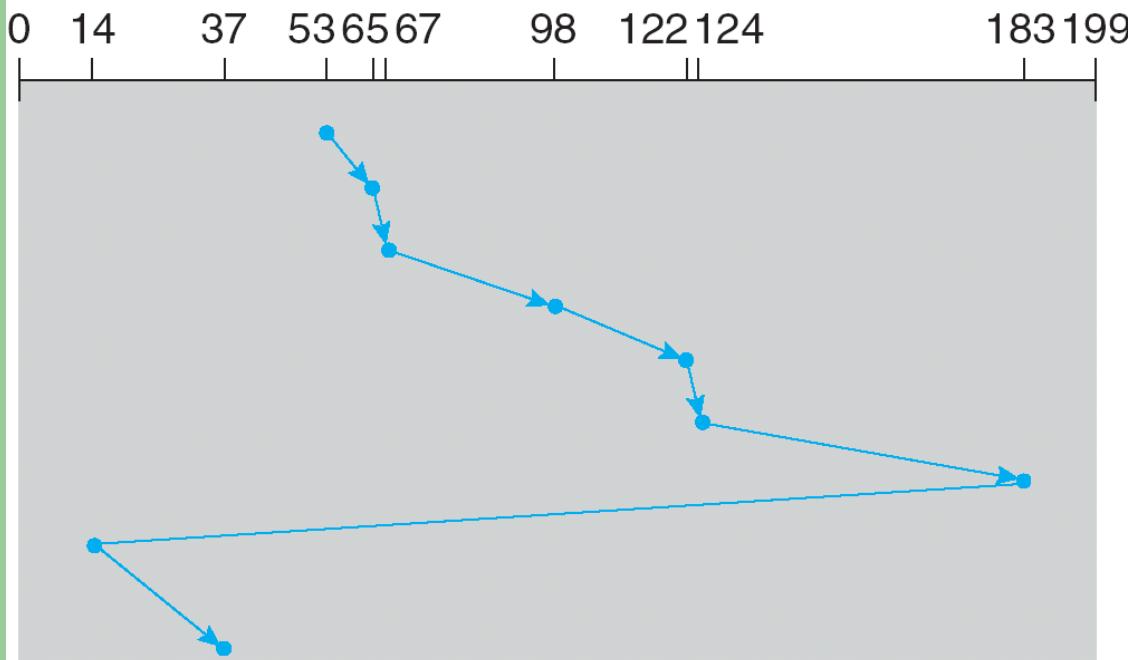


- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

C-LOOK

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without going all the way to the end of the disk.

Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

Another Problem

- Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999.
- The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125.
- The queue of pending requests, in FIFO order, is 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.
- Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk-scheduling algorithms?
 - a) FCFS
 - b) SSTF
 - c) SCAN
 - d) LOOK
 - e) C-SCAN

Solution

- a) a. The FCFS schedule is 143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. The total seek distance is 7081.
- b) The SSTF schedule is 143, 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774. The total seek distance is 1745.
- c) The SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86. The total seek distance is 9769.

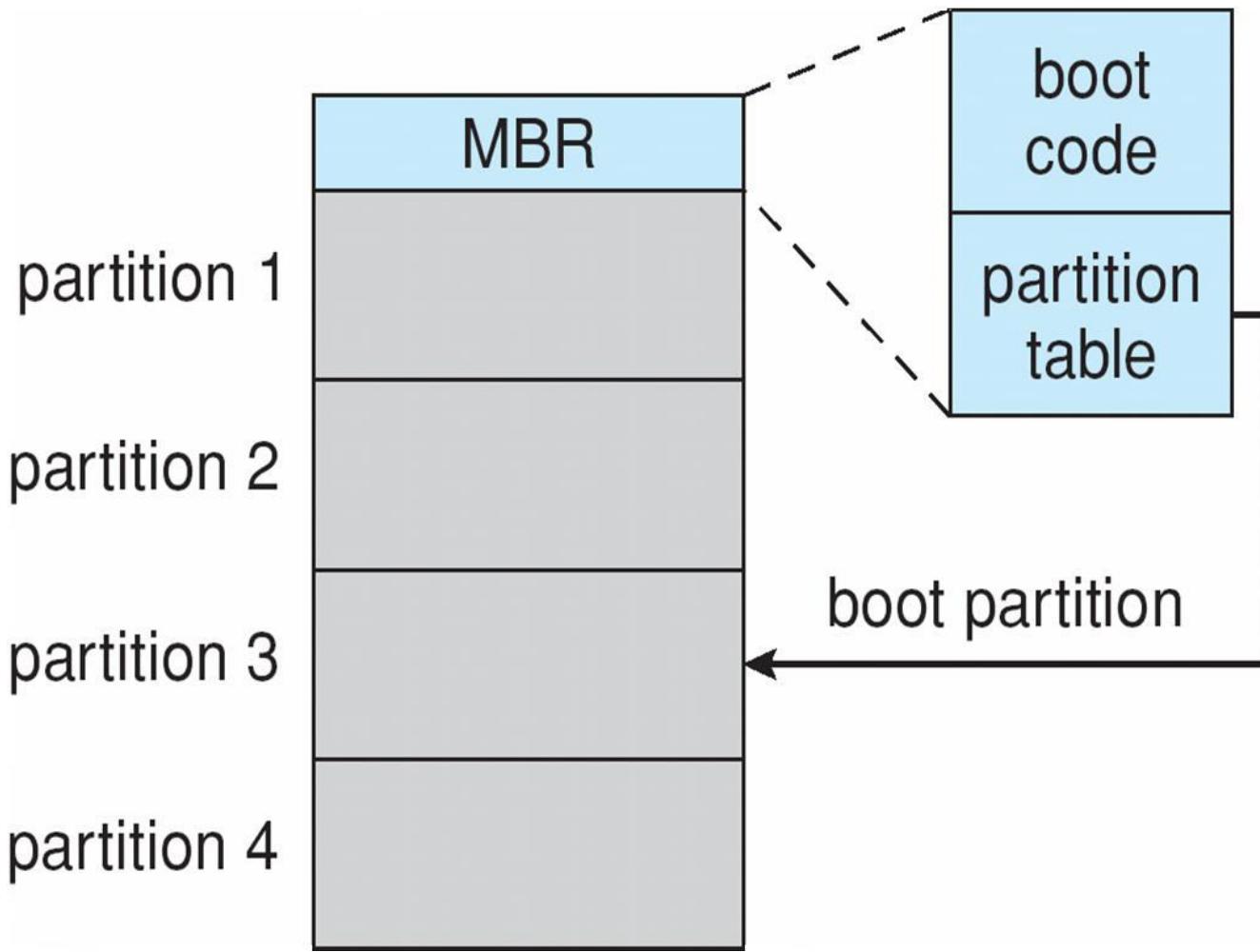
Solution

- d) The LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86. The total seek distance is 3319.
- e) The C-SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 86, 130. The total seek distance is 9813.
- f) The C-LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 86, 130. The total seek distance is 3363.

Disk Management

- *Low-level formatting, or physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
 - *Partition* the disk into one or more groups of cylinders.
 - *Logical formatting* or “making a file system”.
- Boot block initializes system.
 - The bootstrap is stored in ROM.
 - *Bootstrap loader* program.
- Methods such as *sector sparing* used to handle bad blocks.

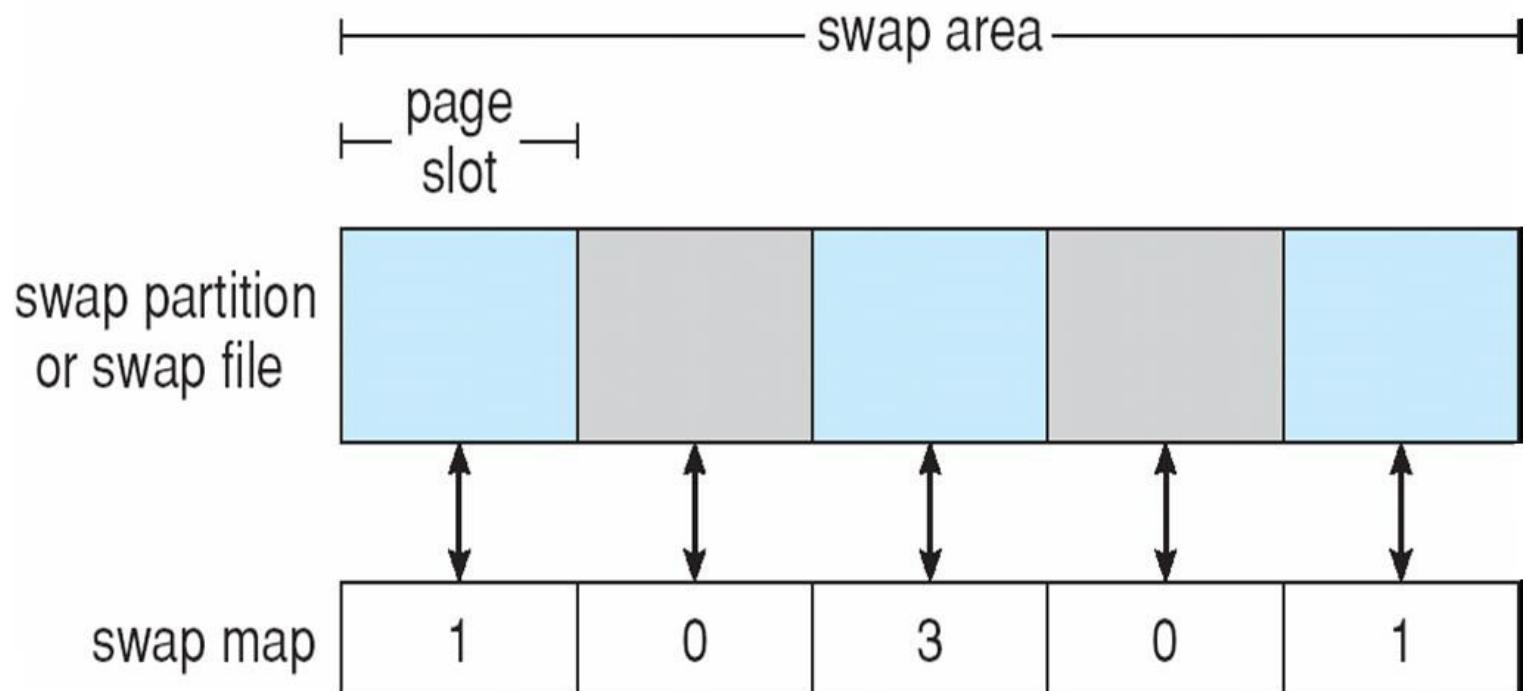
Booting from a Disk in Windows 2000



Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition.
- Swap-space management
 - 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment*.
 - Kernel uses *swap maps* to track swap-space use.
 - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.

Data Structures for Swapping on Linux Systems



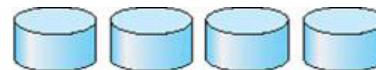
RAID Structure

- **RAID** – multiple disk drives provides **reliability** via **redundancy**.
- RAID is arranged into six different levels.

RAID (cont)

- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
- Disk striping uses a group of disks as one storage unit.
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
 - *Mirroring* or *shadowing* keeps duplicate of each disk.
 - *Block interleaved parity* uses much less redundancy.

RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.

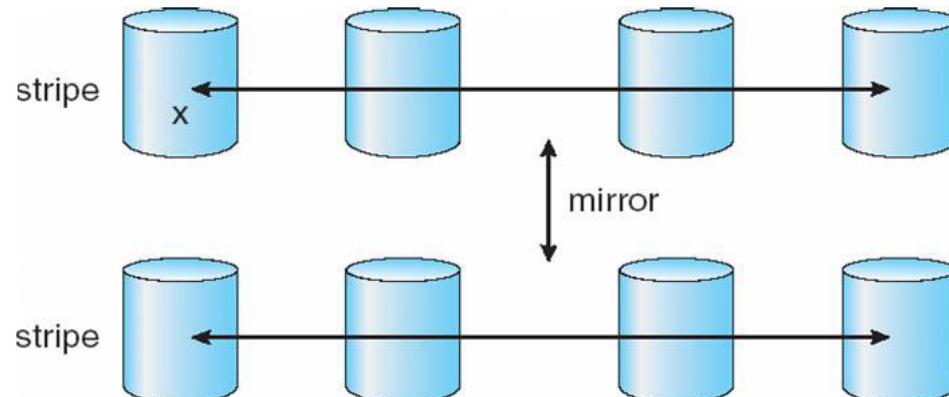


(f) RAID 5: block-interleaved distributed parity.

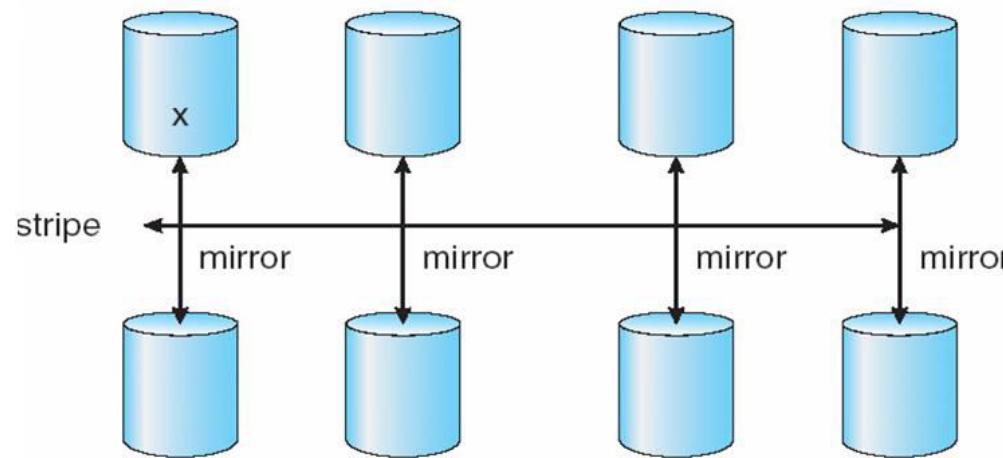


(g) RAID 6: P + Q redundancy.

RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

Stable-Storage Implementation

- Write-ahead log scheme requires stable storage.
- To implement stable storage:
 - Replicate information on more than one nonvolatile storage media with independent failure modes.
 - Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage.
- Generally, tertiary storage is built using *removable media*
- Common examples of removable media are floppy disks and CD-ROMs; other types are available.

Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.
 - Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.
 - Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.

Removable Disks (Cont.)

- A magneto-optic disk records data on a rigid platter coated with magnetic material.
 - Laser heat is used to amplify a large, weak magnetic field to record a bit.
 - Laser light is also used to read data (Kerr effect).
 - The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.
- Optical disks do not use magnetism; they employ special materials that are altered by laser light.

WORM Disks

- The data on read-write disks can be modified over and over.
- WORM (“Write Once, Read Many Times”) disks can be written only once.
- Thin aluminum film sandwiched between two glass or plastic platters.
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.
- Very durable and reliable.
- *Read Only* disks, such ad CD-ROM and DVD, com from the factory with the data pre-recorded.

Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
 - stacker – library that holds a few tapes
 - silo – library that holds thousands of tapes
- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.

Operating System Issues

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications
- For hard disks, the OS provides two abstraction:
 - Raw device – an array of data blocks.
 - File system – the OS queues and schedules the interleaved requests from several applications.

Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device.
- Usually the tape drive is reserved for the exclusive use of that application.
- Since the OS does not provide file system services, the application must decide how to use the array of blocks.
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it.

Tape Drives

- The basic operations for a tape drive differ from those of a disk drive.
- **locate** positions the tape to a specific logical block, not an entire track (corresponds to **seek**).
- The **read position** operation returns the logical block number where the tape head is.
- The **space** operation enables relative motion.
- Tape drives are “append-only” devices; updating a block in the middle of the tape also effectively erases everything beyond that block.
- An EOT mark is placed after a block that is written.

File Naming

- The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.
- Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.
- Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.

Hierarchical Storage Management (HSM)

- A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage to incorporate tertiary storage — usually implemented as a jukebox of tapes or removable disks.
- Usually incorporate tertiary storage by extending the file system.
 - Small and frequently used files remain on disk.
 - Large, old, inactive files are archived to the jukebox.
- HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

Speed

- Two aspects of speed in tertiary storage are bandwidth and latency.
- Bandwidth is measured in bytes per second.
 - Sustained bandwidth – average data rate during a large transfer; # of bytes/transfer time.
Data rate when the data stream is actually flowing.
 - Effective bandwidth – average over the entire I/O time, including **seek** or **locate**, and cartridge switching.
Drive's overall data rate.

Speed (Cont.)

- Access latency – amount of time needed to locate data.
 - Access time for a disk – move the arm to the selected cylinder and wait for the rotational latency; < 35 milliseconds.
 - Access on tape requires winding the tape reels until the selected block reaches the tape head; tens or hundreds of seconds.
 - Generally say that random access within a tape cartridge is about a thousand times slower than random access on disk.
- The low cost of tertiary storage is a result of having many cheap cartridges share a few expensive drives.
- A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.

Reliability

- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
- An optical cartridge is likely to be more reliable than a magnetic disk or tape.
- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

Cost

- Main memory is much more expensive than disk storage
- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.
- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.

