BMS INSTITUTE OF TECHNOLOGY

YELAHANKA, BANGALORE

OPERATING SYSTEMS

SUBJECT CODE: 18CS43

PREPARED BY
PROF.S.MAHALAKSHMI
ASST.PROF
DEPARTMENT OF ISE

TABLE OF CONTENTS

15CS64 Operating Systems

AY:2019-20 Even

Prepared by
Prof.S.Mahalakshmi
AP/ISE

Course Outcomes

- **Acquire** the knowledge on the need of OS, types, structures and its services.
- **Understand** the concepts involved in process, memory, device and file management
- **Apply** suitable techniques for management of different resources
- **Analyse** how deadlock occurs and methods to handle deadlocks
- **Demonstrate** the impact of Operating System in social and Environmental context

Module 1: Introduction

- **What Operating Systems Do**
- Computer-System Organization
- Computer-System Architecture
- Operating-System Structure
- Operating-System Operations
- Process Management
- Memory Management
- Storage Management
- Protection and Security
- Distributed Systems
- Special-Purpose Systems
- Computing Environments

- **Operating System Services**
- User - Operating System interface;
- System calls, Types of system calls;
- System programs
- Operating system design and implementation
- Operating System structure
- Virtual machines
- Operating System generation; System boot.

Module 1 Cont..

- **Process Management :**
- Process concept
- Process scheduling
- Operations on processes
- Inter process communication

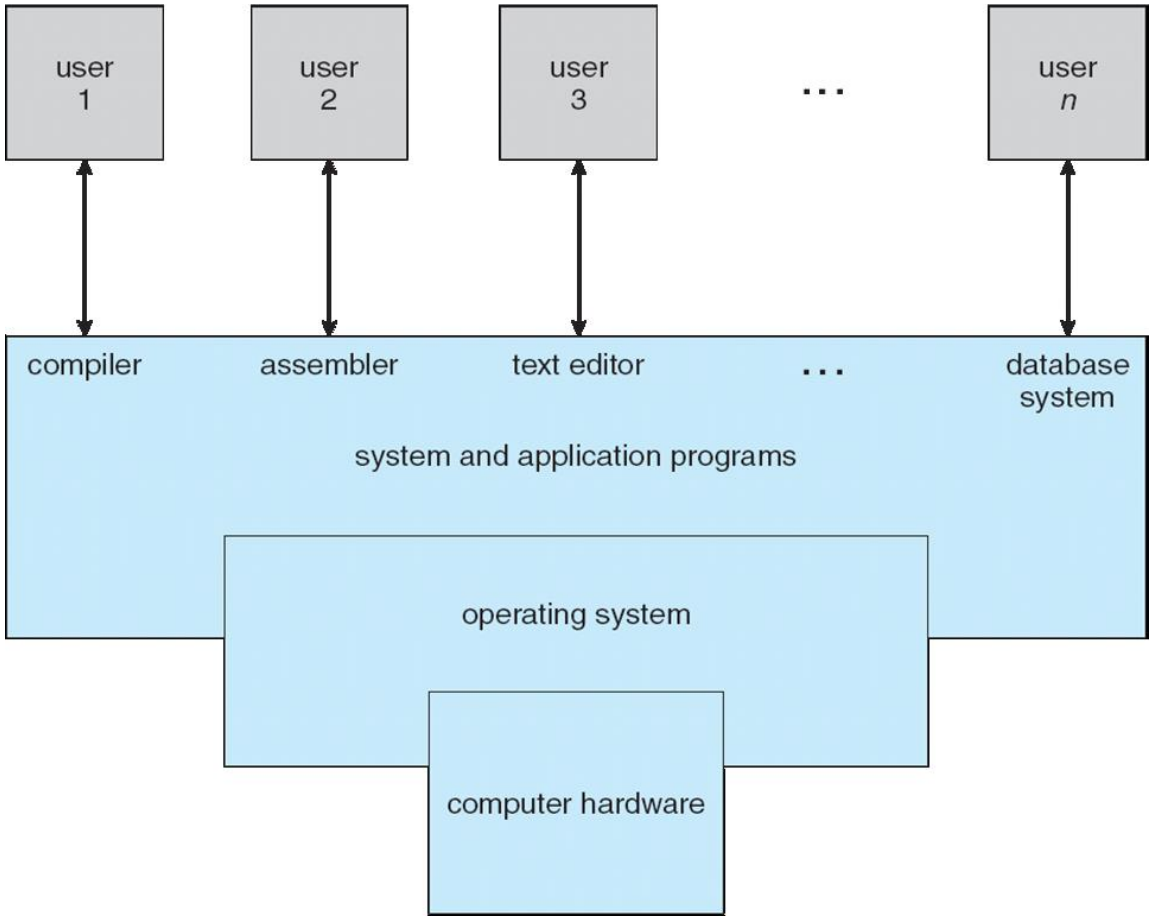
1.What is an Operating System?

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

Computer System Structure

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

Four Components of a Computer System



Operating System : System View

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

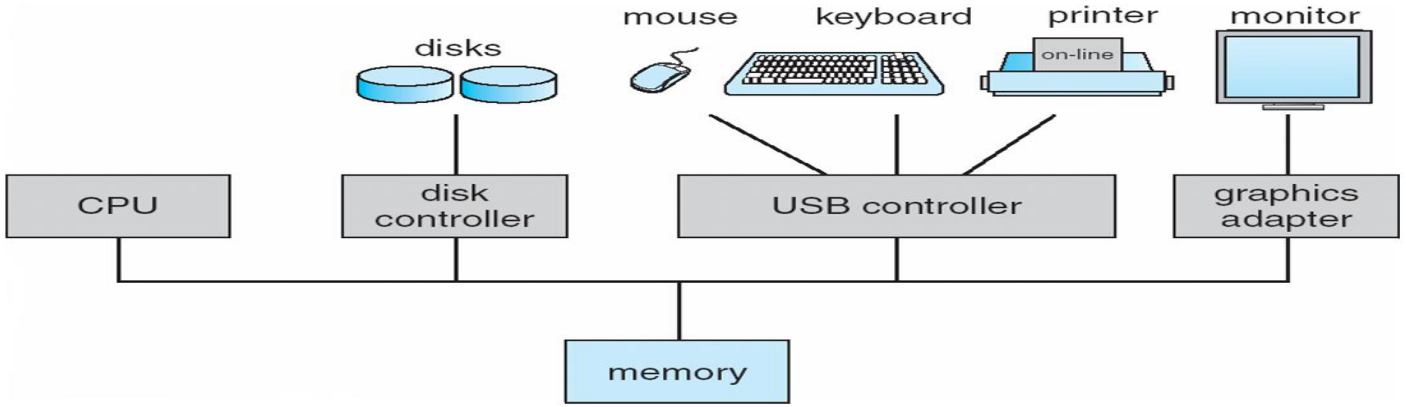
- The one program running at all times on the computer" is the **kernel.**

Computer Startup

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

2.Computer System Organization

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



Computer-System Operation

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

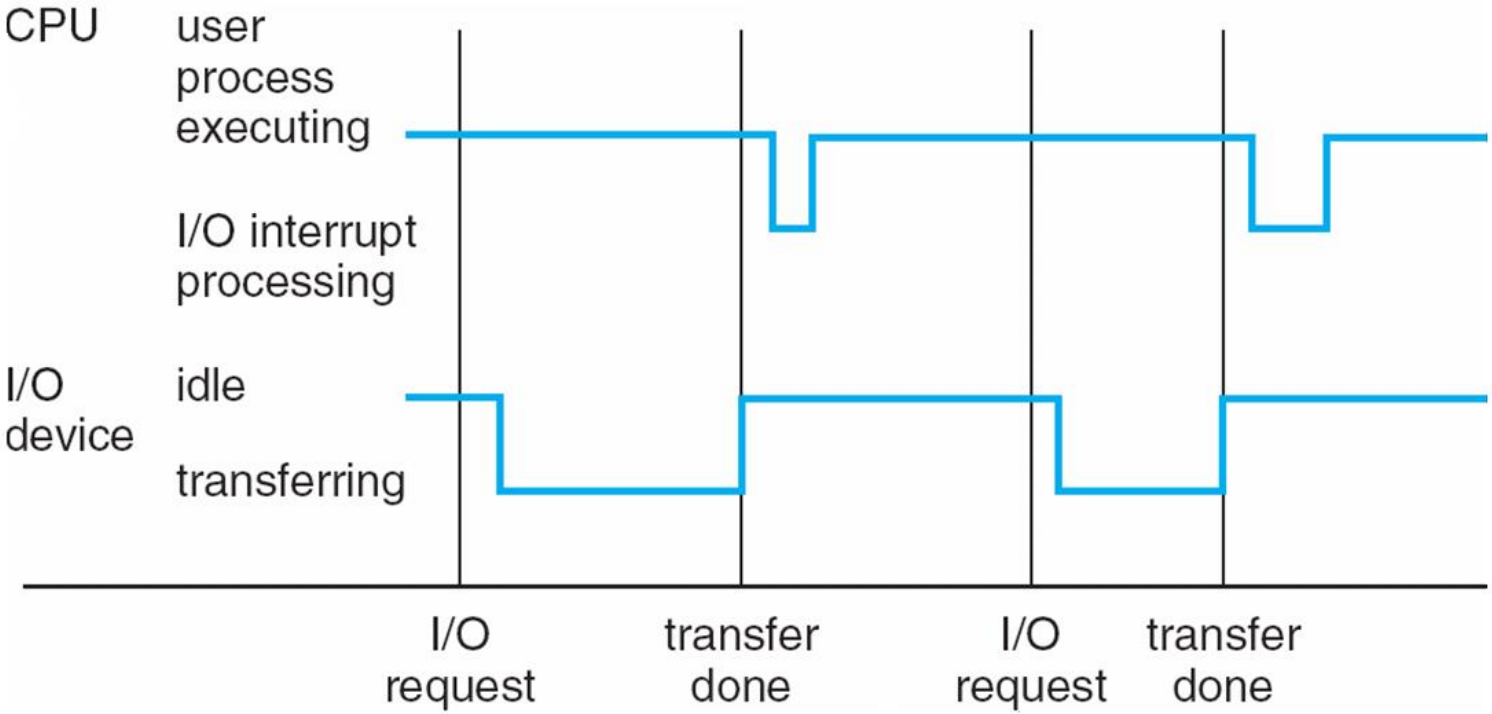
Common Functions of Interrupts

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

Interrupt Handling

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

Interrupt Timeline



I/O Structure

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

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

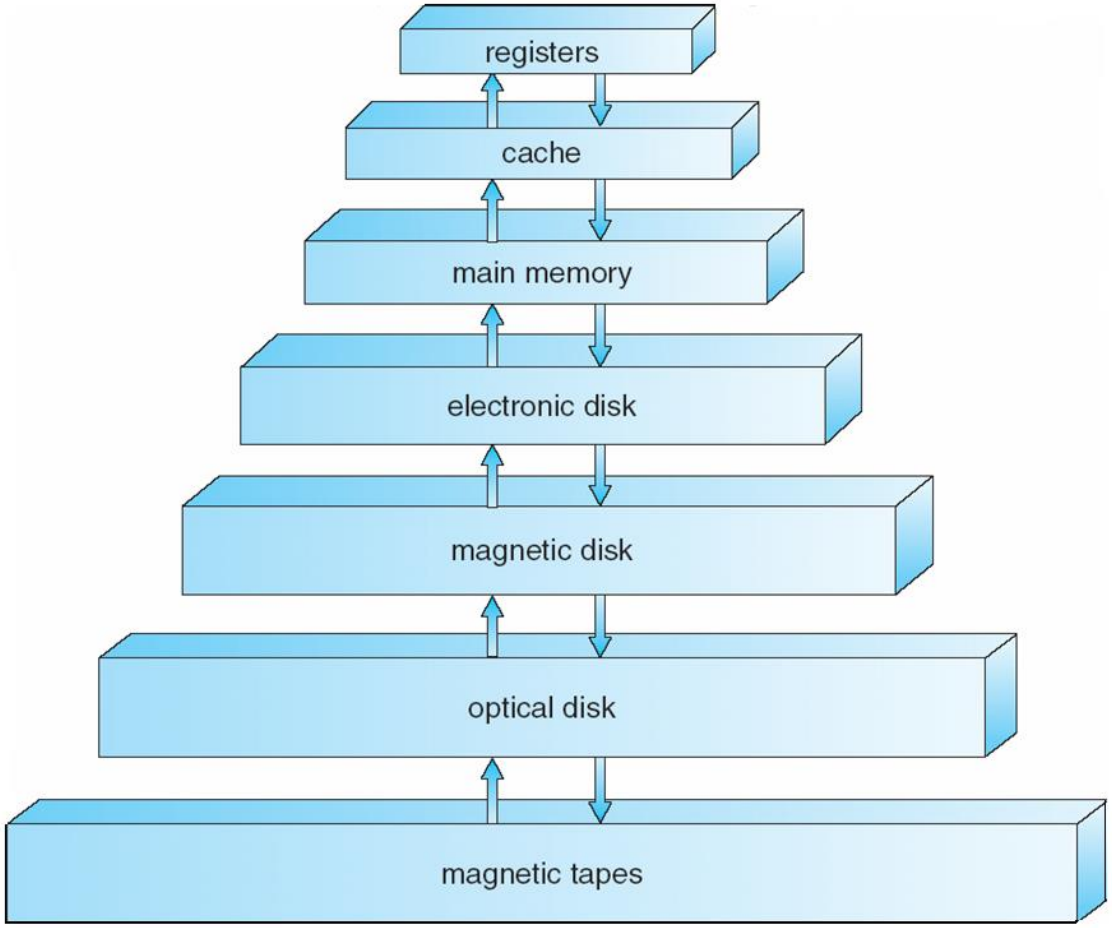
Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer

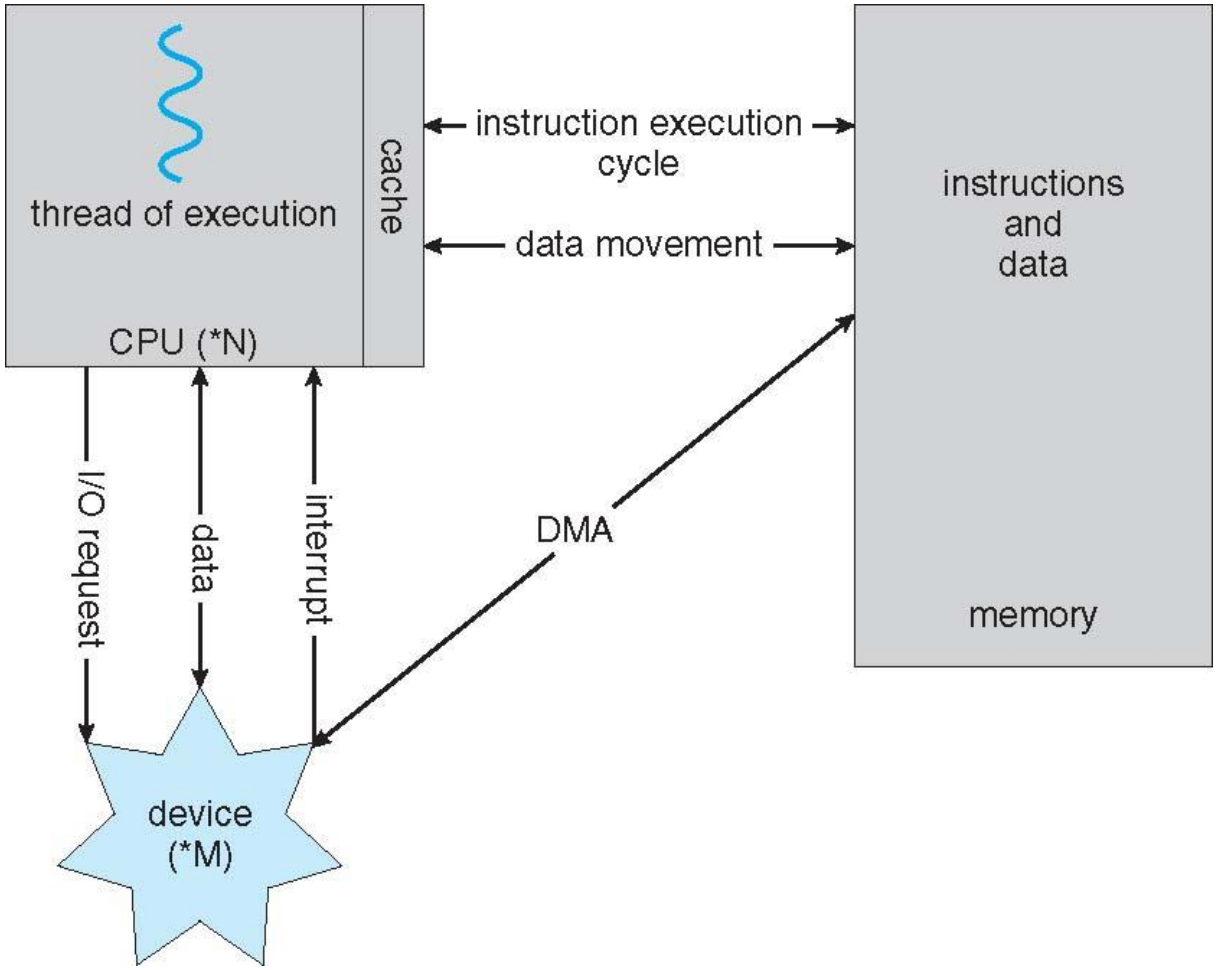
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage

Storage-Device Hierarchy



How a Modern Computer Works



3.Computer-System Architecture

- Single Processor System
- Multiprocessor system
- Clustered system

Single Processor Systems

- Most systems use a single general-purpose processor (PDAs through mainframes)
  - Most systems have special-purpose processors as well
  - Eg: disk controller microprocessor, keyboard and graphics controller
  - Special purpose processors do not run **user processes**

Note: The use of SPP does not make the single processor to a multiprocessor

Multiprocessor Systems

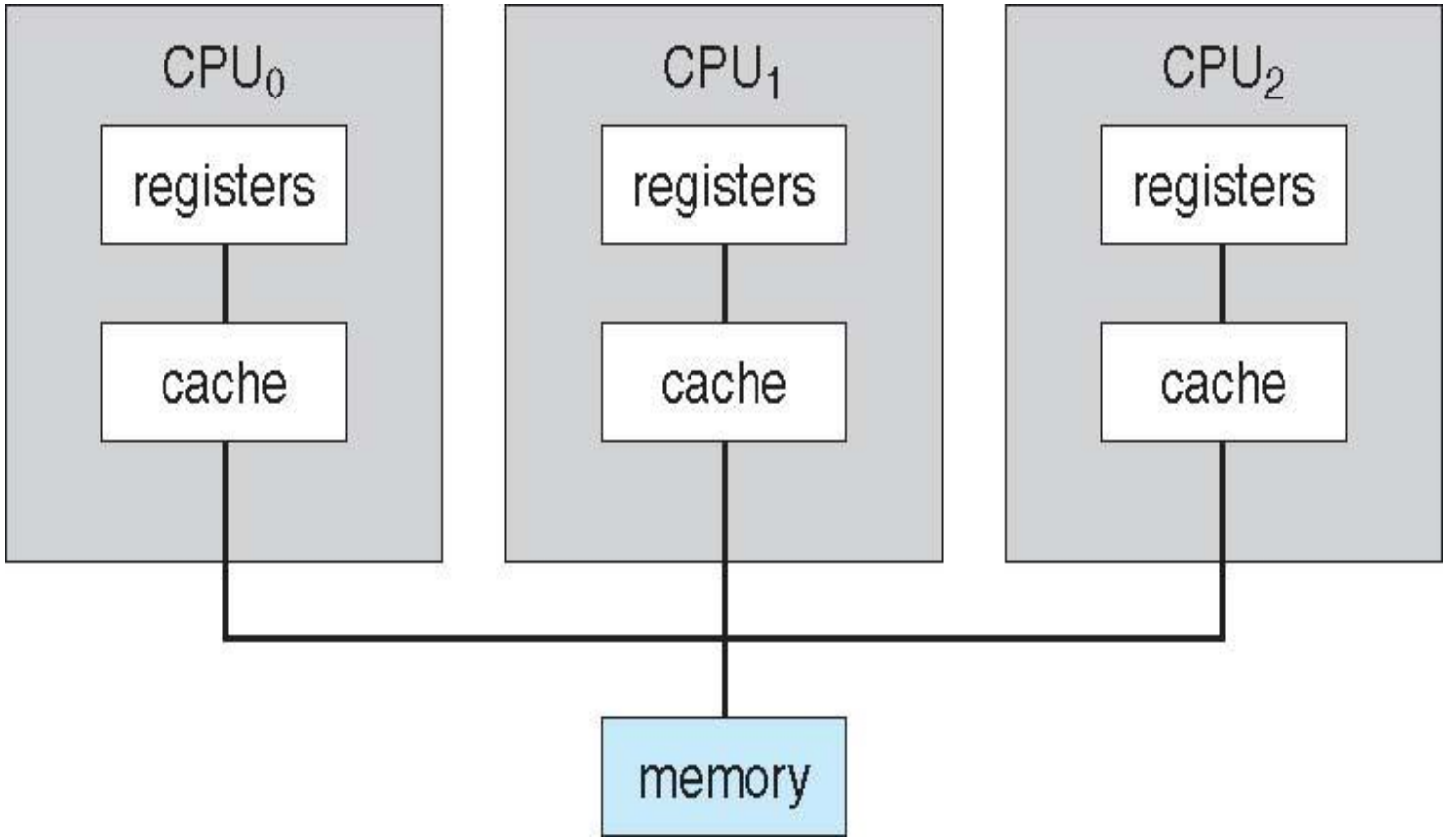
- Multiprocessors systems growing in use and importance
- 2 or more processors, sharing computer bus, memory and peripheral devices
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include
    1. Increased throughput
       - Speed up ratio with N processors ≠ N
    2. Economy of scale
    3. Increased reliability – graceful degradation or fault tolerance
  - Two types
    1. Asymmetric Multiprocessing
    2. Symmetric Multiprocessing

Types of Multiprocessor

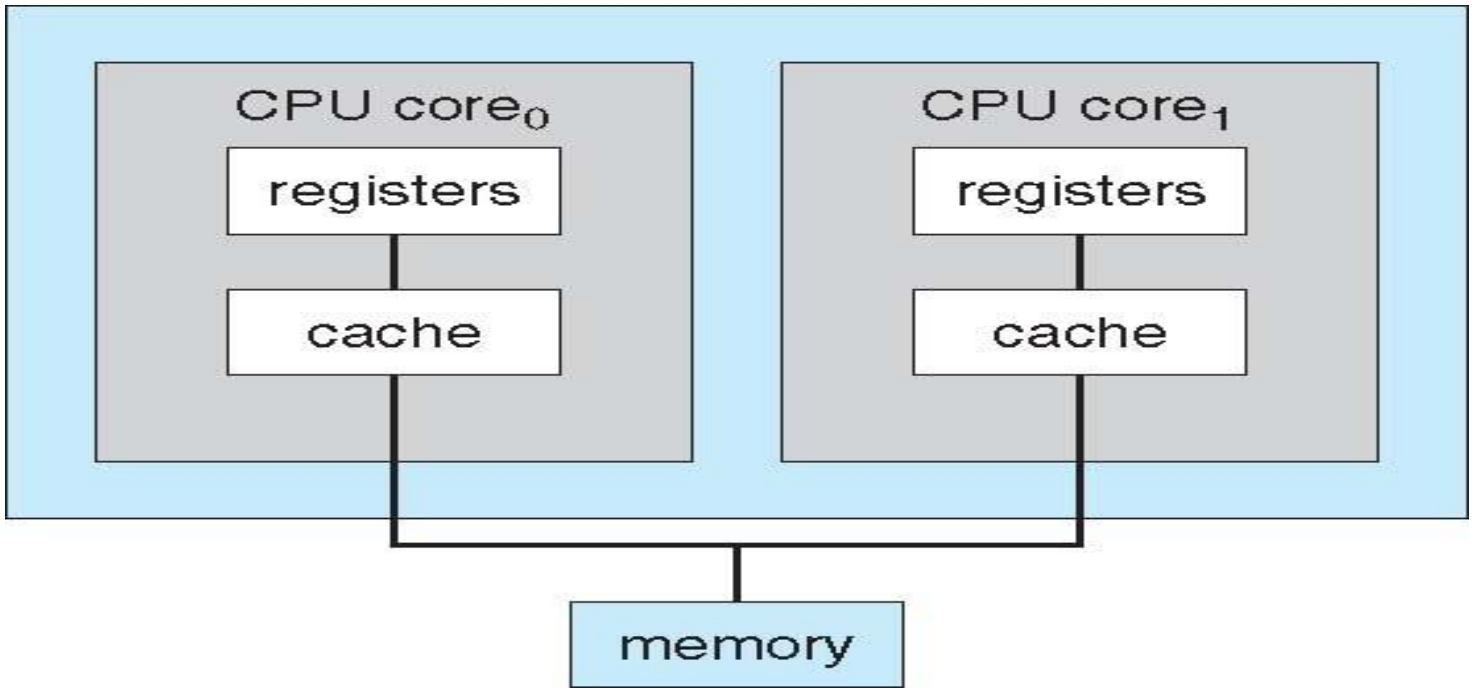
Asymmetric	Symmetric
Each processor is assigned a specific task	Each processor performs all task within the OS
Master slave relationship	All are peers
Eg: ps/2 server 195	Eg: Solaris Disadvantage: one CPU may be idle while another is overloaded

Difference between SMP & ASMP results from either H/W or S/W

Symmetric Multiprocessing Architecture



A Dual-Core Design



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

Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a LAN or faster interconnect storage-area network (SAN)
  - Provides a high-availability service which survives failures
    - **Asymmetric clustering** has one machine in hot-standby mode
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other(uses all available H/W)
  - Some clusters are for high-performance computing (HPC)
    - Applications must be written to use parallelization

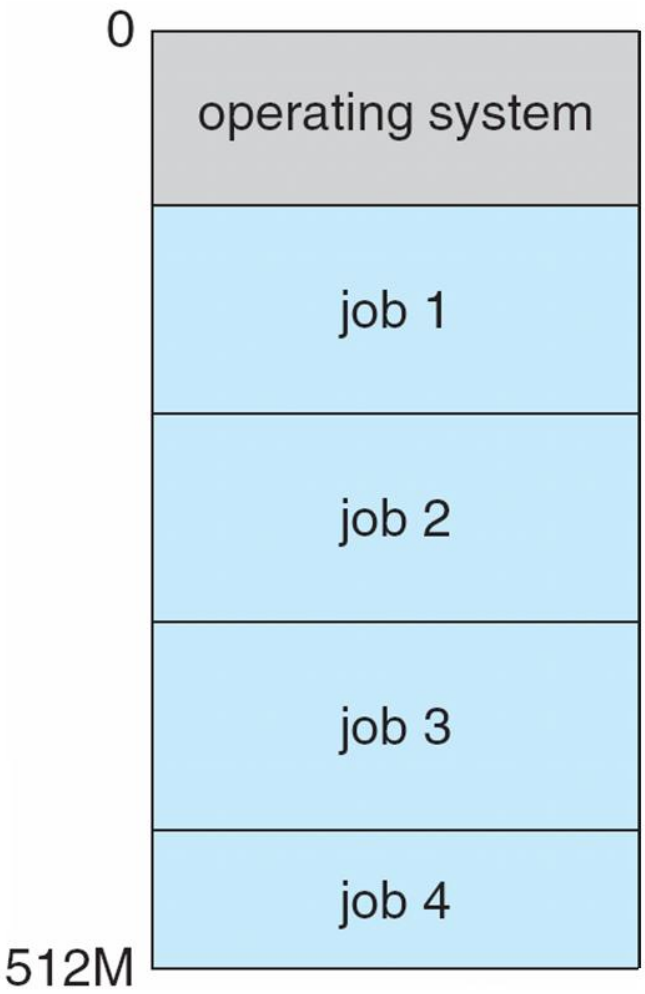
1.4. Operating System Structure

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

Adv: CPU will never be idle

Disadv: do not provide user interaction

Memory Layout for Multiprogrammed System



ISE Dept.
Transform Here

Time sharing Systems

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

1.5. Operating-System Operations

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

1.5.1 Dual Mode

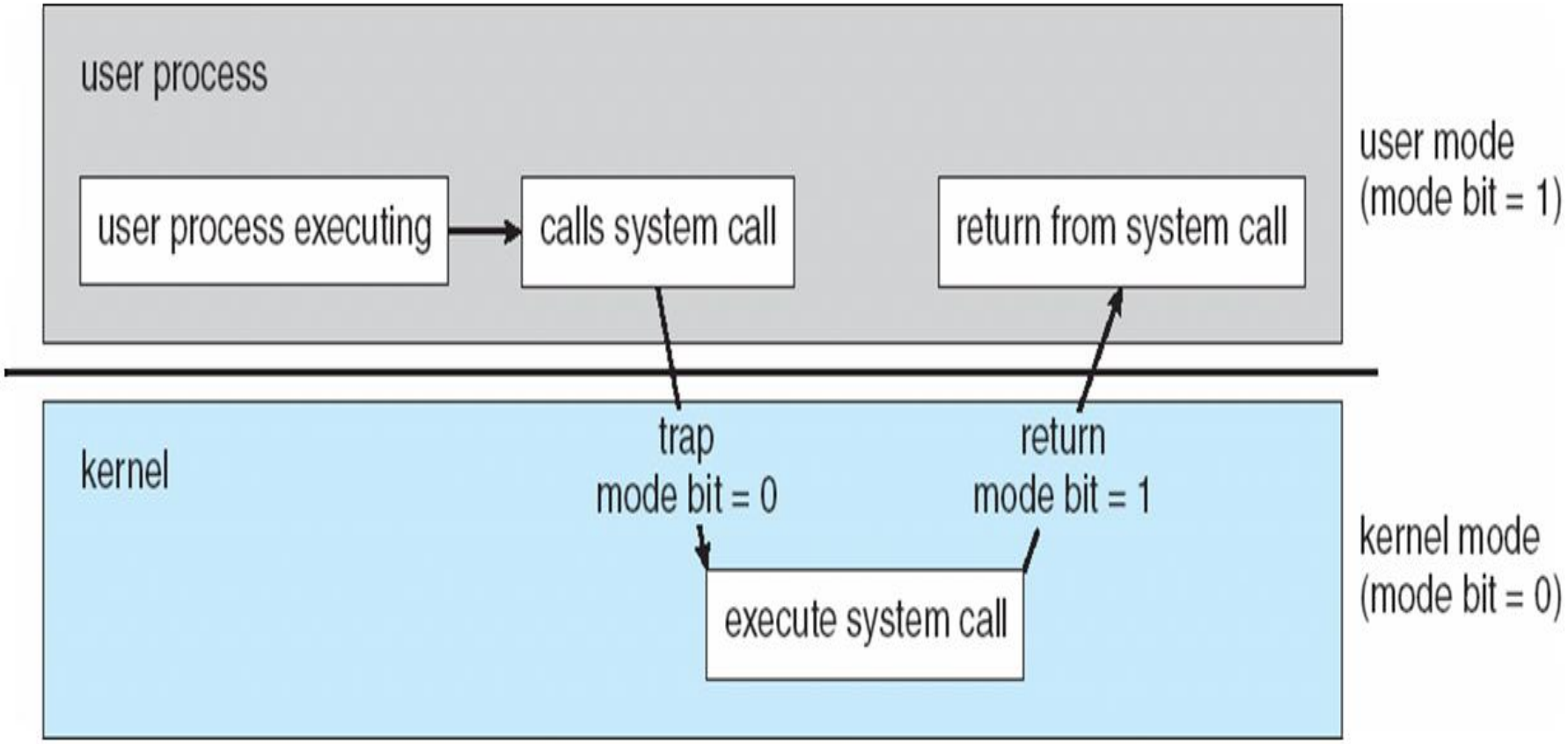
Dual-mode operation allows OS to protect itself and other system components

- **User mode** and **kernel mode (supervisor, system or privileged)**
- **Mode bit** provided by hardware
  - 0→ kernel mode
  - 1→ user mode

- **Advantages of Dual mode**
  - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions designated as **privileged**, only executable in kernel mode