Operating Systems

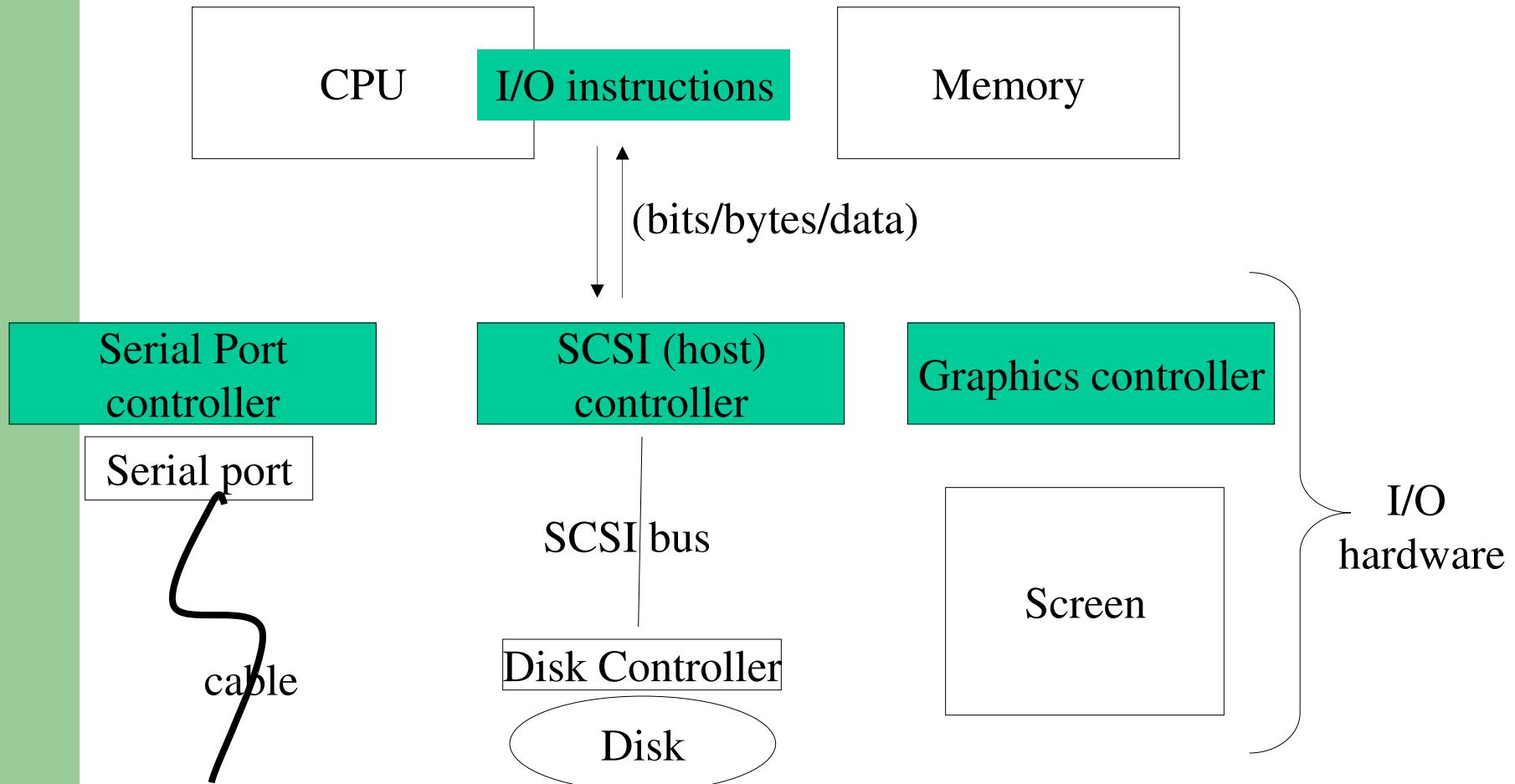
I/O System

Alok Kumar Jagadev

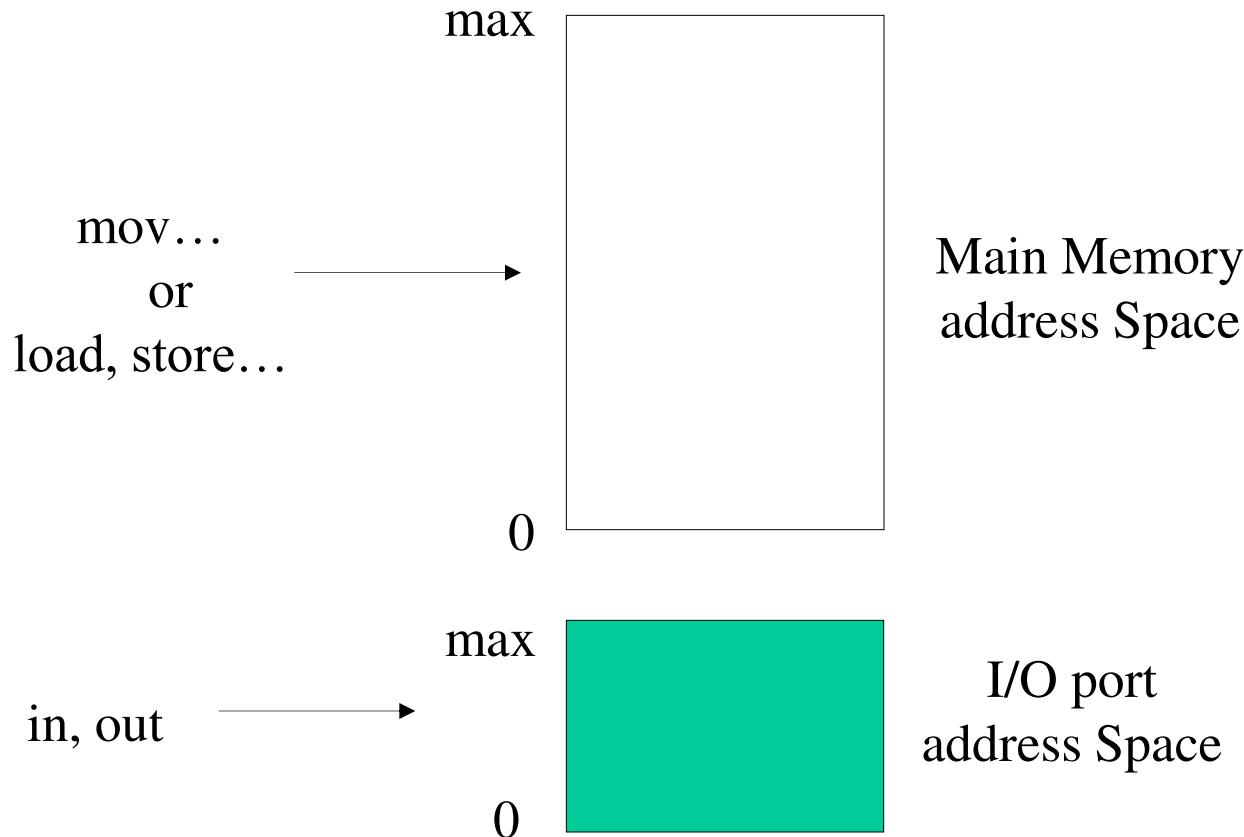
I/O Hardware

- Incredible variety of I/O devices
- Common concepts
 - Port (a connection point through computer accesses a device)
 - Bus (daisy chain or shared direct access)
 - Medium over which signals are sent/received
 - Controller (host adapter)
 - Chip that can be accessed by CPU and that controls the device/port/bus
 - PCI controller, Serial Port controller, keyboard controller, ...
- I/O instructions control devices
- Devices have addresses, used by
 - Direct I/O instructions
 - **Memory-mapped I/O**

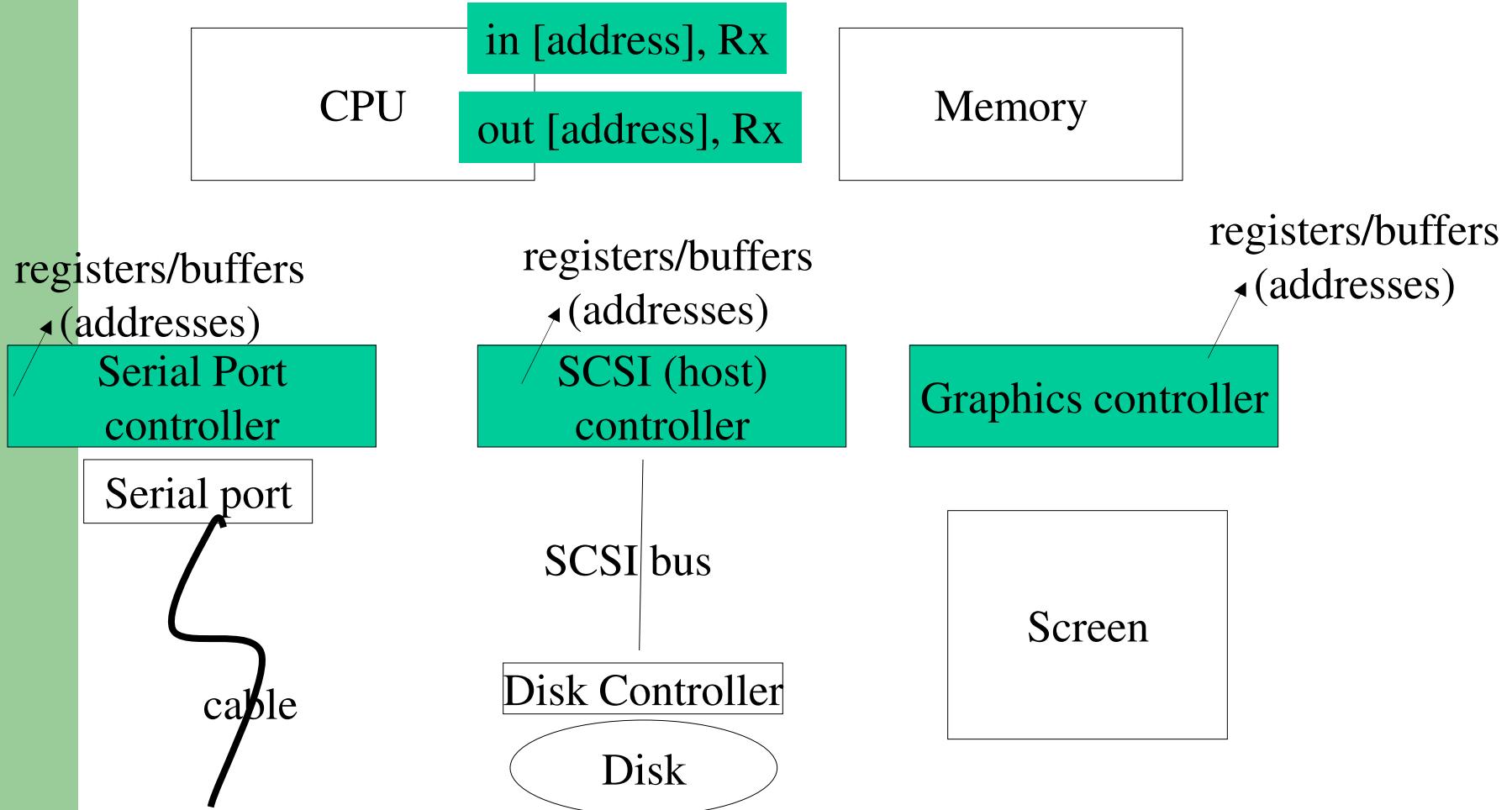
I/O hardware concepts



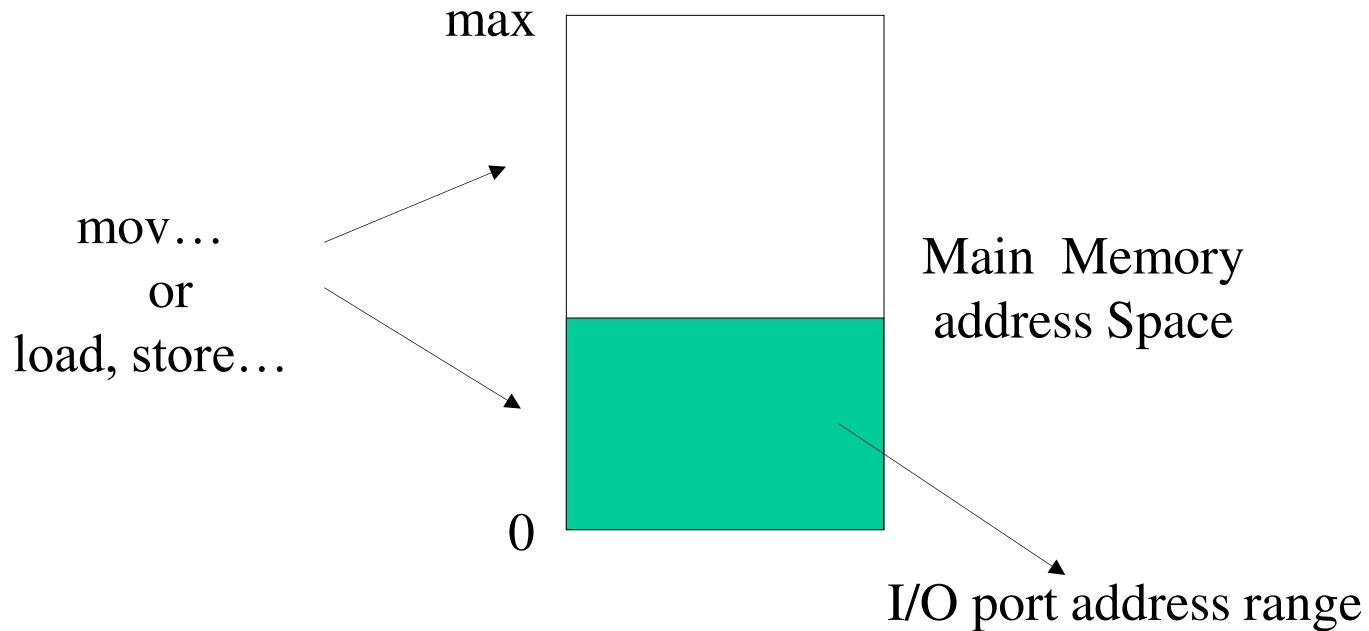
I/O hardware concepts: direct I/O



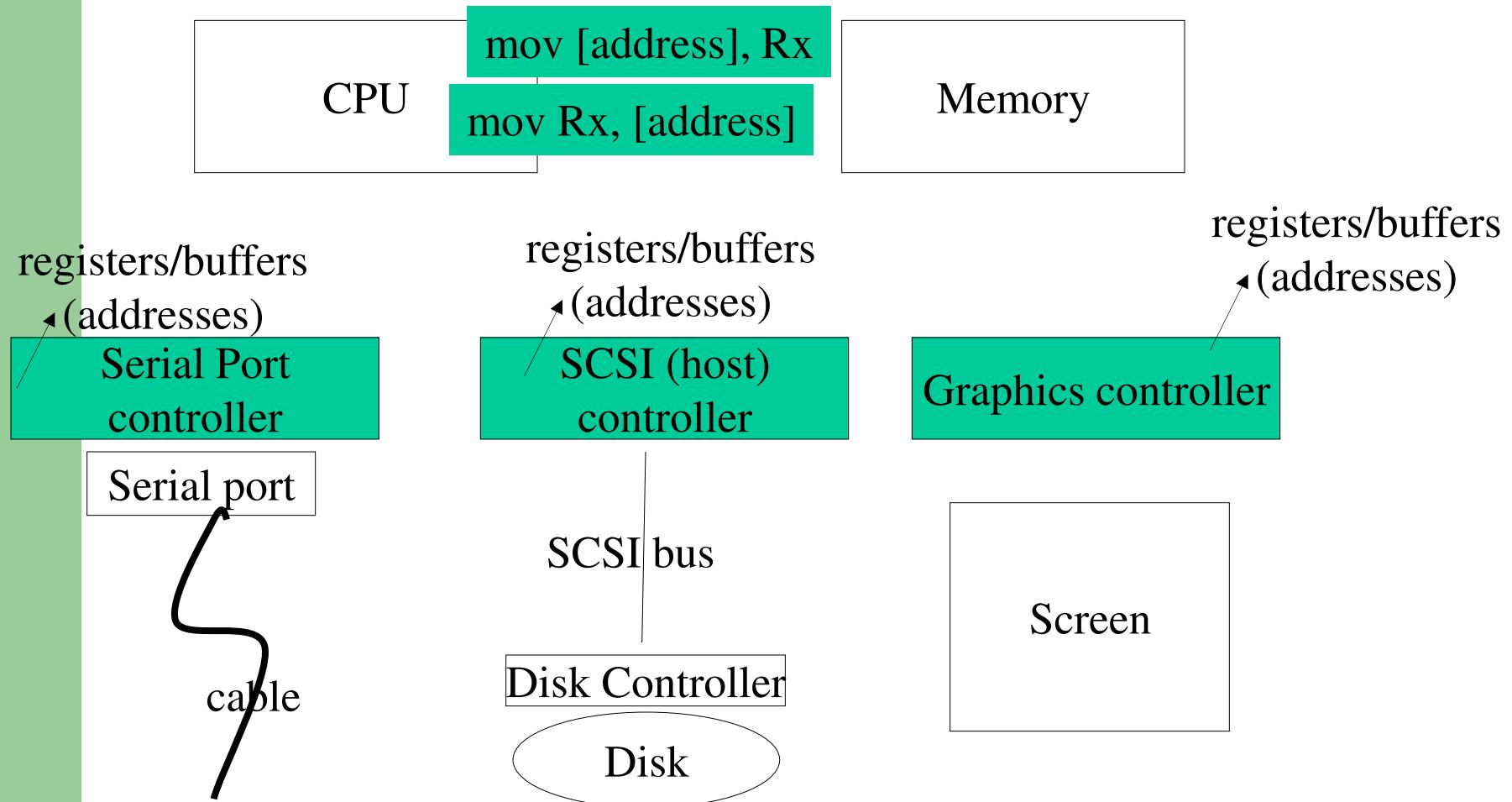
I/O hardware concepts: direct I/O



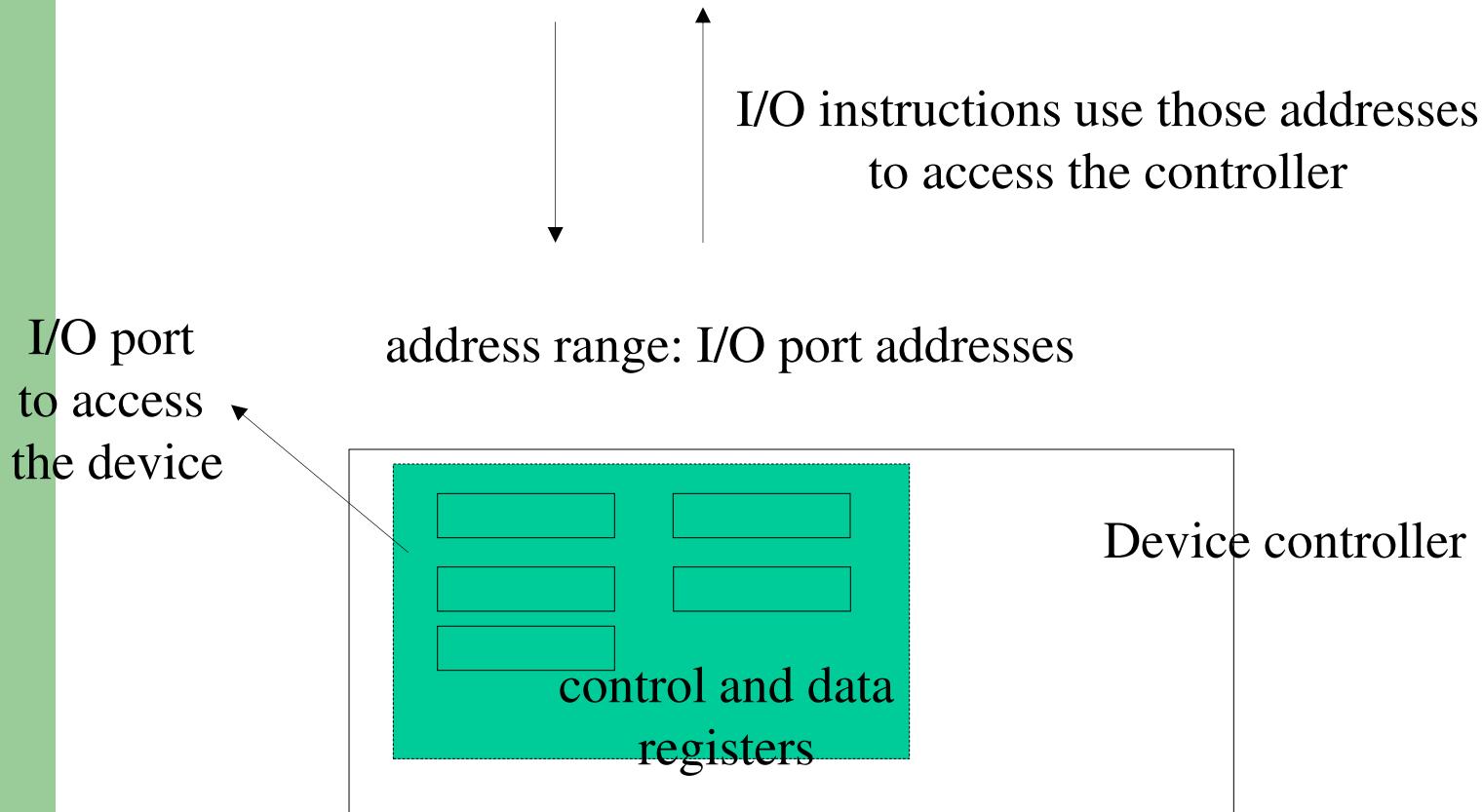
I/O hardware concepts: memory mapped I/O



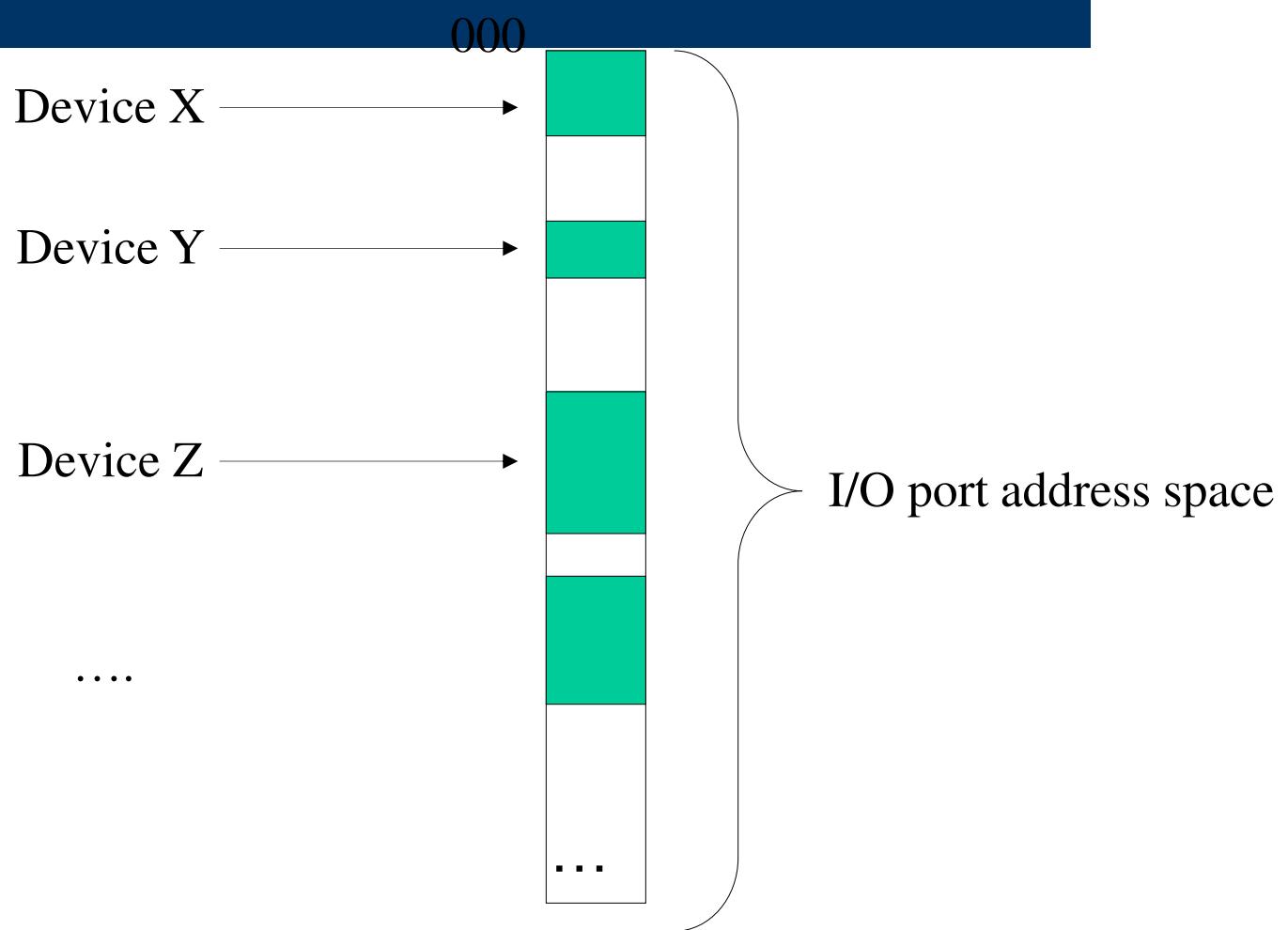
I/O hardware concepts: memory mapped I/O



I/O port concept



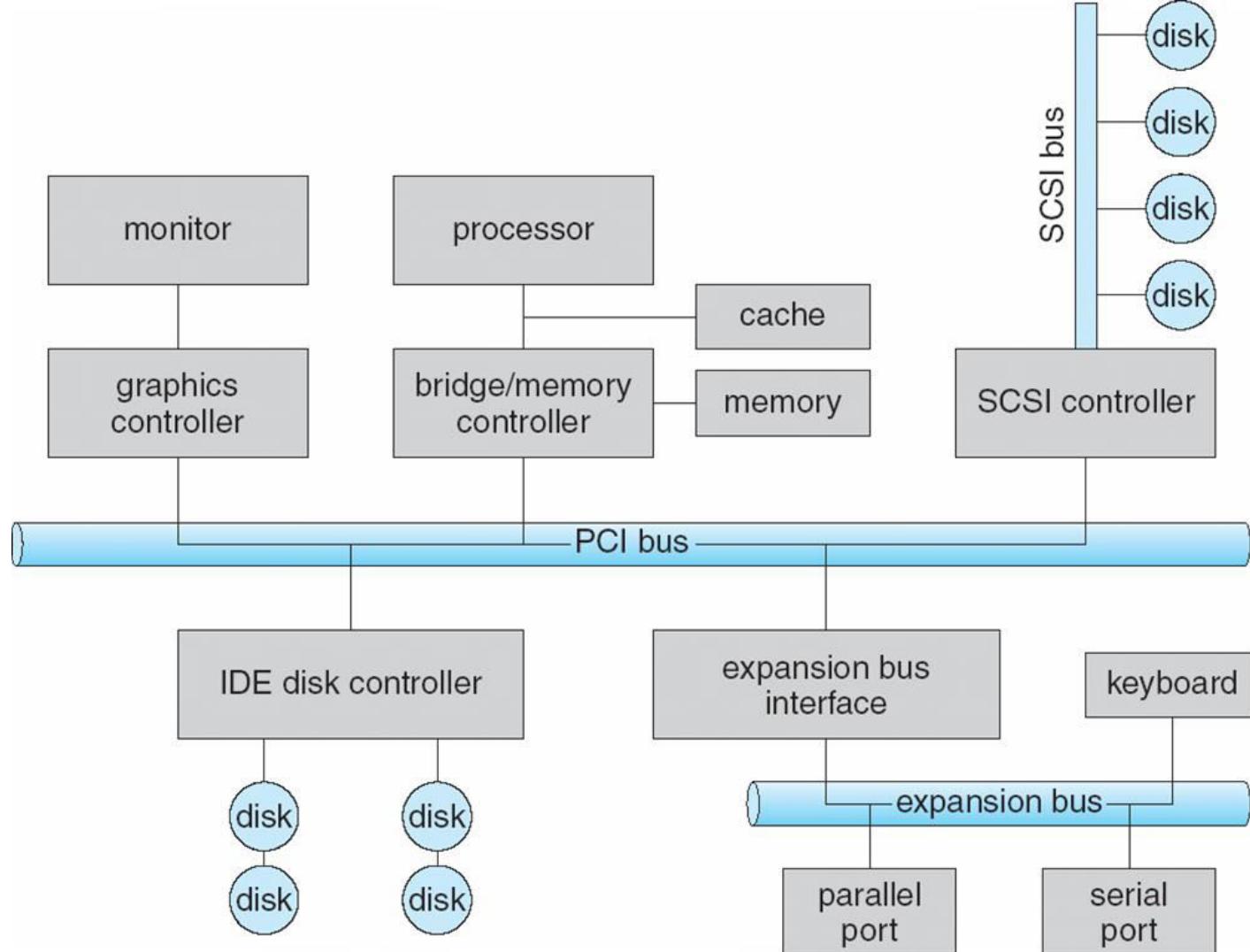
I/O port addresses



Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

A Typical PC Bus Structure



Interacting with the Device controller

- Host (CPU+Memory) and Device Controller interaction (data transfers and control) can be in one of 3 ways:
 - Polling
 - Interrupt driven I/O
 - Interrupt driven with help of DMA

Interrupts vs. Polling

- An interrupt is an external or internal event that interrupts the microcontroller
 - To inform it that a device needs its service
- A single microcontroller can serve several devices by two ways
 - **Interrupts**
 - Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal
 - Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device

Interrupts vs. Polling (cont.)

- The program which is associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler
- **Polling**
 - The microcontroller continuously monitors the status of a given device
 - ex. JNB TF, target
 - When the conditions met, it performs the service
 - After that, it moves on to monitor the next device until every one is serviced
 - Polling can monitor the status of several devices and serve each of them as certain conditions are met
 - The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service

Interrupts vs. Polling (cont.)

- The advantage of interrupts is:
 - The microcontroller can serve many devices (not all at the same time)
 - Each device can get the attention of the microcontroller based on the assigned priority
 - For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion
 - The microcontroller can also ignore (mask) a device request for service
 - This is not possible for the polling method

Interrupt Service Routine

- For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler
 - When an interrupt is invoked, the microcontroller runs the interrupt service routine
 - There is a fixed location in memory that holds the address of its ISR
 - The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table

Steps in Executing an Interrupt

- Upon activation of an interrupt, the microcontroller goes through:
 - It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack
 - It also saves the current status of all the registers internally (not on the stack)
 - It jumps to a fixed location in memory, called the interrupt vector table, that holds the address of the ISR

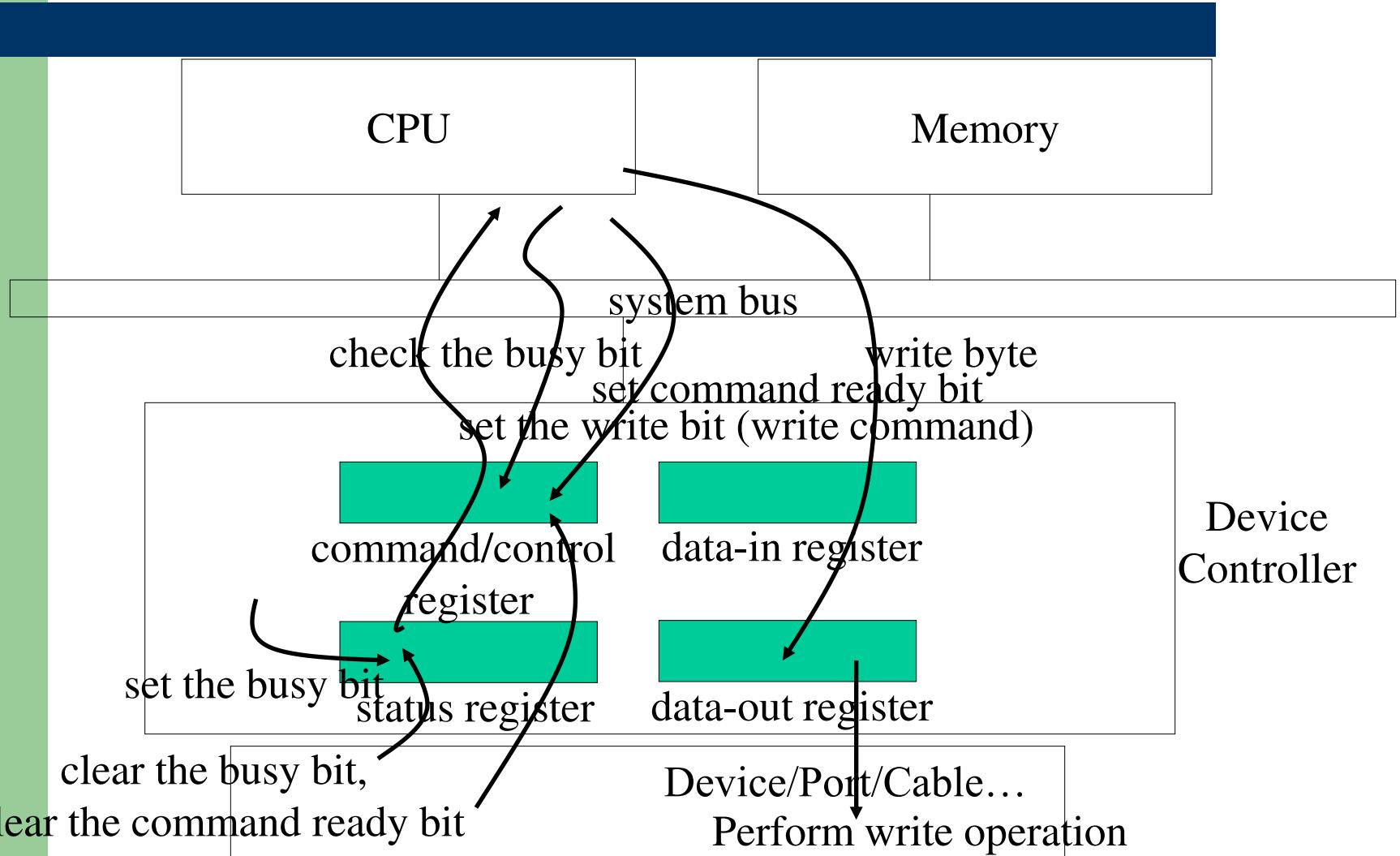
Steps in Executing an Interrupt (cont.)