- Whenever a user program request a service from os, a system call is invoked
  - A system call is treated by the H/W as a software interrupt(**trap**)
  - Ctrl passes through the Interrupt vector to a service routine in OS and **mode bit is set to kernel mode**.
  - The kernel examines the interrupting instruction to **determine what sys call has occurred?**
  - A parameter indicates what type of service the user program is requesting?
  - ```
    if (divisor == 0) {
    //Writing a message to stderr, and exiting with failure.
     fprintf(stderr, "Division by zero! Aborting...\n");
    exit(EXIT_FAILURE);   /* indicate failure.*/ }
    ```
  - Returns control to the instructions following the Sys call.
    ```

Transition from User to Kernel Mode



1.5.2 Timer

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

6.Process Management

- A process is a program in execution. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- **Multi-threaded process has multiple program counter each per thread**

Process Management Activities

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

1. Scheduling processes and threads on the CPU
2. Creating and deleting both user and system processes
3. Suspending and resuming processes
4. Providing mechanisms for process synchronization
5. Providing mechanisms for process communication

7.Memory Management

- Is a large array of words/bytes
- CPU reads instruction from main memory
- All instructions in memory will execute in order
- When the program terminates, mem space is declared available and next pgm is loaded.
- **Memory management activities**
  1. Keeping track of which parts of memory are currently being used and by whom
  2. Deciding which processes (or parts thereof) and data to move into and out of memory
  3. Allocating and de allocating memory space as needed

8. Storage Management-
8.1 File system Mgmt

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

- **File-System management**
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    1. Creating and deleting files and directories
    2. Primitives to manipulate files and dirs
    3. Mapping files onto secondary storage
    4. Backup files onto stable (non-volatile) storage media

8.2. Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- **OS activities**
  1. Free-space management
  2. Storage allocation
  3. Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed
  - Varies between WORM (write-once, read-many-times) and RW (read-write)

8.3 Caching

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

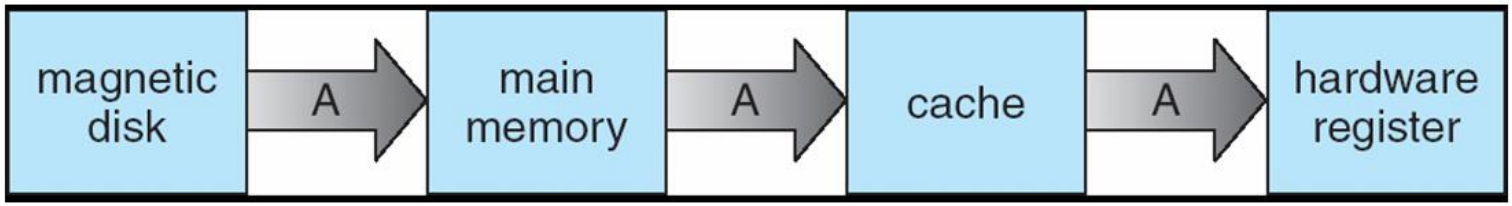
Performance of Various Levels of Storage

-

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist

8.4 I/O Subsystem

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

9. Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control

Protection and Security

- Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

10. Distributed Systems

- Collection of separate, possibly heterogeneous, systems networked together to provide users with access to the various resources.
  - Network is a communications path b/w 2 or more systems
  - Networks vary by
    - Protocols used, distance b/w nodes and transport media
  - Based on distance
    - Local Area Network (**LAN**)
    - Wide Area Network (**WAN**)
    - Metropolitan Area Network (**MAN**)
- **Network Operating System** provides features between systems across network Communication scheme allows systems to exchange messages
- Illusion of a single system

11.Special Purpose systems

- Functions are more limited and objectives is to deal with limited computations domains
  1. Real time Embedded systems
  2. Multimedia Systems
  3. Hand held Systems

11.1Real time Embedded Systems

- Specific task with little or no user intervention, ,spend time in monitoring and managing H/W devices
  - General purpose computers with Std OS
  - H/W devices with special purpose OS
  - H/W devices with ASIC without OS
- Almost always run **RTOS**
- Real time system has well defined, fixed time constraints

11.2 Multimedia Systems

- To handle Multimedia Data(audio, video)
- Frames of video must be delivered with certain time restrictions(30 FPS)
- Applications
  - MP3, DVD movies, video conferencing, live webcasts

11.3 Hand held Systems

- PDAs, smart phones,
- limited CPU, memory, power
- Reduced feature set OS, limited I/O
  - Web clipping

12. Computing Environments

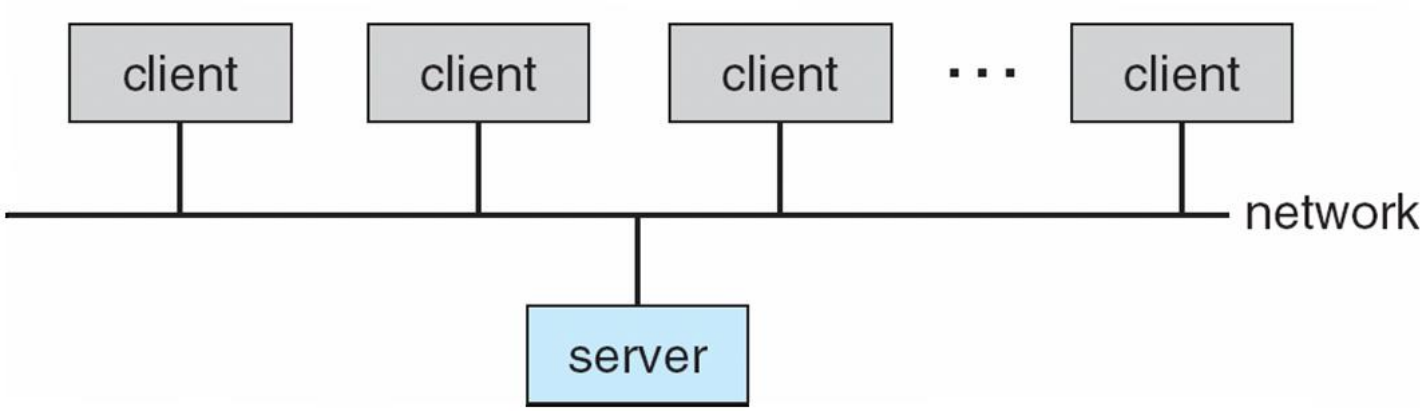
12.1 Traditional computer

- Blurring over time
- Office environment
  - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
  - Now portals allowing networked and remote systems access to same resources
- Home networks
  - Used to be single system, then modems
  - Now firewalled, networked

Computing Environments (Cont)

12.2 Client-Server Computing

- Dumb terminals supplanted by smart PCs
- Many systems now **servers**, responding to requests generated by **clients**
  - **Compute-server** provides an interface to client to request services (i.e. database)
  - **File-server** provides interface for clients to store and retrieve files



Computing Environments..

12.3 Peer-to-Peer Computing

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

Computing Environments

12.4 Web Based Computing

- Web has become ubiquitous
- PCs most prevalent devices
- More devices becoming networked to allow web access
- New category of devices to manage web traffic among similar servers: **load balancers**
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement
- Started by Free Software Foundation (FSF), which has “copyleft” GNU Public License (GPL)
- Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris

End of Chapter 1

Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

1. Operating System Services
2. User Operating System Interface
3. System Calls
4. Types of System Calls
5. System Programs
6. Operating System Design and Implementation
7. Operating System Structure
8. Virtual Machines
9. Operating System Debugging
10. Operating System Generation
11. System Boot

Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

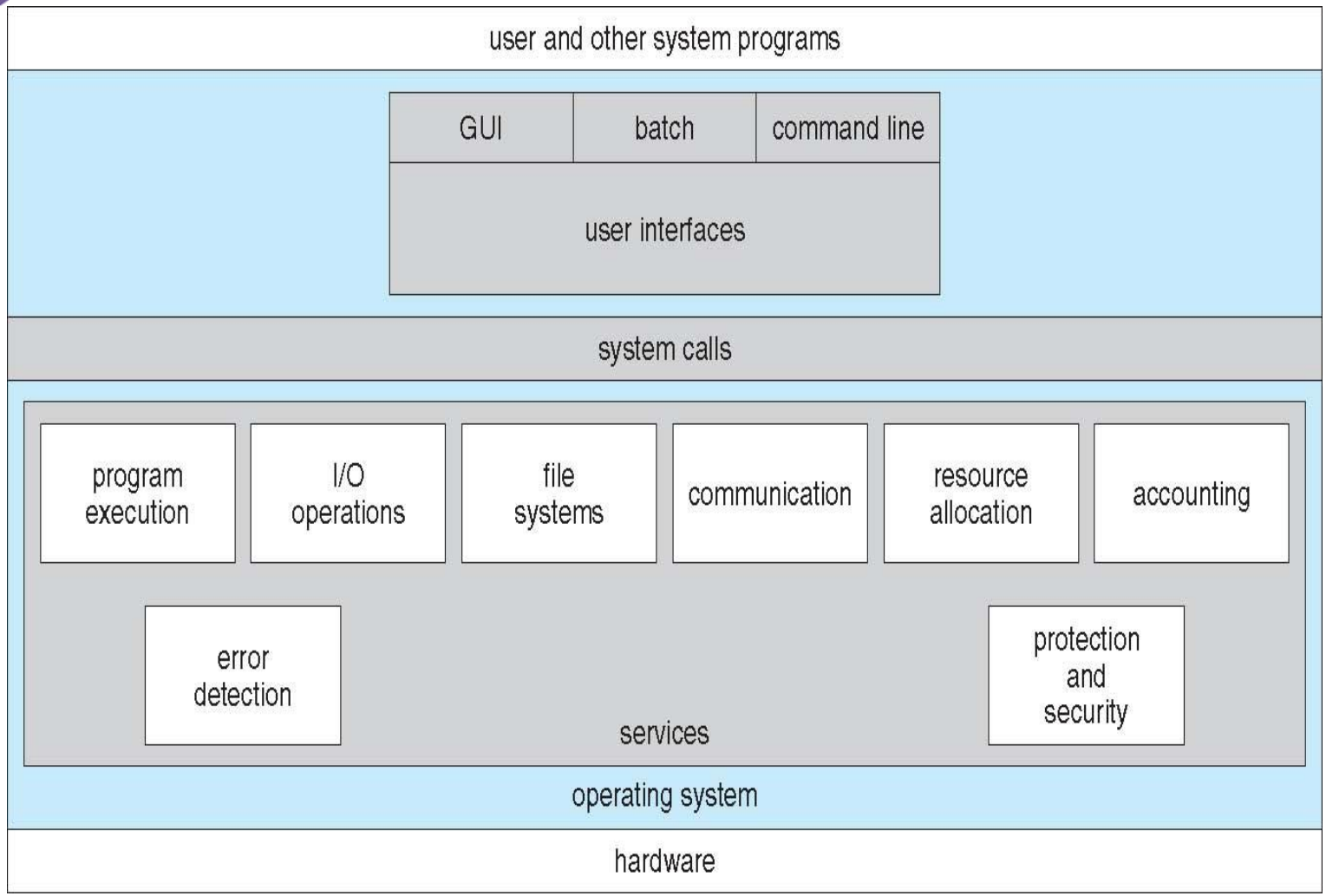
1.Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont)

- Communications – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

A View of Operating System Services



Operating System Services (Cont)

- Another set of OS functions exists for ensuring the **efficient operation of the system itself via resource sharing**
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

2.User Operating System Interface - CLI

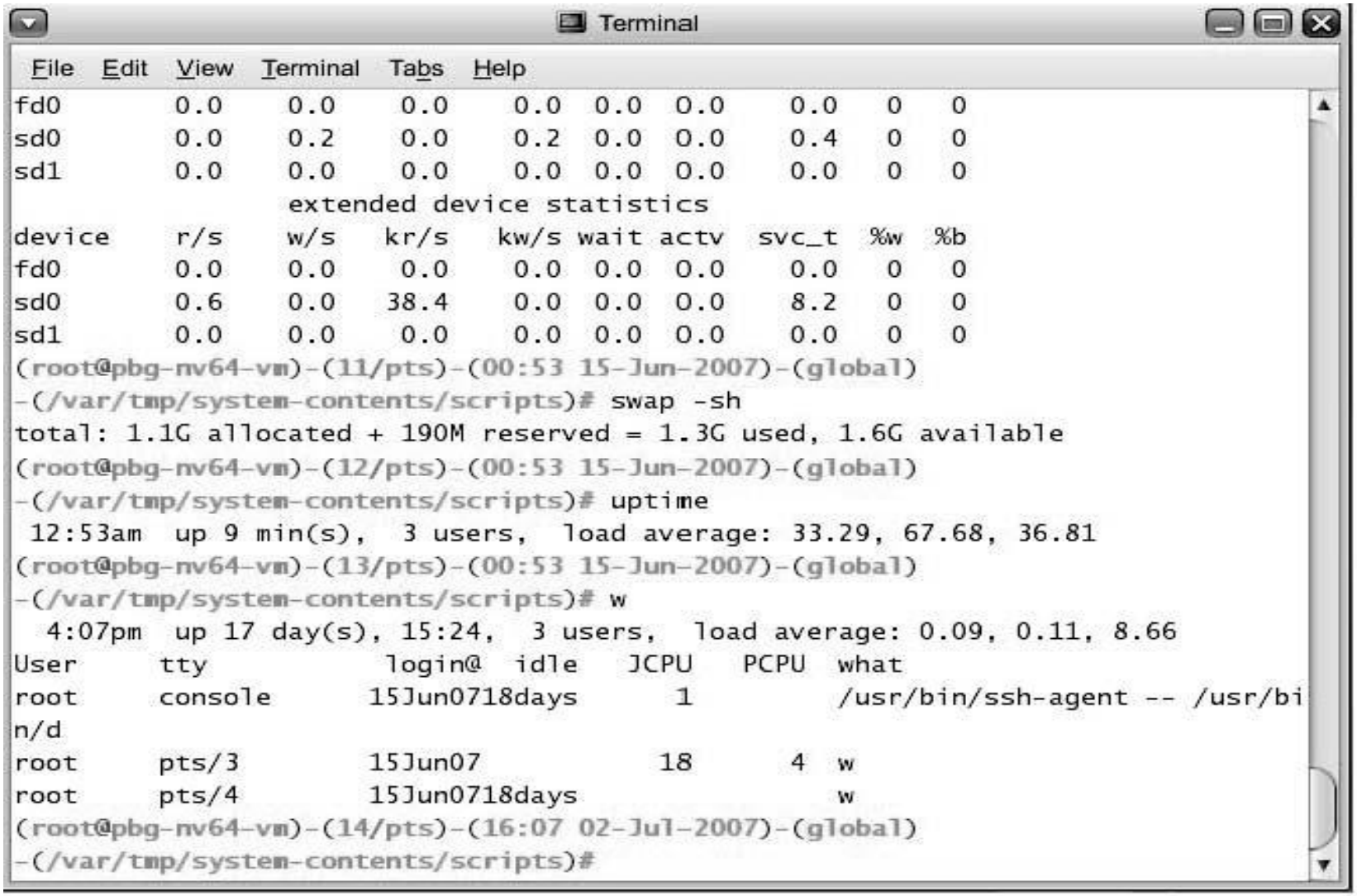
Command Line Interface (CLI) or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors of command interpreter implemented – shells
- Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
    - » If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Bourne Shell Command Interpreter



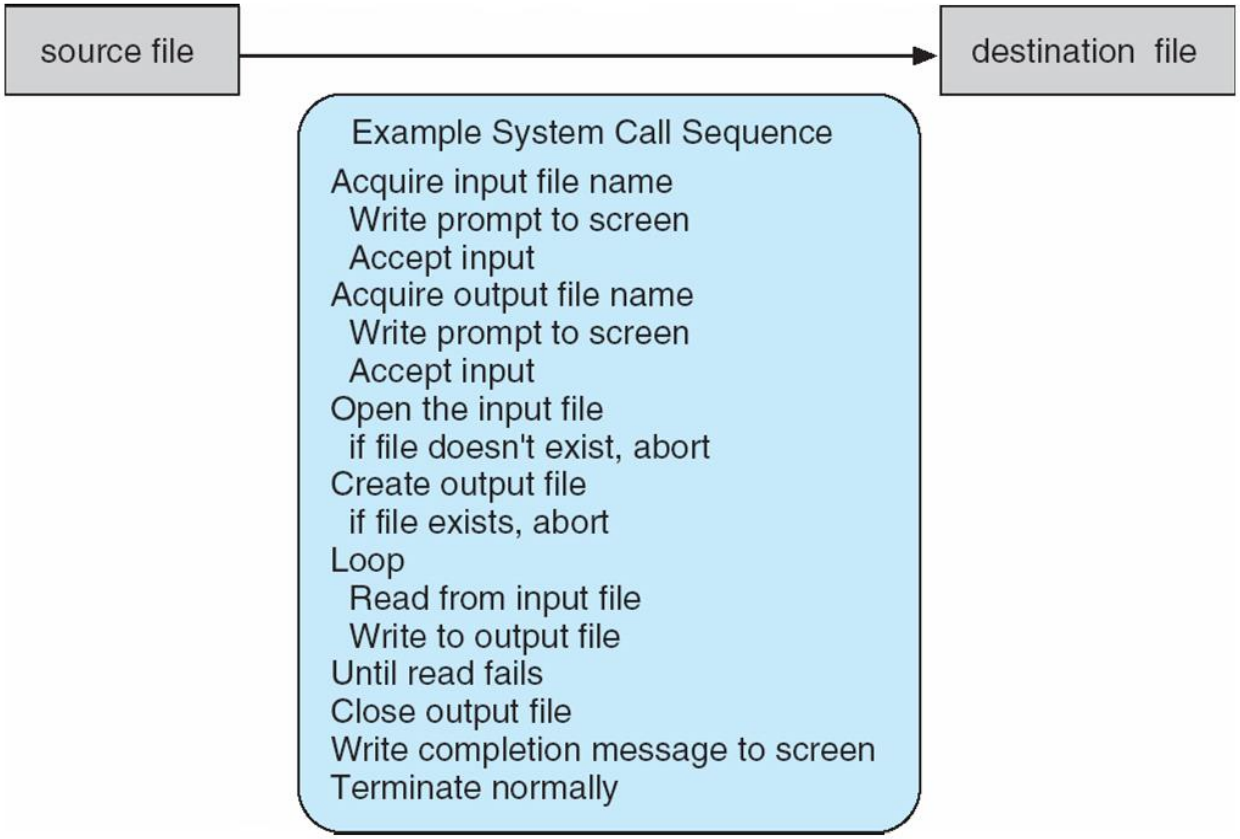
3. System Calls

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

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

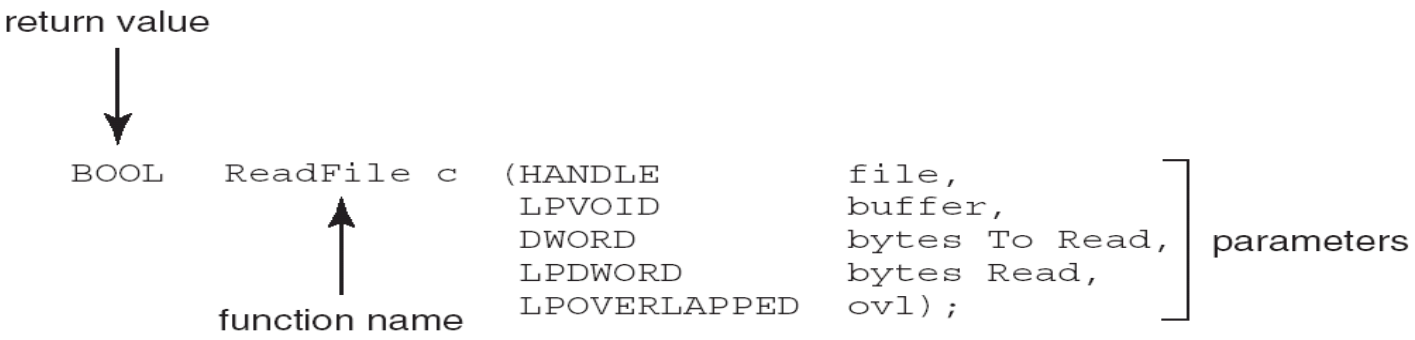
Example of System Calls

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



Example of Standard API

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

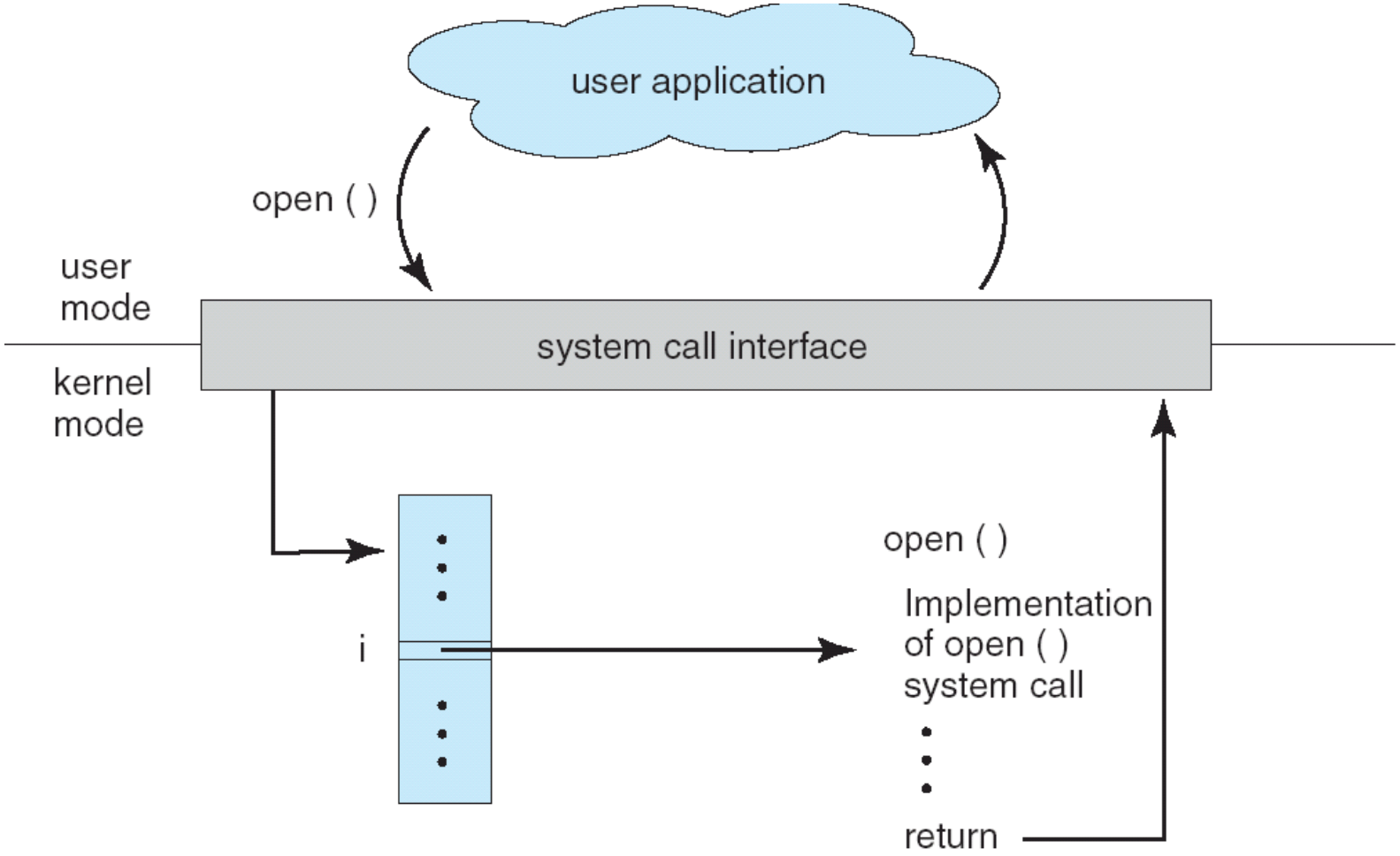


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read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

System Call Implementation

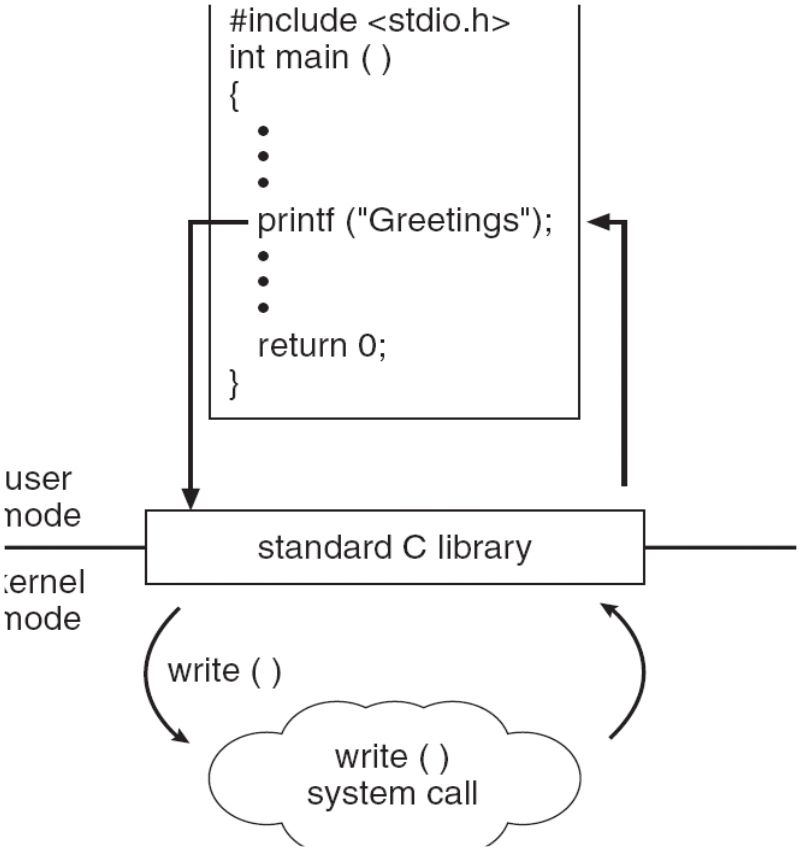
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

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

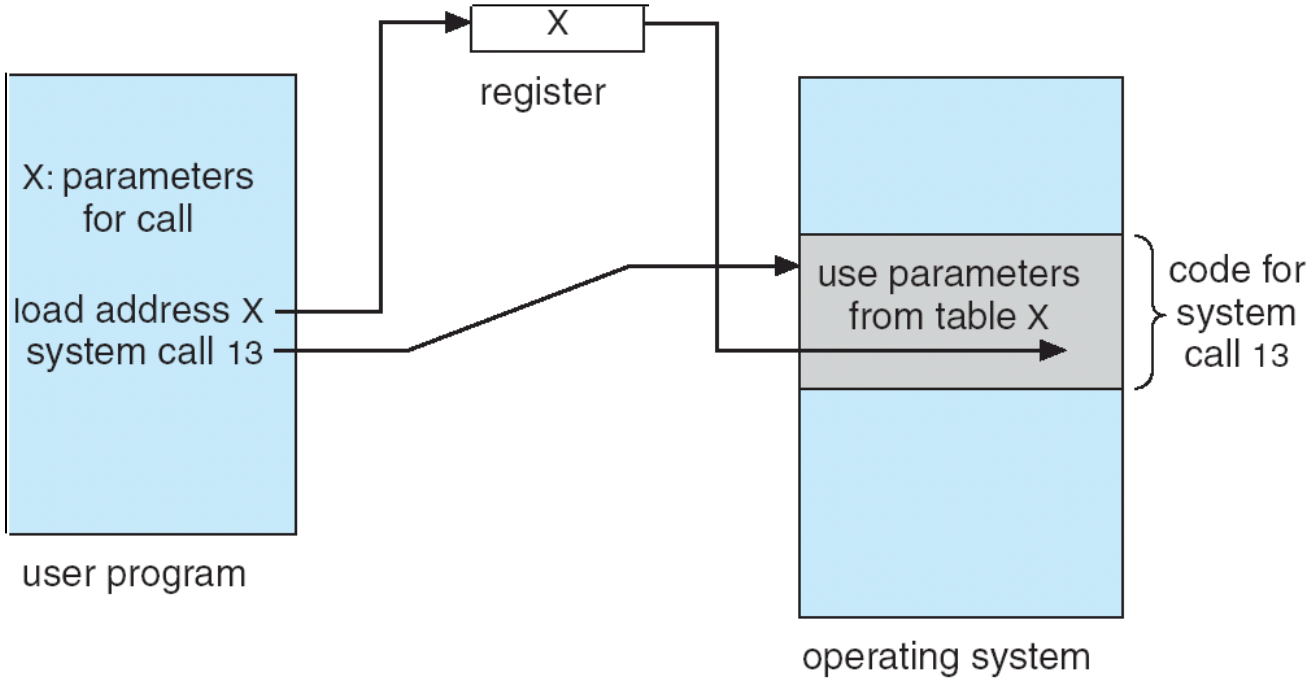


System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- **Three general methods used to pass parameters to the OS**
  1. Simplest: pass the parameters in *registers*
     - In some cases, may be more parameters than registers
  2. Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
     - This approach taken by Linux and Solaris
  3. Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system

Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



4. Types of System Calls

- Process control
  - End, abort
  - Load, execute
  - Create process, terminate process
  - Get process attributes, set process attributes
  - Wait for time
  - Wait event, signal event
  - Allocate and free memory

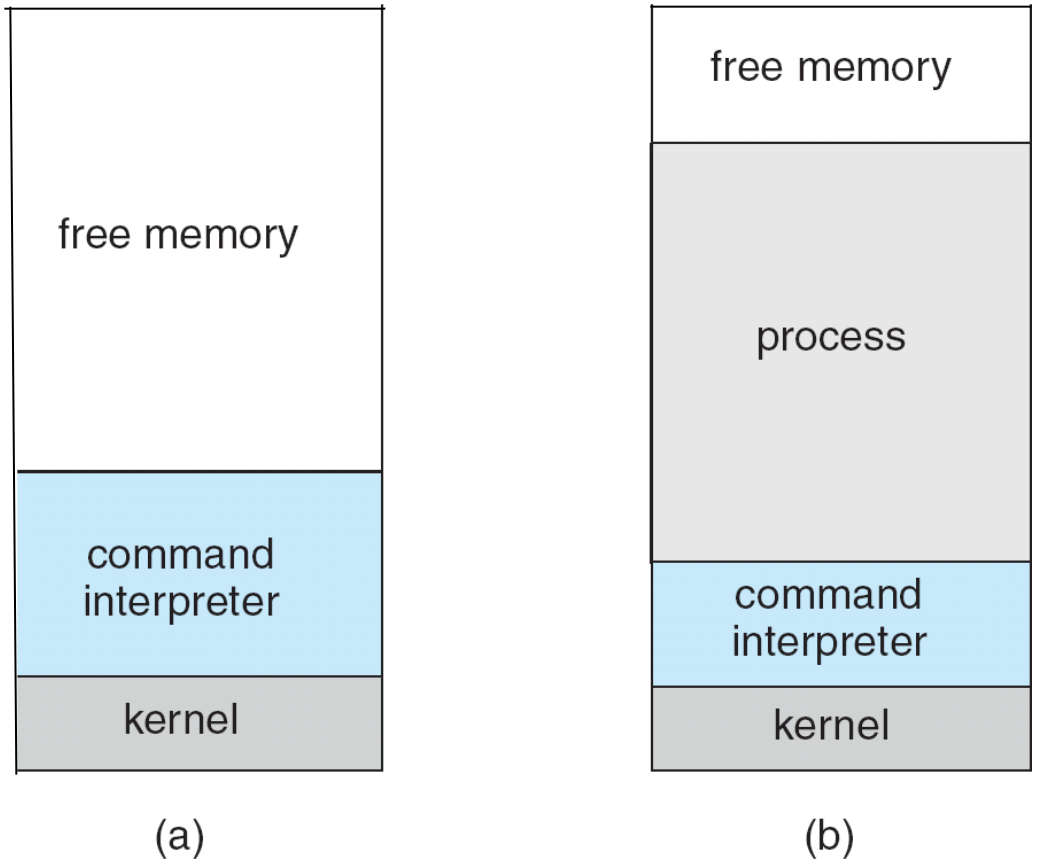
- File management
  - Create file, delete file
  - Open, close
  - Read, write, reposition
  - Get file attributes, set file attributes
- Device management
  - Request device, release device
  - Read, write, reposition
  - Get device attributes, set device attributes
  - Logically attach or detach devices

- Information maintenance
  - Get time or date, set time or date
  - Get system data, set system data
  - Get process , file or device attributes
  - set process , file or device attributes
- Communications
  - Create delete communication connection
  - Send, receive message
  - Transfer status information
  - Attach or detach remote devices
- Protection

Examples of Windows and Unix System Calls

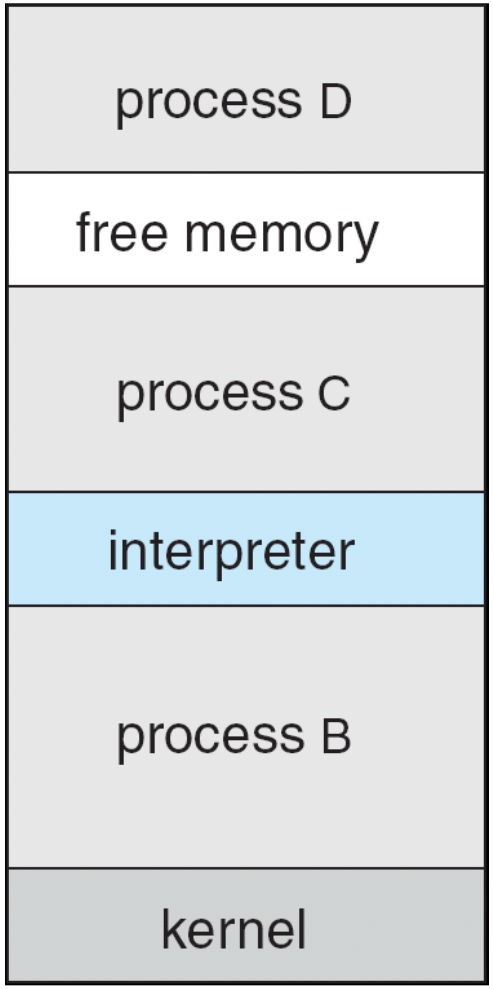
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

MS-DOS execution



(a) At system startup (b) running a program

FreeBSD Running Multiple Programs



5.System Programs

- System programs provide a convenient environment for program development and execution.
- **Also called as System Utilities**
- They are divided into
  - File manipulation (create, delete, copy…)