- It gets the address of the ISR from the interrupt vector table and jumps to ISR
 - It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt)
- Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted
 - It gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC
 - It starts to execute from that address

Six Interrupts in 8051

- Six interrupts are allocated as follows
 - Reset – power-up reset
 - Two interrupts are set aside for the timers:
 - One for timer 0 and one for timer 1
 - Two interrupts are set aside for hardware external interrupts
 - P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2)
 - Serial communication has a single interrupt that belongs to both receive and transfer

Example: polling based writing



Example: polling based writing

CPU

1. read and check the busy bit
2. if busy go to 1.
3. set the write bit in command register
4. Write byte (word) into data out register
5. Set the command ready bit
6. Go to 1(maybe after doing something else)

Controller

- notices command ready bit set
- set the busy bit
- controller reads the command (it is write command), gets byte from data out register and write the byte out (this may take time)
- clears the busy bit
- clears the command ready bit
- clears the error bit

Transferring Data between host and device: Polling

- Determines state of device
 - command-ready
 - busy
 - Error
- Busy-wait cycle to wait for I/O from device

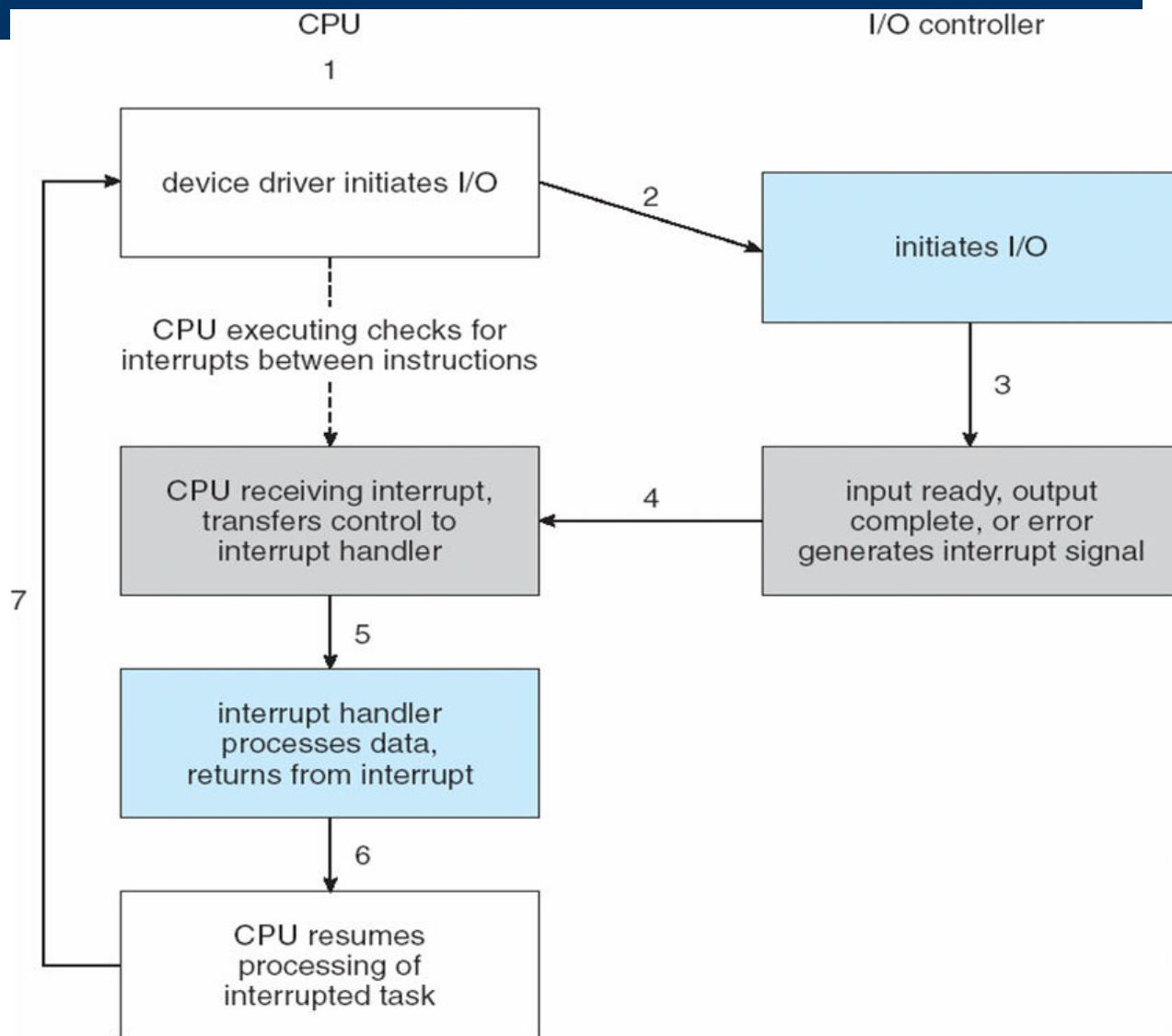
Transferring Data between host and device: Interrupts

- CPU Interrupt-request line triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to the correct handler
 - Based on priority
 - Some nonmaskable
- Interrupt mechanism also used for exceptions

Interrupts

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
 - Based on priority
 - Some **nonmaskable**
- Interrupt mechanism also used for exceptions

Interrupt-Driven I/O Cycle



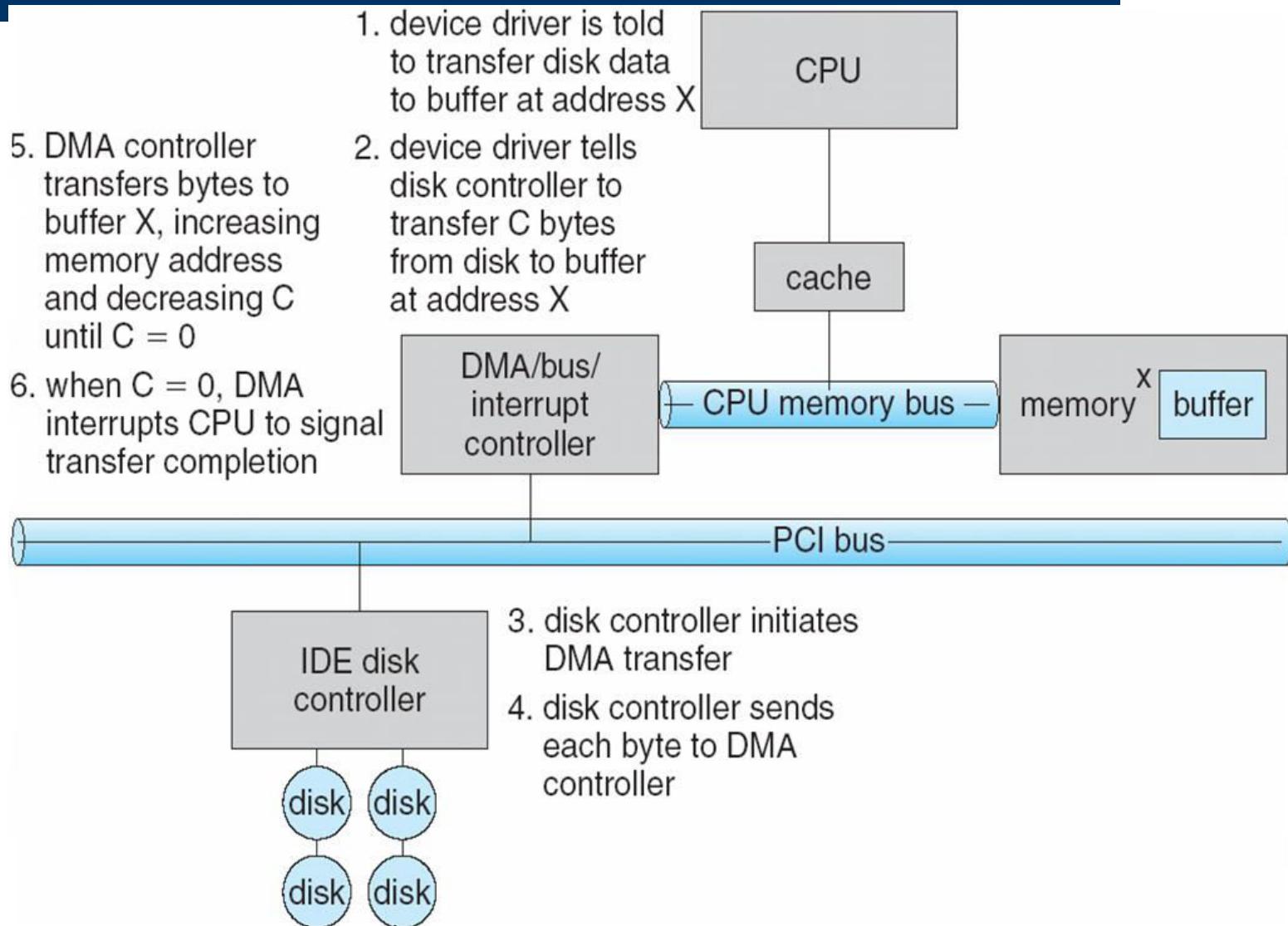
Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Direct Memory Access

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

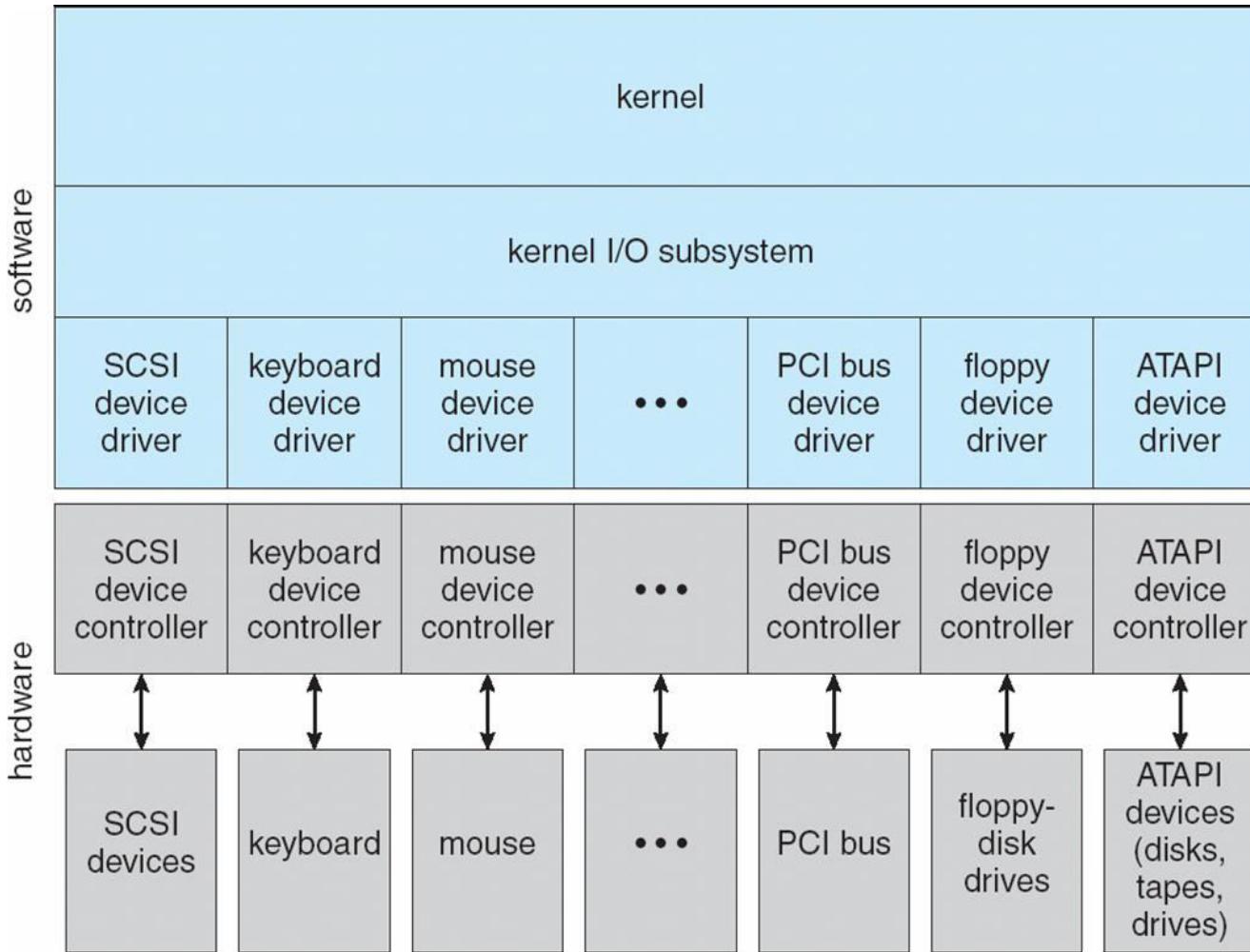
Six Step Process to Perform DMA Transfer



Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
 - Character-stream or block
 - Sequential or random-access
 - Sharable or dedicated
 - Speed of operation
 - read-write, read only, or write only

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

Block and Character Devices

- Block devices include disk drives
 - Commands include read, write, seek
 - Raw I/O or file-system access
 - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
 - Commands include get, put
 - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9x/2000 include socket interface
 - Separates network protocol from network operation
 - Includes select functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

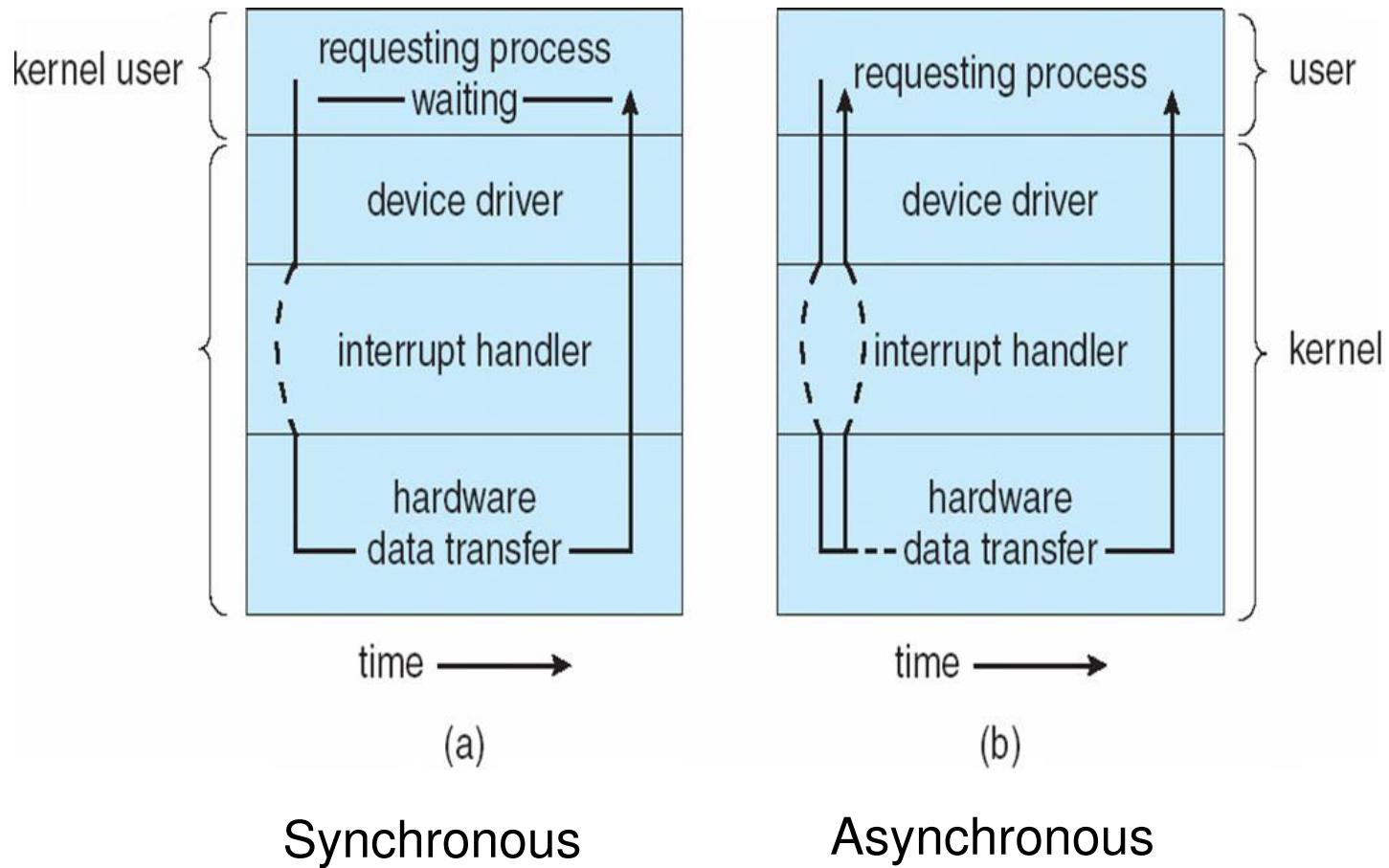
Clocks and Timers

- Provide current time, elapsed time, timer
- **Programmable interval timer** used for timings, periodic interrupts
- ioctl (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O

- **Blocking** - process suspended until I/O completed
 - Easy to use and understand
 - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
 - User interface, data copy (buffered I/O)
 - Implemented via multi-threading
 - Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
 - Difficult to use
 - I/O subsystem signals process when I/O completed

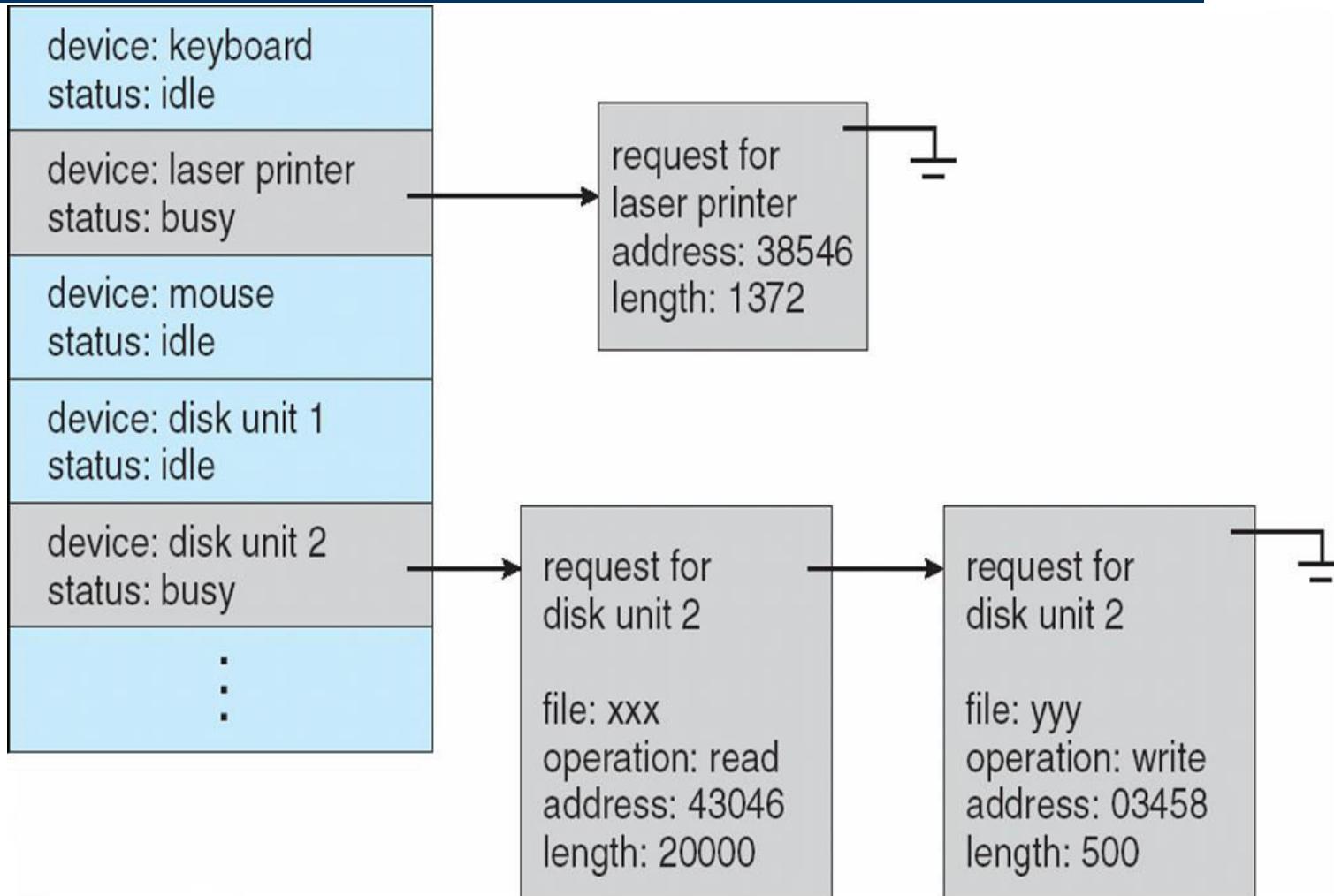
Two I/O Methods



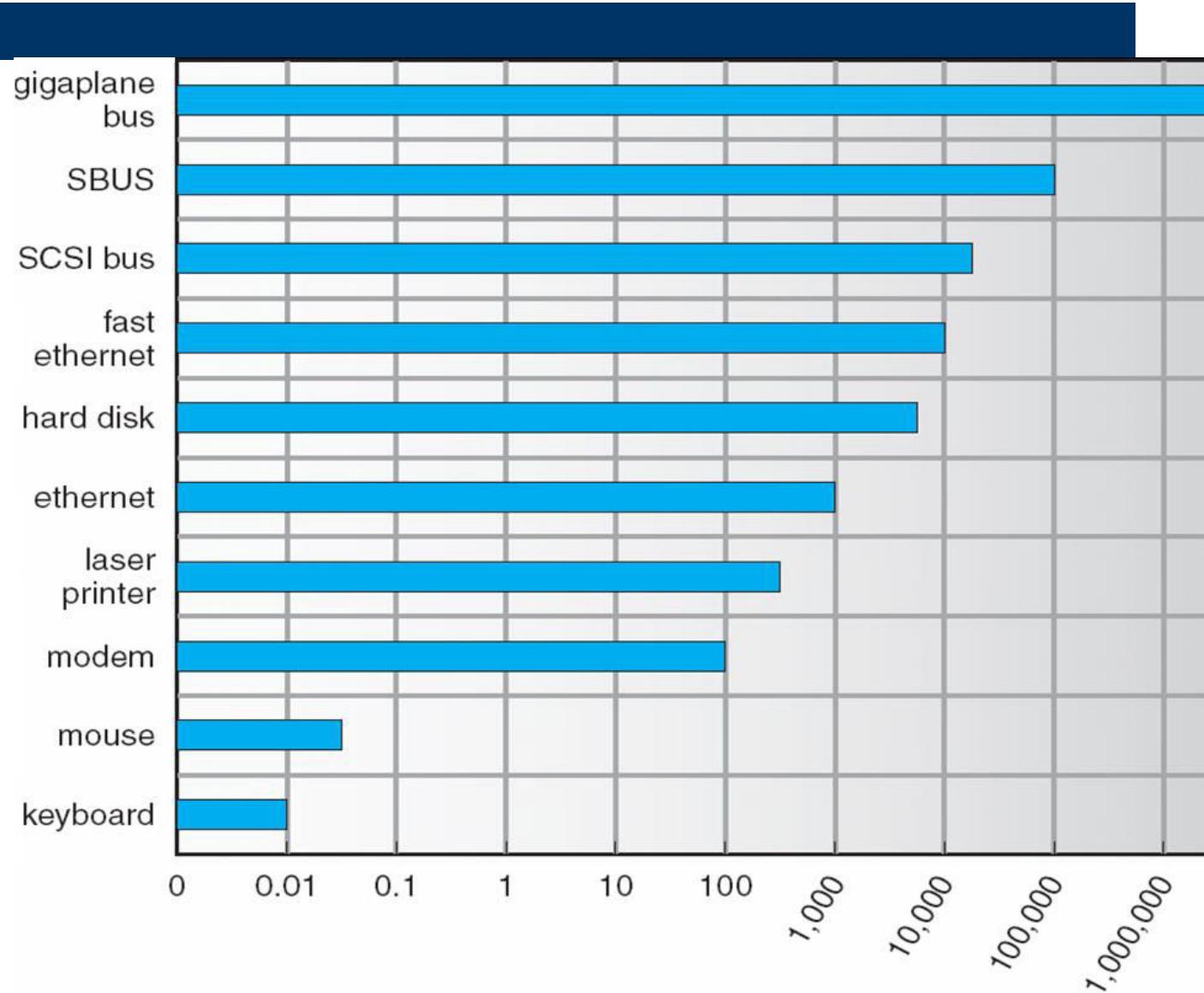
Kernel I/O Subsystem

- **Scheduling**
 - Some I/O request ordering via per-device queue
 - Some OSs try fairness
- **Buffering** - store data in memory while transferring between devices
 - To cope with device speed mismatch
 - To cope with device transfer size mismatch
 - To maintain “copy semantics”

Device-status Table



Sun Enterprise 6000 Device-Transfer Rates



Kernel I/O Subsystem

- **Caching** - fast memory holding copy of data
 - Always just a copy
 - Key to performance
- **Spooling** - hold output for a device
 - If device can serve only one request at a time
 - i.e., Printing
- **Device reservation** - provides exclusive access to a device
 - System calls for allocation and deallocation
 - Watch out for deadlock

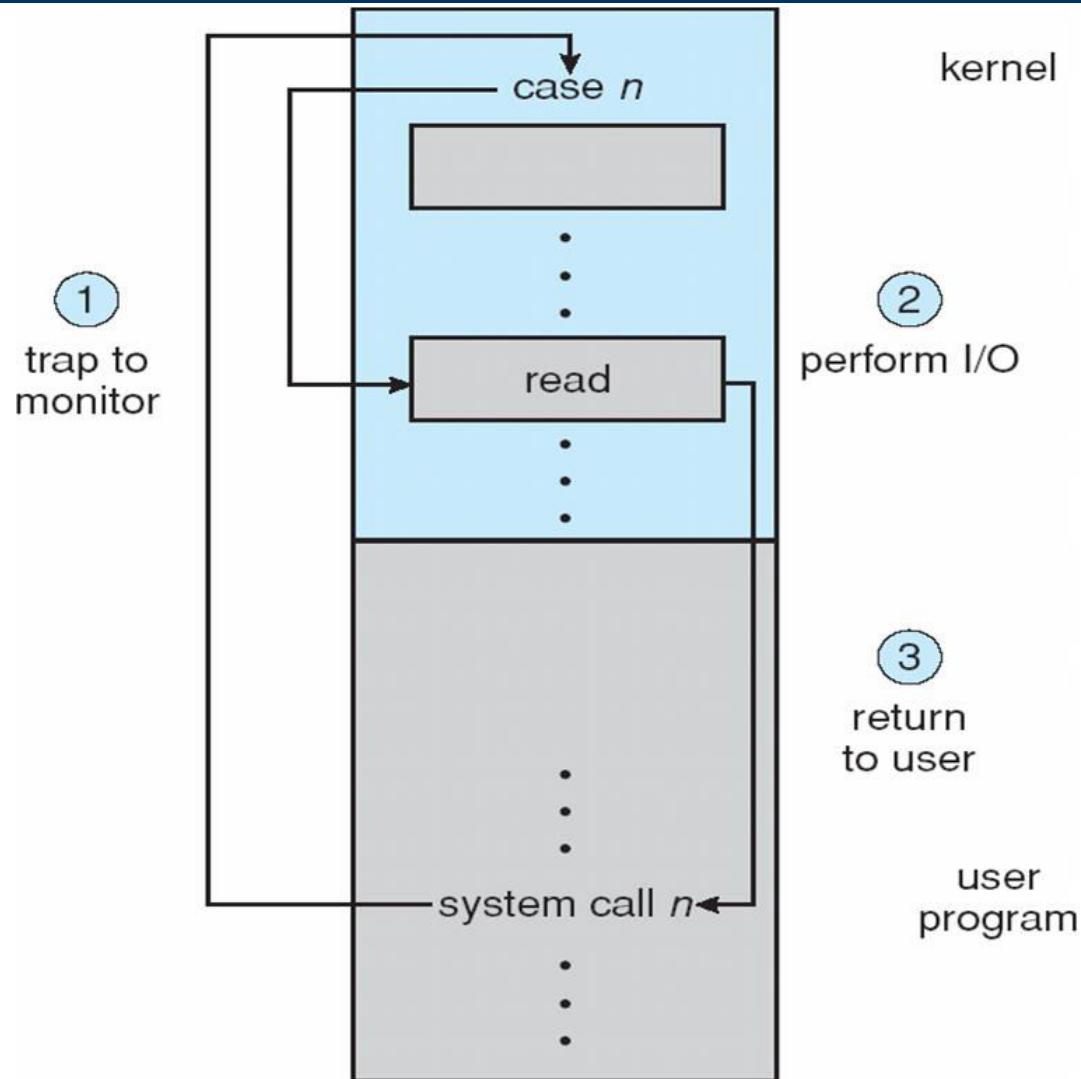
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Must return an error number or code when I/O request fails
- System error logs hold problem reports

I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
 - All I/O instructions defined to be privileged
 - I/O must be performed via system calls
 - Memory-mapped and I/O port memory locations must be protected too

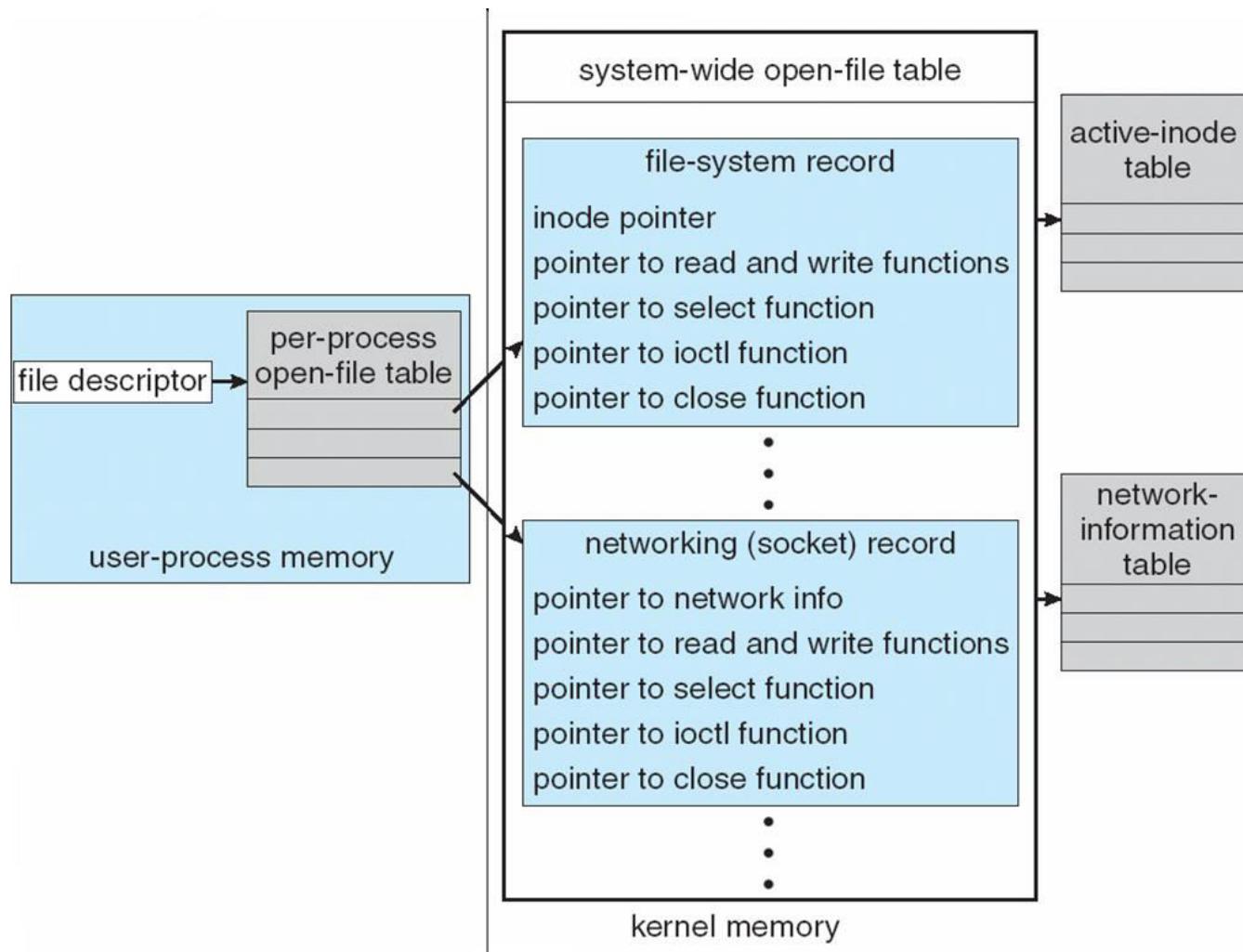
Use of a System Call to Perform I/O



Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O

UNIX I/O Kernel Structure



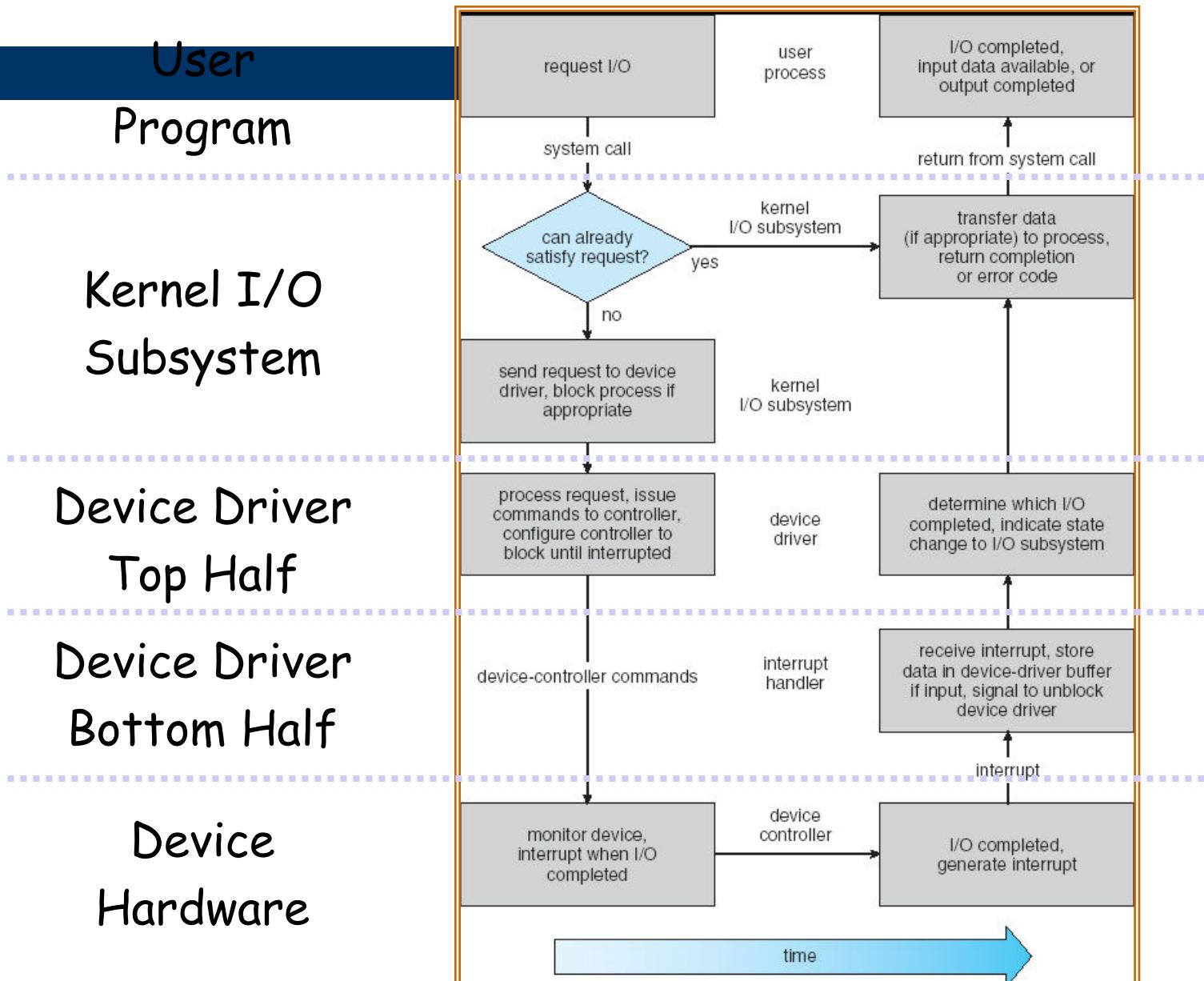
I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
 - Determine device holding file
 - Translate name to device representation
 - Physically read data from disk into buffer
 - Make data available to requesting process
 - Return control to process

Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
 - **Top half:** accessed in call path from system calls
 - implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
 - This is the kernel's interface to the device driver
 - Top half will *start* I/O to device, may put thread to sleep until finished
 - **Bottom half:** run as interrupt routine
 - Gets input or transfers next block of output
 - May wake sleeping threads if I/O now complete