  - Status information (date, time, logging, debugging)
  - File modification (using text editors, commands)
  - Programming language support (compilers, assemblers, debuggers)
  - Program loading and execution (absolute loaders, linkage editors, relocatable loaders)
  - Communications(send email, remote login, browse web pages..)
  - Application programs (web browsers, spread sheets..)
- Most users' view of the operation system is defined by system programs, not the actual system calls

6.Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- **Design Goals**
  - Problem in designing a system is to define goals and specifications
  - Will be affected by choice of hardware, type of system
  - *User* goals and *System* goals
    - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
    - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont)

Mechanism and Policies

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

- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- **Identify which is policy and mechanism in the following?**
  - Timer construct for CPU protection
  - How long the timer to be set for a particular user

7. OS structure

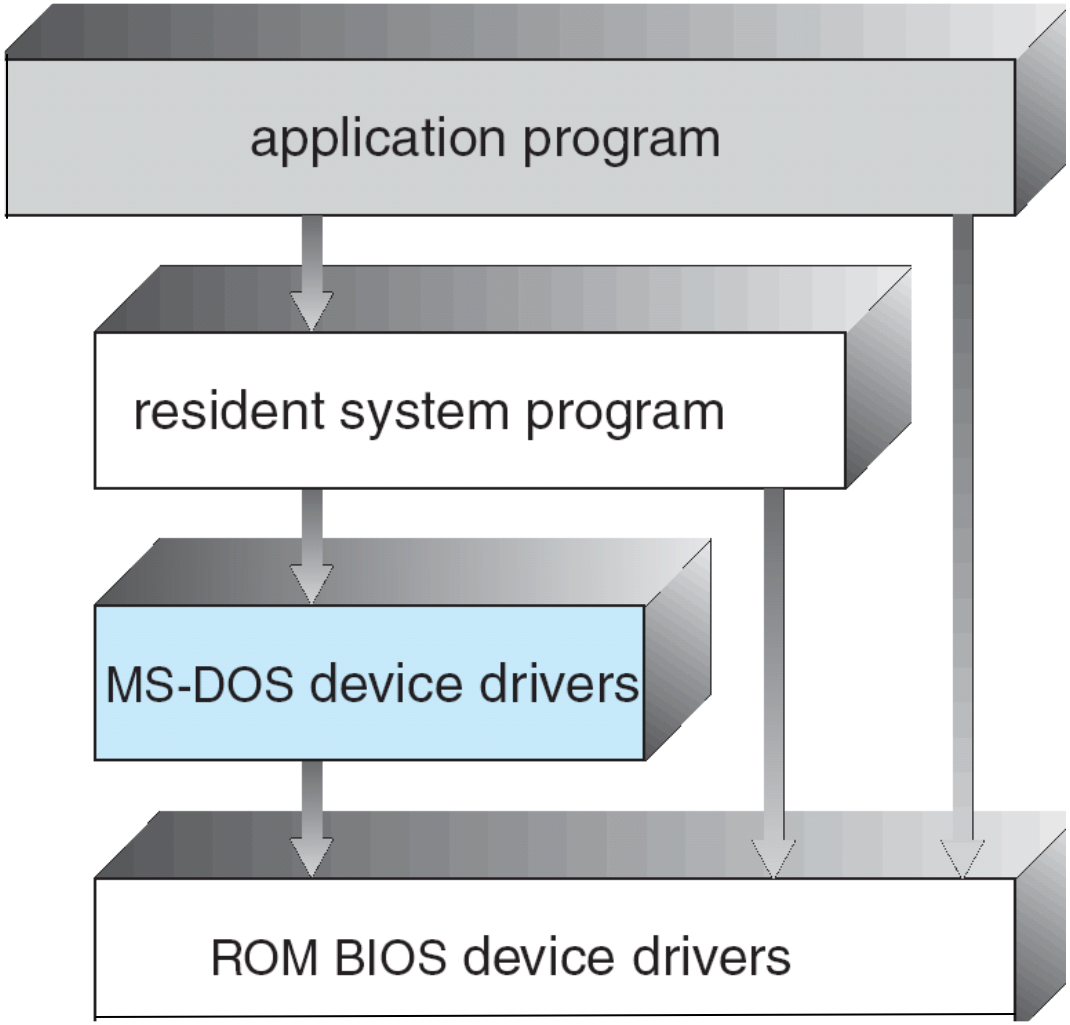
- Simple structure
- Layered Approach
- Micro kernels
- Modules

Simple structure

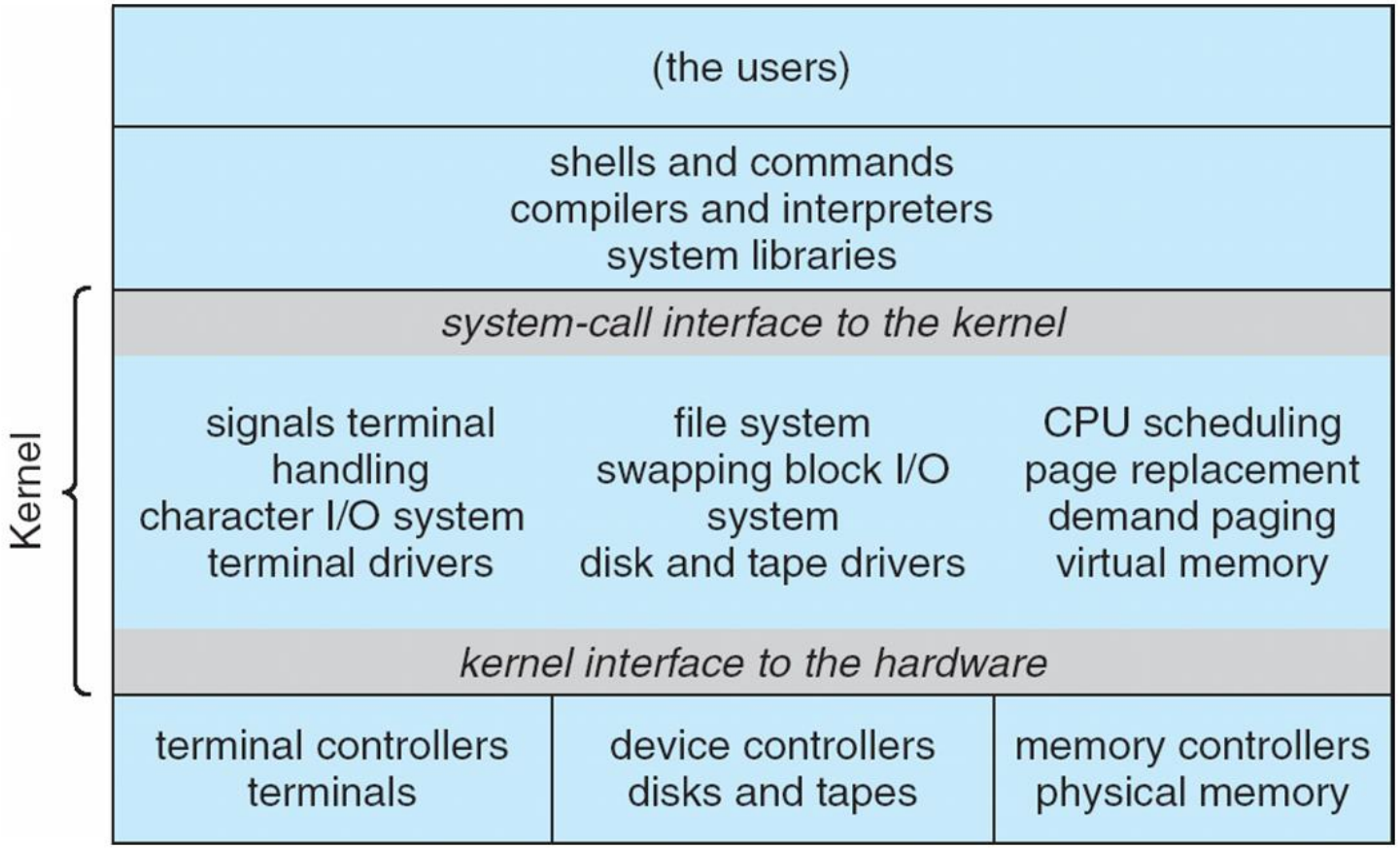
- Many commercial systems started as simple, small and limited systems then grew beyond their original scope.(no well defined structure)
- Eg1→**MS-DOS** – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
  - Limited by hardware , no dual mode, no h/w protection
- Eg2→ Traditional Unix system
  - Limited by H/w
  - 2 seperable parts(kernel, system pgms)
  - Monolithic structure was difficult to implement and maintain

ISE Dept.
Transform Here

MS-DOS Layer Structure



Traditional UNIX System Structure

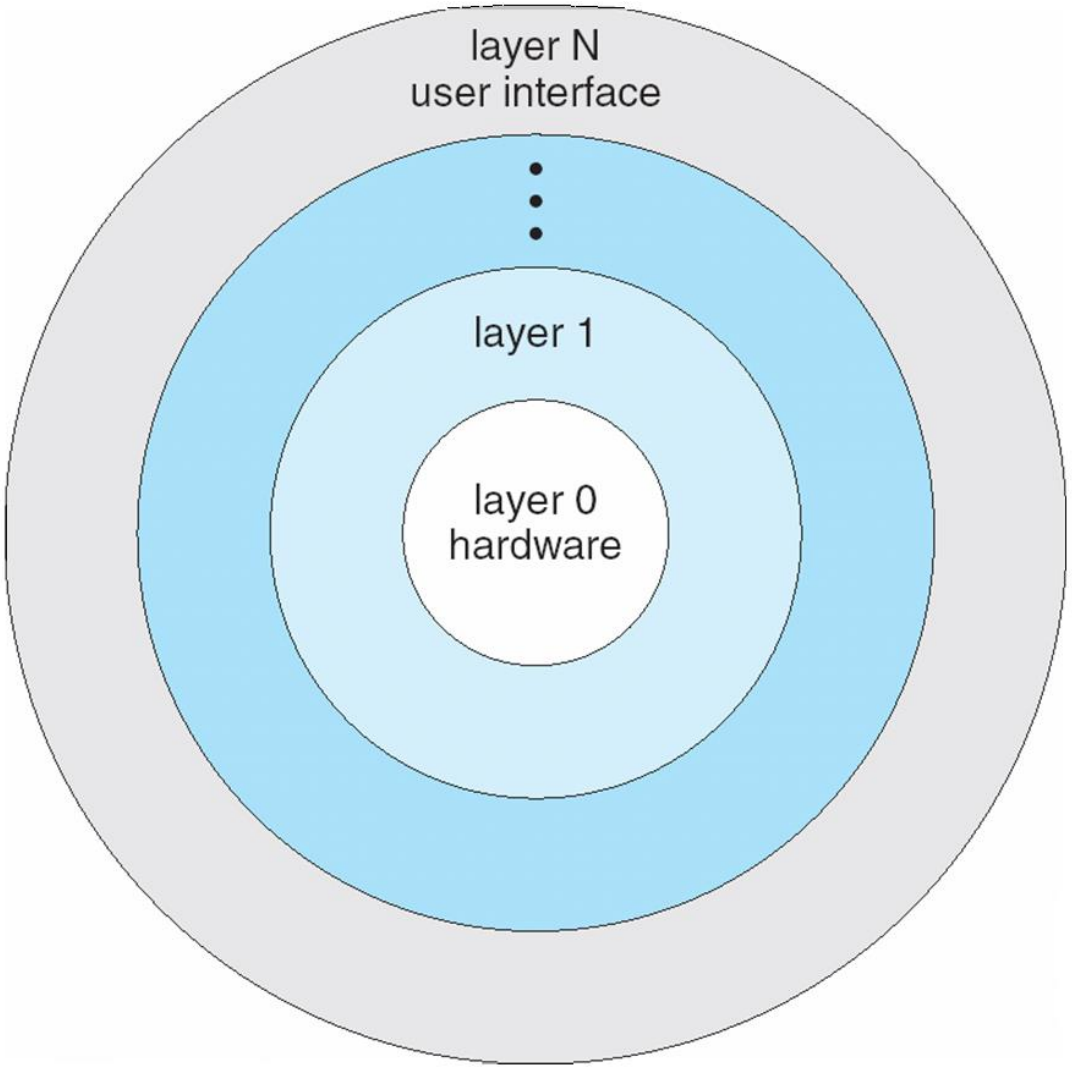


Layered Approach

- The operating system is divided into a number of layers (levels),
- each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware;
  - the highest (layer N) is the user interface.
- **Advantages**
  - Simplicity of construction and debugging
  - layers are selected such that each uses functions (operations) and services of only lower-level layers
  - Each layer is implemented with operations provided by lower layers . Need not to know how it is implemented (simplify the design and implementation)

ISE Dept.
Transform Here

Layered Operating System



Layered Approach

- Disadvantages
  - Appropriate definition of various layers(eg device driver << mem mgmt)
  - Less efficient than other types (more time on sys call)

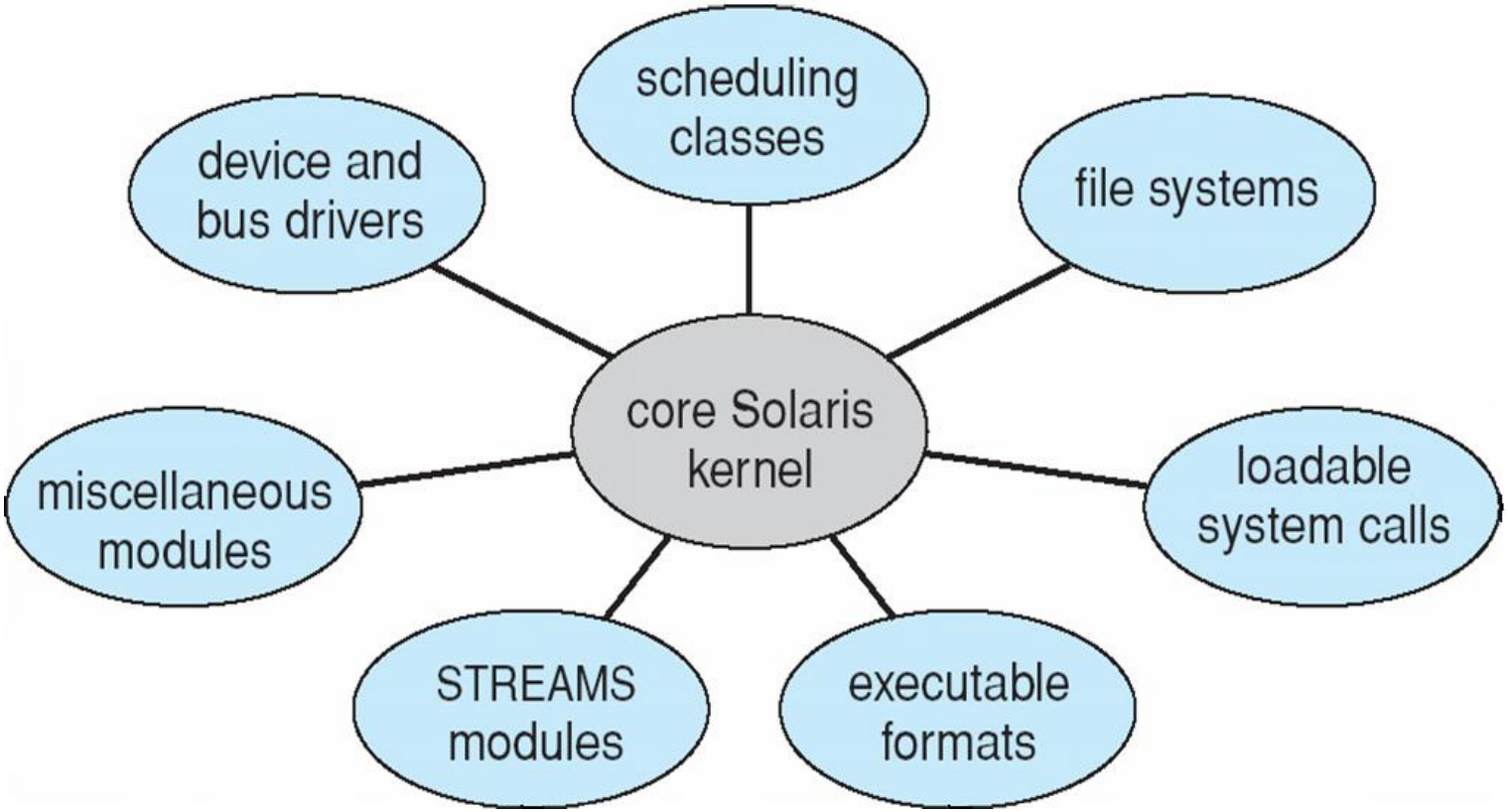
Microkernel System Structure

- Carnegie Mellon university developed an os called **Mach** that uses micro kernel approach
- Structuring the os by removing all non essential components on the kernel.
- Main functionality → communication b/w client pgm and services that are running in user space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel (OS) by adding services to user space
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication
- Eg: Tru64 UNIX, QNX

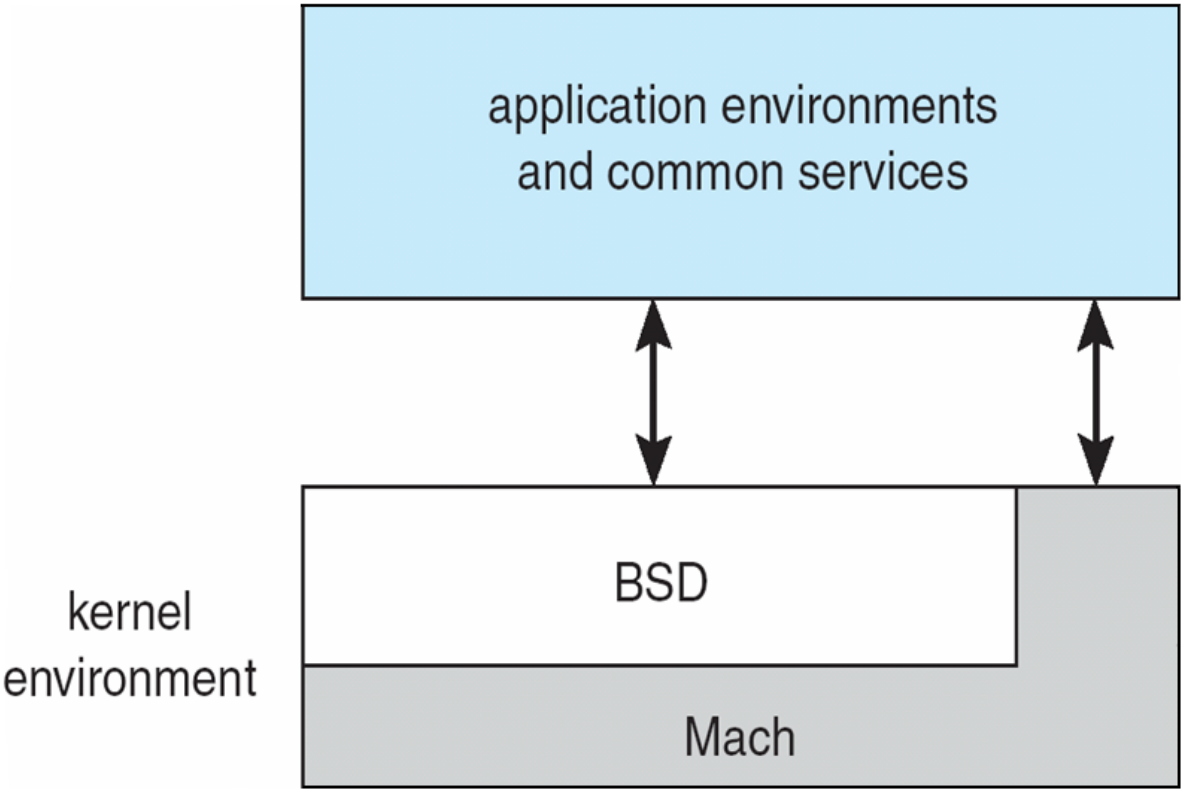
Modules structure

- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- similar to layers (kernel is defined, protected interfaces)but with more flexible any module can call any other module
- Similar to microkernel (core functions and knowledge of how to communicate with other modules) but more efficient bcoz no message passing

Solaris Modular Approach



Mac OS X Structure



8. Virtual Machines

- The logical conclusion of layered approach leads to virtual machine.
- The basic idea→ abstract the H/W of a single computer in to several different exe environment, thereby creating the illusion that each separate exe environment is running its own computer
- 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 History and Benefits

- First appeared commercially in IBM mainframes in 1972
- VM has evolved and still available

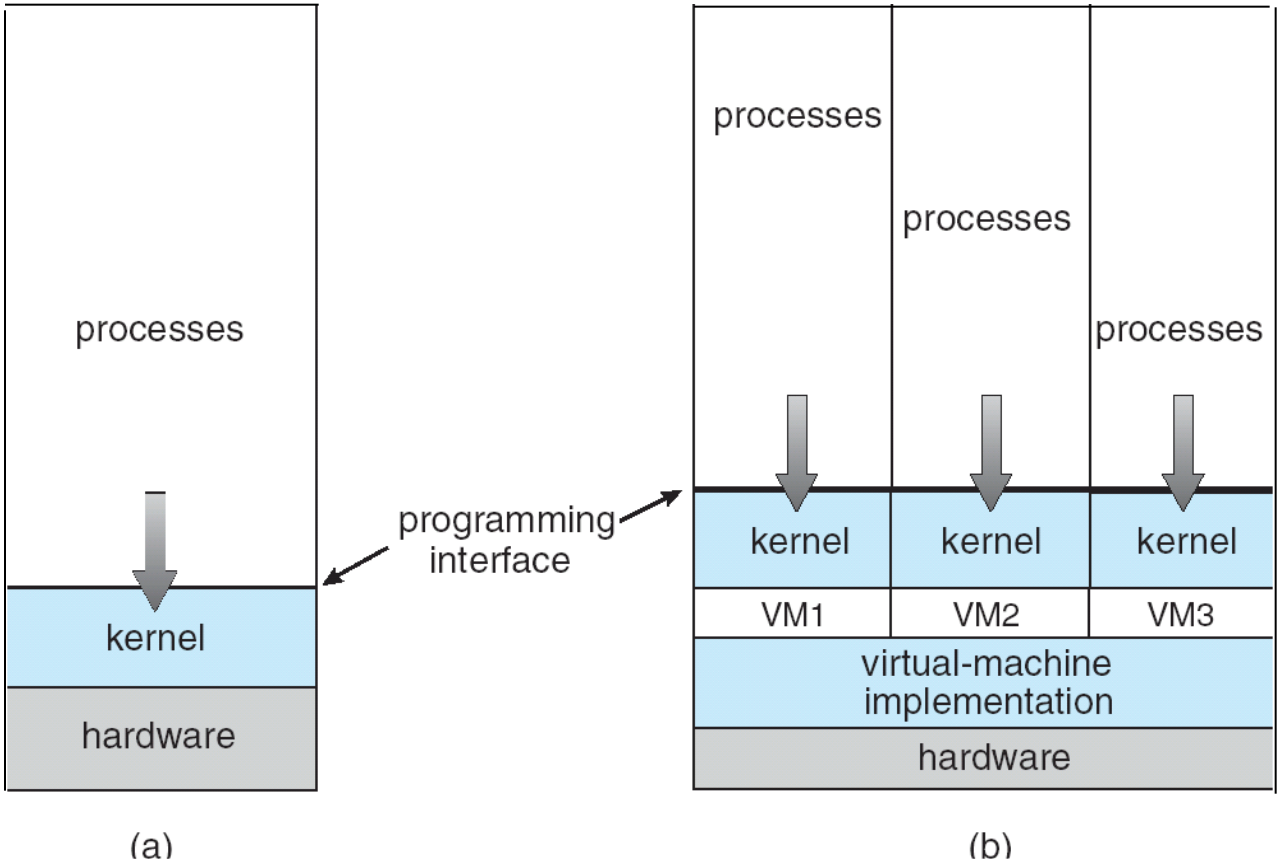
BENEFITS

- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Host system is protected from virtual m/c & VM are protected from each other (no protection problems)
- No direct sharing of resources. It is done in 2 ways
  - Network of VM each of which send info over the virtual comm network
  - Share a file system volume and thus share files

Benefits contd..

- Useful for development, testing and porting(normal systems are not disrupted)
- Consolidation of many low-resource use systems onto fewer busier systems
- "Open Virtual Machine Format", standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms

Virtual Machines (Cont)



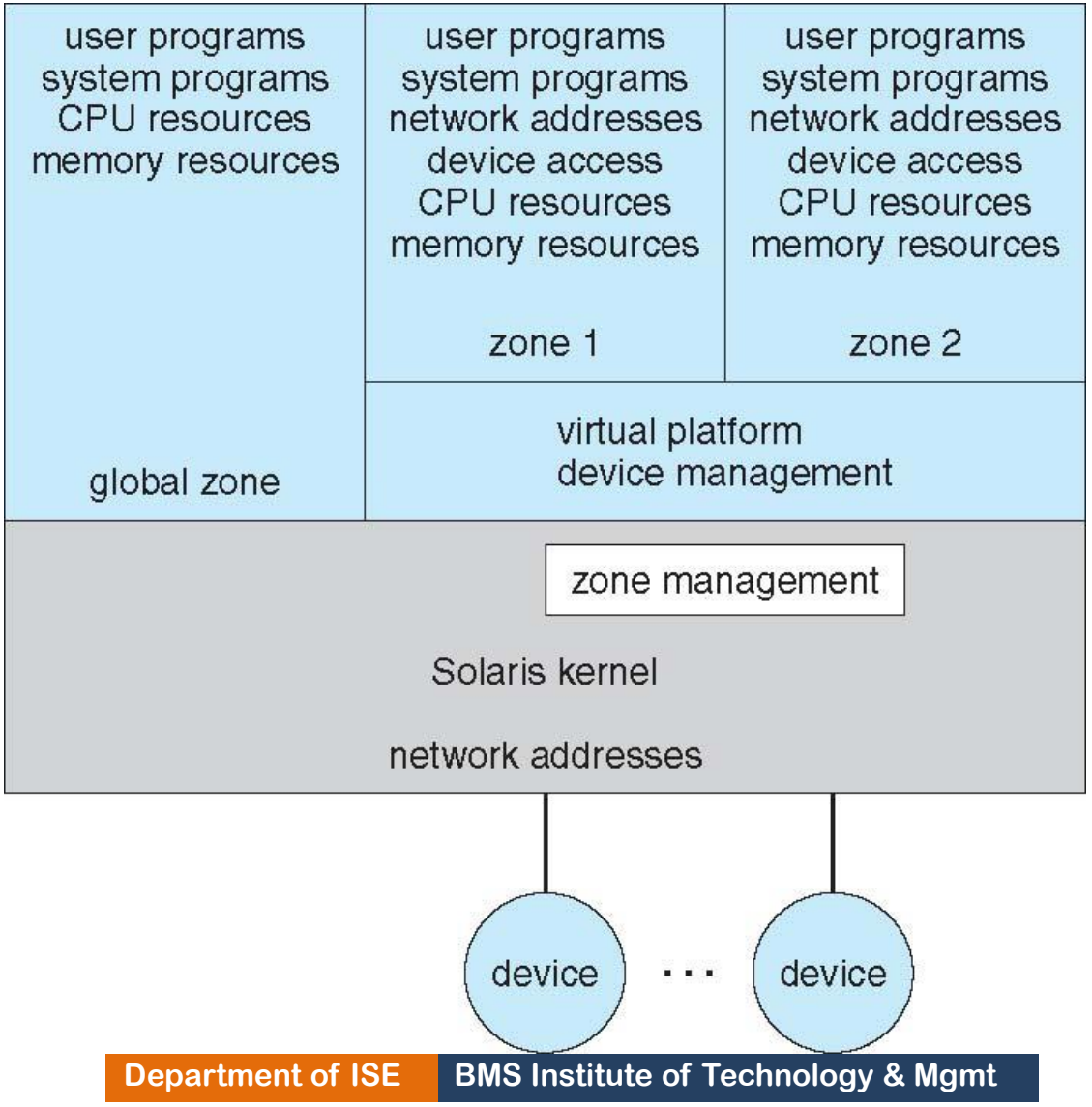
a) Non virtual machines

b) virtual machine

Para-virtualization

- Presents guest with system similar but not identical to hardware
- Guest must be modified to run on paravirtualized hardware
- Guest can be an OS, or in the case of Solaris 10 applications running in containers or zones.
- Adv→more efficient use if resources and a smaller virtualization layer.
- H/W is not virtualized rather os and its devices are virtualized

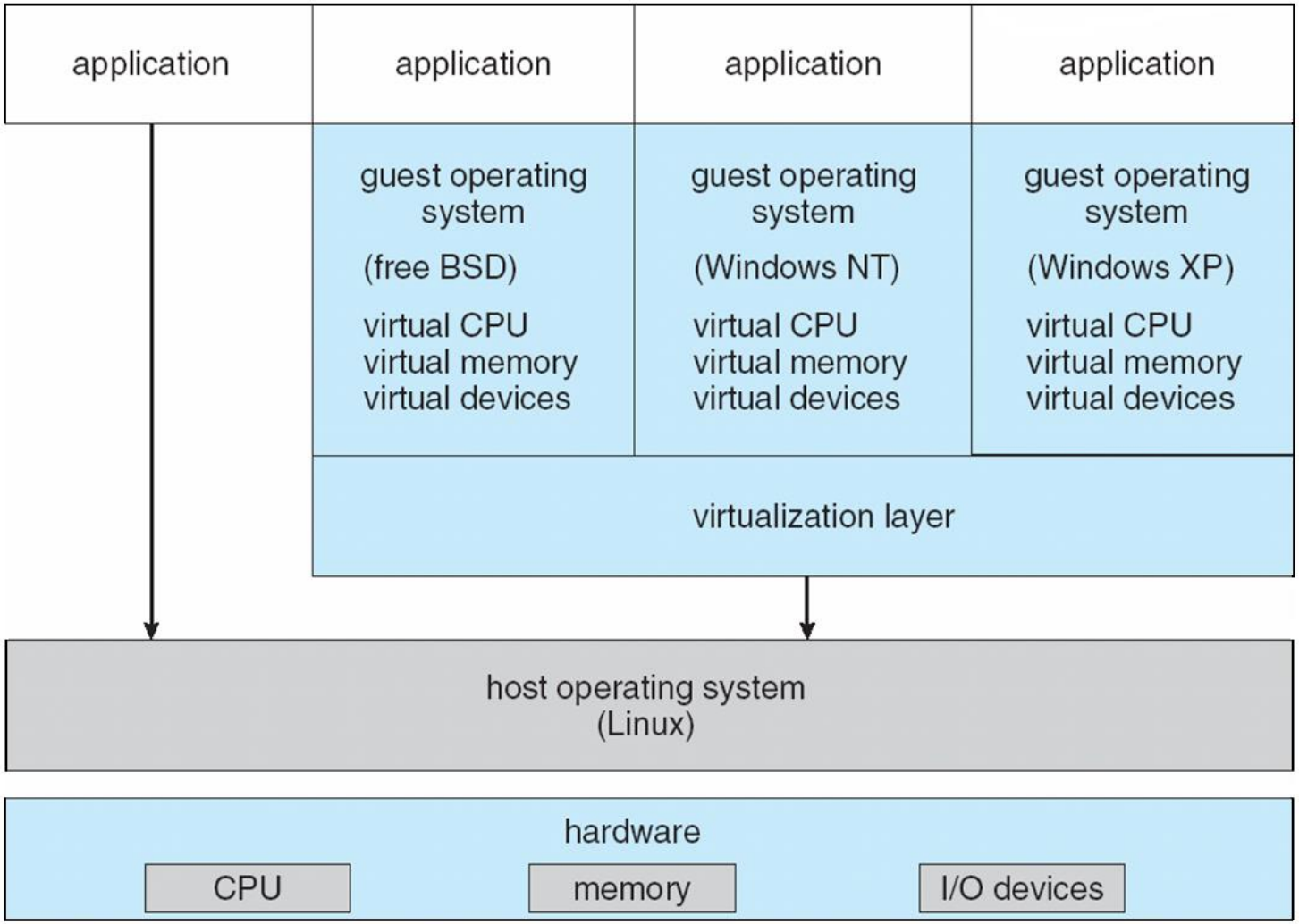
Solaris 10 with Two Containers



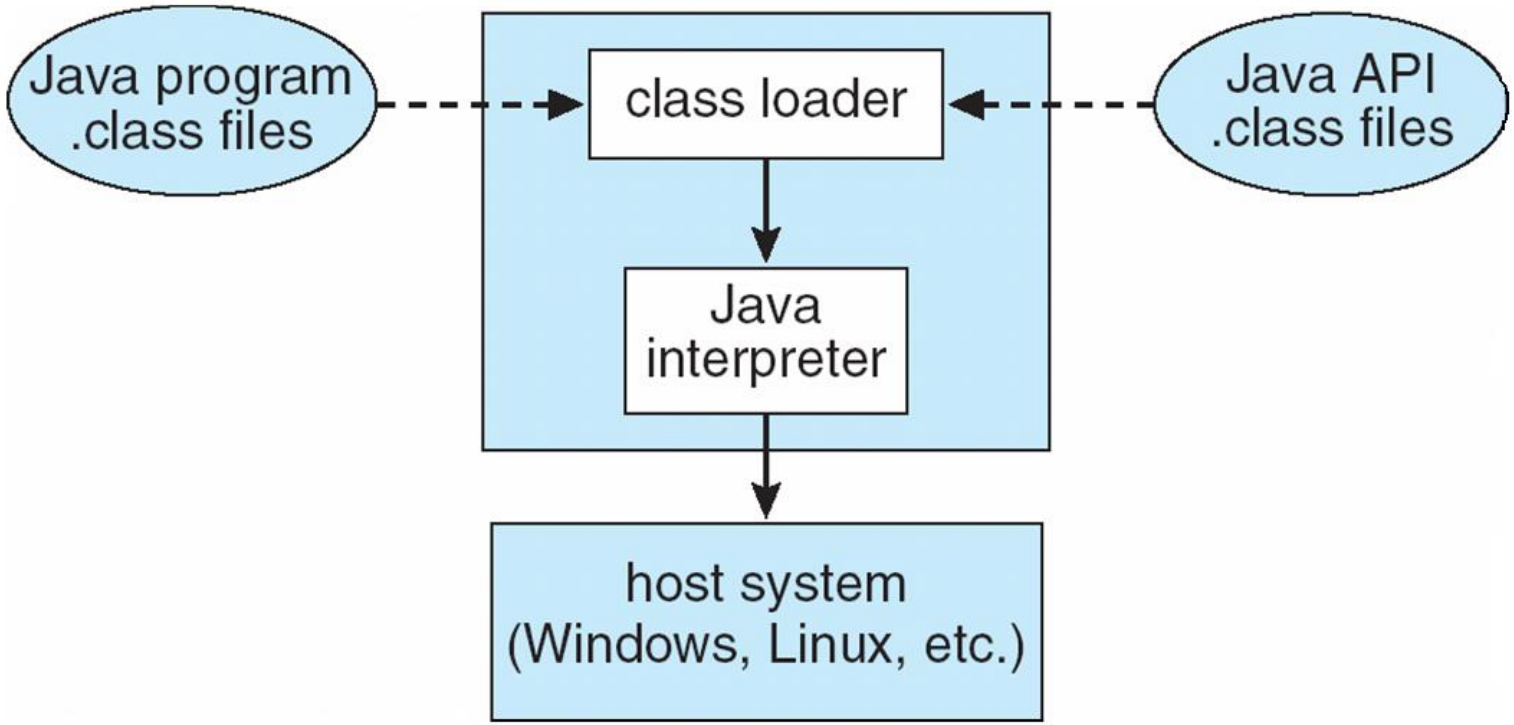
Virtual Machine Implementation

- Difficult to implement – must provide an *exact* duplicate of underlying machine
  - Typically runs in user mode, creates virtual user mode and virtual kernel mode
- Timing can be an issue – slower than real machine
- Hardware support needed
  - More support-> better virtualization
  - i.e. AMD provides “host” and “guest” modes

VMware Architecture



The Java Virtual Machine



9.Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- **Kinds of info to be determined?**
  - What CPU to be used? What options are installed? For multiple CPU each CPU may be described.
  - How will the boot disk be formatted? How many sections or partitions will be separated in to, and what will go into each partitions?
  - How much memory is available?
  - What devices are available?
  - What OS options are desired?

- The previous information is determined and used in several ways
  1. A sys admin can use it to modify a copy of the source code of OS. The result is tailored
  2. System description can lead to the creation of tables and the selections of modules from a precompiled library.

11.System Boot

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

End of Chapter 2

Chapter 3: Process Management

Chapter 3: Processes

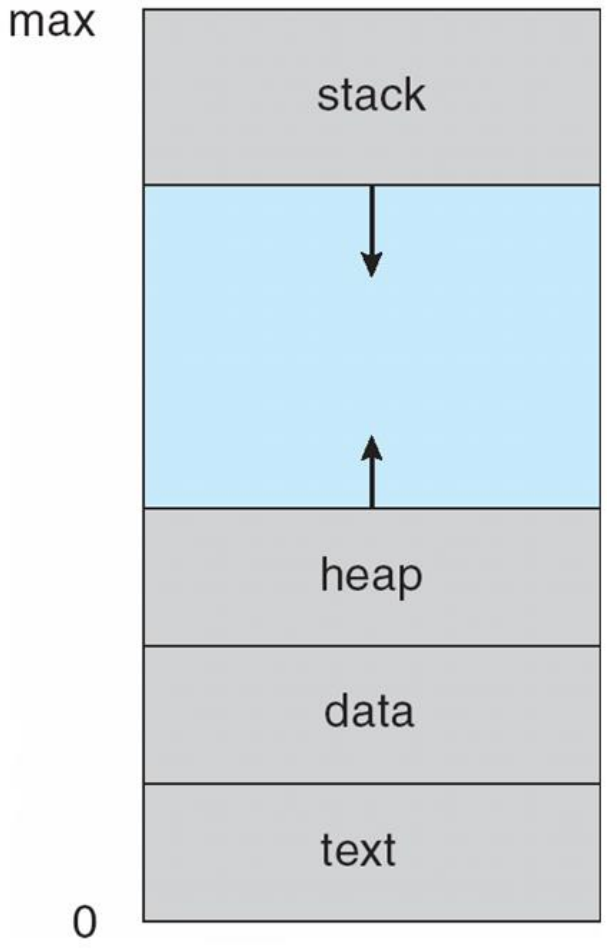
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

1.1Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

ISE Dept.
Transform Here

Process in Memory

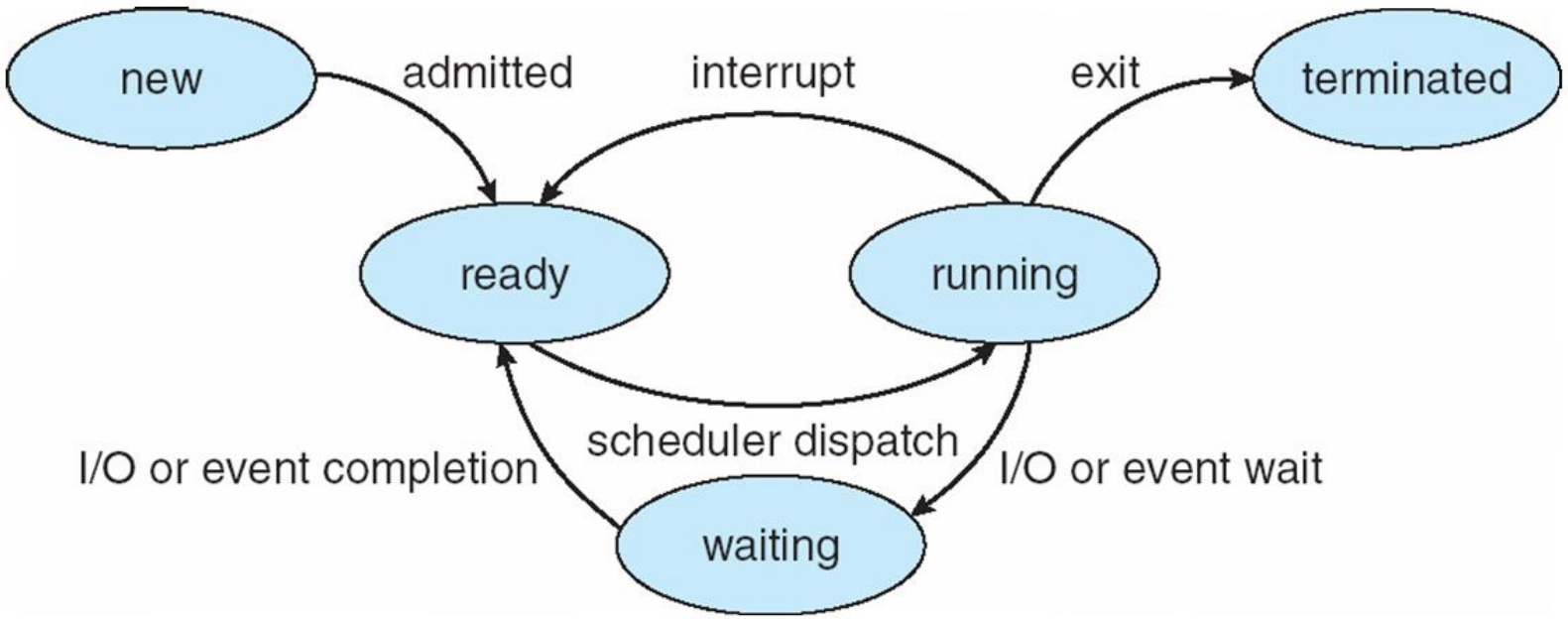


1.2Process State

- 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

ISE Dept.
Transform Here

Diagram of Process State



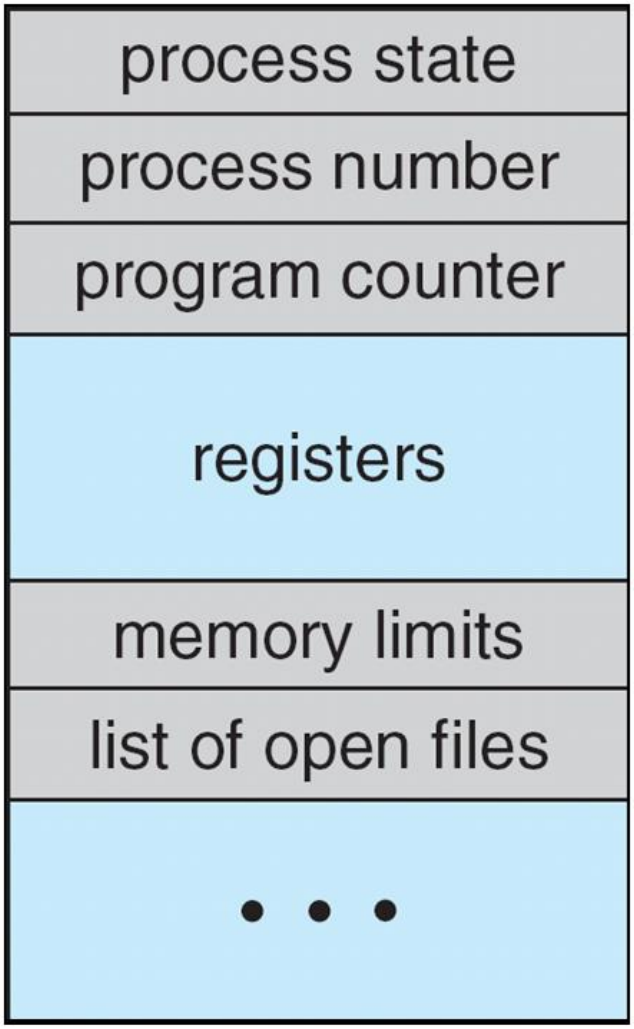
1.3 Process Control Block (PCB)

- Also called as task control block
- Each process is represented in the OS by a PCB

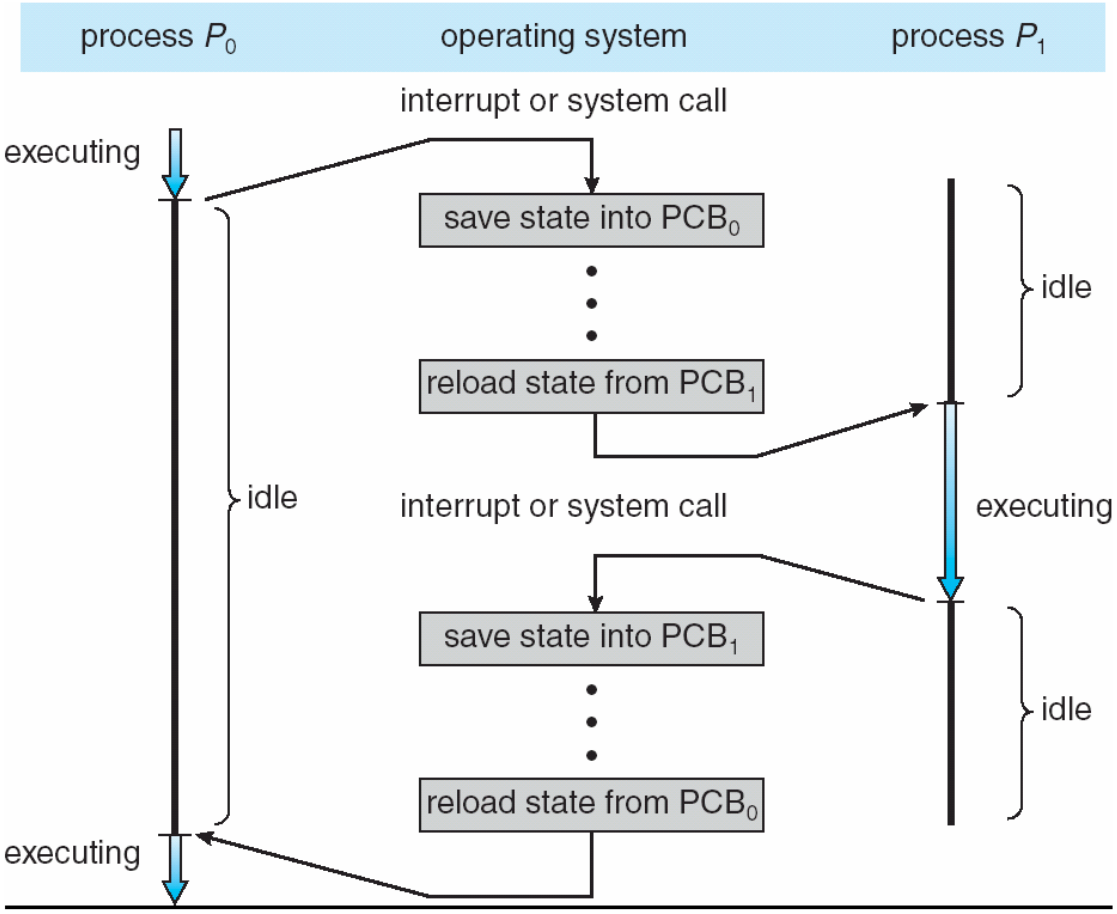
  Information associated with each process
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

- Process state→ new, ready, running, waiting,,,,
- Pgm counter→ addres of next inst to be executed
- Cpu registers→ acc, index reg, SP, GPR
  - Along with the pgm counter ,this state info to be saved when an interrupt occurs, to allow the process to be continued correctly
- CPU scheduling info→ priority, scheduling queues
- Mem mgmt info →value of BR, LR, PT, ST
- Accounting info→amount of CPU and real time used, time limits, acc no, process no
- I/O status info→ list of I/O devices allocated, list of open file

Process Control Block (PCB)



CPU Switch From Process to Process

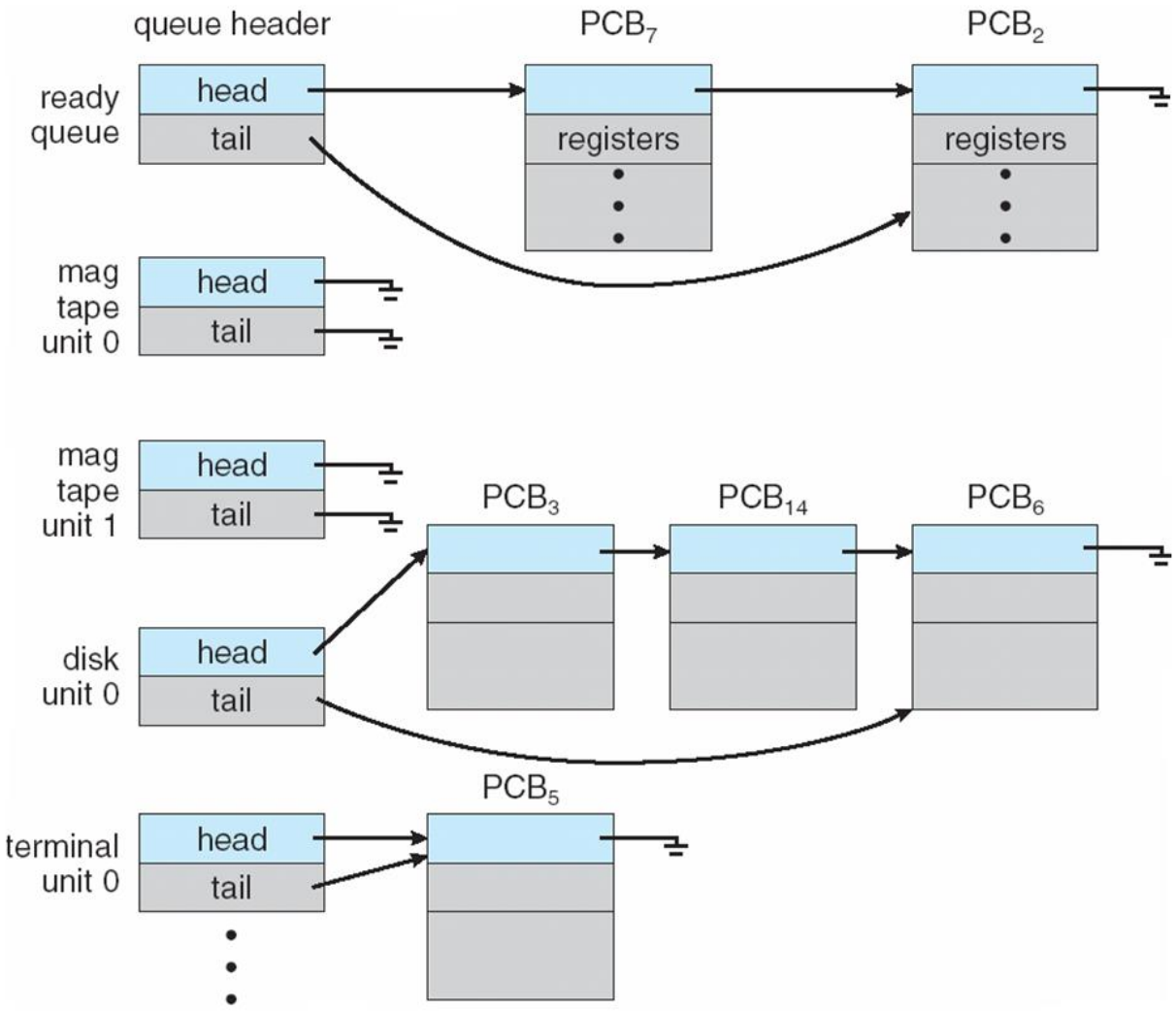


2.1 Process Scheduling Queues

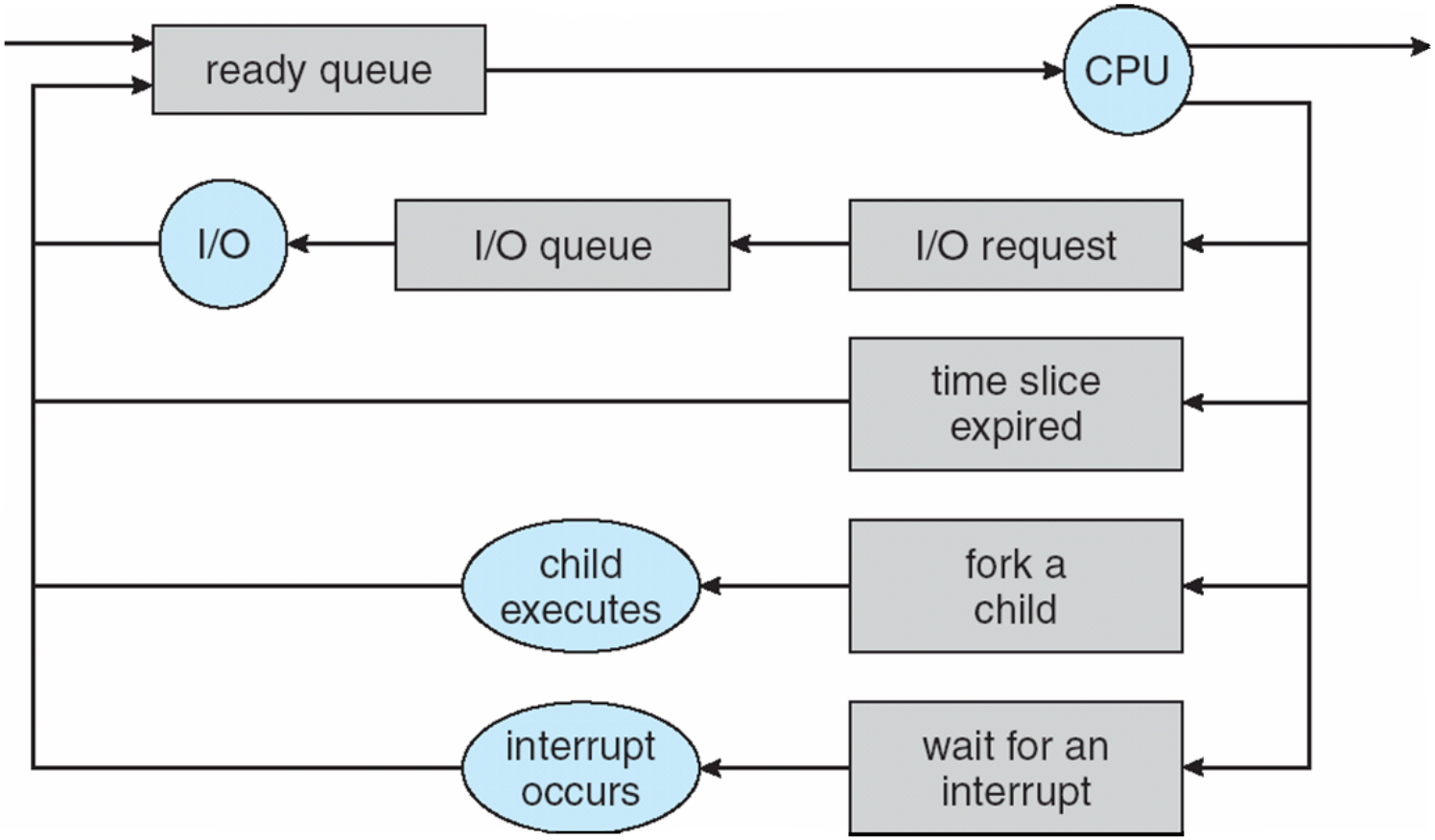
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

ISE Dept.
Transform Here

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



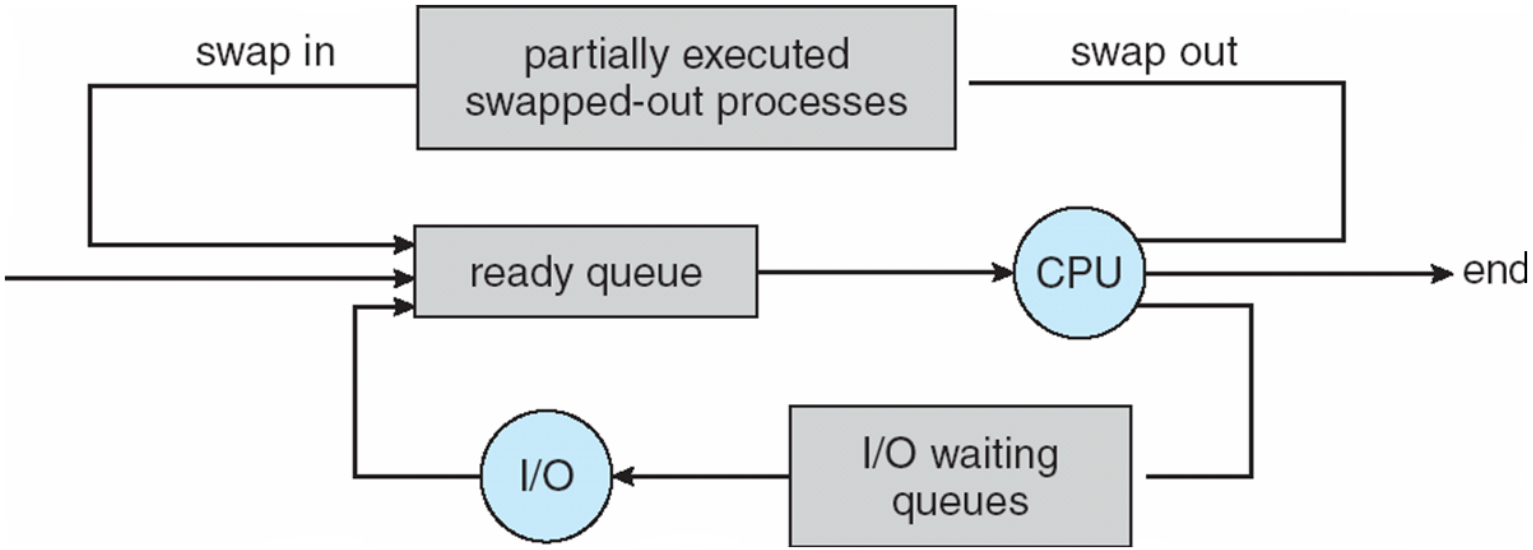
2.2 Schedulers

- **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

Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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



2.3 Context Switch

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

Operations on Procesess

- Process Creation
- Process Termination

2.3 Process Creation

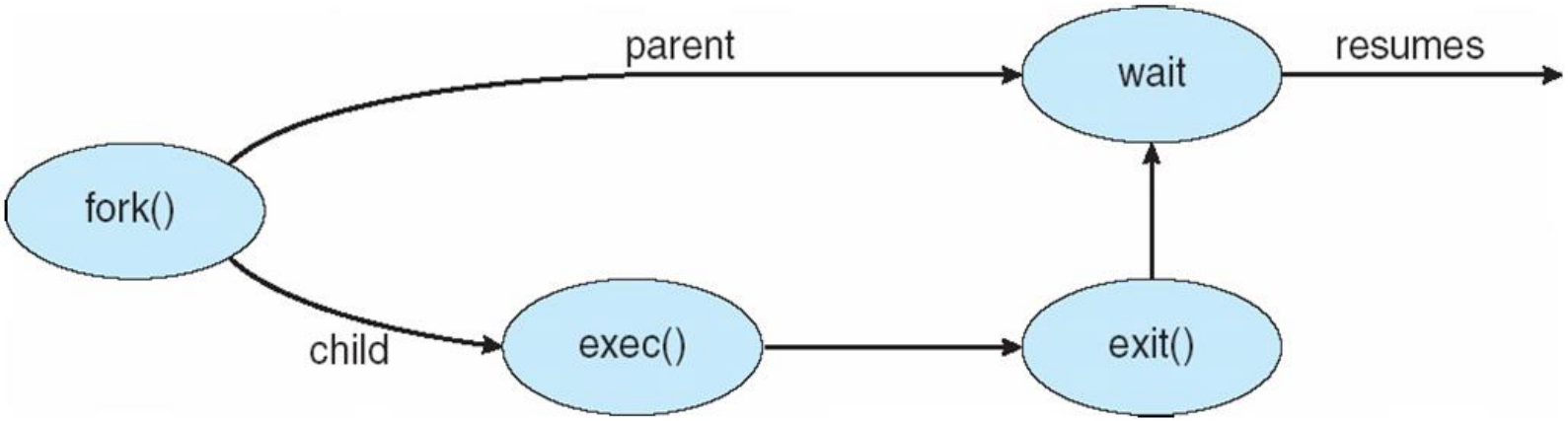
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's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

Process Creation (Cont)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

ISE Dept.
Transform Here

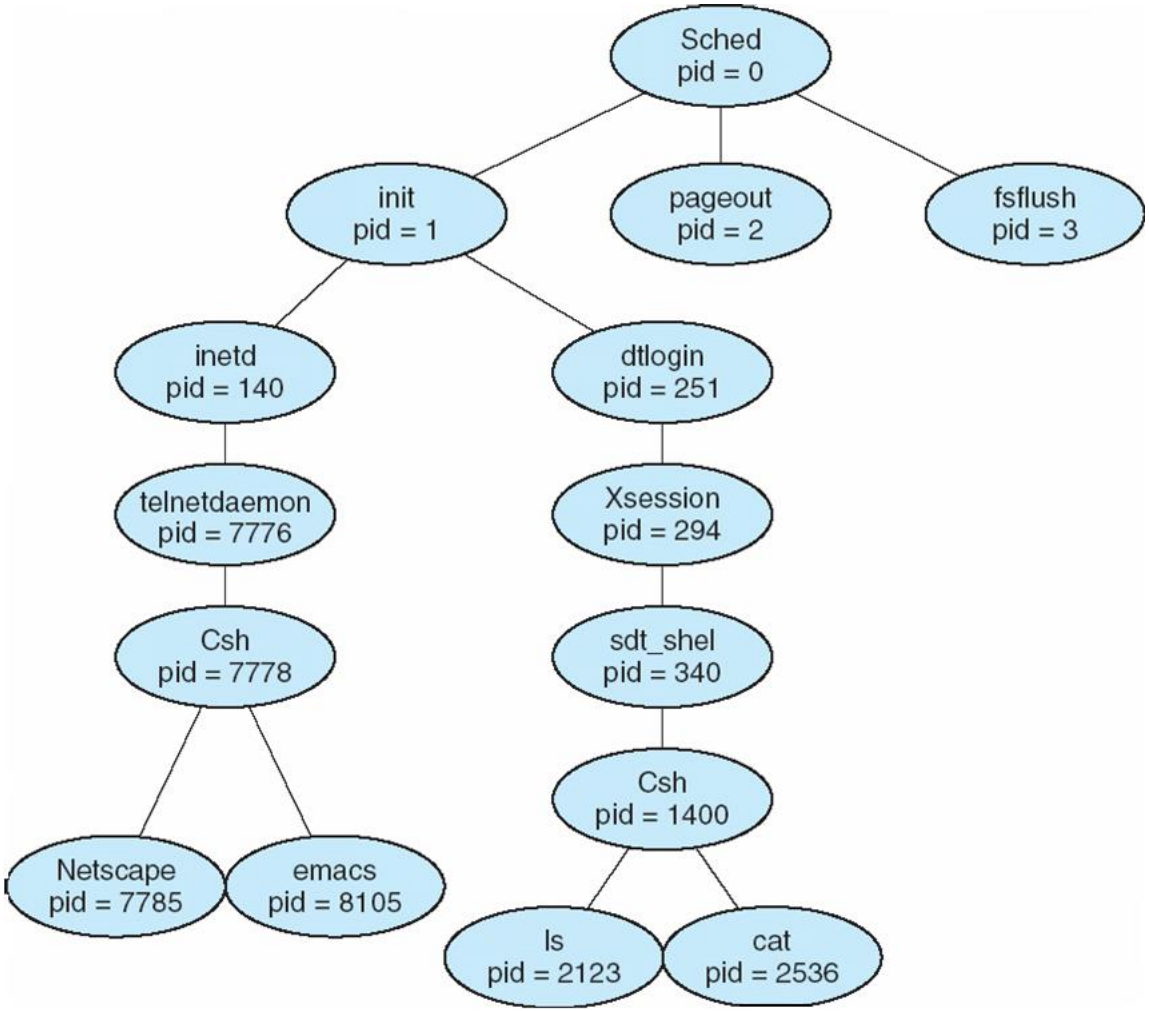
Process Creation



C Program Forking Separate Process

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

A tree of processes on a typical Solaris



Process Termination

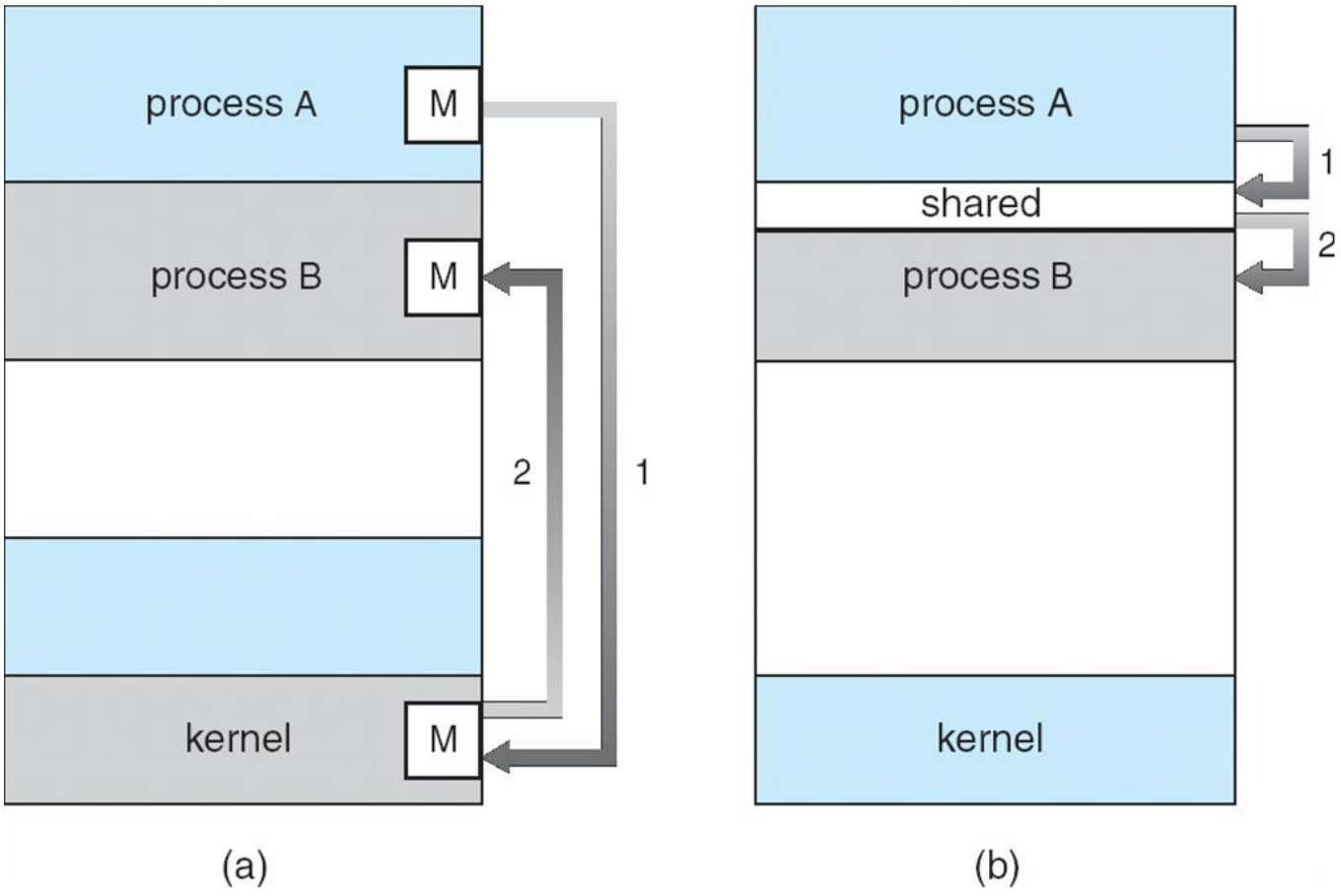
- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



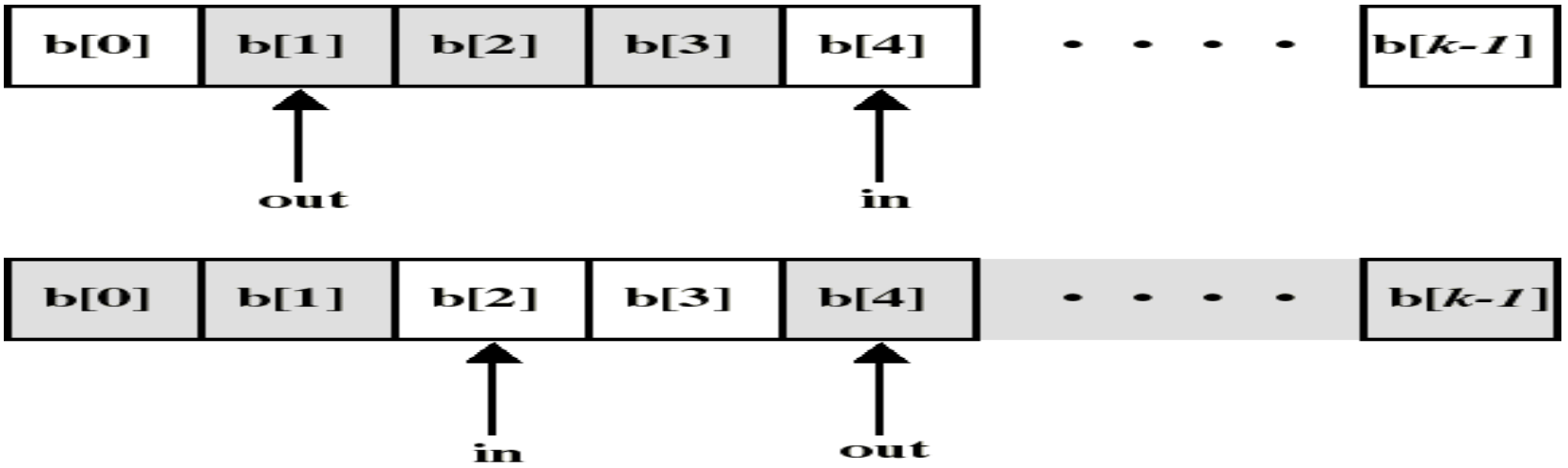
Cooperating Processes

- **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

- The bounded buffer is implemented as a circular array with 2 logical pointers: **`in`** and **`out`**.
- The variable **`in`** points to the next free position in the buffer.
- The variable **`out`** points to the first full position in the buffer.



Bounded-Buffer – Shared-Memory Solution

- Shared data implemented as circular array