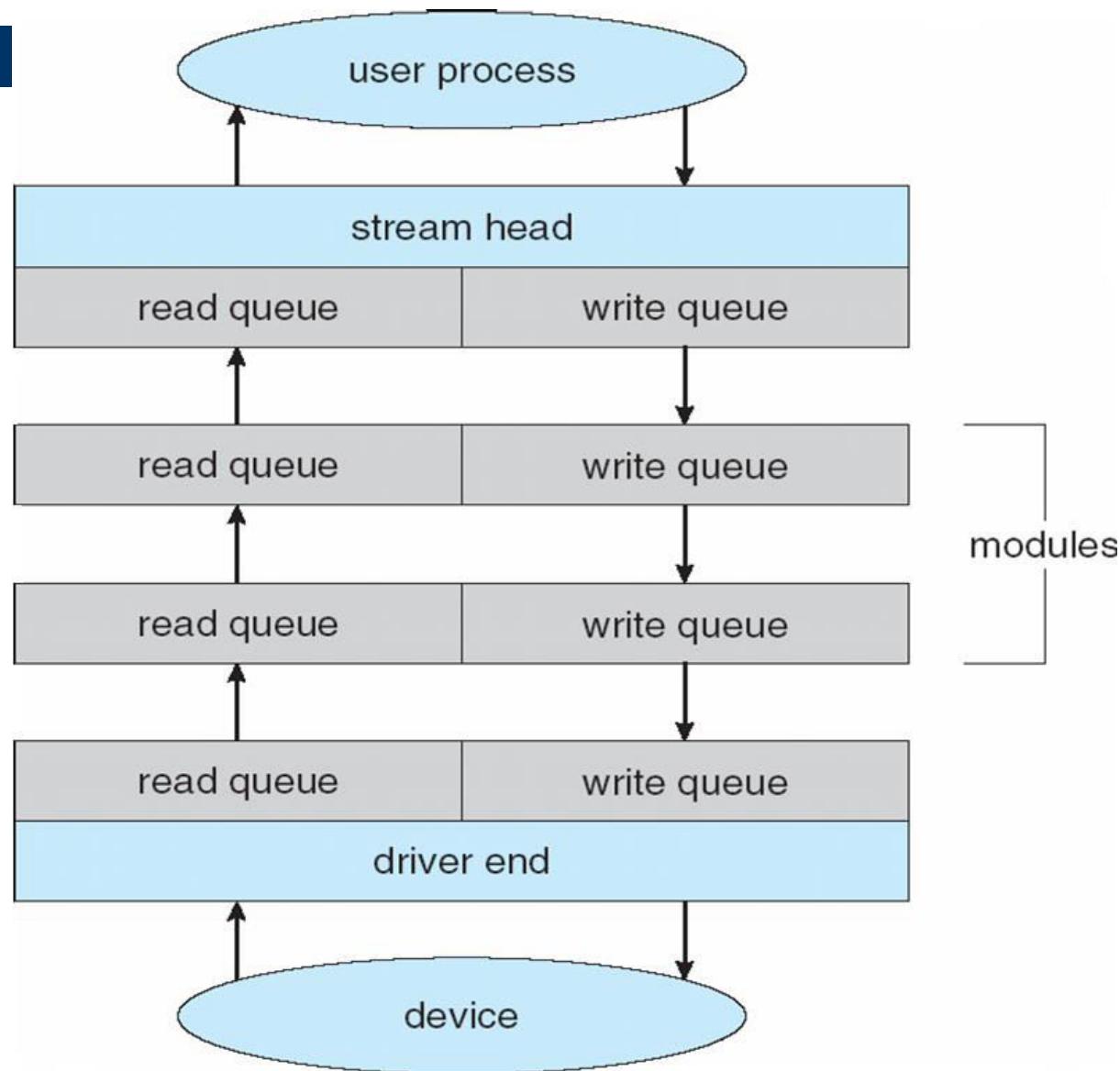
Life Cycle of An I/O Request



STREAMS

- **STREAM** – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
 - STREAM head interfaces with the user process
 - driver end interfaces with the device
 - zero or more STREAM modules between them.
- Each module contains a **read queue** and a **write queue**
- Message passing is used to communicate between queues

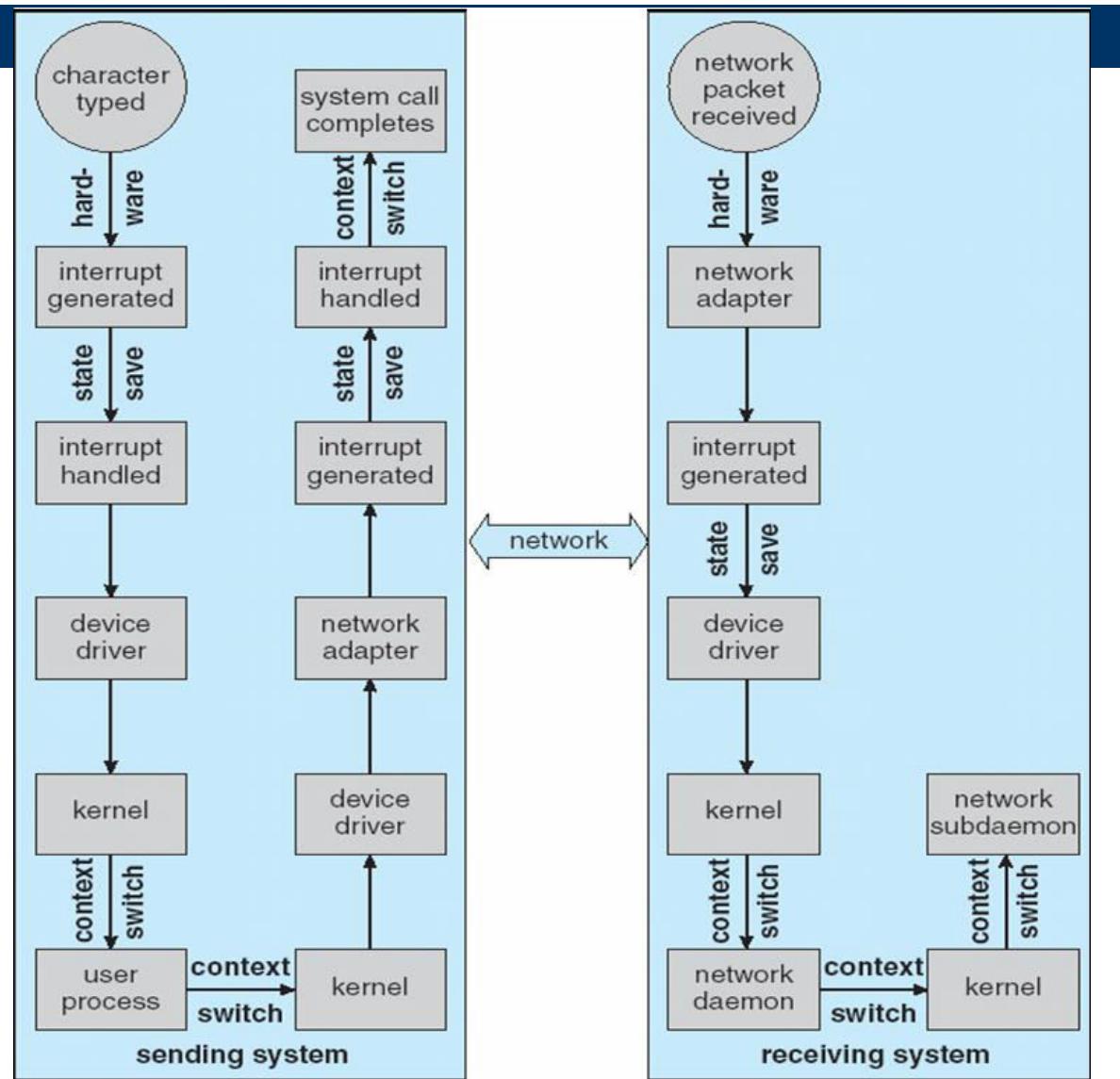
The STREAMS Structure



Performance

- I/O a major factor in system performance:
 - Demands CPU to execute device driver, kernel I/O code
 - Context switches due to interrupts
 - Data copying
 - Network traffic especially stressful

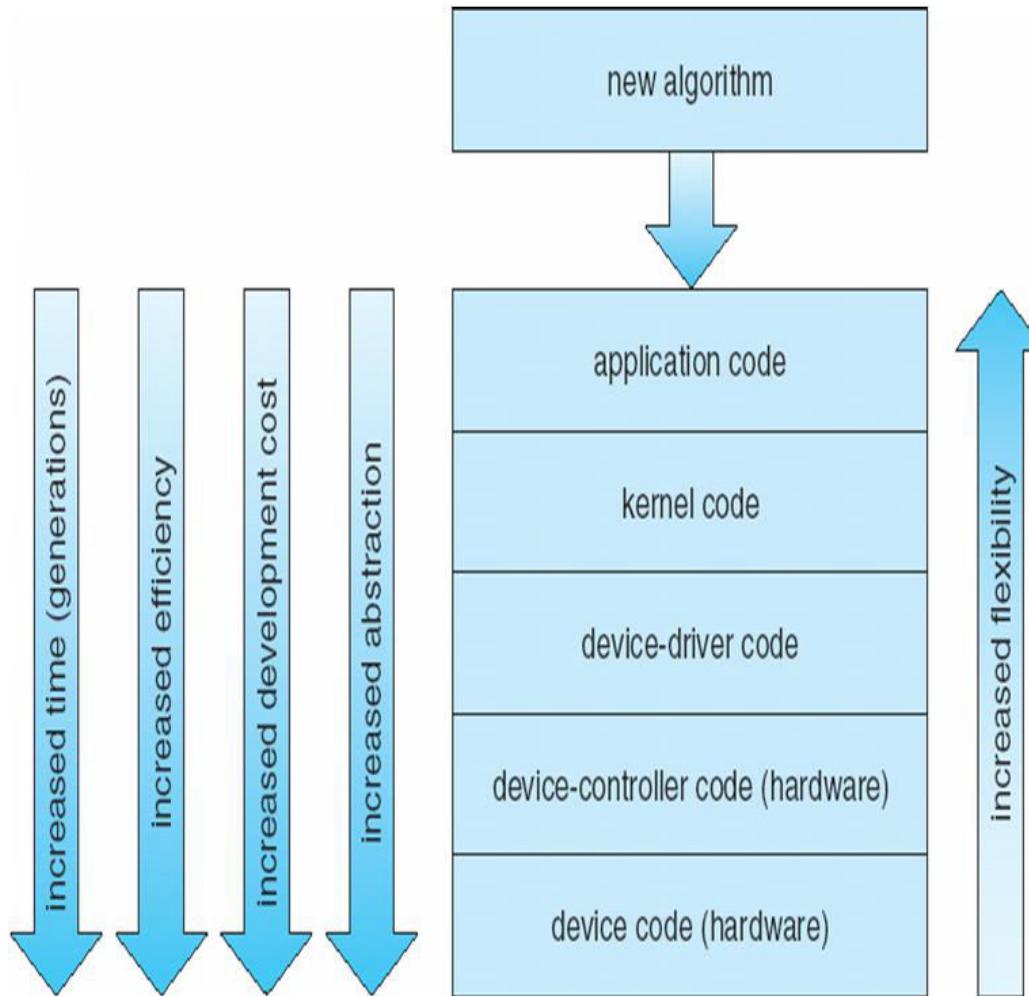
Intercomputer Communications



Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

Device-Functionality Progression



Operating Systems

Protection

Alok Kumar Jagadev

Protection

- Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.
- Protection ensures that the resources of the computer are used in a consistent way.
- It also ensures that each object accessed correctly and only by those processes that are allowed to do so.

Goals of Protection

- As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown.
- We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by user.
- An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.
- A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

Goals of Protection (Contd...)

- The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use.
- These policies can be established in a variety of ways.
- Some are fixed in the design of the system, while others are formulated by the management of a system.
- Still others are defined by the individual users to protect their own files and programs.
- A protection system must have the flexibility to enforce a variety of policies.

Principles of Protection

- The time-tested guiding principle for protection is the **Principle of least privilege**.
- It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.
- Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that he guards, then misuse of the key will result in minimal damage.
- If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater.

Principles of Protection (Contd...)

- An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done.
- The principle of least privilege can help produce a more secure computing environment.

Domain of Protection

- A computer system is a collection of processes and objects.
- By objects, we mean both hardware objects (such as the CPU, printer) and software objects(such as files, programs).
- Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations.

Domain of Protection (Contd...)

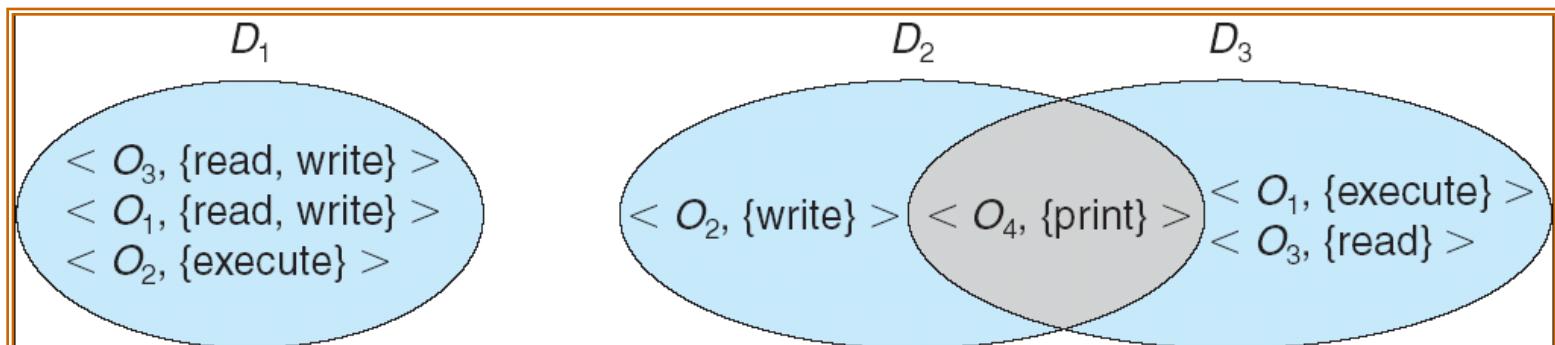
- A process should be allowed to access only those resources for which it has authorization.
- Furthermore, at any time, a process should be able to access only those resources that it currently requires to complete its task.

Domain Structure

- A process operates within a Protection Domain that specifies the resources that the process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- The ability to execute an operation on an object is an access right.

Domain Structure (Contd...)

- **Access-right** = $\langle \text{object-name}, \text{rights-set} \rangle$
where *rights-set* is a subset of all valid operations that can be performed on the object.
- **Domain** = set of access-rights
- For example, if domain D has the access right $\langle \text{file F}, \{\text{read, write}\} \rangle$, then a process executing in domain D can both read and write file F; it cannot perform any other operation on that object.



Domain Structure (Contd...)

- A domain can be realized in a variety of ways:
- Each user may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user.
- Each process may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process.

Domain Implementation (UNIX)

- System consists of 2 domains:
 - User
 - Supervisor
- UNIX
 - Domain = user-id
 - Domain switch accomplished via file system.
 - Each file has associated with it a domain bit (setuid bit).
 - When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.

Access Matrix

- The model of protection can be viewed abstractly as a matrix, called an Access Matrix.
- The rows of the access matrix represent domains, and the columns represent objects.
- Each entry in the matrix consists of a set of access rights.
- Entry $\text{Access}(i, j)$ is the set of operations that a process executing in Domain_i can invoke on Object_j

Access Matrix

object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain D_i tries to do “op” on object O_j , then “op” must be in the access matrix.
- Can be expanded to dynamic protection.
 - Operations to add, delete access rights.
 - Special access rights:
 - *owner* of O_i
 - *copy op from O_i to O_j*
 - *control – D_i can modify D_j access rights*
 - *transfer – switch from domain D_i to D_j*

Use of Access Matrix (Cont.)

- Access matrix design separates **mechanism** from **policy**.
 - **Mechanism**
 - Operating system provides access-matrix + rules.
 - Ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.
 - **Policy**
 - User dictates policy.
 - Who can access what object and in what mode.

Implementation of Access Matrix

- Each column = Access-control list for one object
Defines who can perform what operation.

Domain 1 = Read, Write

Domain 2 = Read

Domain 3 = Read

:

- Each Row = Capability List (like a key)
For each domain, what operations allowed on what objects.

Object 1 – Read

Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy

Access Matrix of Figure A With Domains as Objects

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure B

Access Matrix with *Copy* Rights

- The ability to *copy* rights is denoted by an **asterisk**, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object.
- There are two important variations: If the asterisk is removed from the original access right, then the right is *transferred*, rather than being copied. This may be termed a *transfer* right as opposed to a *copy* right.
- If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a *limited copy* right.

Access Matrix with *Copy* Rights

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Access Matrix With *Owner* Rights

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Modified Access Matrix of Figure B

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Access Control

- The basic problem of computer protection is to control which objects a given program can access, and in what ways.
- Objects are things like files, sound cards, other programs, the network, modem etc.
- Access means what kind of operations can be done on these objects.
- Examples include reading a file, writing to a file and creating or deleting objects.

Access Control (Contd...)

- When we talk about ``controlling access," we are really talking about four kinds of things:
 - Preventing access.
 - Limiting access.
 - Granting access .
 - Revoking access.
- A good example of this is found in Solaris 10.
- Solaris uses Role-based access control(RBAC) to adding the principle.

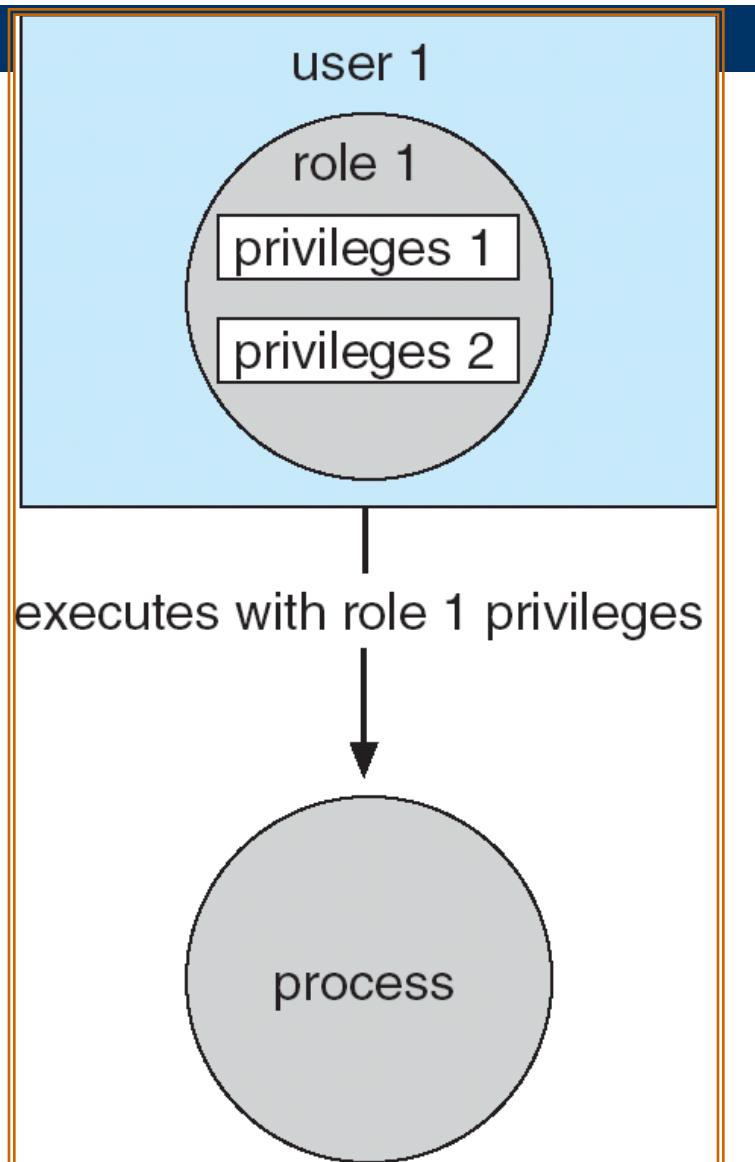
Access Control

- Protection can be applied to non-file resources
- Solaris 10 provides **role-based access control** to implement least privilege
 - Privilege is right to execute system call or use an option within a system call
 - Can be assigned to processes
 - Users assigned roles granting access to privileges and programs

A Role-based access control (RBAC)

- Role-based access control (RBAC) is a security feature for controlling user access to tasks that would normally be restricted to the root user.
- In conventional UNIX systems, the root user, also referred to as superuser.
- The root user has the ability to read and write to any file, run all programs, and send kill signals to any process.

Role-based Access Control in Solaris 10



Revocation of Access Rights

- *Access List* – Delete access rights from access list.
 - Simple
 - Immediate
- *Capability List* – Scheme required to locate capability in the system before capability can be revoked.
 - Reacquisition
 - Back-pointers
 - Indirection
 - Keys

Capability-Based Systems

- **Hydra**
 - Fixed set of access rights known to and interpreted by the system.
 - Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.
- **Cambridge CAP System**
 - **Data capability** - provides standard read, write, execute of individual storage segments associated with object.
 - **Software capability** - interpretation left to the subsystem, through its protected procedures.

Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

Operating Systems

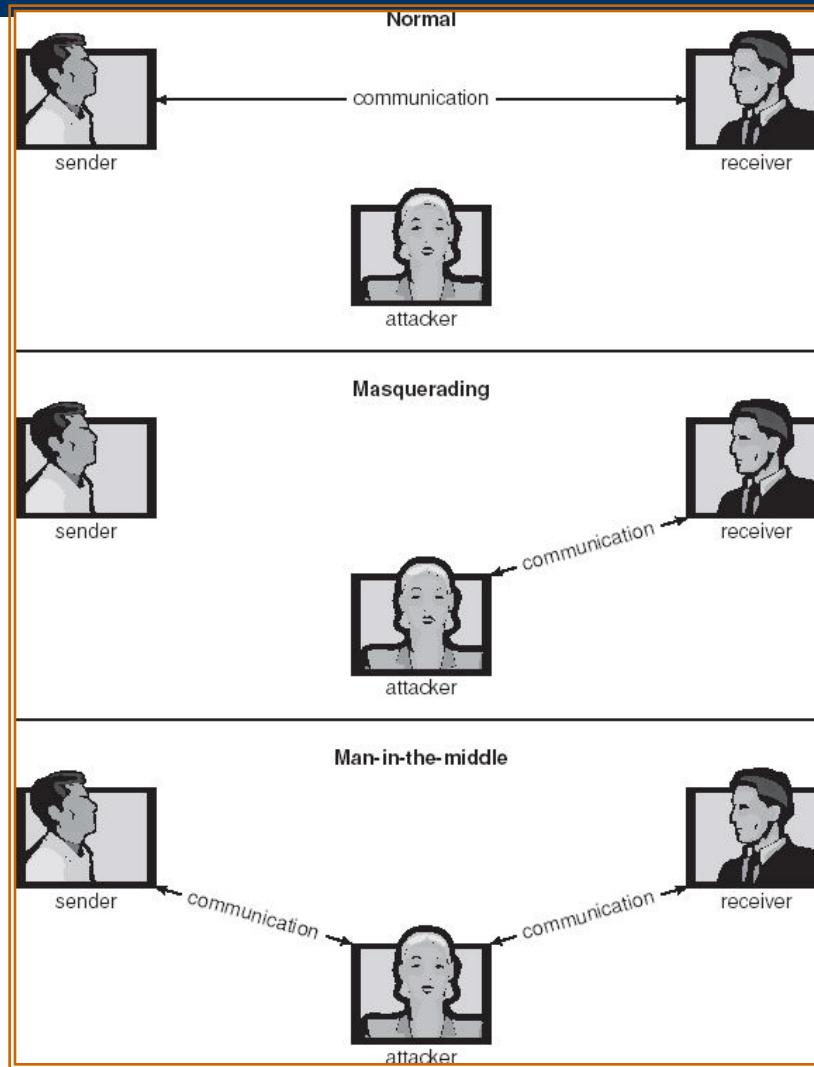
Security

Alok Kumar Jagadev

The Security Problem

- Security must consider external environment of the system, and protect the system resources
- Intruders (crackers) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
- Easier to protect against accidental than malicious misuse

Standard Security Attacks



Security Measure Levels

- Four levels of security measures
 - Physical: Physical protection of the computer system
 - Human: Screening of users given access to the computer system
 - Avoid **social engineering, phishing**
 - Operating System
 - Network: OS must be capable of protecting itself from accidental or intentional security breaches

Operating System Security

User authentication

Based on

- User possession (of key or card)
- User knowledge (user identifier + password)
- User attribute (fingerprint, retina pattern, signature)

Program Threats

- Trojan Horse
 - Trojans are a class of malware that take their name from the way they infect computers.
 - Code segment that misuses its environment
 - Exploits mechanisms for allowing programs written by users to be executed by other users

Program Threats

- **Creating backdoors:** some Trojans will make changes to the security system so that the data and device can be accessed by their controller.
- **Spying:** some Trojans are designed to wait until you access your online accounts or enter your credit card details, and then send your data back to whoever is in control.
- **Steal you passwords:** some Trojans are made to steal your passwords for your most important online accounts.
- **Turn your computer into a Zombie:** some Trojans just want to use your computer as a slave in a network controlled by a single hacker.
- **Send costly SMS messages:** even smartphones get Trojans, and the most common way for criminals to make money is by using them to make your phone send costly SMS messages to premium numbers.

Program Threats

- Trap Door
 - Trap doors are bits of code embedded in programs by the programmer(s) to quickly gain access at a later time, often during the testing or debugging phase.
 - If an unscrupulous programmer purposely leaves this code in or simply forgets to remove it, a potential security hole is introduced.
 - Trap doors can be almost impossible to remove in a reliable manner. Often, reformatting the system is the only sure way.

Program Threats

- Logic Bomb
 - A logic bomb is a malicious program timed to cause harm at a certain point in time, but is inactive up until that point.
 - A set trigger, such as a preprogrammed date and time, activates a logic bomb.
 - Once activated, a logic bomb implements a malicious code that causes harm to a computer.
- Stack and Buffer Overflow
 - Exploits a bug in a program (overflow either the stack or memory buffers)

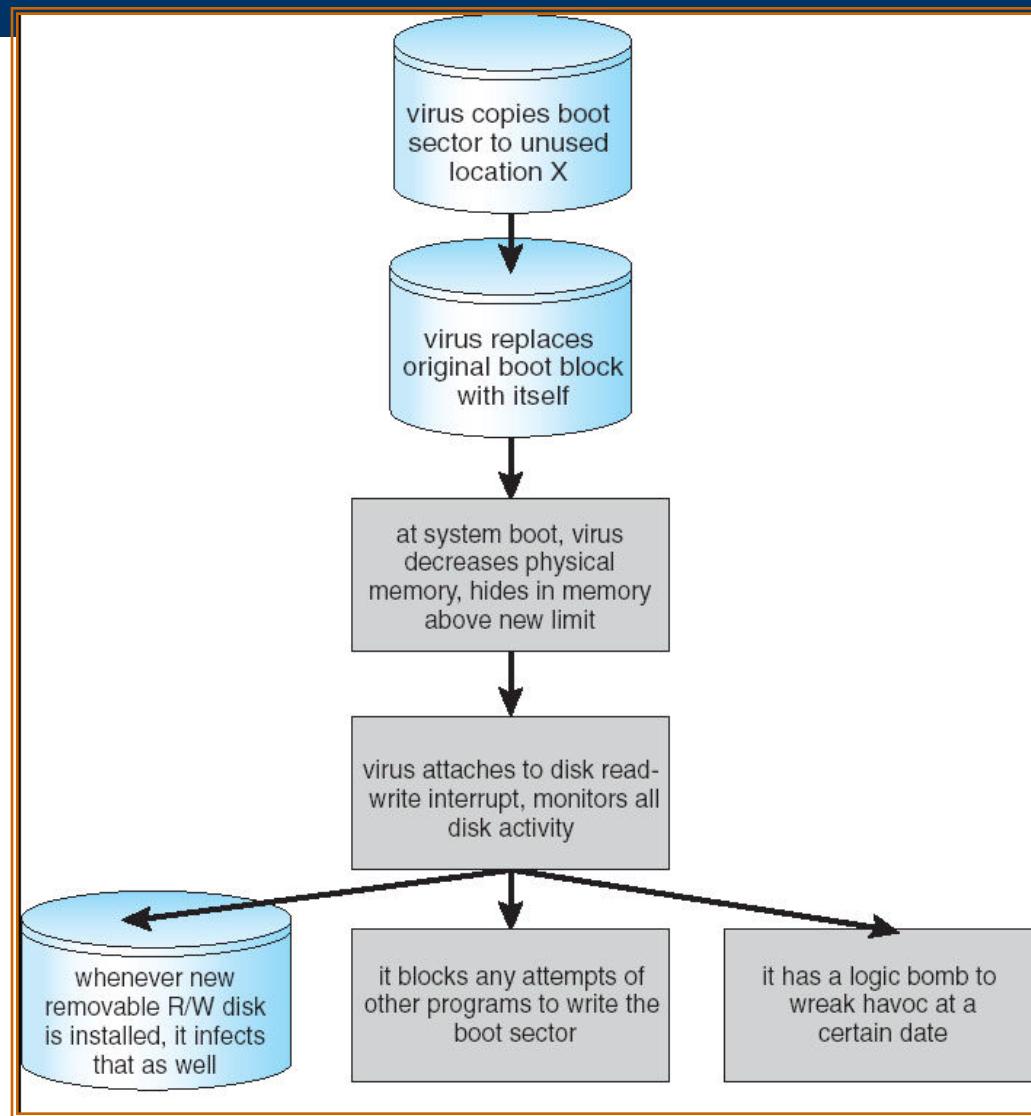
Program Threats

- **Virus**
 - Virus as name suggest can replicate themselves on computer system.
 - They are highly dangerous and can modify/delete user files, crash systems. A virus is generally a small code embedded in a program.
 - As user accesses the program, the virus starts getting embedded in other files/ programs and can make system unusable for user.

Program Threats

- **Virus dropper** inserts virus onto the system
- Many categories of viruses, literally many thousands of viruses
 - File
 - Boot
 - Macro
 - Source code
 - Polymorphic
 - Encrypted
 - Stealth
 - Tunneling
 - Multipartite
 - Armored

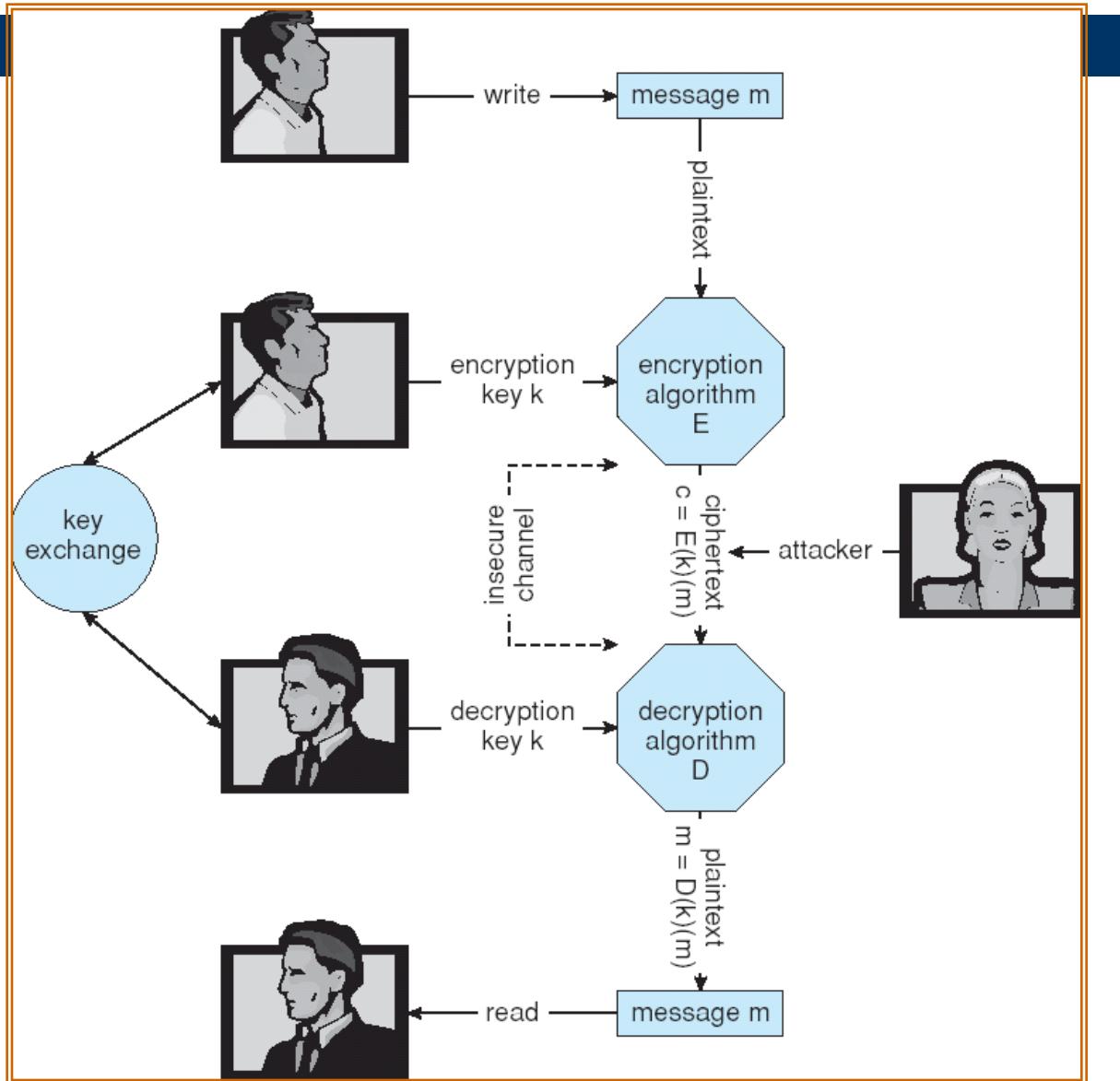
A Boot-sector Computer Virus



Cryptography as a Security Tool

- Broadest security tool available
 - Source and destination of messages cannot be trusted without cryptography
 - Means to constrain potential senders (*sources*) and / or receivers (*destinations*) of *messages*
- Based on secrets (**keys**)

Secure Communication over Insecure Medium



Encryption

- Encryption algorithm consists of
 - Set of K keys
 - Set of M Messages
 - Set of C ciphertexts (encrypted messages)
 - A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, $E(k)$ is a function for generating ciphertexts from messages.
 - Both E and $E(k)$ for any k should be efficiently computable functions.
 - A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, $D(k)$ is a function for generating messages from ciphertexts.
 - Both D and $D(k)$ for any k should be efficiently computable functions.