```
#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 count)  == 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)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
- Logical implementations of link and send() and receive operations
  - Direct/indirect communication
  - Synchronous / asynchronous communication
  - Automatic / explicit buffering

1. NAMING: DIRECT COMM

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

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

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

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

Indirect Communication

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

2. Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

3.Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – queue lengh is 0
     Sender must block till the recipient receives the msg
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

End of Chapter 3

Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

Objectives

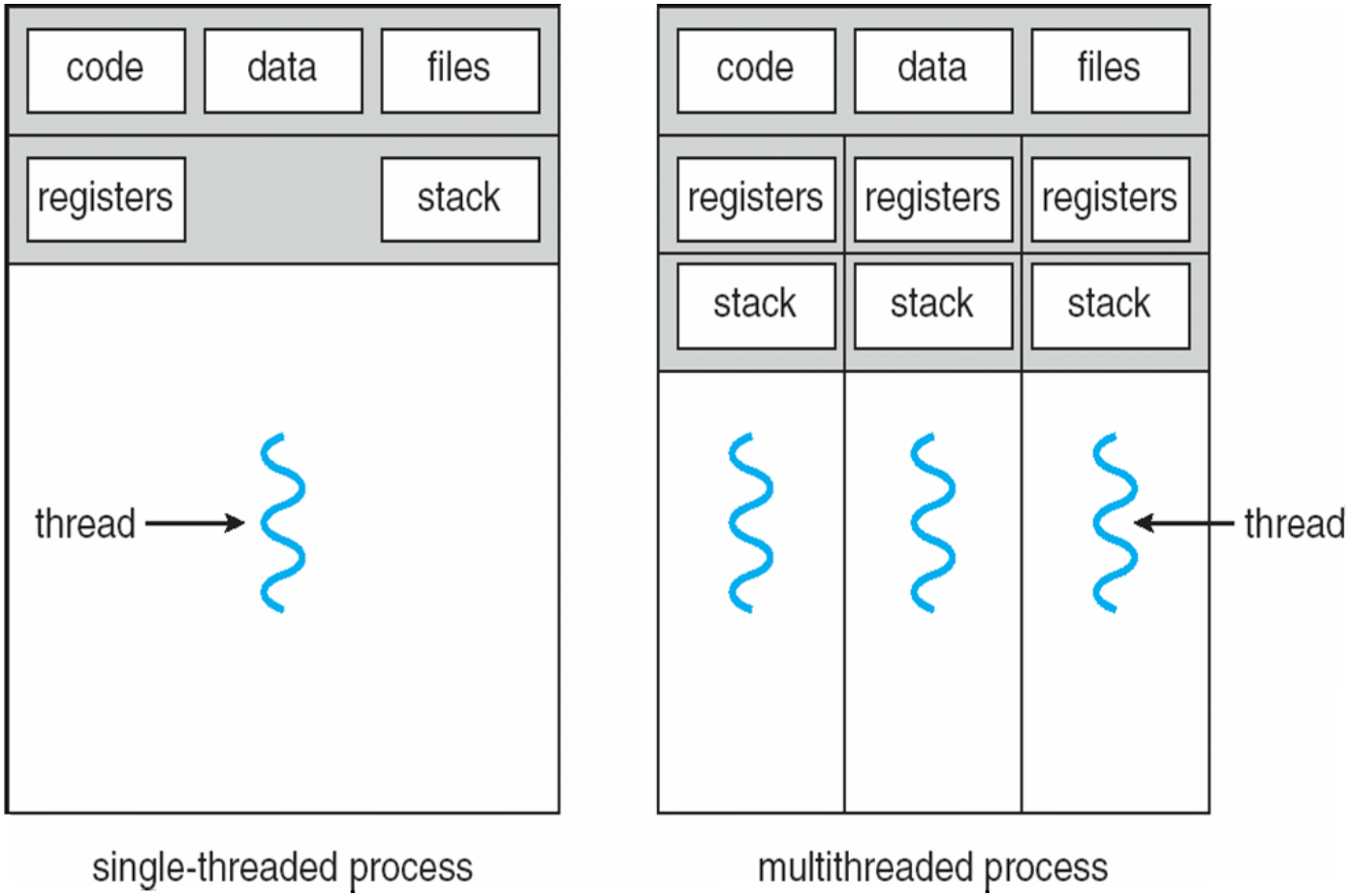
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

- To examine issues related to multithreaded programming

Threads

- Thread is a light weight process
- A ***thread*** is a basic unit of CPU utilization.
- It consists of a thread ID, program counter, a stack, and a set of registers.
- Process→ single thread of control →heavy weight process
  - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- A multi-threaded application → multiple threads within a single process→ light weight process
  - each having their own program counter, stack and set of registers,
  - but sharing common code, data, and certain structures such as open files

Single and Multithreaded Processes



Benefits

- Responsiveness: One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations
- Resource Sharing: threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
- Economy: Creating and managing threads is much faster , Context switching between threads takes less time
- Scalability: **Utilization of multiprocessor architectures**

- threads can share common data, they do not need to use interprocess communication

Draw back

- no protection between threads

Multicore Programming

- Recently produce chips with multiple ***cores***, or CPUs on a single chip is a trend.
- The threads could be spread across the available cores, allowing true parallel processing.