Encryption

- An encryption algorithm must provide this essential property: Given a ciphertext $c \in C$, a computer can compute m such that $E(k)(m) = c$ only if it possesses $D(k)$.
 - Thus, a computer holding $D(k)$ can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding $D(k)$ cannot decrypt ciphertexts.
 - Since ciphertexts are generally exposed (for example, sent on the network), it is important that it be infeasible to derive $D(k)$ from the ciphertexts

Symmetric Encryption

- Same key used to encrypt and decrypt
 - $E(k)$ can be derived from $D(k)$, and vice versa
- DES is most commonly used symmetric block-encryption algorithm (created by US Govt)
 - Encrypts a block of data at a time
- Triple-DES considered more secure
- Advanced Encryption Standard (**AES**), **twofish** up and coming
- RC4 is most common symmetric stream cipher, but known to have vulnerabilities
 - Encrypts/decrypts a stream of bytes (i.e wireless transmission)
 - Key is a input to psuedo-random-bit generator
 - Generates an infinite **keystream**

Asymmetric Encryption

- Public-key encryption based on each user having two keys:
 - public key – published key used to encrypt data
 - private key – key known only to individual user used to decrypt data
- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme
 - Most common is RSA block cipher
 - Efficient algorithm for testing whether or not a number is prime
 - No efficient algorithm is known for finding the prime factors of a number

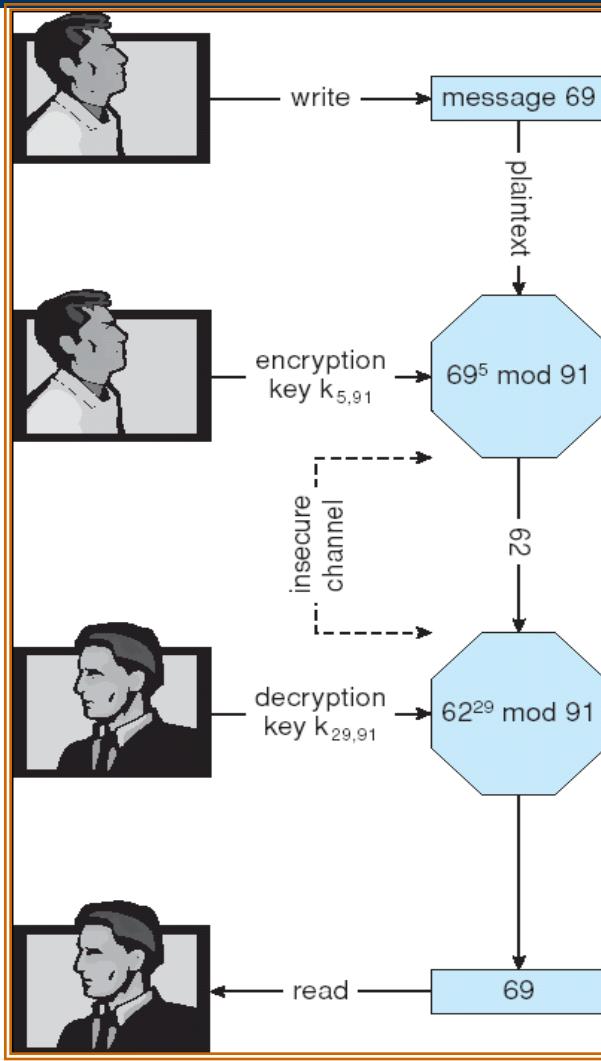
Asymmetric Encryption (Cont.)

- Formally, it is computationally infeasible to derive $D(k_d, N)$ from $E(k_e, N)$, and so $E(k_e, N)$ need not be kept secret and can be widely disseminated
 - $E(k_e, N)$ (or just k_e) is the **public key**
 - $D(k_d, N)$ (or just k_d) is the **private key**
 - N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each)
 - Encryption algorithm is $E(k_e, N)(m) = m^{k_e} \text{ mod } N$, where k_e satisfies $k_e k_d \text{ mod } (p-1)(q-1) = 1$
 - The decryption algorithm is then $D(k_d, N)(c) = c^{k_d} \text{ mod } N$

Asymmetric Encryption Example

- For example, make $p = 7$ and $q = 13$
- We then calculate $N = 7 * 13 = 91$ and $(p-1)(q-1) = 72$
- We next select k_e relatively prime to 72 and < 72, yielding 5
- Finally, we calculate k_d such that $k_e k_d \text{ mod } 72 = 1$, yielding 29
- We now have our keys
 - Public key, $k_e, N = 5, 91$
 - Private key, $k_d, N = 29, 91$
- Encrypting the message 69 with the public key results in the ciphertext 62
- Ciphertext can be decoded with the private key
 - Public key can be distributed in cleartext to anyone who wants to communicate with holder of public key

Encryption and Decryption using RSA Asymmetric Cryptography



Cryptography (Cont.)

- Note symmetric cryptography based on transformations, asymmetric based on mathematical functions
 - Asymmetric much more compute intensive
 - Typically not used for bulk data encryption

Authentication

- Constraining set of potential senders of a message
 - Complementary and sometimes redundant to encryption
 - Also can prove message unmodified
- Algorithm components
 - A set K of keys
 - A set M of messages
 - A set A of authenticators
 - A function $S : K \rightarrow (M \rightarrow A)$
 - That is, for each $k \in K$, $S(k)$ is a function for generating authenticators from messages
 - Both S and $S(k)$ for any k should be efficiently computable functions
 - A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \in K$, $V(k)$ is a function for verifying authenticators on messages
 - Both V and $V(k)$ for any k should be efficiently computable functions

Authentication (Cont.)

- For a message m , a computer can generate an authenticator $a \in A$ such that $V(k)(m, a) = \text{true}$ only if it possesses $S(k)$
- Thus, computer holding $S(k)$ can generate authenticators on messages so that any other computer possessing $V(k)$ can verify them
- Computer not holding $S(k)$ cannot generate authenticators on messages that can be verified using $V(k)$
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive $S(k)$ from the authenticators

Authentication – Hash Functions

- Basis of authentication
- Creates small, fixed-size block of data (**message digest, hash value**) from m
- Hash Function H must be collision resistant on m
 - Must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$
- If $H(m) = H(m')$, then $m = m'$
 - The message has not been modified
- Common message-digest functions include **MD5**, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash

Authentication - MAC

- Symmetric encryption used in **message-authentication code (MAC)** authentication algorithm
- Simple example:
 - MAC defines $S(k)(m) = f(k, H(m))$
 - Where f is a function that is one-way on its first argument
 - k cannot be derived from $f(k, H(m))$
 - Because of the collision resistance in the hash function, reasonably assured no other message could create the same MAC
 - A suitable verification algorithm is $V(k)(m, a) \equiv (f(k, m) = a)$
 - Note that k is needed to compute both $S(k)$ and $V(k)$, so anyone able to compute one can compute the other

Authentication – Digital Signature

- Based on asymmetric keys and digital signature algorithm
- Authenticators produced are **digital signatures**
- In a digital-signature algorithm, computationally infeasible to derive $S(k_s)$ from $V(k_v)$
 - V is a one-way function
 - Thus, k_v is the public key and k_s is the private key
- Consider the RSA digital-signature algorithm
 - Similar to the RSA encryption algorithm, but the key use is reversed
 - Digital signature of message $S(k_s)(m) = H(m)^{k_s} \text{ mod } N$
 - The key k_s again is a pair d, N , where N is the product of two large, randomly chosen prime numbers p and q
 - Verification algorithm is $V(k_v)(m, a) \equiv (a^{k_v} \text{ mod } N = H(m))$
 - Where k_v satisfies $k_v k_s \text{ mod } (p - 1)(q - 1) = 1$

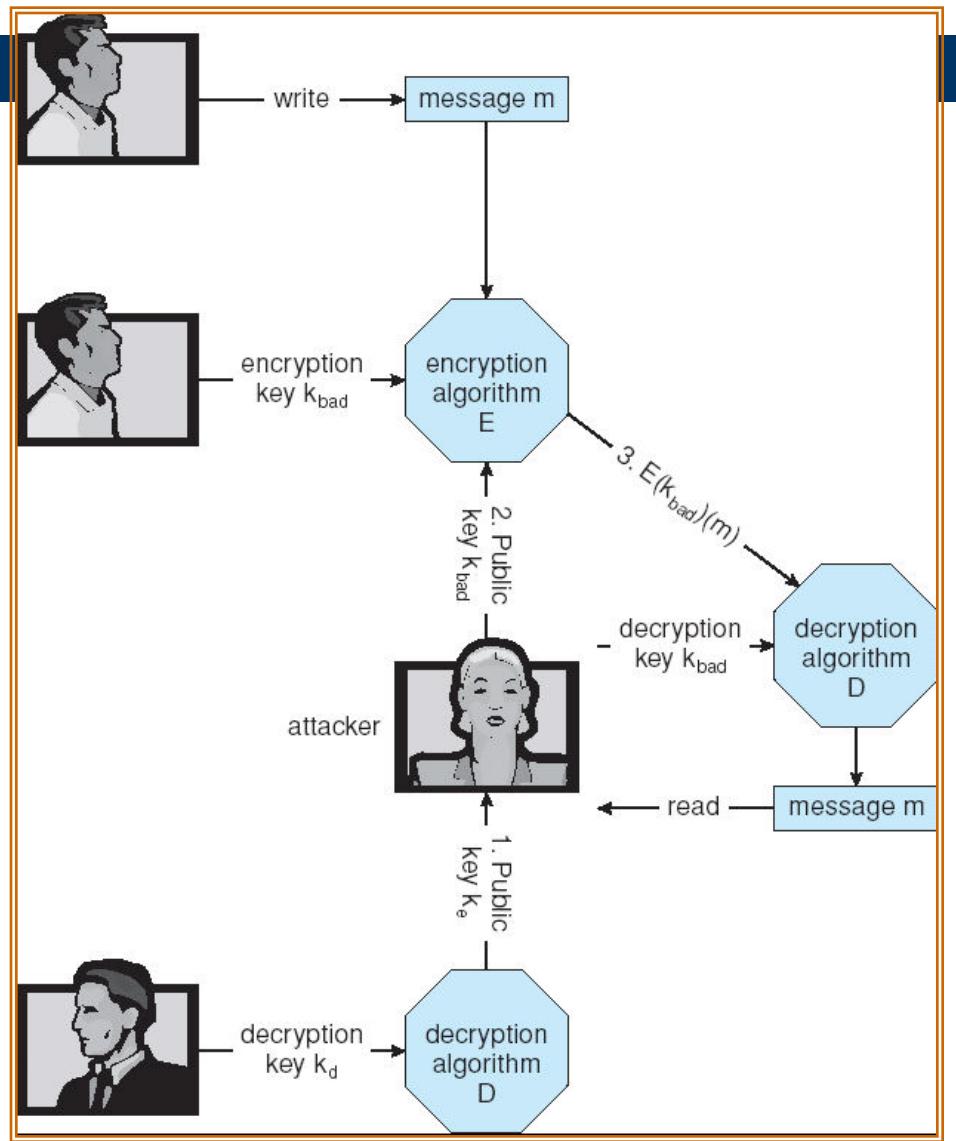
Authentication (Cont.)

- Why authentication if a subset of encryption?
 - Fewer computations (except for RSA digital signatures)
 - Authenticator usually shorter than message
 - Sometimes want authentication but not confidentiality
 - Signed patches et al
 - Can be basis for **non-repudiation**

Key Distribution

- Delivery of symmetric key is huge challenge
 - Sometimes done **out-of-band**
- Asymmetric keys can proliferate – stored on **key ring**
 - Even asymmetric key distribution needs care – man-in-the-middle attack

Man-in-the-middle Attack on Asymmetric Cryptography



Digital Certificates

- Proof of who or what owns a public key
- Public key digitally signed a trusted party
- Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- Certificate authority are trusted party – their public keys included with web browser distributions
 - They vouch for other authorities via digitally signing their keys, and so on

Encryption Example - SSL

- Insertion of cryptography at one layer of the ISO network model (the transport layer)
- SSL – Secure Socket Layer (also called TLS)
- Cryptographic protocol that limits two computers to only exchange messages with each other
 - Very complicated, with many variations
- Used between web servers and browsers for secure communication (credit card numbers)
- The server is verified with a **certificate** assuring client is talking to correct server
- Asymmetric cryptography used to establish a secure **session key** (symmetric encryption) for bulk of communication during session
- Communication between each computer theb uses symmetric key cryptography

User Authentication

- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through *passwords*, can be considered a special case of either keys or capabilities
 - Also can include something user has and /or a user attribute
- Passwords must be kept secret
 - Frequent change of passwords
 - Use of “non-guessable” passwords
 - Log all invalid access attempts
- Passwords may also either be encrypted or allowed to be used only once

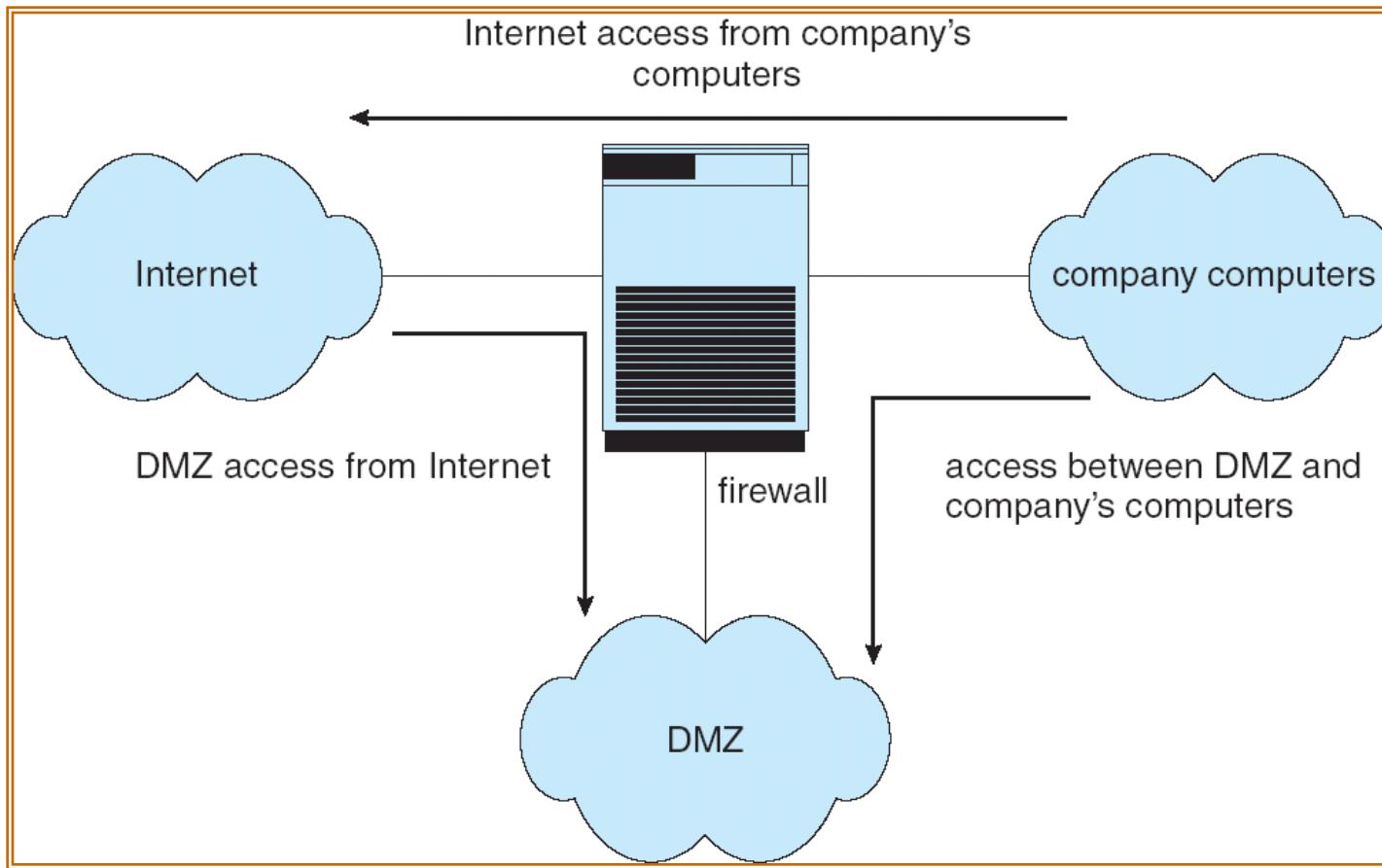
Implementing Security Defenses

- **Defense in depth** is most common security theory – multiple layers of security
- Security policy describes what is being secured
- Vulnerability assessment compares real state of system / network compared to security policy
- Intrusion detection endeavors to detect attempted or successful intrusions
 - **Signature-based** detection spots known bad patterns
 - **Anomaly detection** spots differences from normal behavior
 - Can detect **zero-day** attacks
 - **False-positives** and **false-negatives** a problem
- Virus protection
- Auditing, accounting, and logging of all or specific system or network activities

Firewalling to Protect Systems and Networks

- A network firewall is placed between trusted and untrusted hosts
 - The firewall limits network access between these two security domains
- Can be tunneled or spoofed
 - Tunneling allows disallowed protocol to travel within allowed protocol (i.e. telnet inside of HTTP)
 - Firewall rules typically based on host name or IP address which can be spoofed
- **Personal firewall** is software layer on given host
 - Can monitor / limit traffic to and from the host
- **Application proxy firewall** understands application protocol and can control them (i.e. SMTP)
- **System-call firewall** monitors all important system calls and apply rules to them (i.e. this program can execute that system call)

Network Security Through Domain Separation Via Firewall



Computer Security Classifications

- U.S. Department of Defense outlines four divisions of computer security: **A**, **B**, **C**, and **D**.
- **D** – Minimal security.
- **C** – Provides discretionary protection through auditing. Divided into **C1** and **C2**. **C1** identifies cooperating users with the same level of protection. **C2** allows user-level access control.
- **B** – All the properties of **C**, however each object may have unique sensitivity labels. Divided into **B1**, **B2**, and **B3**.
- **A** – Uses formal design and verification techniques to ensure security.

Example: Windows XP

- Security is based on user accounts
 - Each user has unique security ID
 - Login to ID creates **security access token**
 - Includes security ID for user, for user's groups, and special privileges
 - Every process gets copy of token
 - System checks token to determine if access allowed or denied
- Uses a subject model to ensure access security. A subject tracks and manages permissions for each program that a user runs
- Each object in Windows XP has a security attribute defined by a security descriptor
 - For example, a file has a security descriptor that indicates the access permissions for all users