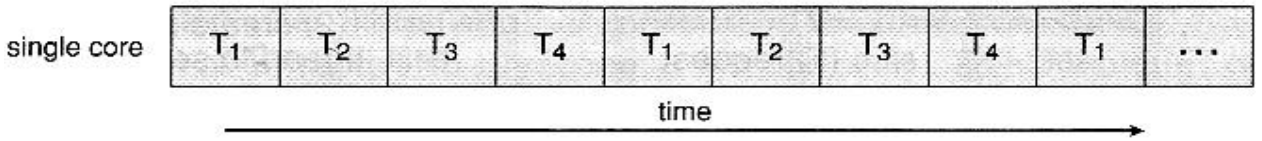


Figure 4.3 Concurrent execution on a single-core system.

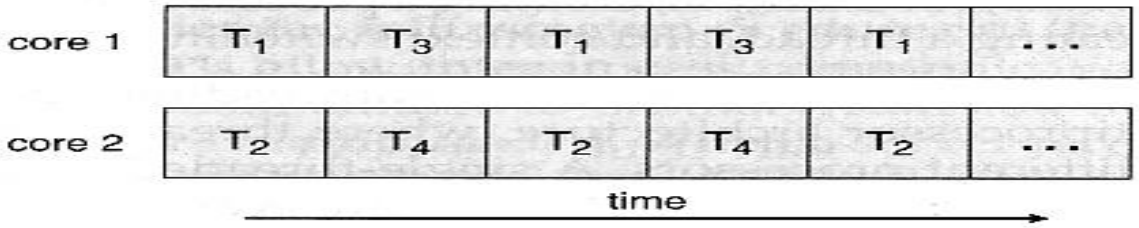
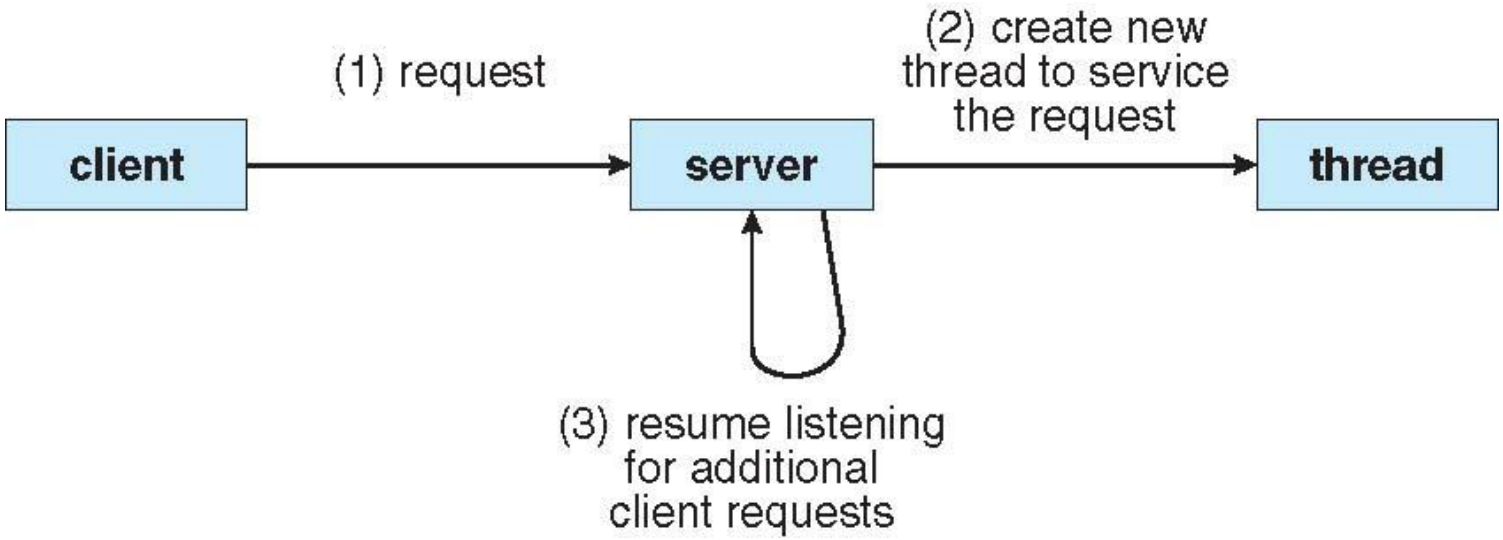


Figure 4.4 Parallel execution on a multicore system.

Challenges of Multicore programming

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

Multithreaded Server Architecture



User Threads

- Thread management done by user-level threads library rather than via systems calls so thread switching does not need to call operating system and to cause interrupt to the kernel
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

- Advantages
  - user-level threads package can be implemented on an Operating System that does not support threads
  - User-level threads does not require modification to operating systems
  - Simple representation (PC, registers, stack and a small control block
  - Simple Management (no intervention of kernel)
  - Fast and Efficient (Thread switching is not expensive)
- **Disadvantages:**
  - There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within.
  - User-level threads requires non-blocking systems call i.e., a multithreaded kernel.

Kernel Threads

- Supported by the Kernel
- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system
- In addition, the kernel also maintains the traditional process table to keep track of processes
- Operating Systems kernel provides system call to create and manage threads.
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

- **Advantages:**
  - Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads
  - Kernel-level threads are especially good for applications that frequently block
- **Disadvantages**
  - The kernel-level threads are slow and inefficient
  - Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.
    - As a result there is significant overhead and increased in kernel complexity.

Multithreading Models

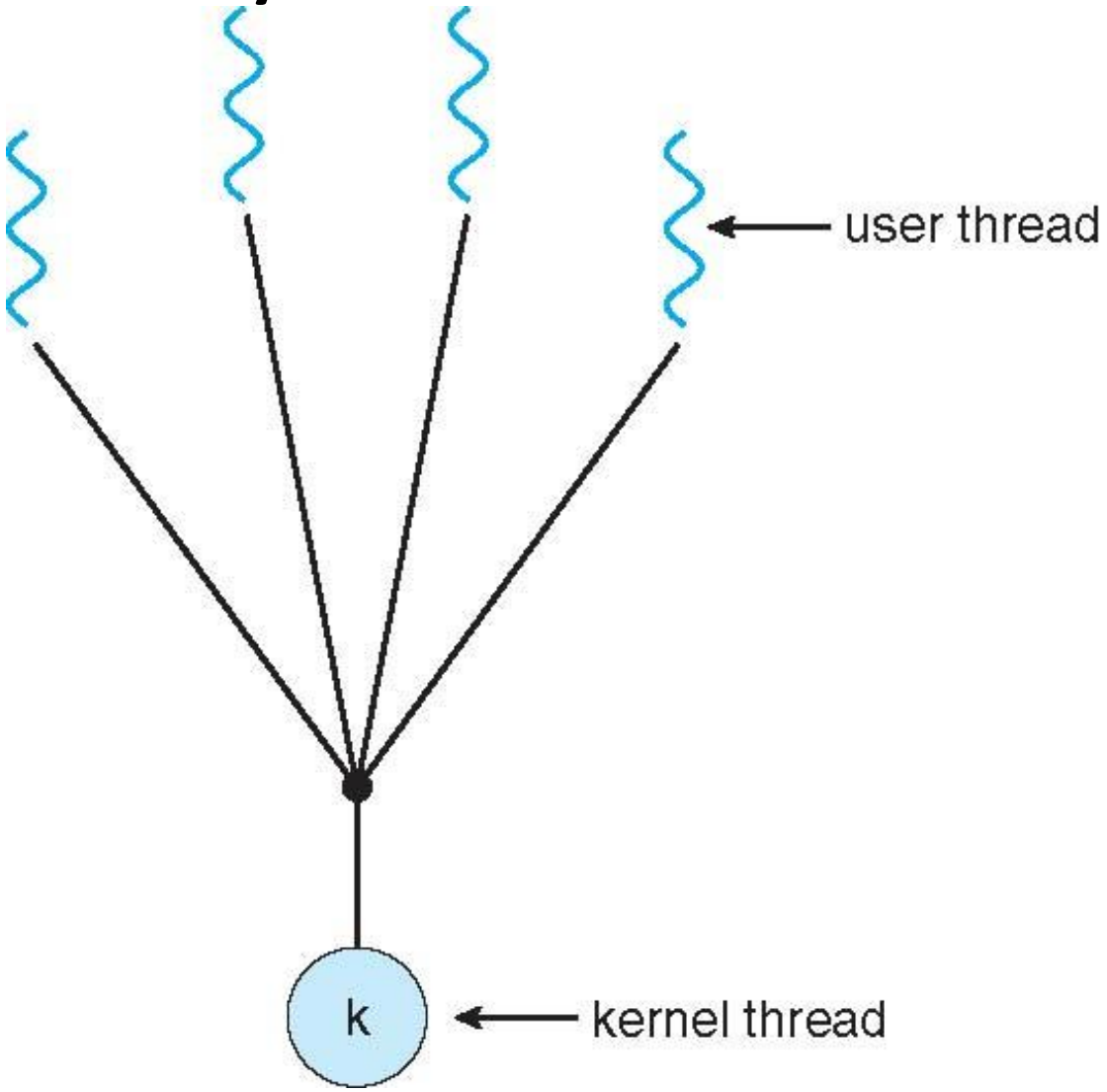
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread

- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

Many-to-One Model

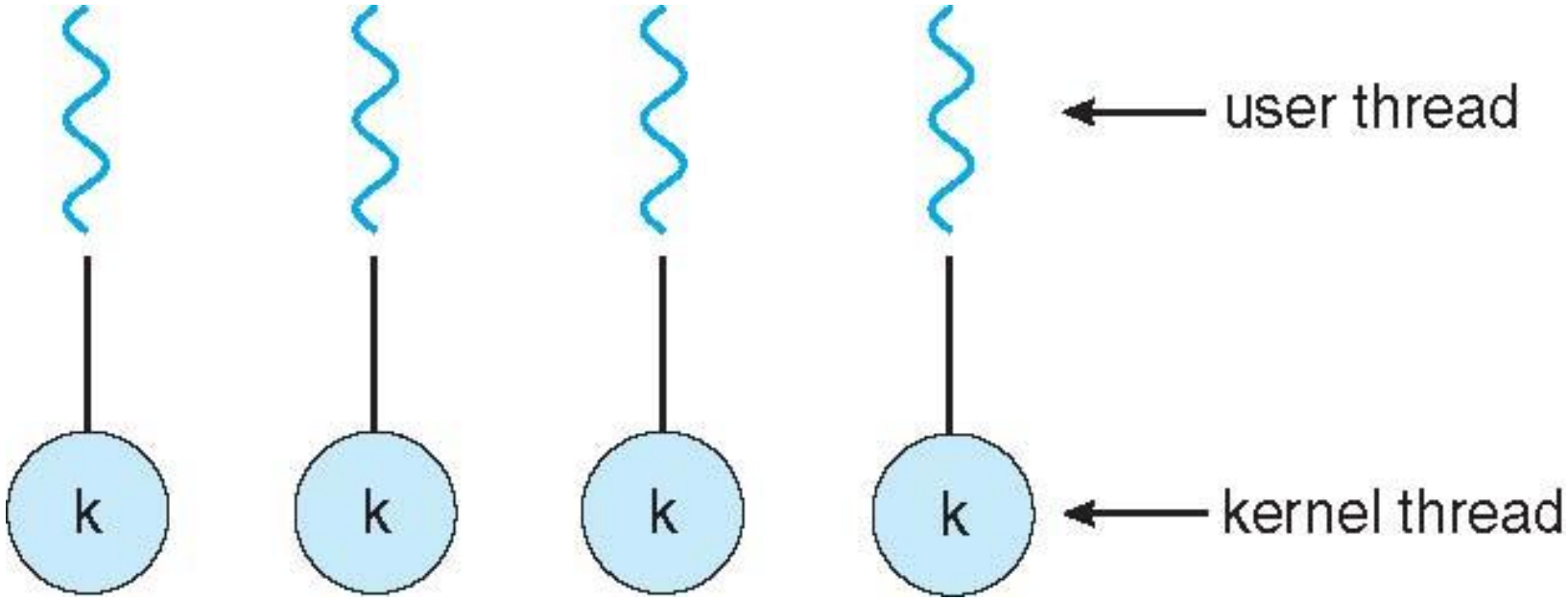


One-to-One

- Each user-level thread maps to kernel thread

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

One-to-one Model

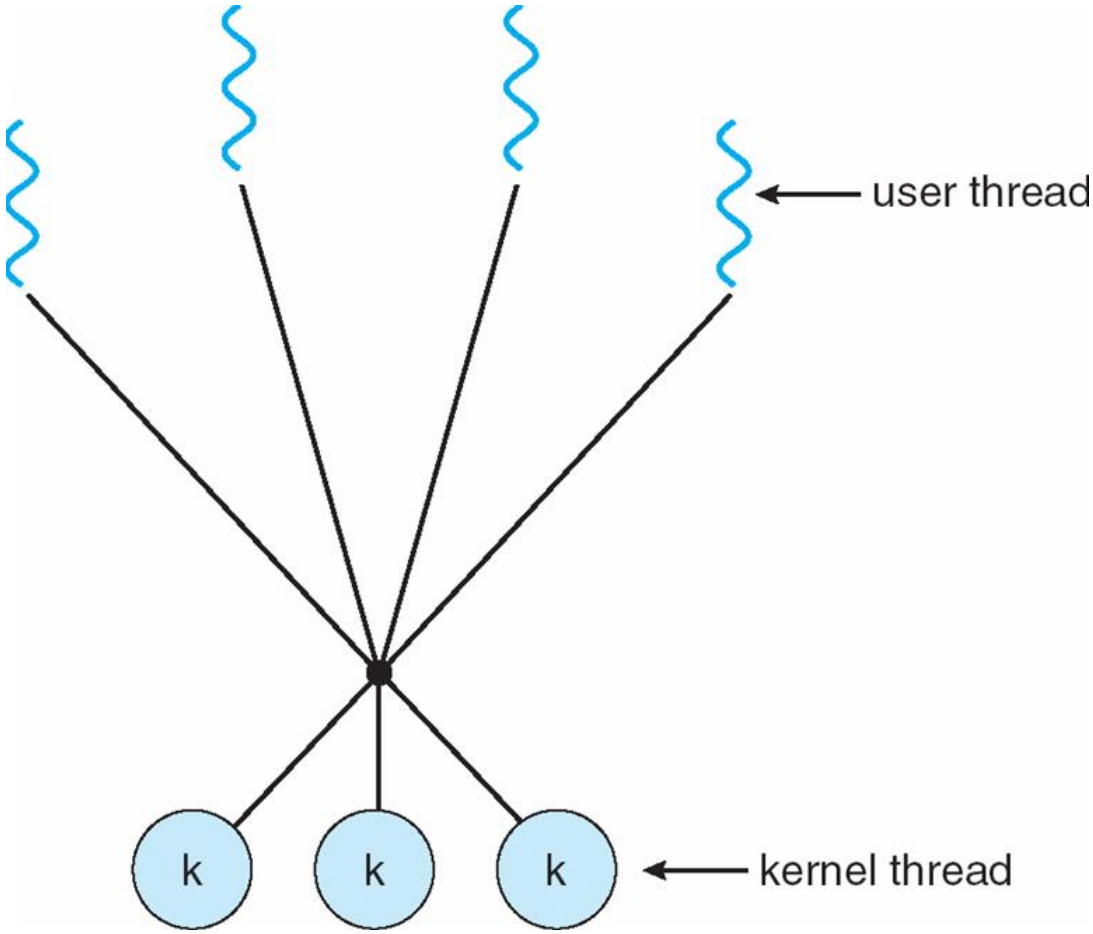


Many-to-Many Model

- 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

ISE Dept.
Transform Here

Many-to-Many Model



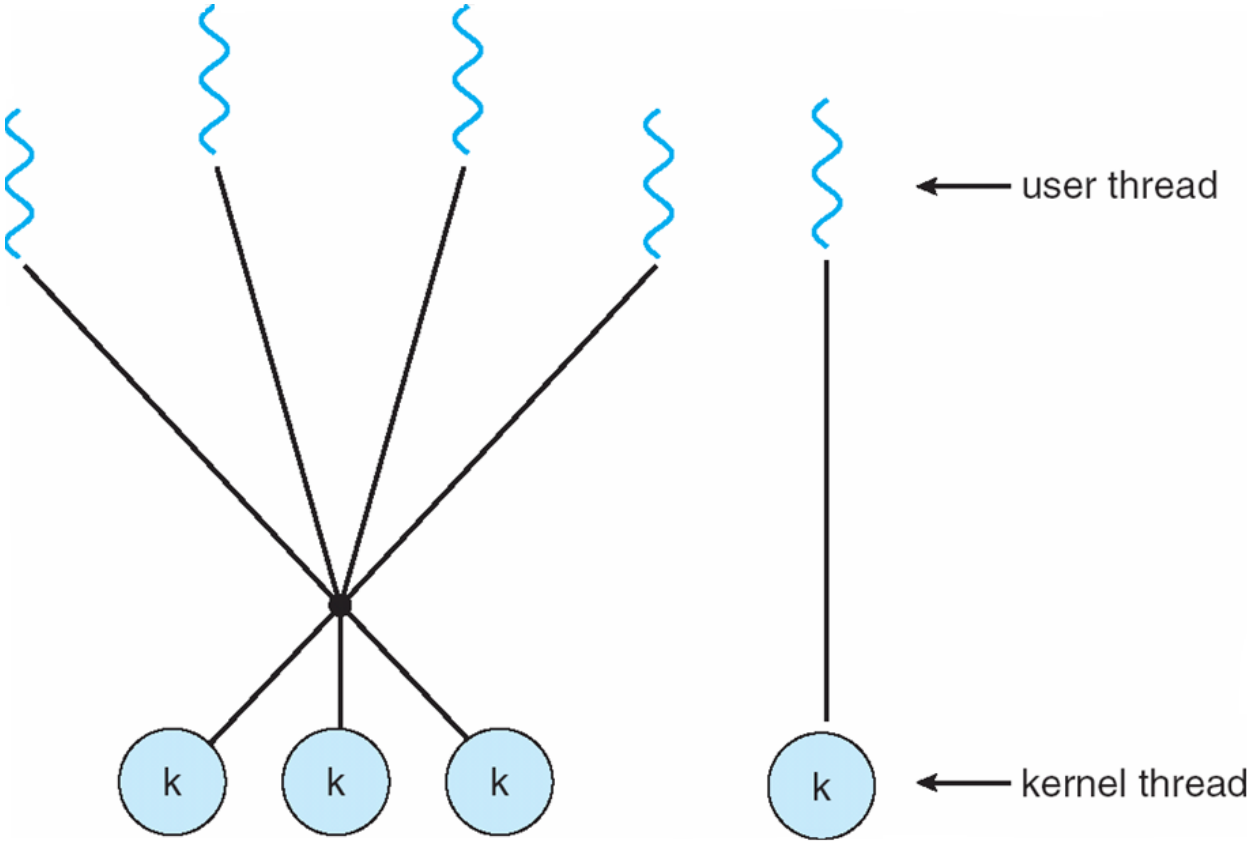
Two-level Model

- 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

ISE Dept.
Transform Here

Two-level Model



Thread Libraries

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

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

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

P threads example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

```
    // create the thread
    ThreadHandle = CreateThread(
       NULL, // default security attributes
       0, // default stack size
       Summation, // thread function
       &Param, // parameter to thread function
       0, // default creation flags
       &ThreadId); // returns the thread identifier

    if (ThreadHandle != NULL) {
       // now wait for the thread to finish
       WaitForSingleObject(ThreadHandle,INFINITE);

       // close the thread handle
       CloseHandle(ThreadHandle);

       printf("sum = %d\n",Sum);
    }
}
```

Figure 4.10 Multithreaded C program using the Win32 API.

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  Extending Thread class
  - Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         // create the object to be shared
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.

Threading Issues

a) Semantics of **fork()** and **exec()** system calls

b) Thread cancellation of **target thread**

   a) Asynchronous or deferred

c) Signal handling

d) Thread pools

e) Thread-specific data

f) Scheduler activations

a.Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- It depends on exec()→ if called immediately after fork then duplicating all threads are un necessary

b.Thread Cancellation

- Terminating a thread before it has finished
- Eg. Web page loading, searching a database
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately.
    - Disadv: may not free necessary system wide resources
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

c.Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Types
  - Synchronous
  - Asynchronous
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

- Synchronous→ signal is generated by the same process that performed the operation that caused the signal and delivered to the same.
  - Eg. Divide by 0, illegal memory access
- Asynchronous→ signal is generated by an event external to the running process and delivered to another process
  - Eg: ctrl+c , having a timer to expire

d.Thread Pools

- Create a number of threads in a pool where they await work

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

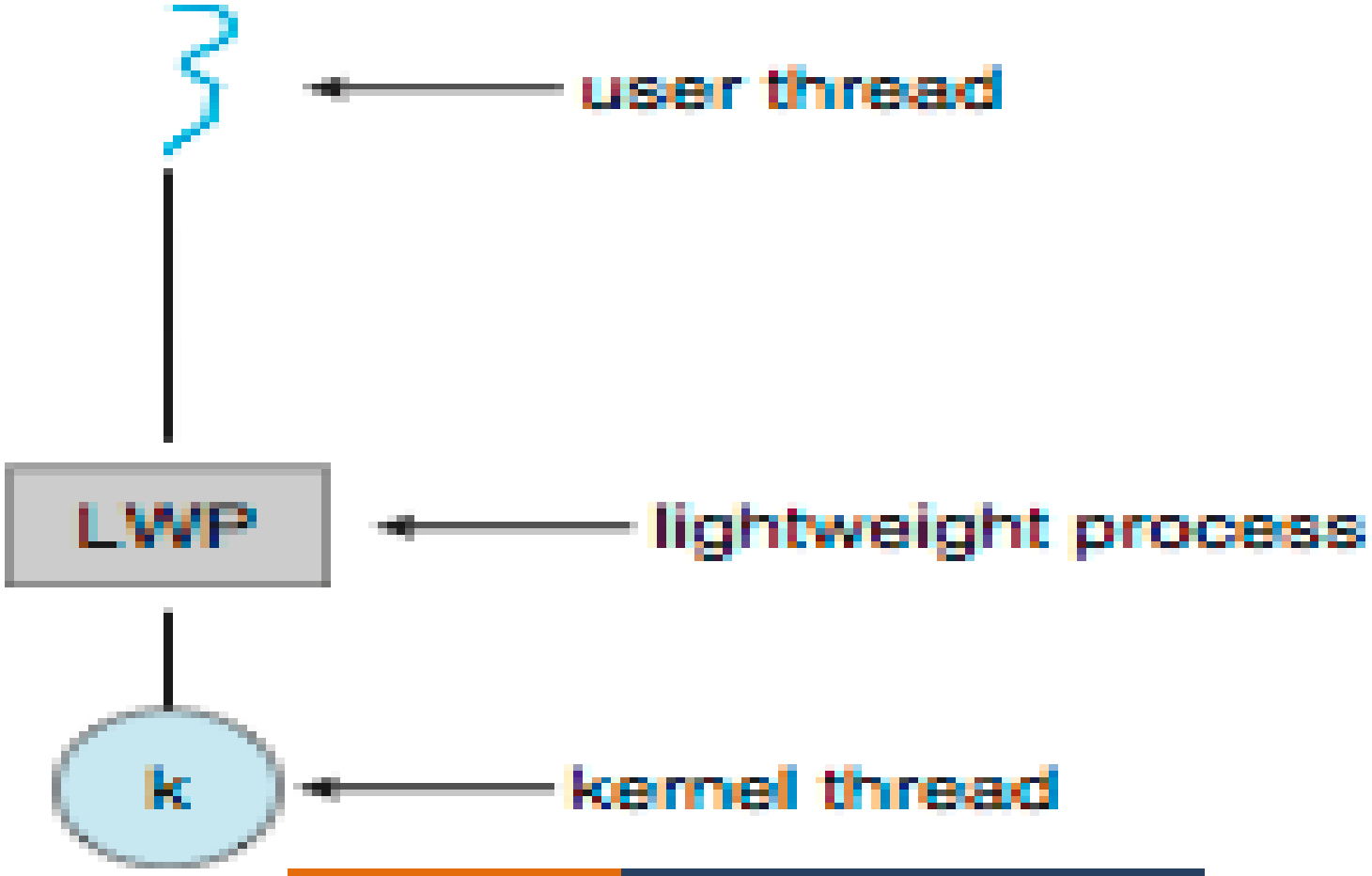
e.Thread Specific Data

- Allows each thread to have its own copy of data

- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

f.Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads


user thread
LWP
lightweight process
k
kernel thread

End of Chapter 4

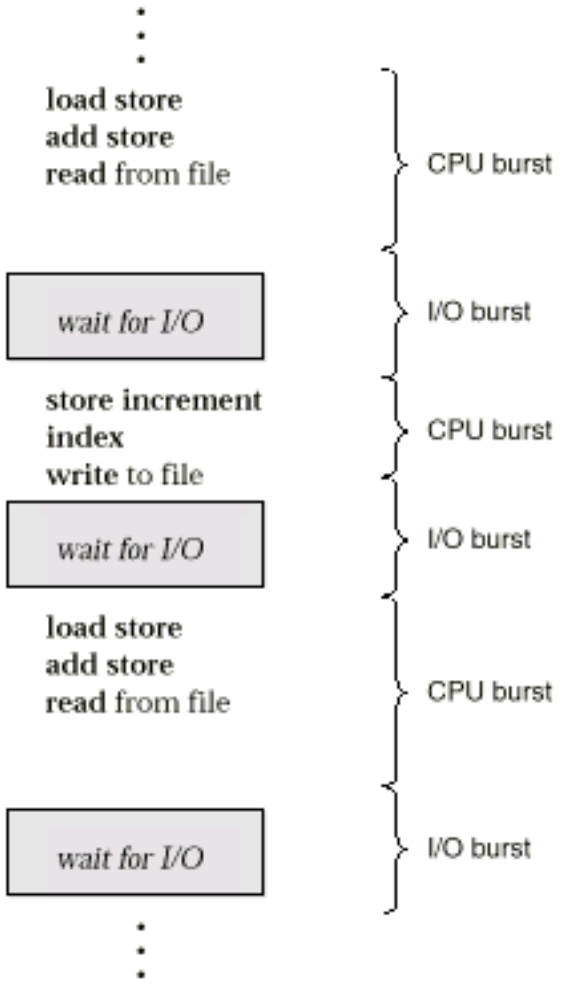
Module 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

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 distribution

Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

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

Dispatcher

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

Scheduling Criteria

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

Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

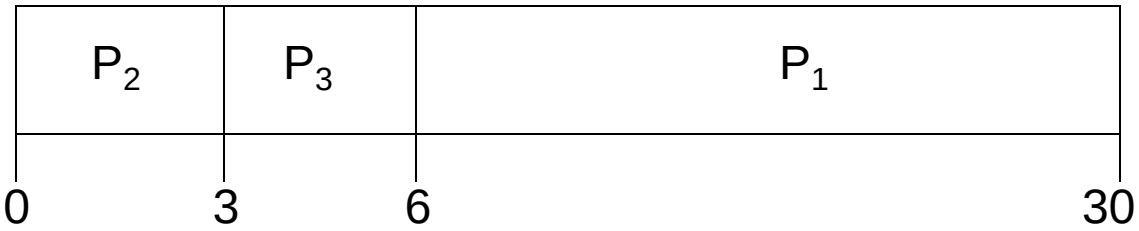
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1 .$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.
- Good for batch systems, not suitable for time sharing systems.
- *Convoy effect* short process behind long process

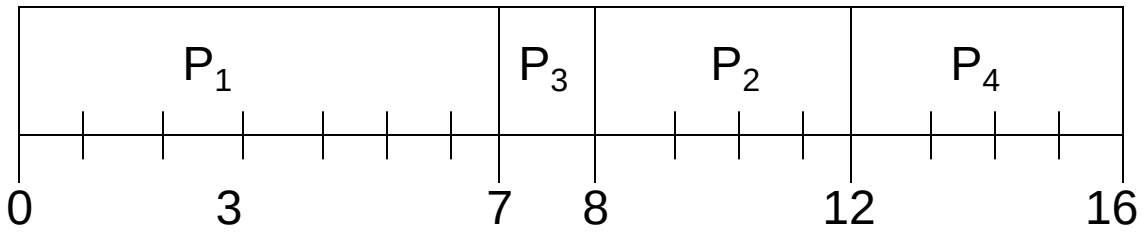
Shortest-Job-First (SJR) Scheduling

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

Example of Non-Preemptive SJF

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

- SJF (non-preemptive)

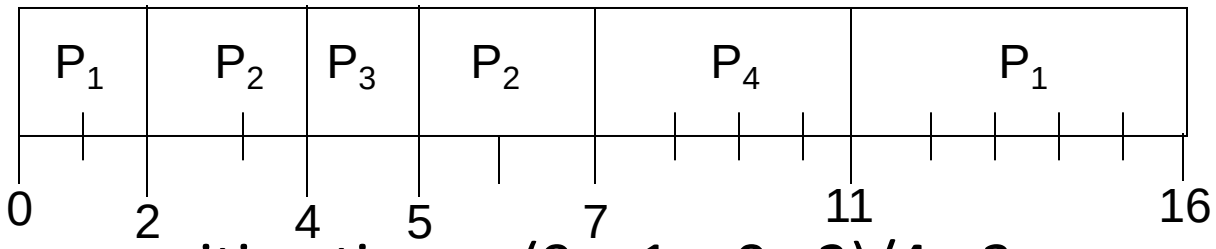


- Average waiting time = (0 + 6 + 3 + 7)/4 - 4

Example of Preemptive SJF

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

- SJF (preemptive)



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

Determining Length of Next CPU Burst

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

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

Examples of Exponential Averaging

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

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

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

Priority Scheduling

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

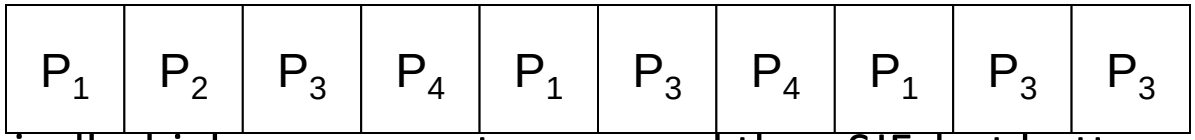
Round Robin (RR)

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

Example: RR with Time Quantum = 20

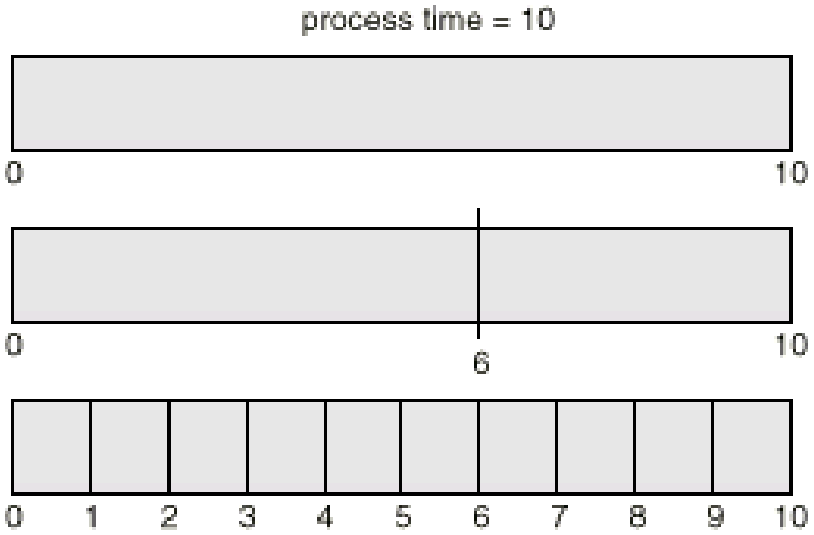
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



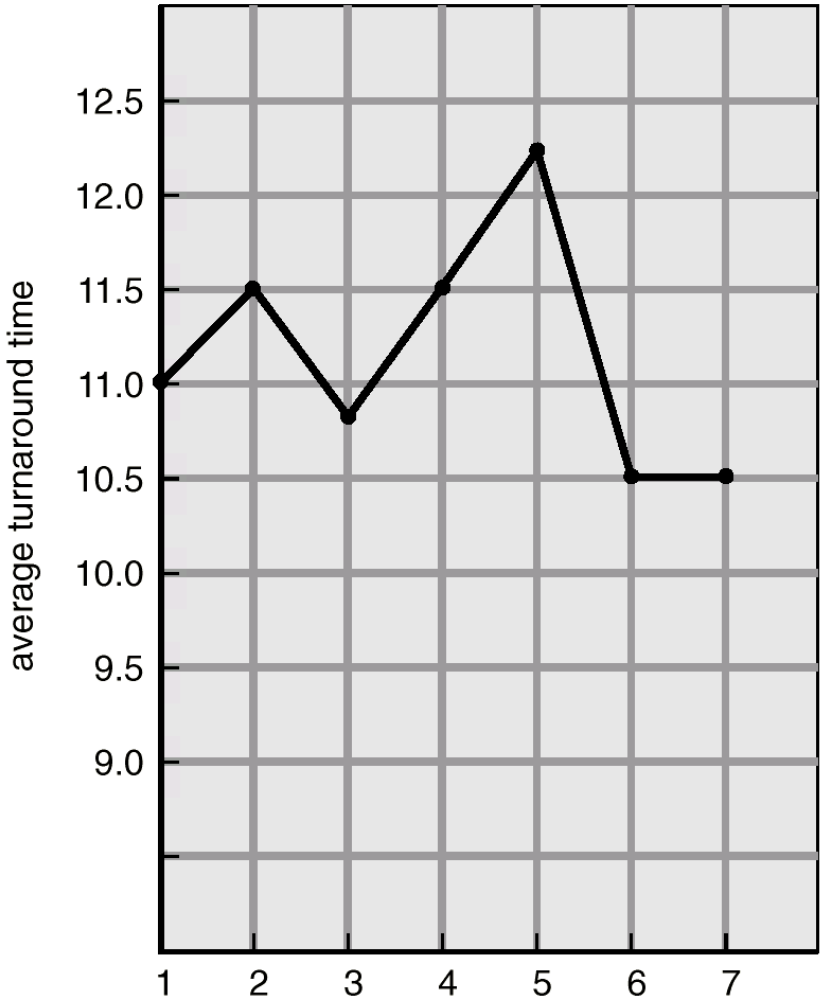
- Typically, higher average turnaround than SJF, but better *response*.

How a Smaller Time Quantum Increases Context Switches



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

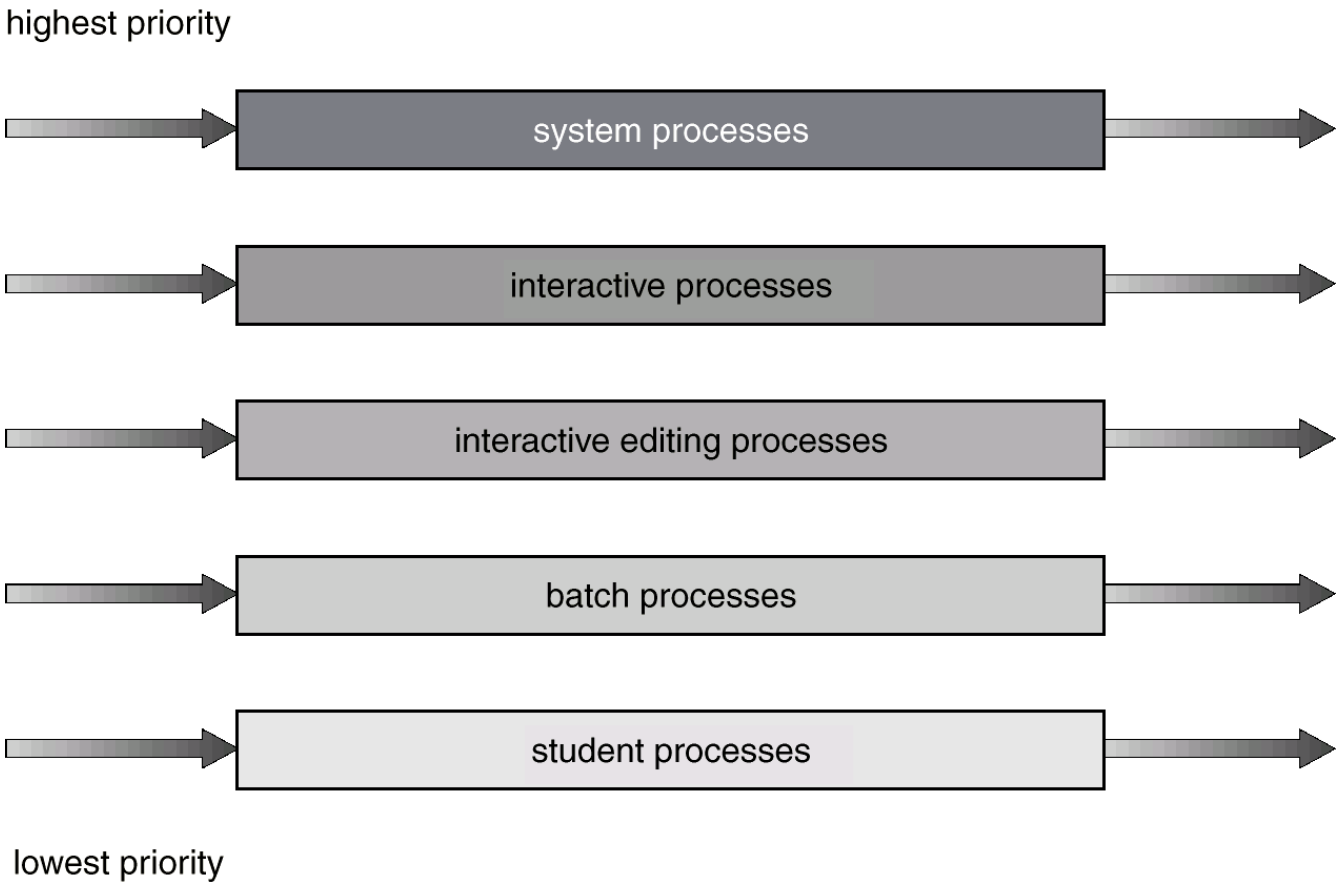


process	time
P_1	6
P_2	3
P_3	1
P_4	7

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

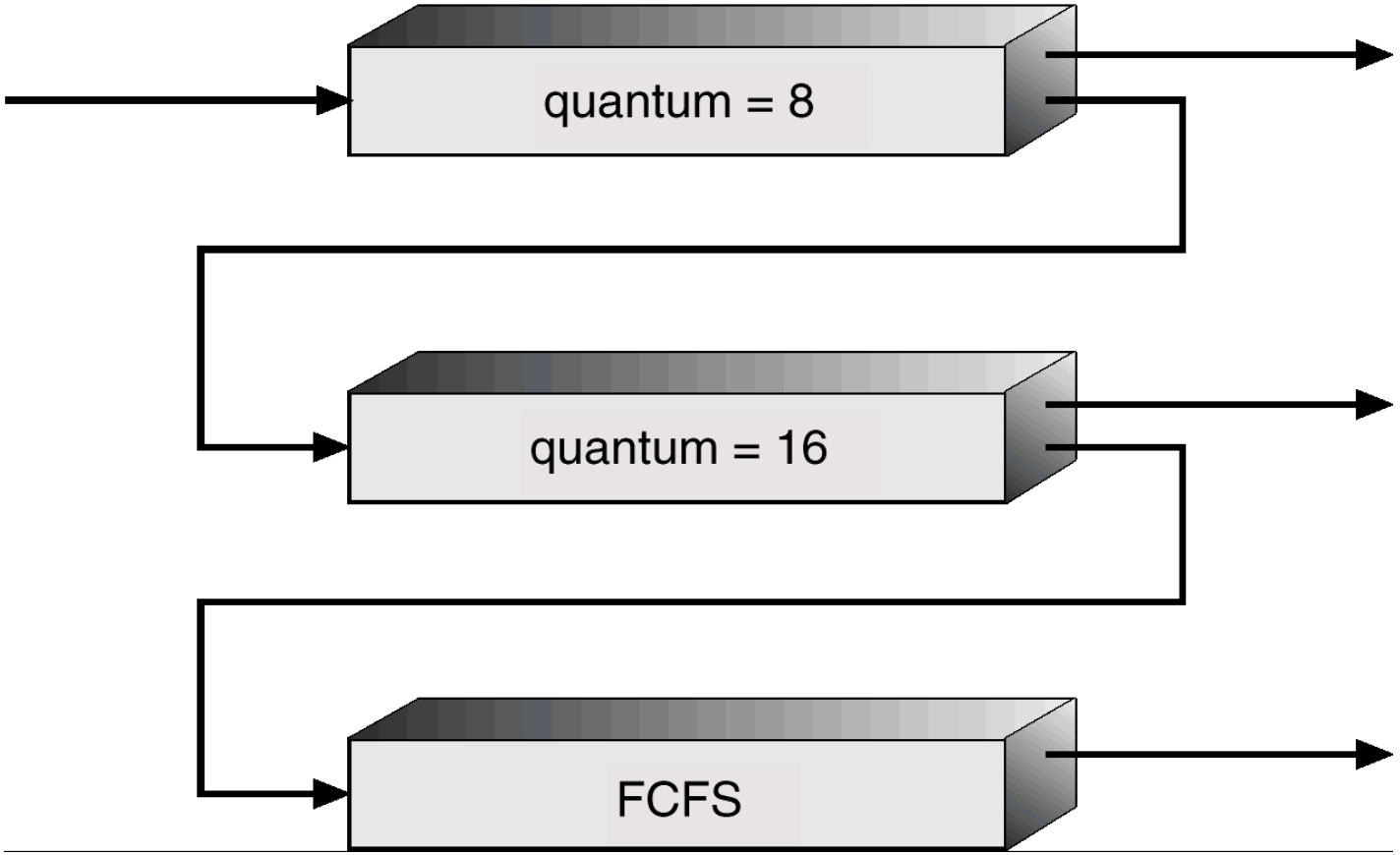
Multilevel Queue Scheduling



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

Multilevel Feedback Queues



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – 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.

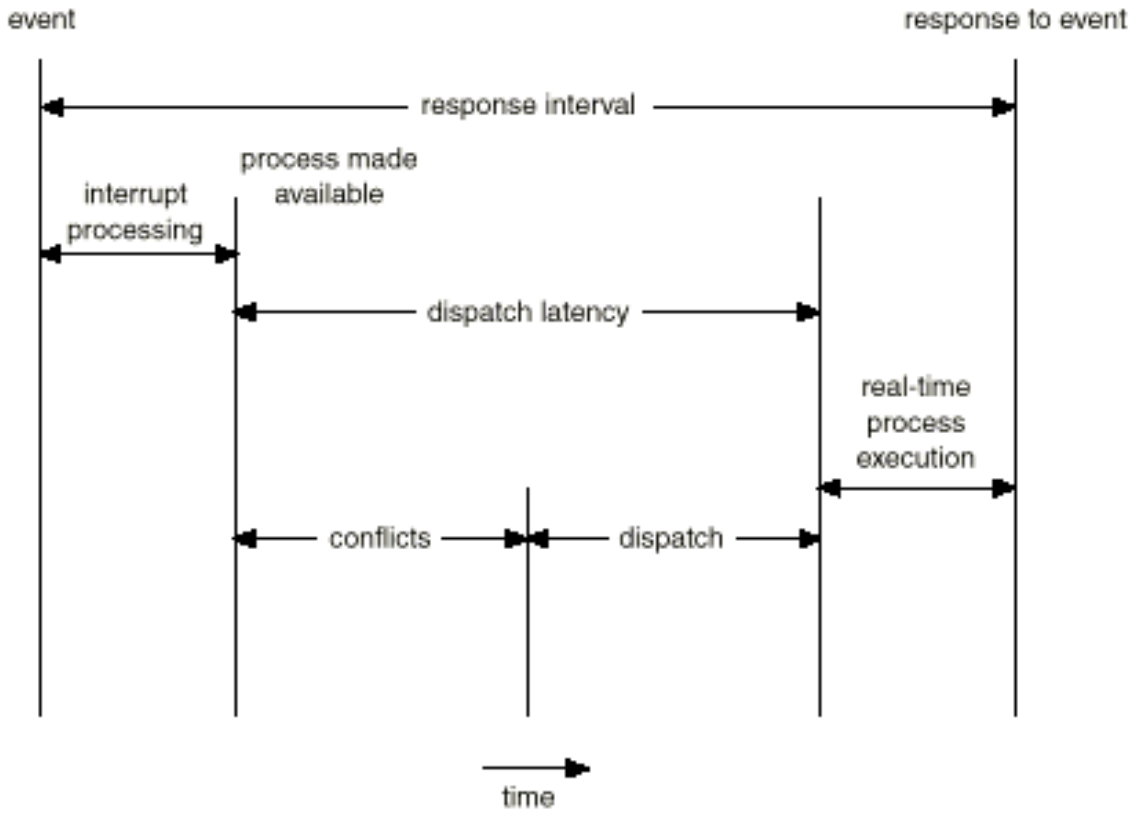
Multiple-Processor Scheduling

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

Real-Time Scheduling

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

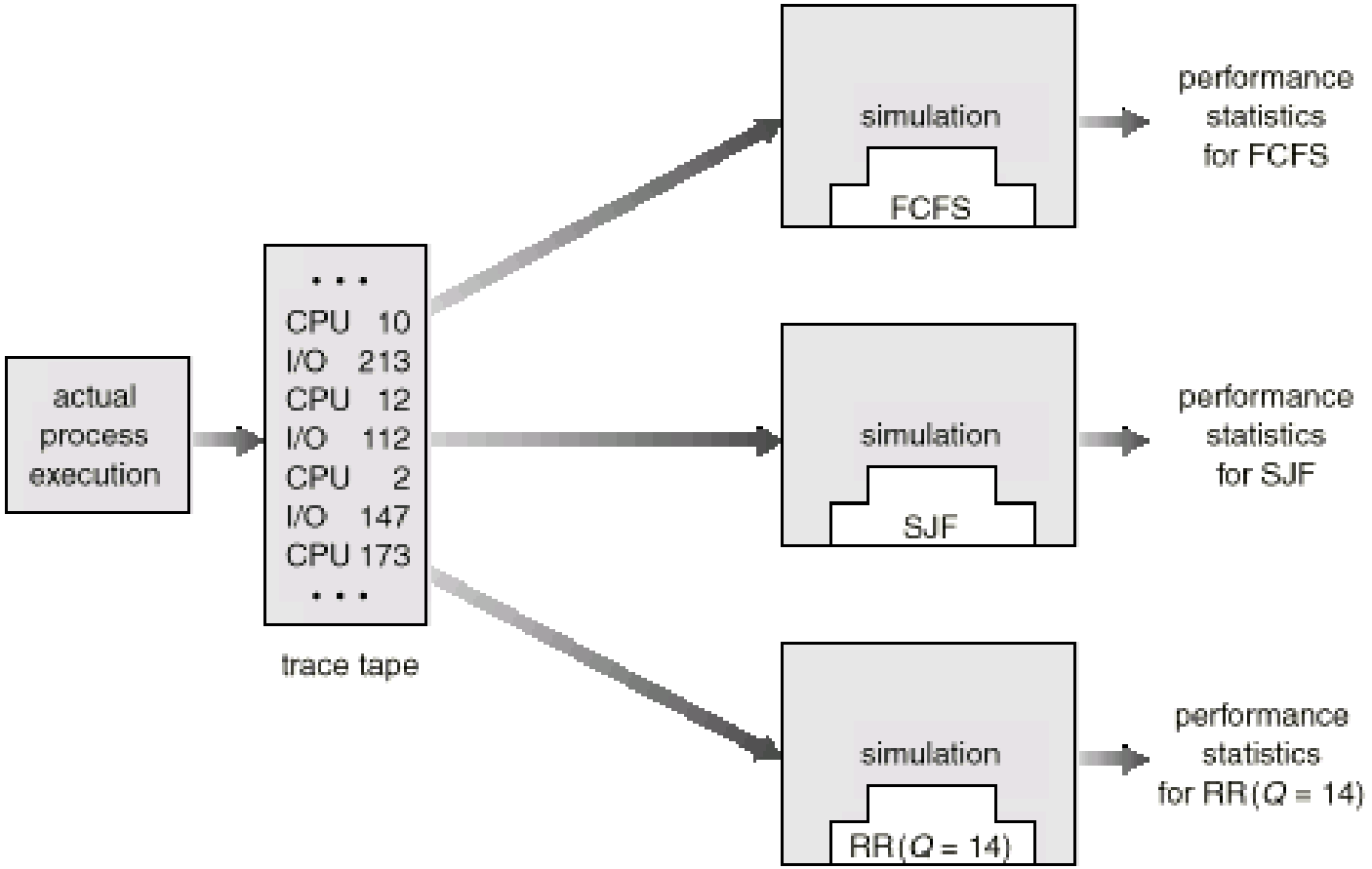
Dispatch Latency



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queuing models
- Implementation

Evaluation of CPU Schedulers by Simulation



Chapter 6: Process Synchronization

Module 6: Process Synchronization

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

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data

- To present both software and hardware solutions of the critical-section problem

- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

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

Producer

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

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                  count--;
                     /*  consume the item in
nextConsumed
}
```

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

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 the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the **N** processes

Peterson's Solution

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

Algorithm for Process P_i

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

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-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);
```

TestAndSet Instruction

- Definition:

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

Must be executed atomically

Solution using TestAndSet

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

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
                //    critical section
        lock = FALSE;
                //      remainder section
} while (TRUE);
```

Swap Instruction

- Definition:

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

Solution using Swap

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

```
do {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );
                 //    critical section
         lock = FALSE;
                 //      remainder section
 } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

Semaphore

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

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

  - signal (S) {

```
    S++;
}
```

Semaphore as General Synchronization Tool

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

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
       // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the 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 on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

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

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

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

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

Bounded-Buffer Problem

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

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        // produce an item in nextp
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
        wait (full);
        wait (mutex);
                //  remove an item from  buffer to nextc
        signal (mutex);
        signal (empty);
                //  consume the item in nextc
} 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 one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1 (controls access to readcount)
  - Semaphore wrt initialized to 1 (writer access)
  - Integer readcount initialized to 0 (how many processes are reading object)

Readers-Writers Problem (Cont.)

- The structure of a writer process

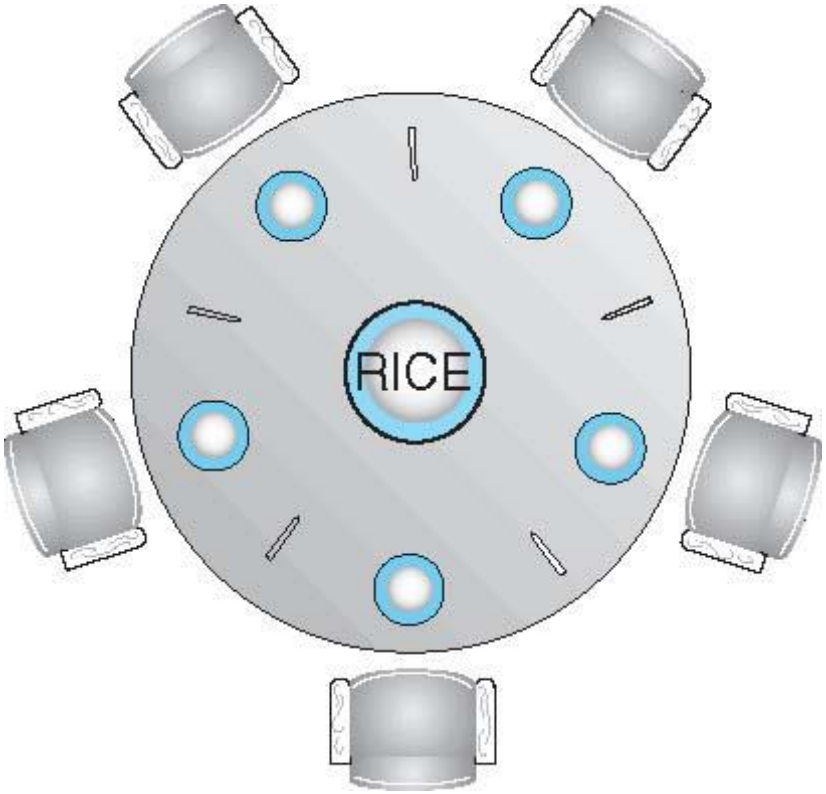
```
do {
        wait (wrt) ;
            //    writing is performed
        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
            // reading is performed
         wait (mutex) ;
         readcount  - - ;
         if (readcount  == 0)
                signal (wrt) ;
         signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



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

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?

More Problems with Semaphores

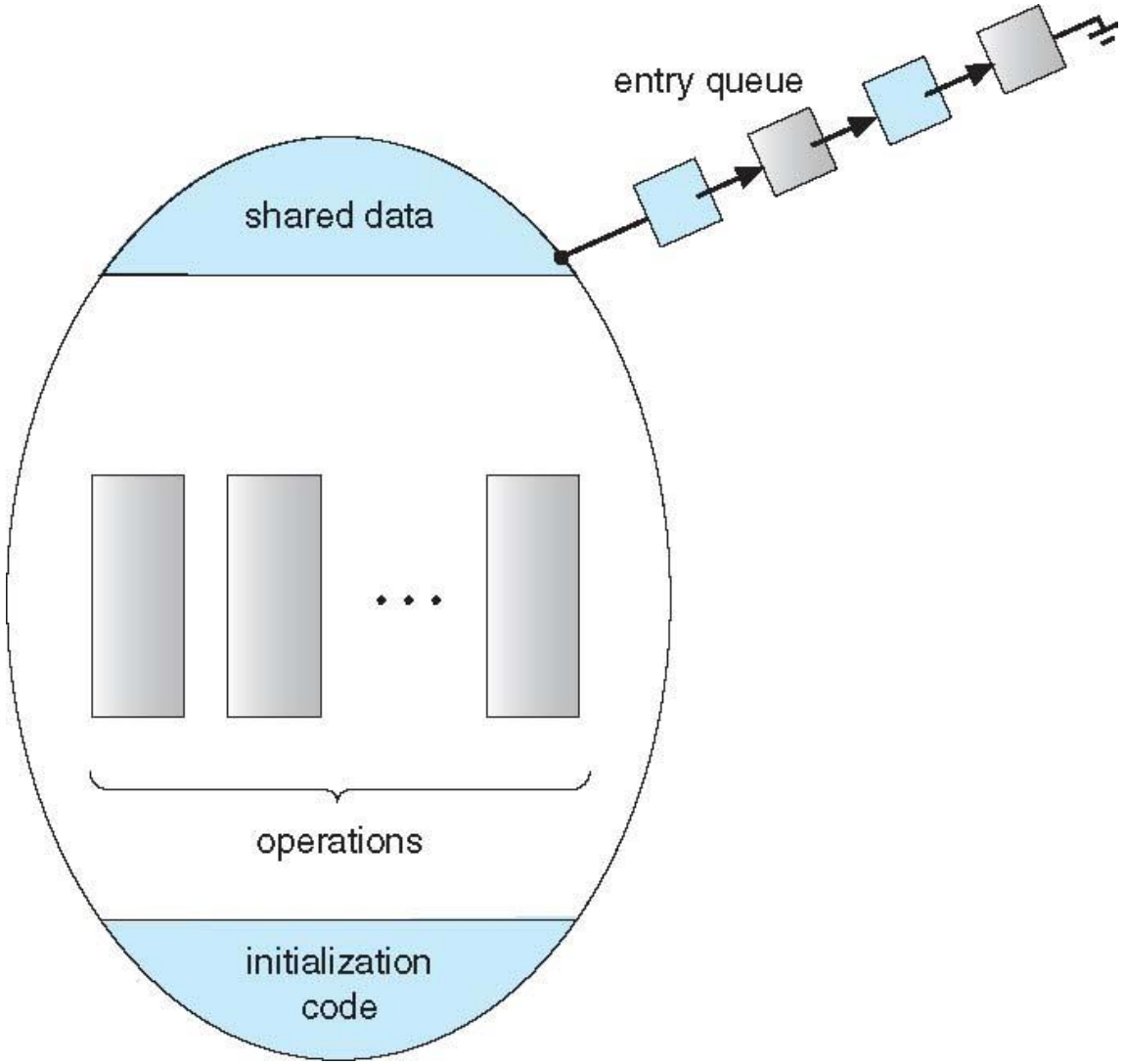
- Relies too much on programmers not making mistakes (accidental or deliberate)
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

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

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

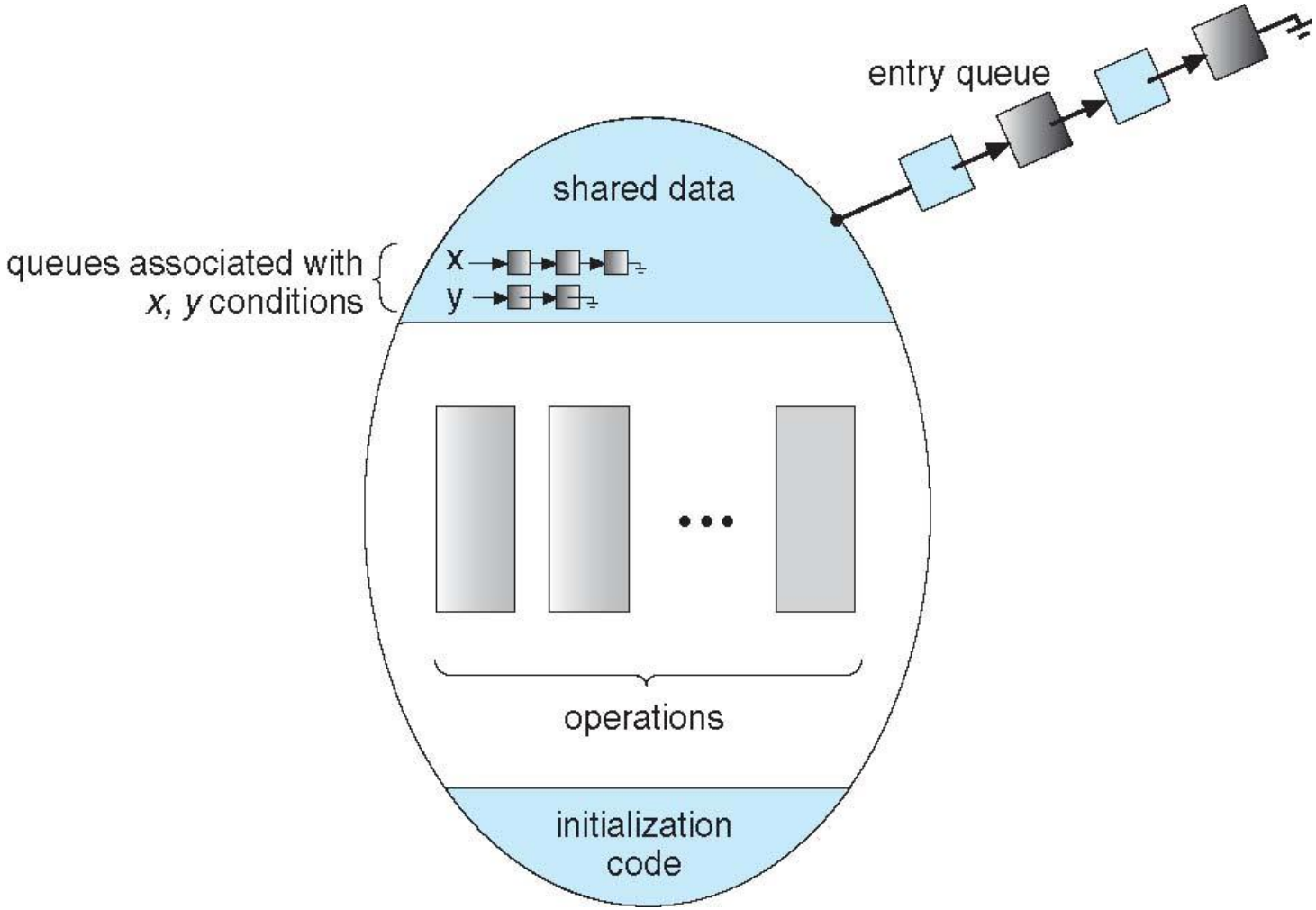
Schematic view of a Monitor



Condition Variables

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

Monitor with Condition Variables



Solution to Dining Philosophers

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

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

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

Solution to Dining Philosophers (Cont.)

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

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

Solution to Dining Philosophers (Cont.)

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

```
DiningPhilosophters.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

Monitor Implementation Using Semaphores

- Variables

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

- Each procedure ***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

- 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--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

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

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
        boolean busy;
        condition x;
        void acquire(int time) {
                        if (busy)
                                x.wait(time);
                        busy = TRUE;
        }
        void release() {
                        busy = FALSE;
                        x.signal();
        }
initialization code() {
         busy = FALSE;
        }
}
```

Chapter 7: Deadlocks

ISE Dept.
Transform Here

Chapter 7: Deadlocks

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

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks

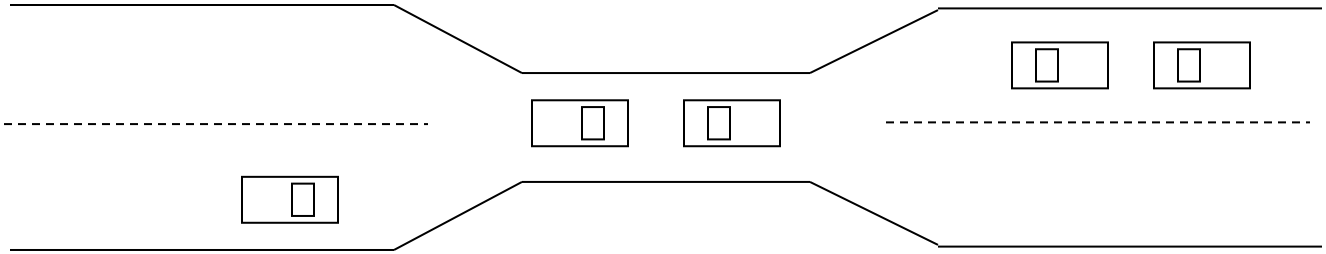
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1 P_0 P_1

    wait (A); wait(B) wait (B);
    wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

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

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request** → request the resource, if not granted immediately requesting process enter the waiting queue
  - **use** → operate on the resource
  - **release**

Deadlock Characterization

7.1 **Necessary conditions**

7.2 Resource Allocation Graph

7.3 Methods for handling Deadlocks

7.4 Deadlock prevention

Mutual Exclusion, Hold and wait, No preemption, Circular wait

7.5 Deadlock Avoidance

Safe state

Resource allocation graph algorithm

Bankers Algorithm

7.6 Deadlock Detection

Single instances of each resource type

Several instances of a resource type

7.7 Recovery from Deadlock

Process Termination

Resource Pre emption

7.1 Necessary Conditions

Deadlock can arise if four conditions hold simultaneously.

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

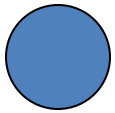
7.2 Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.(directed graph)

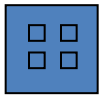
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

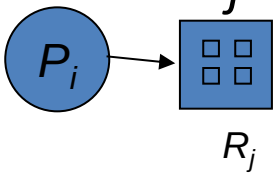
- Process



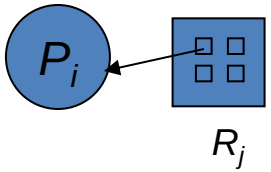
- Resource Type with 4 instances



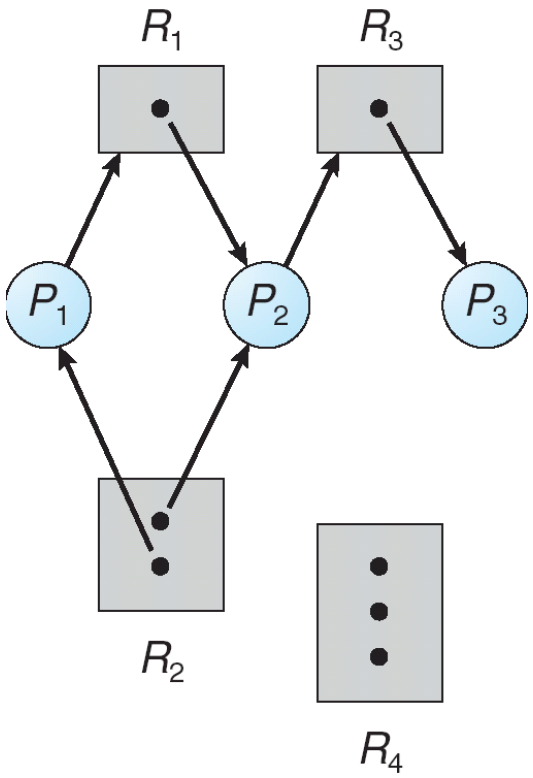
- P_i requests instance of R_j



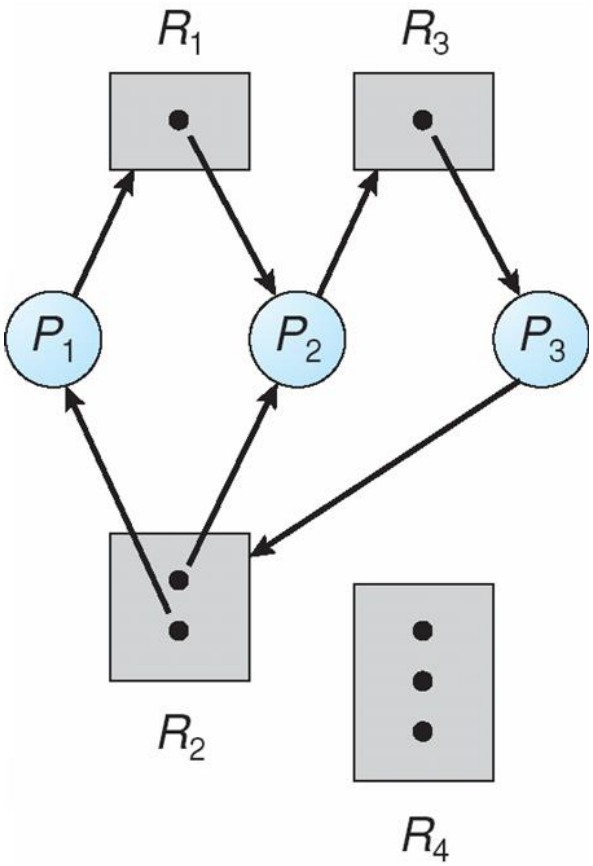
- P_i is holding an instance of R_j



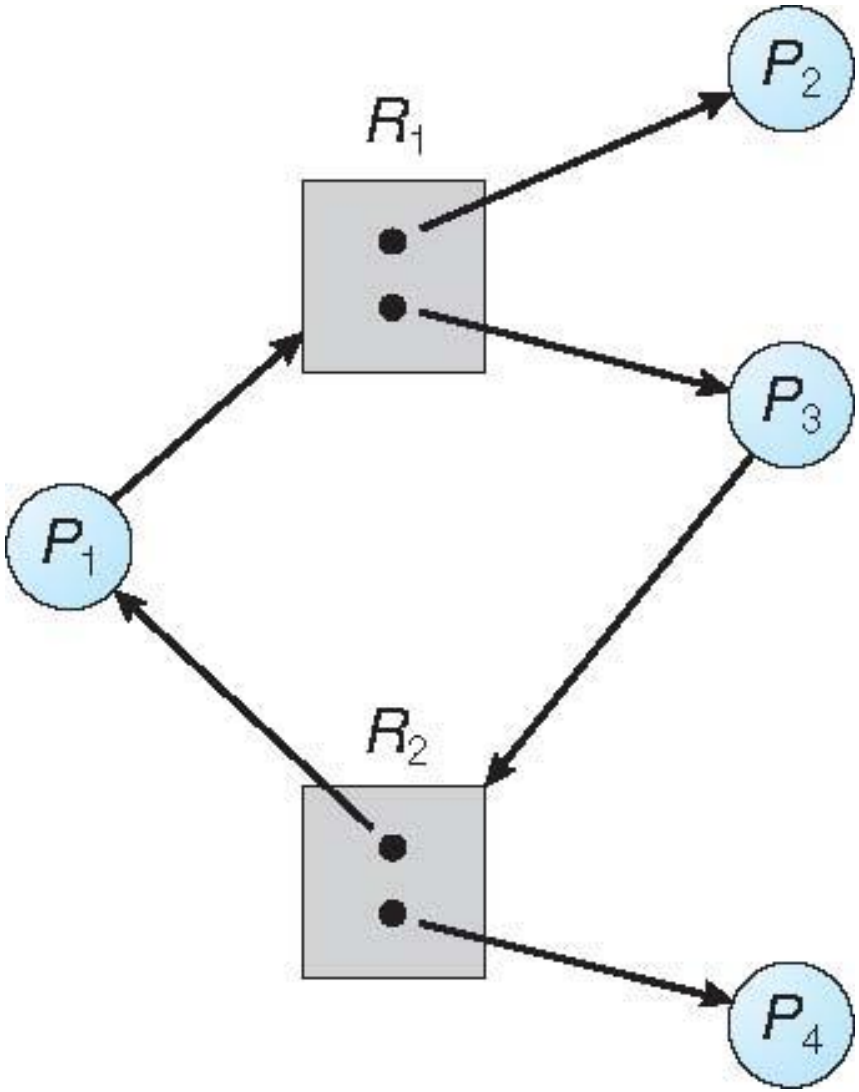
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

7.3 Methods for Handling Deadlocks

1. Ensure that the system will ***never*** enter a deadlock state

   (Deadlock Prevention or avoidance)

   – Prevention provides set of methods for ensuring that atleast one of the necessary conditions cannot hold

   – Avoidance by giving advance additional information about resources and process.

2. Allow the system to enter a deadlock state and then recover (Deadlock detection and Recovery)

3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

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

  DISADVANTAGES
  - 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

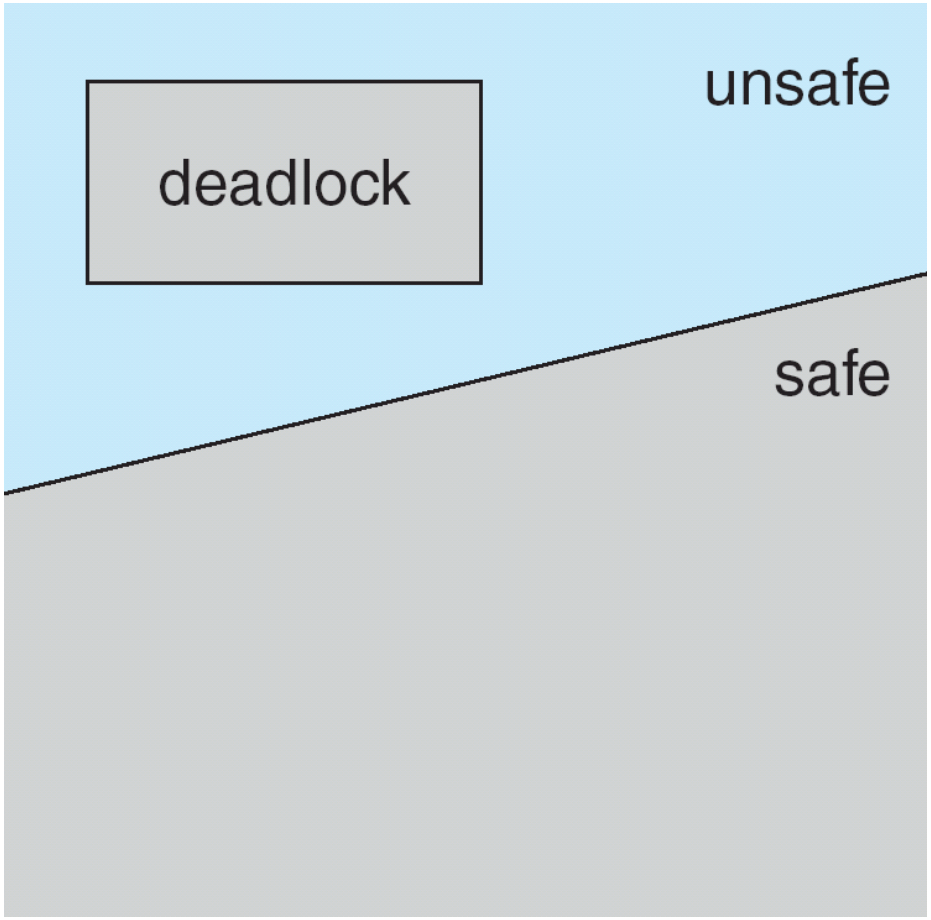
Safe State

- 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 $<P_1, P_2, ..., P_n>$ 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 $j < I$

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

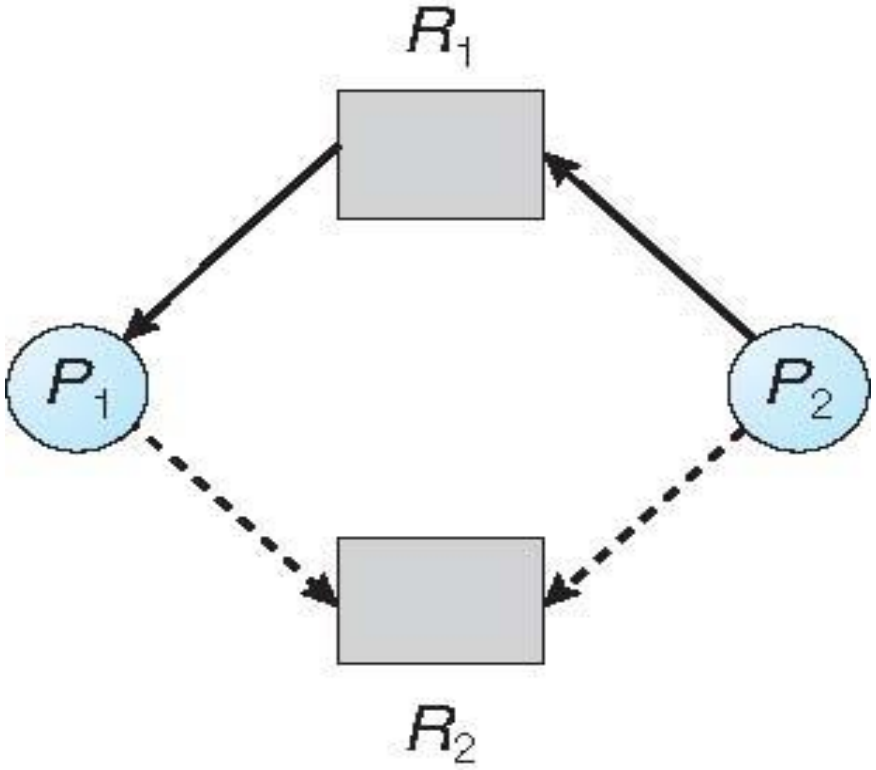
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

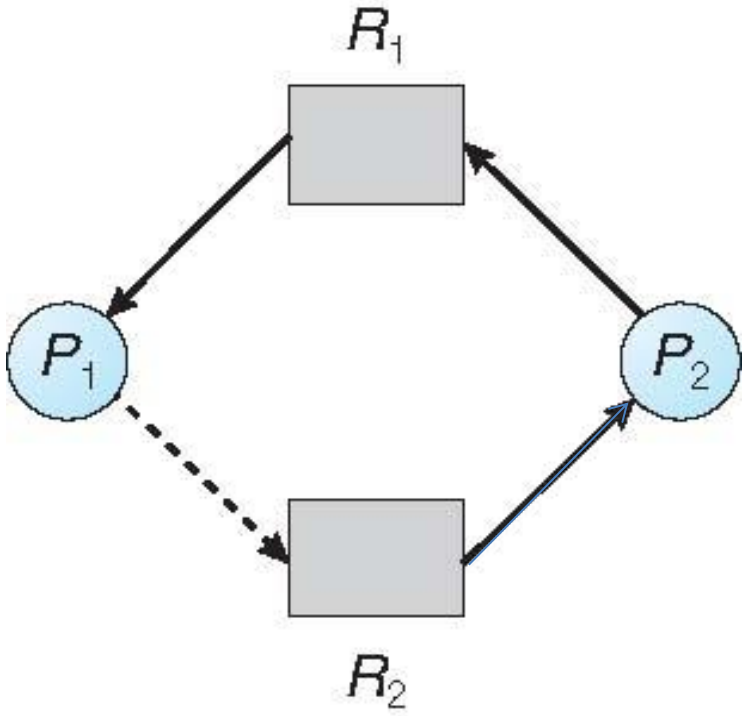
- **Claim edge** P_i --> R_j indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.($P_i \rightarrow R_j$)
- Request edge converted to an assignment edge when the resource is allocated to the process. ($R_j \rightarrow P_i$)
- 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

Resource-Allocation Graph



Unsafe State



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j

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

Banker's Algorithm

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

Data Structures for the Banker's Algorithm

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

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

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

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

   Work = Available

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

2. Find 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 ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

Snapshot at time T_0:

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

Example (Cont.)

- The content of the matrix *Need* is defined to be

  Max – Allocation

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

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

Deadlock Detection

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

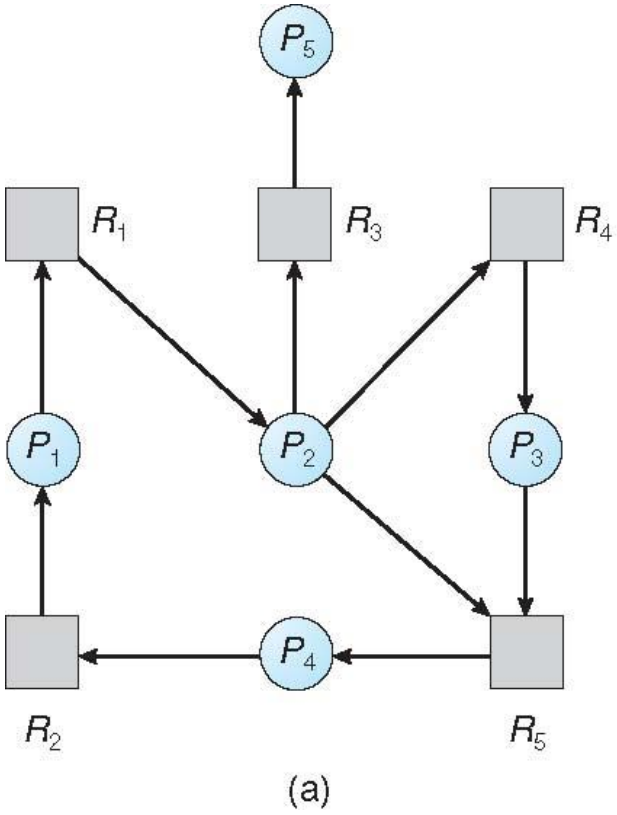
Single Instance of Each Resource Type

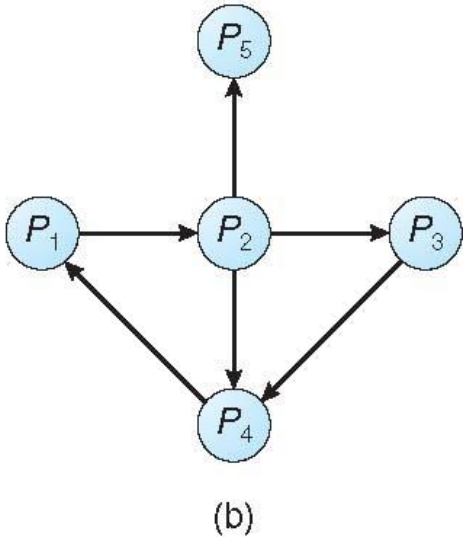
- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available***:* A vector of length *m* indicates the number of available resources of each type.

- **Allocation***:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

- **Request***:* An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][*j*] = *k*, then process P_i is requesting *k* more instances of resource type.R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*

2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

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

Detection Algorithm (Cont.)

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

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

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

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

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

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

Example (Cont.)

- P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 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
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

End of Chapter 7

Chapter 8: Main Memory

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Background

- Memory consists of a large array of words or bytes each with its own address
- Program must be brought (from disk) into memory and placed within a process for it to be run

- Main memory and registers are only storage CPU can access directly

- Memory unit only sees a stream of addresses + read requests, or address + data and write requests

- Register access in one CPU clock (or less)

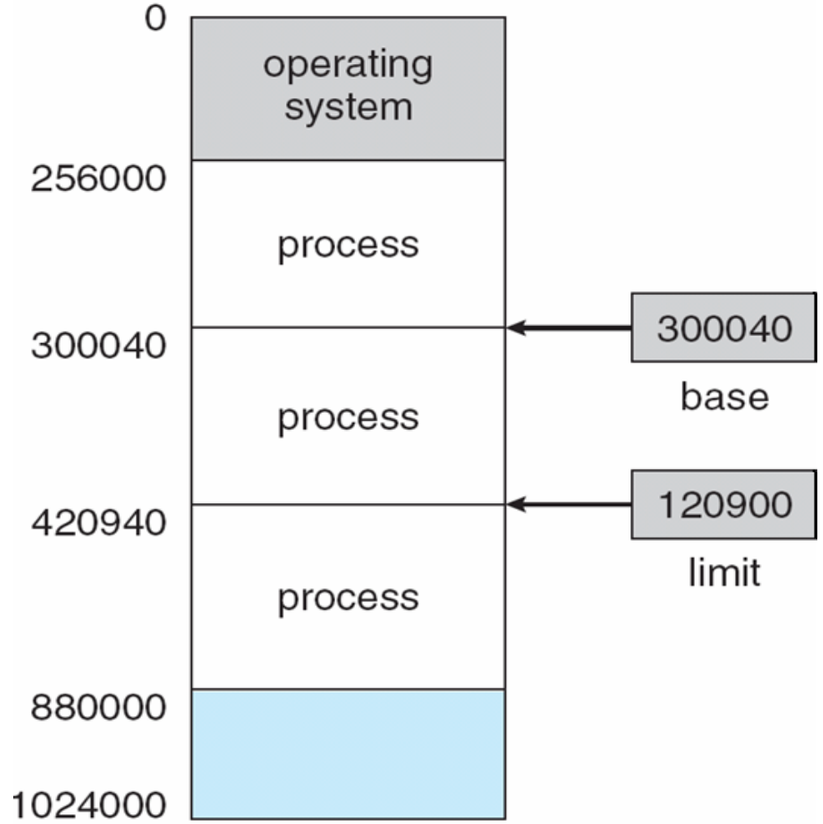
- Main memory can take many cycles, causing a **stall**

- **Cache** sits between main memory and CPU registers

- Protection of memory required to ensure correct operation

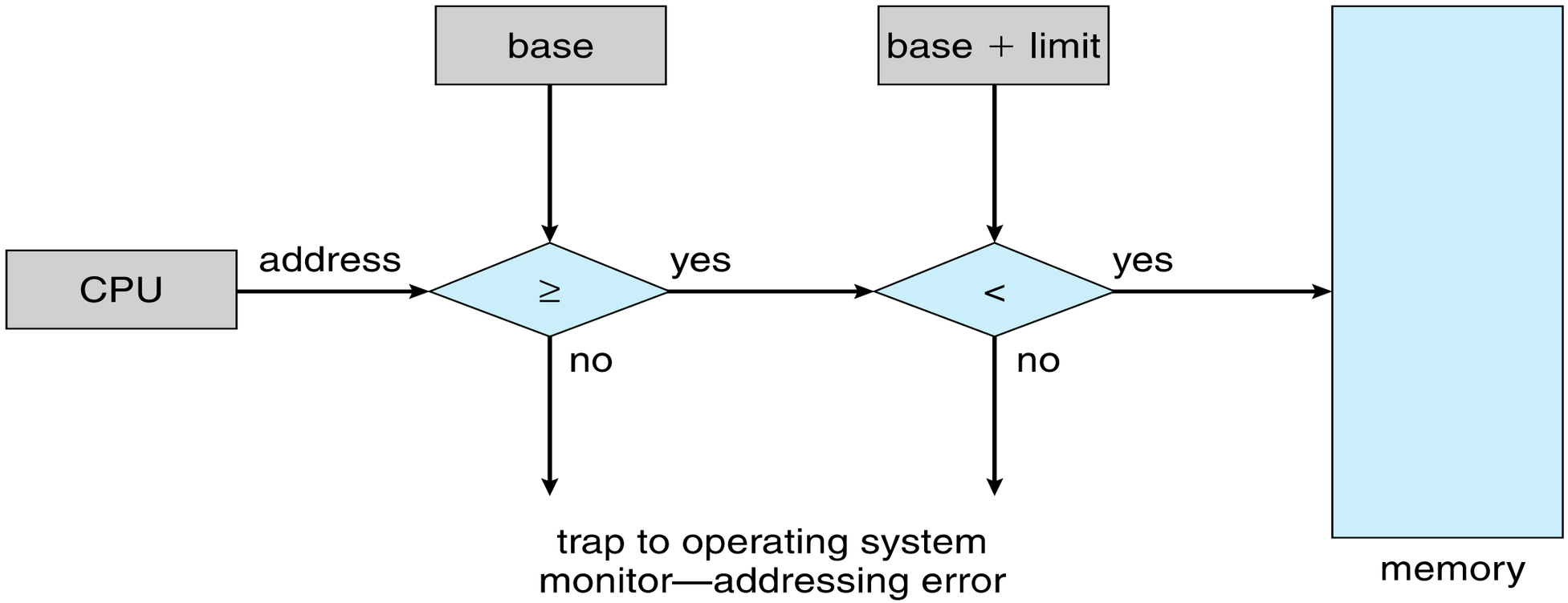
Base and Limit Registers

- Base register→ holds the smallest legal physical address
- Limit register→ specifies the size of the range
- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user, if not trap occurs (fatal error)



Hardware Address Protection with Base and Limit Registers

- Base and limit registers can be loaded by the OS



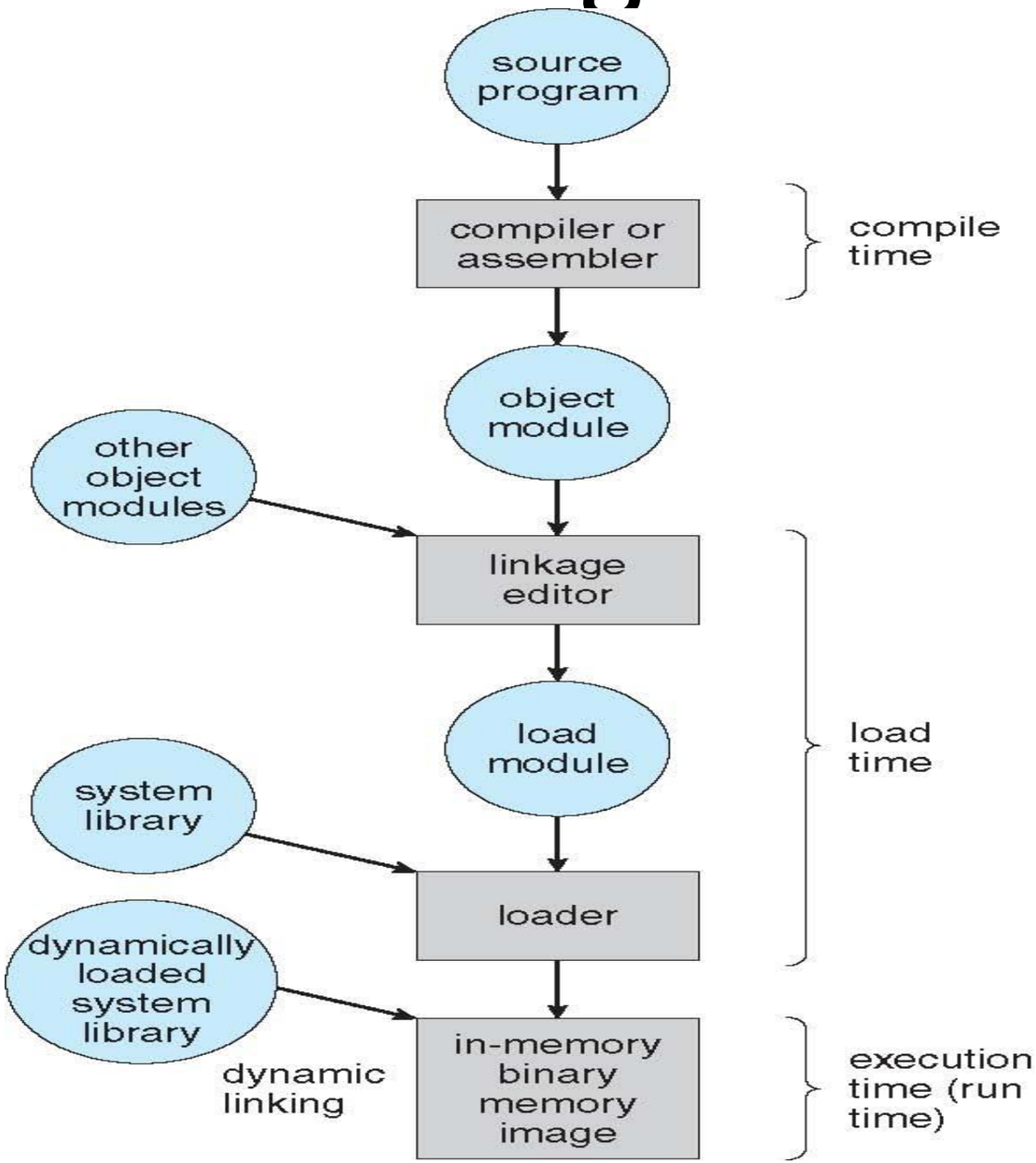
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 00000
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to 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. If starting address changes we need only reload the user code.
  - **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



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

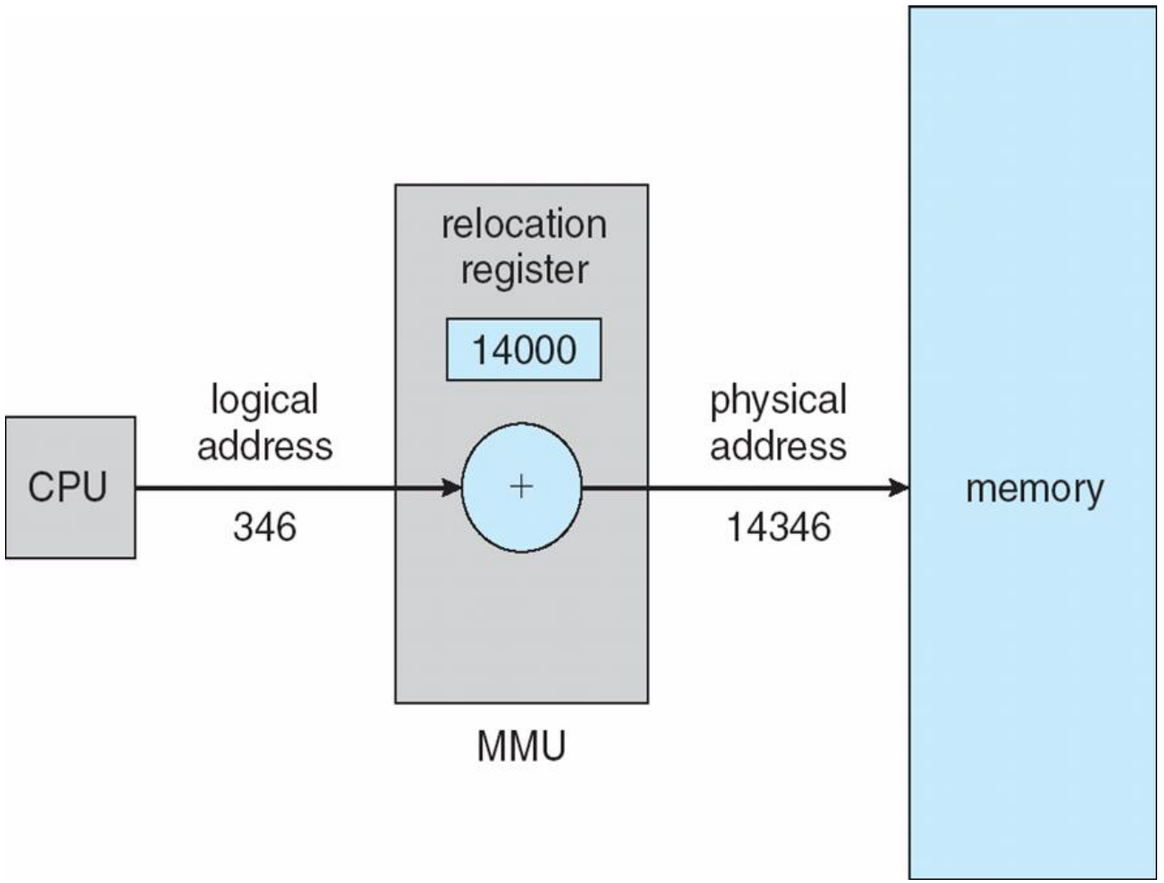
Continue.. (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
  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**
  - MS-DOS on Intel 80x86 used 4 relocation registers

- 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

- Routine is not loaded until it is called
- Advantages
  - 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



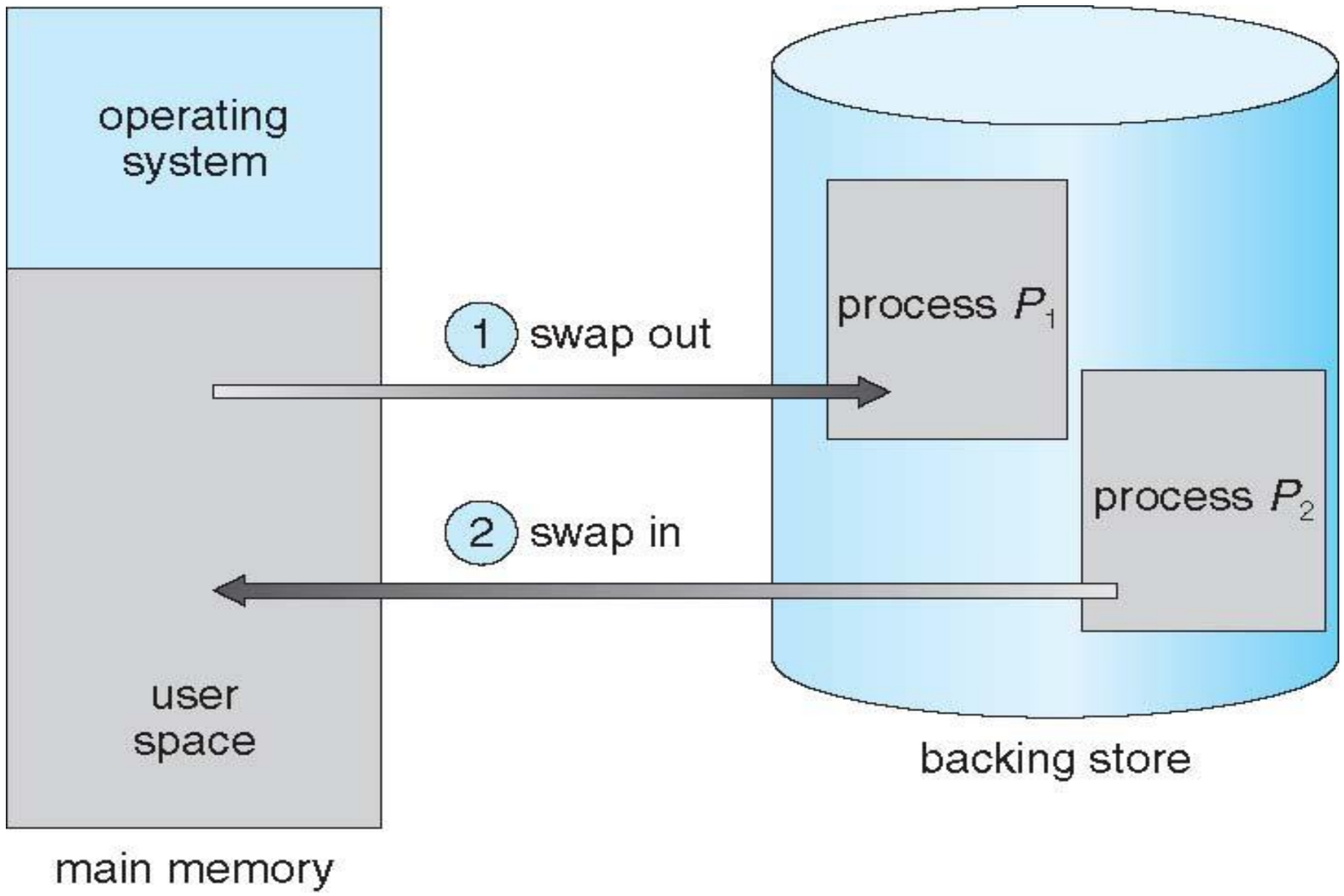
Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

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
- 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



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- Assume 100MB process . hard disk transfer rate = 50MB/sec
  - Swap out time = (100 MB/50 MB/sec) 2sec ➔2000ms
  - Total context switch =swapping in + swap out

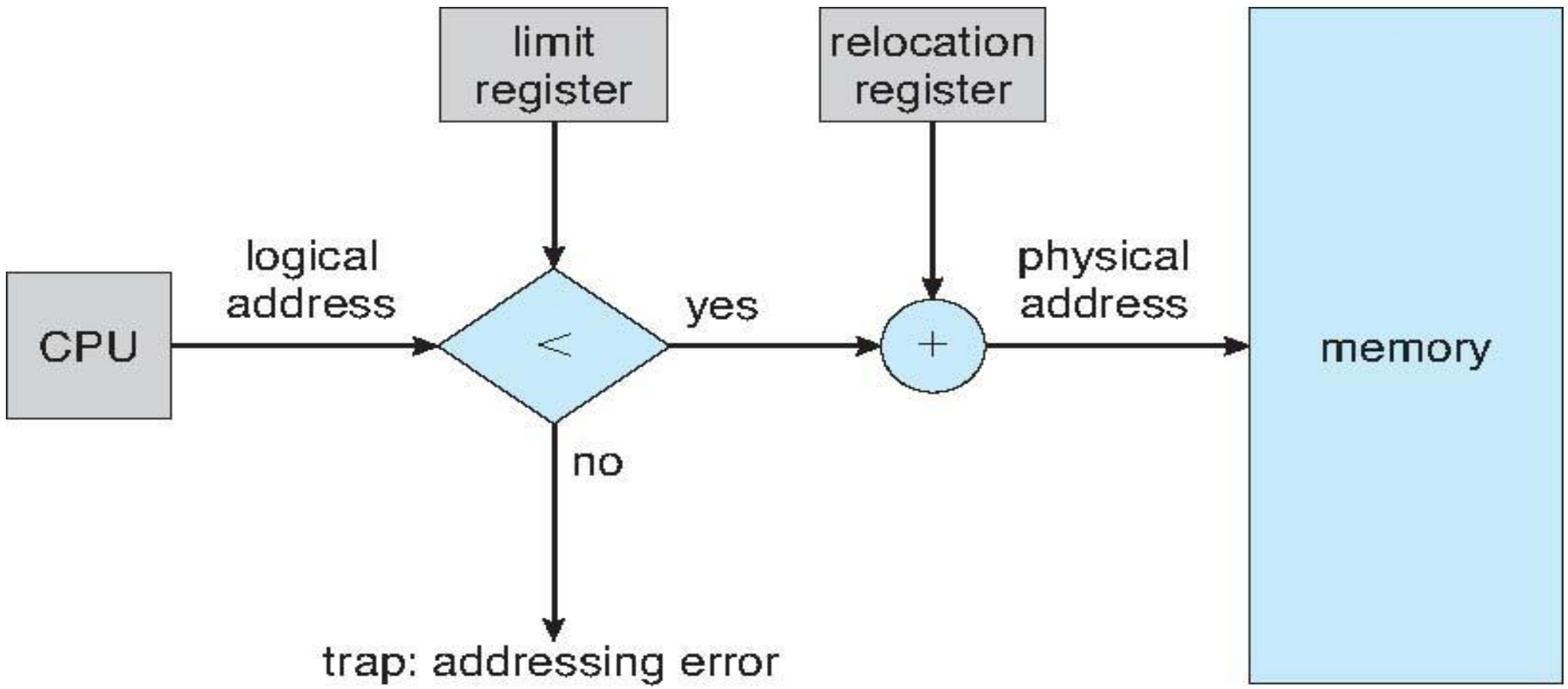
    2000ms + 2000ms=4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
    - Swap only when free memory extremely low

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

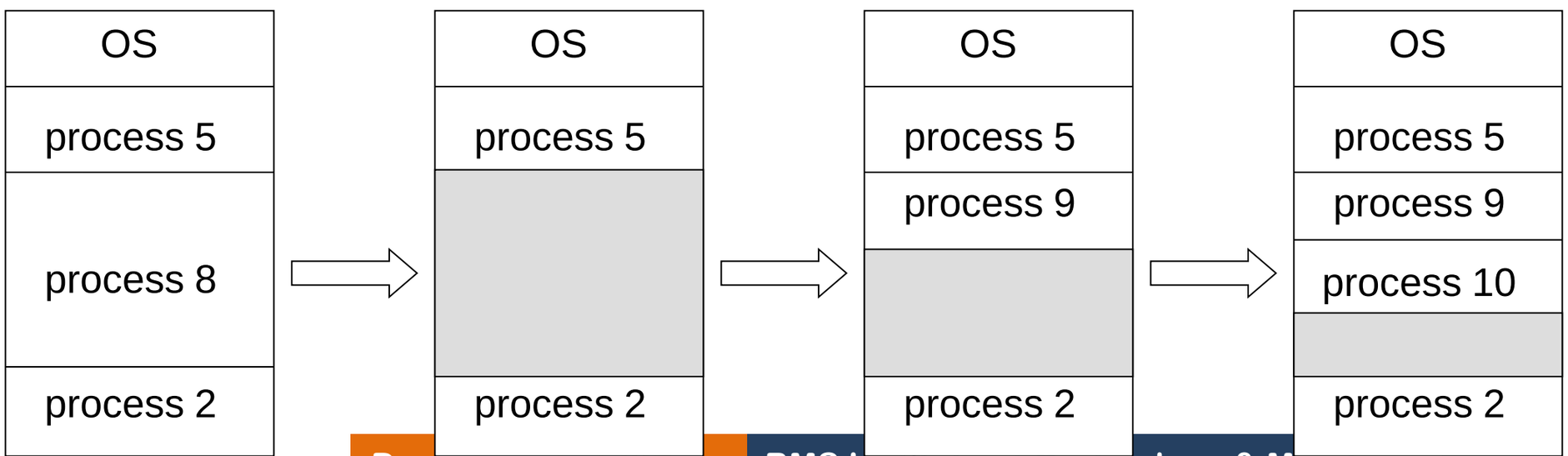
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **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
  - Process exiting frees its partition, adjacent free partitions combined
  - 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

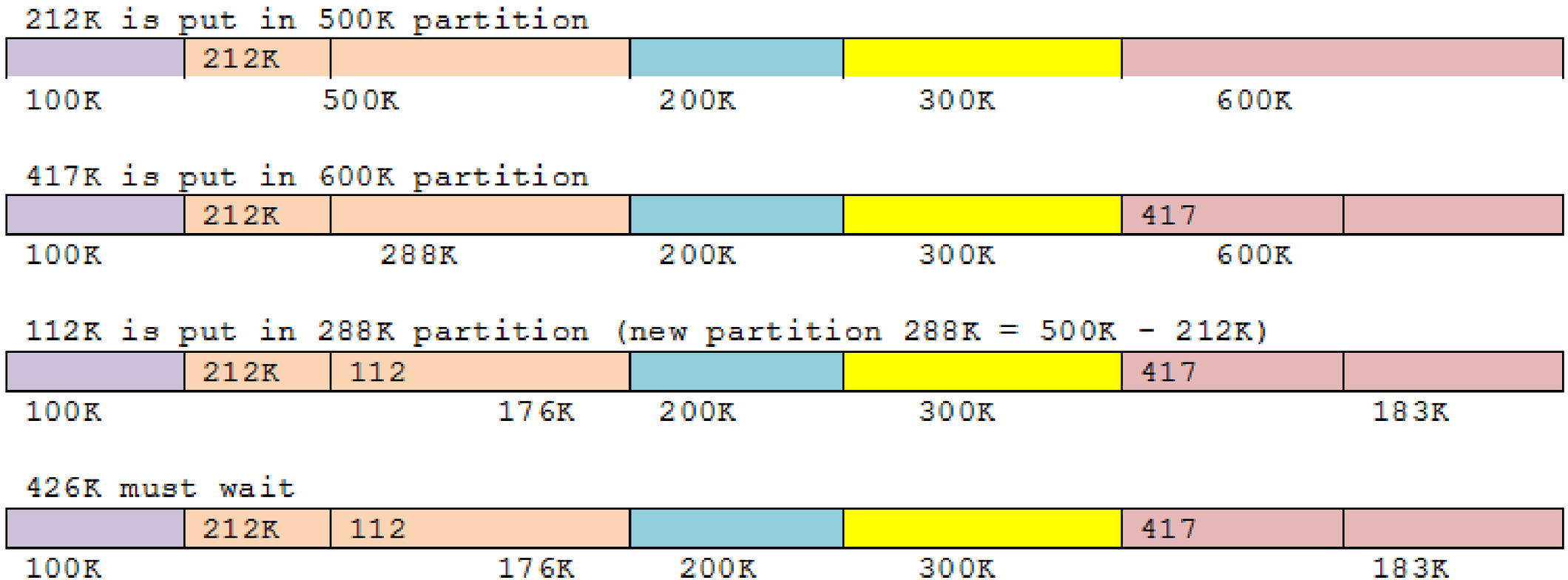
- Given five memory partitions of **100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order)**, how would the first-fit, best-fit, and worst-fit algorithms place processes of **212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)?** Which algorithm makes the most efficient use of memory?

First-fit:

- 212K is put in 500K partition (left over 500k-212= 288k)
- 417K is put in 600K partition
- 112K is put in 288K partition (left over 288K - 212K=76K)
- 426K must wait

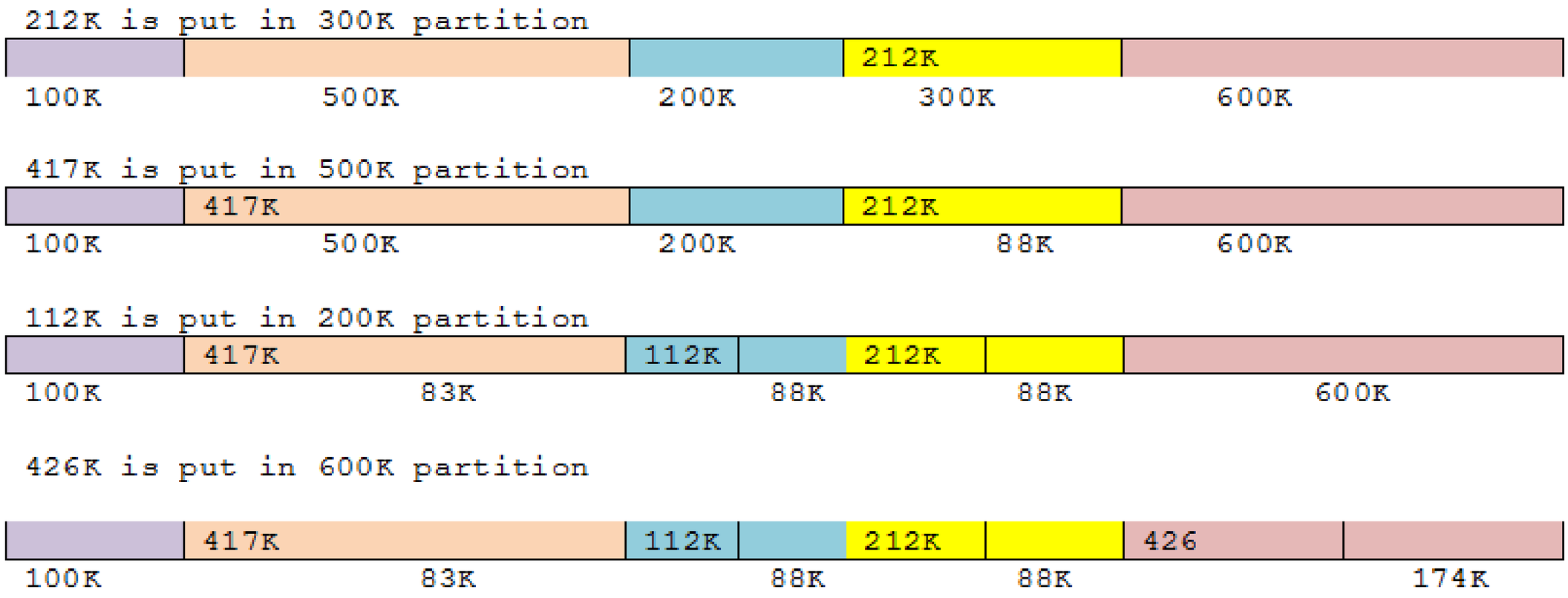
First Fit

- memory partitions of **100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order)**
- processes of **212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)**



Best fit

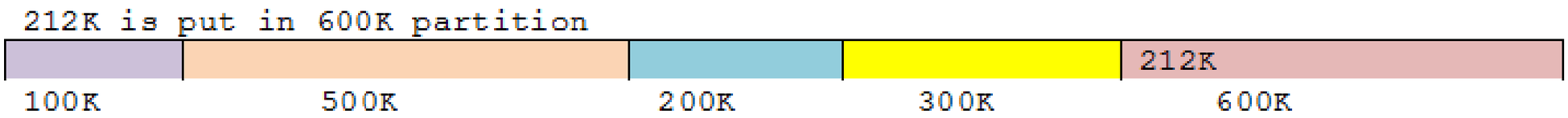
- memory partitions of **100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order)**
- processes of **212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)**



- 212K is put in 300K partition
- 417K is put in 500K partition
- 112K is put in 200K partition
- 426K is put in 600K partition

Worst Fit

- memory partitions of **100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order)**
- processes of **212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)**



- 212K is put in 600K partition (LEFT OVER 600K-212K=388K)
- 417K is put in 500K partition (LEFT OVER 500K-4172K=83K)
- 112K is put in 388K partition (LEFT OVER 500K-4172K=83K)
- 426K must wait

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

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, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- 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

Paging

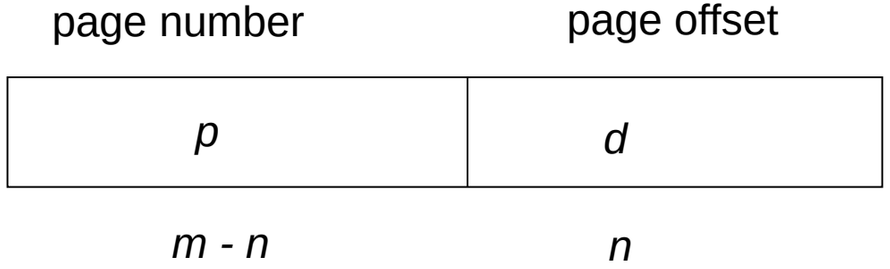
- Is a memory management Scheme that permits Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

Basic Method

- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- 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
- Backing store likewise split into pages
- Still have Internal fragmentation

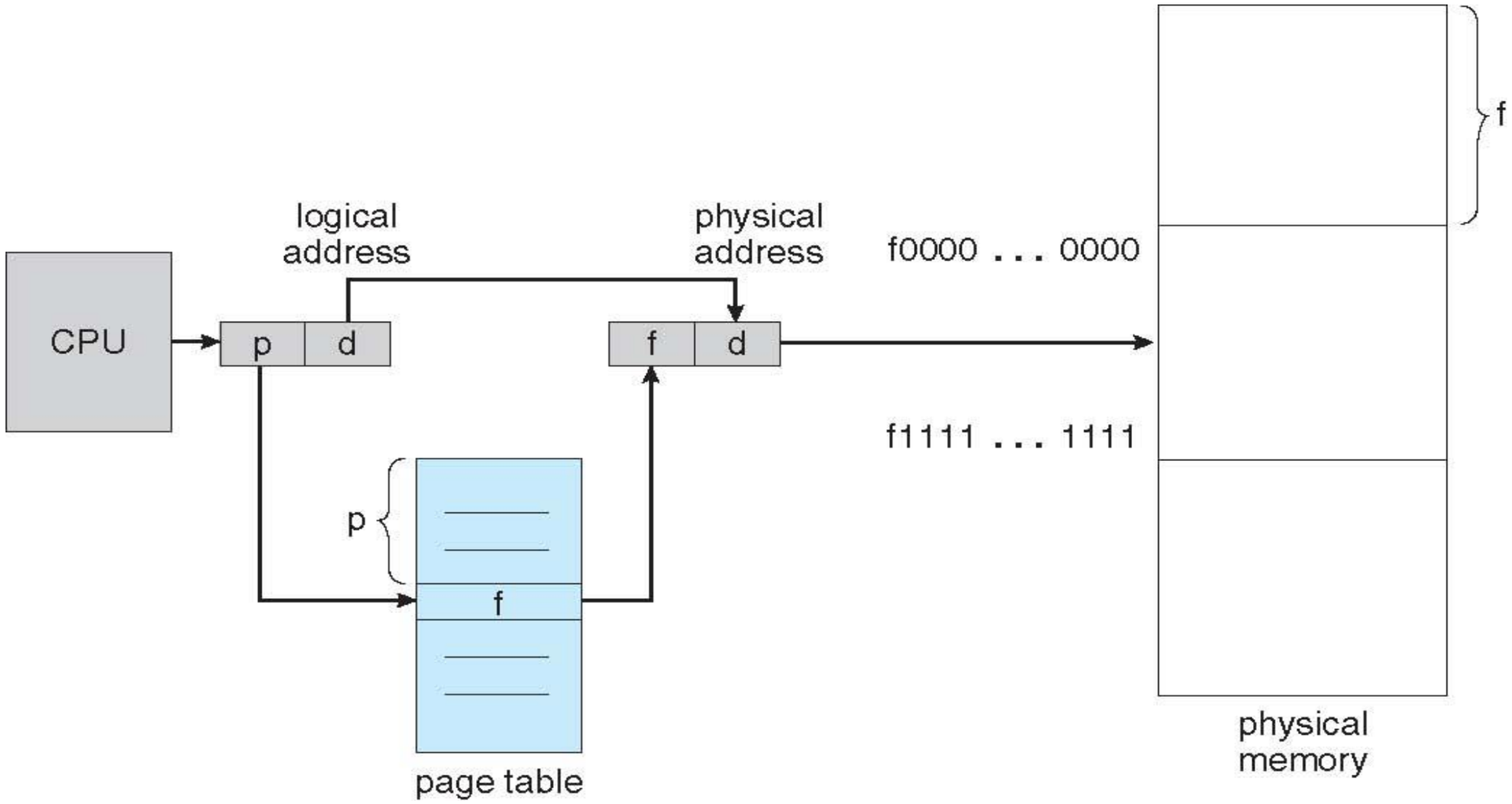
Address Translation Scheme

- 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

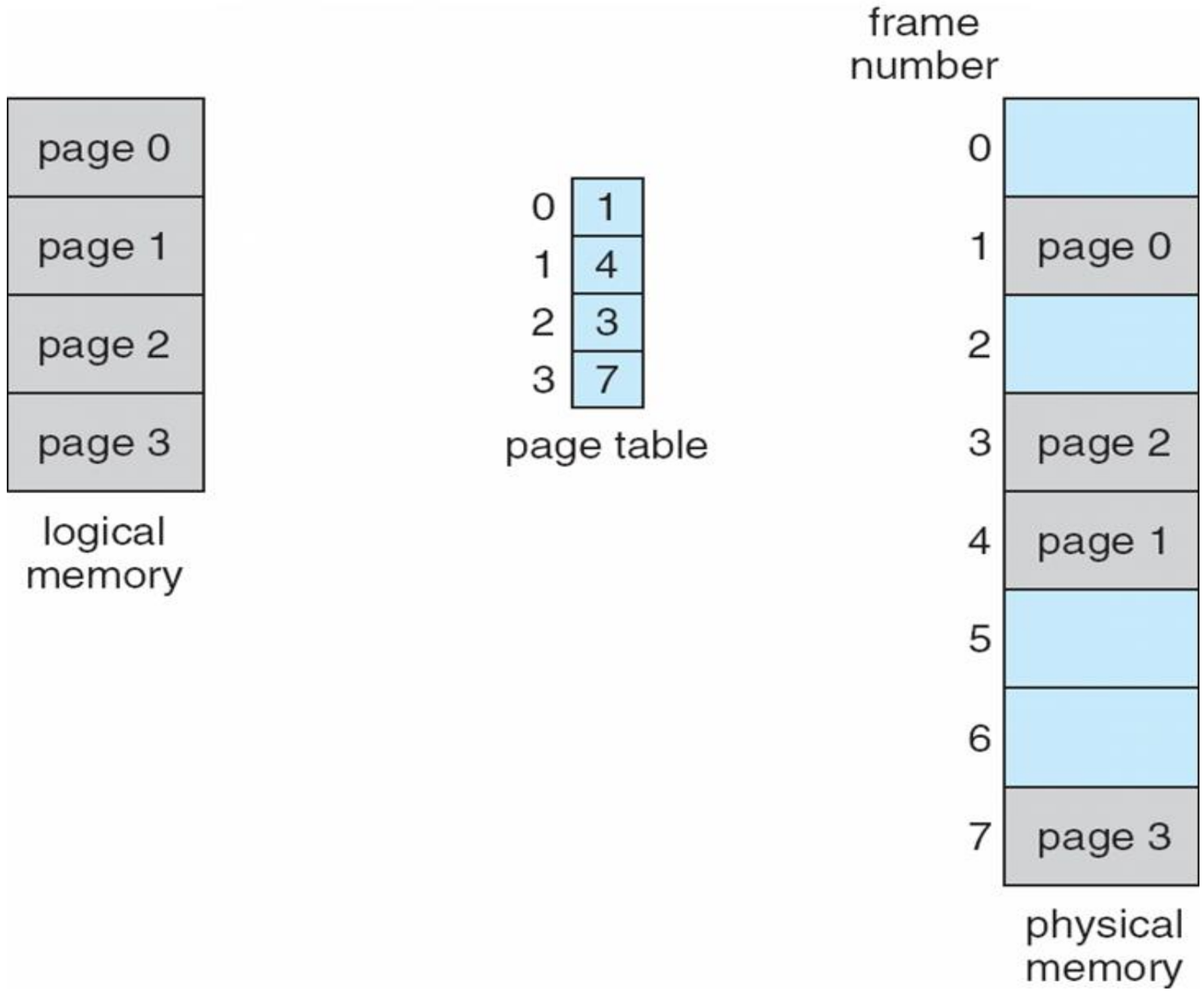


  - For given logical address space 2^m and page size 2^n

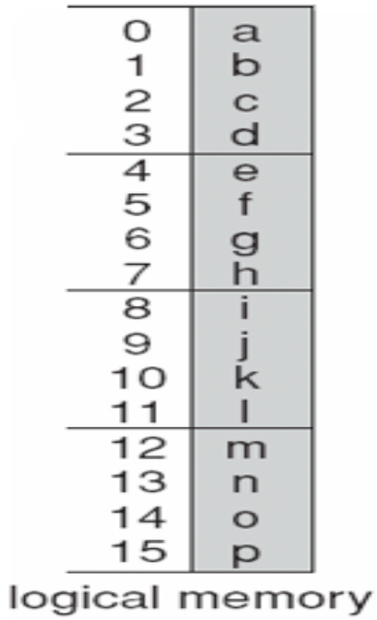
Paging Hardware

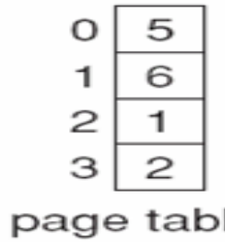


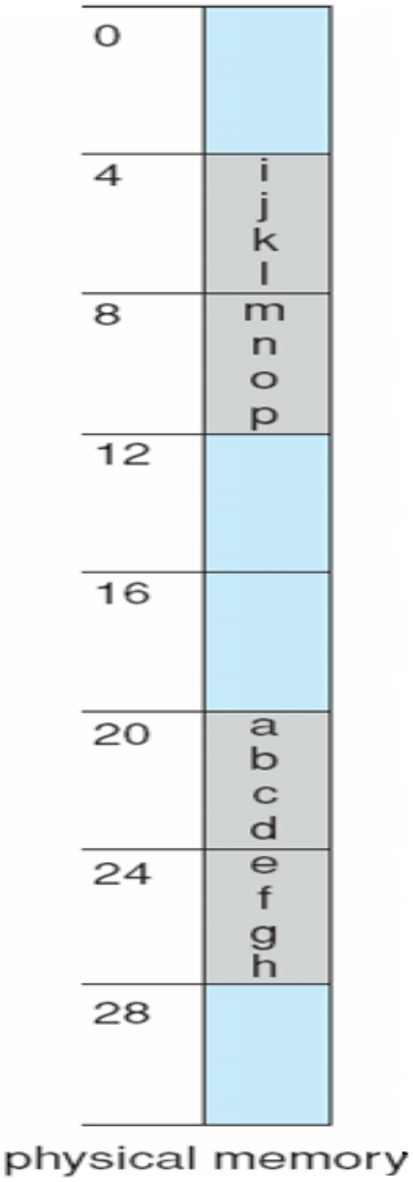
Paging Model of Logical and Physical Memory



Paging Example







n=2 and *m*=4 32-byte

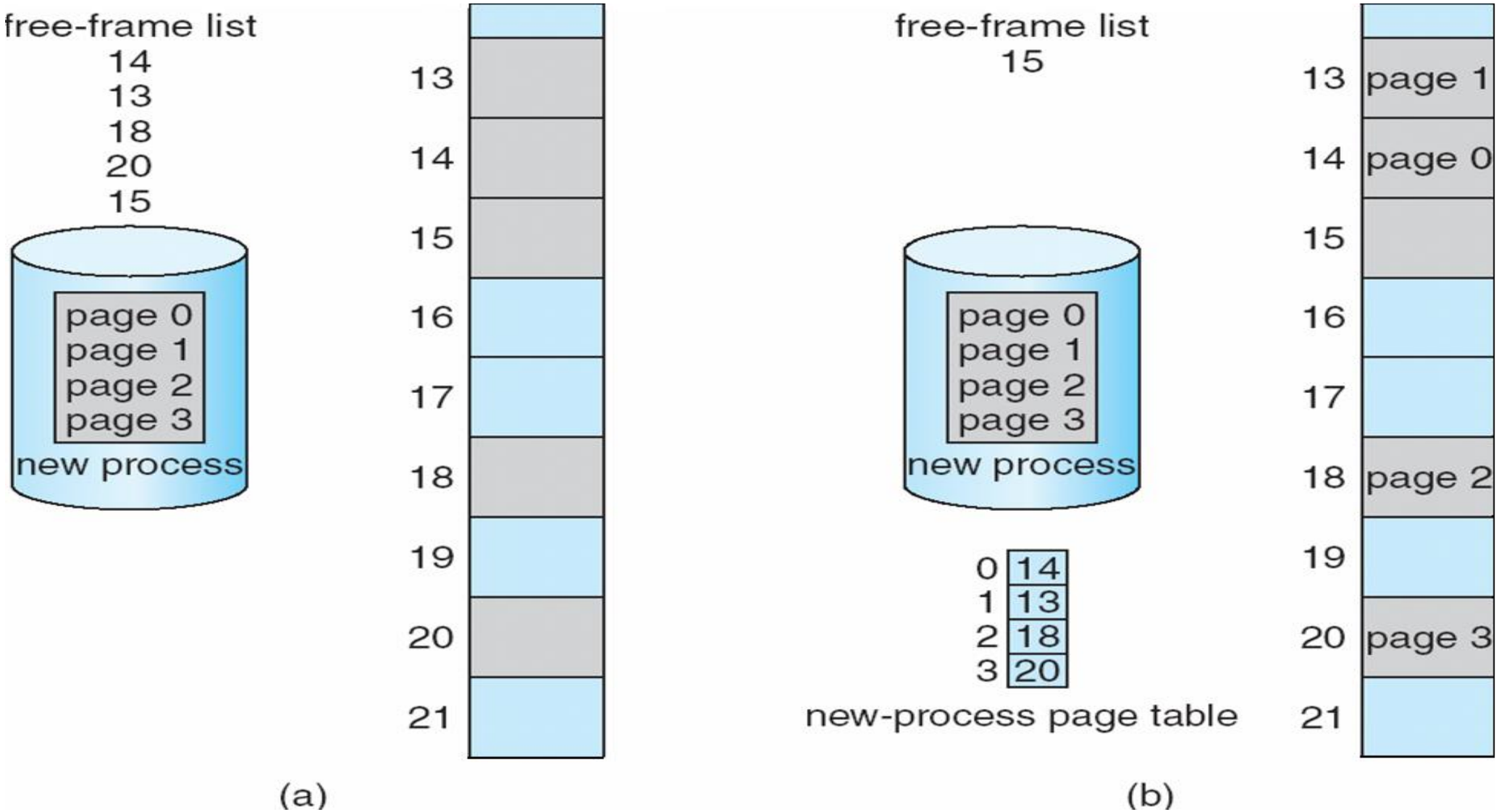
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different

Advantage

- Clear separation between the user's view of memory and the actual physical memory
- Process can only access its own memory

Free Frames



Before allocation

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
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered

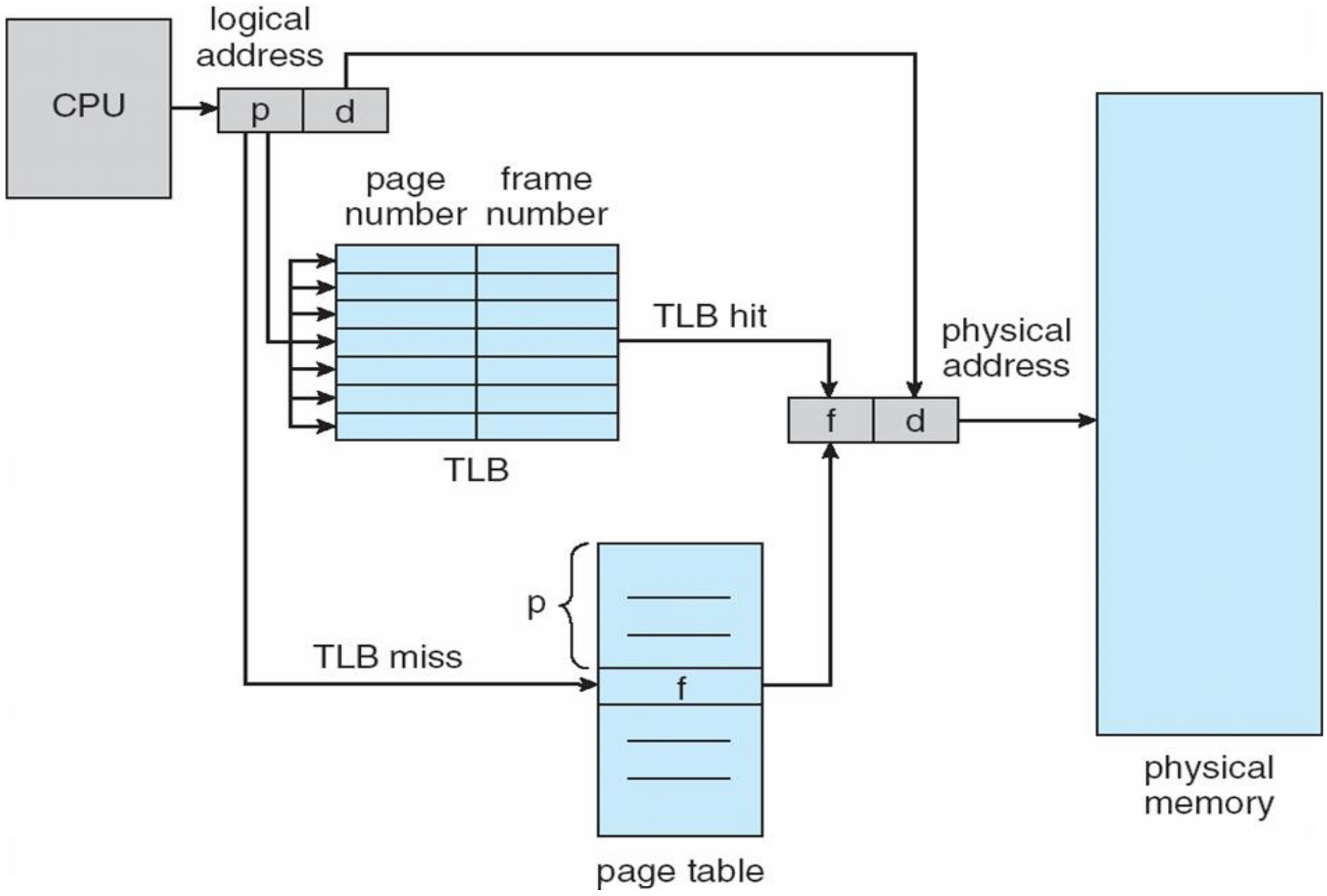
Associative Memory

- 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



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio –> percentage of times that a page number is found in the associative registers(TLB);
  - TLB miss-> if page number is not found in the TLB
- Consider α = 80%, ε = 20ns for TLB search, 100ns for access page table and 100 ns for accessing desired byte in physical memory
- **Effective Access Time** (**EAT**)

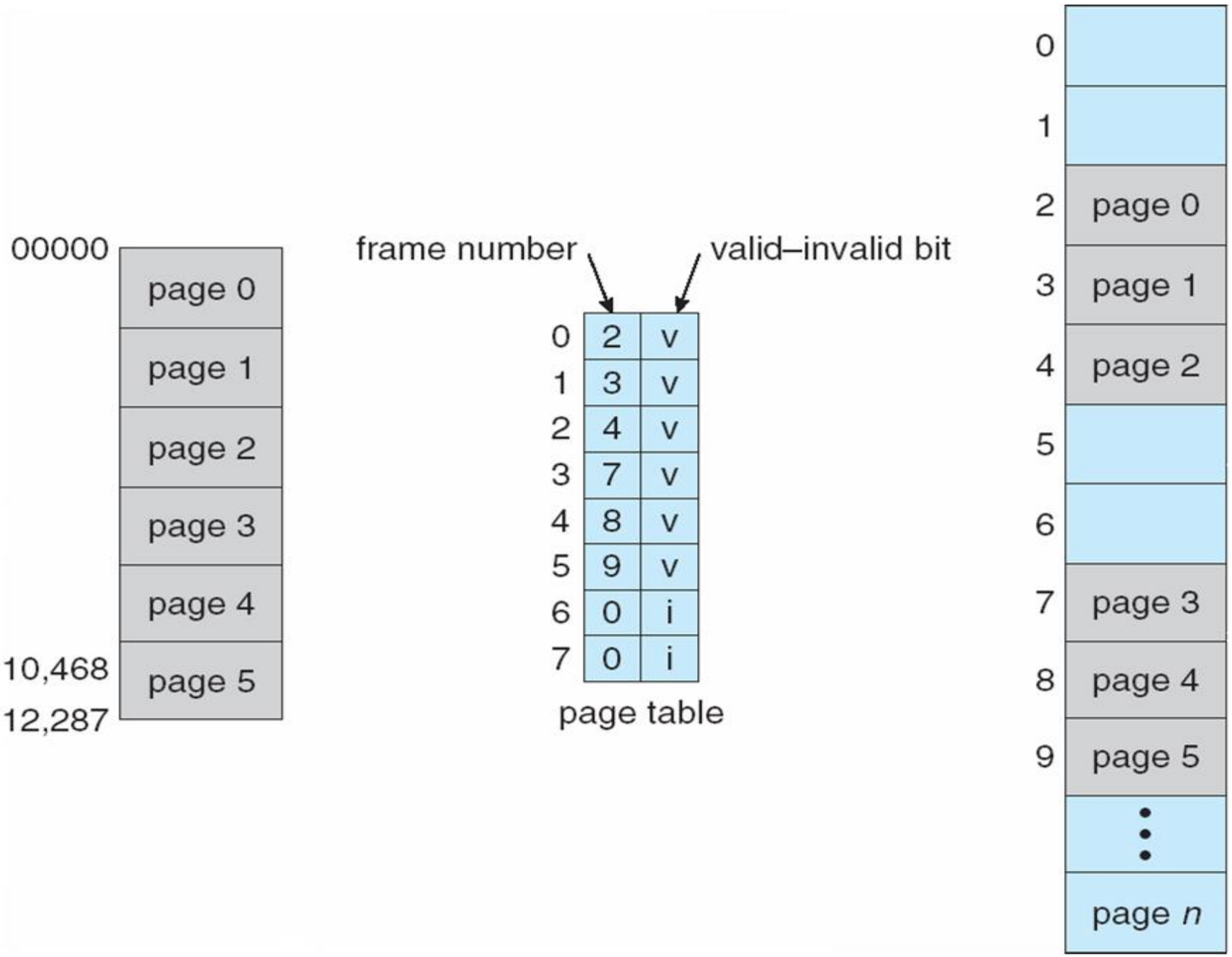
  EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 98%, ε = 20ns for TLB search, 100ns for memory access
  - **EAT = 0.98 x 120 + 0.02 x 220 = 122ns**

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
  - **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
  - Or use **page-table length register** (**PTLR**)

- Any violations result in a trap to the kernel

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

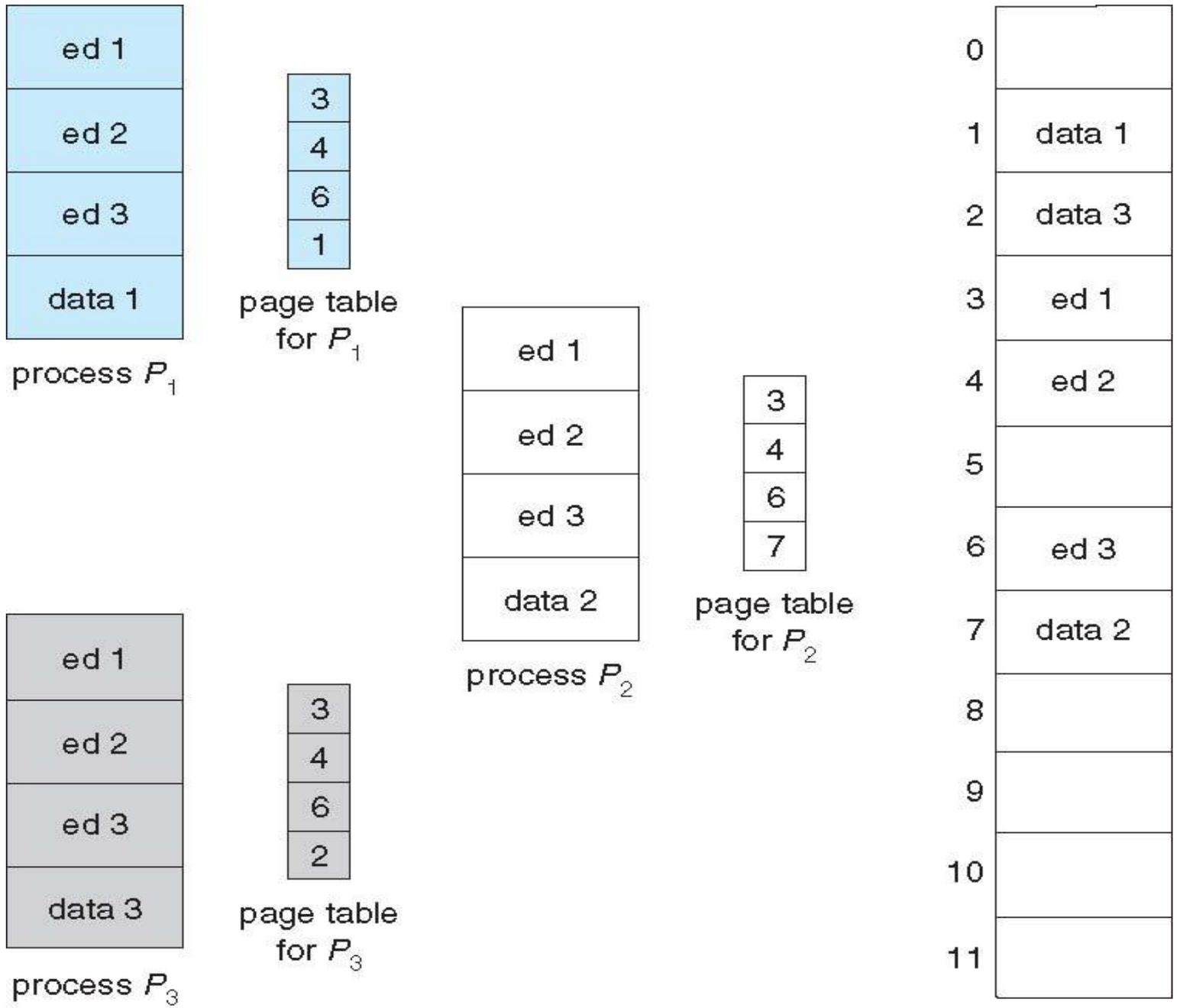


Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

- **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

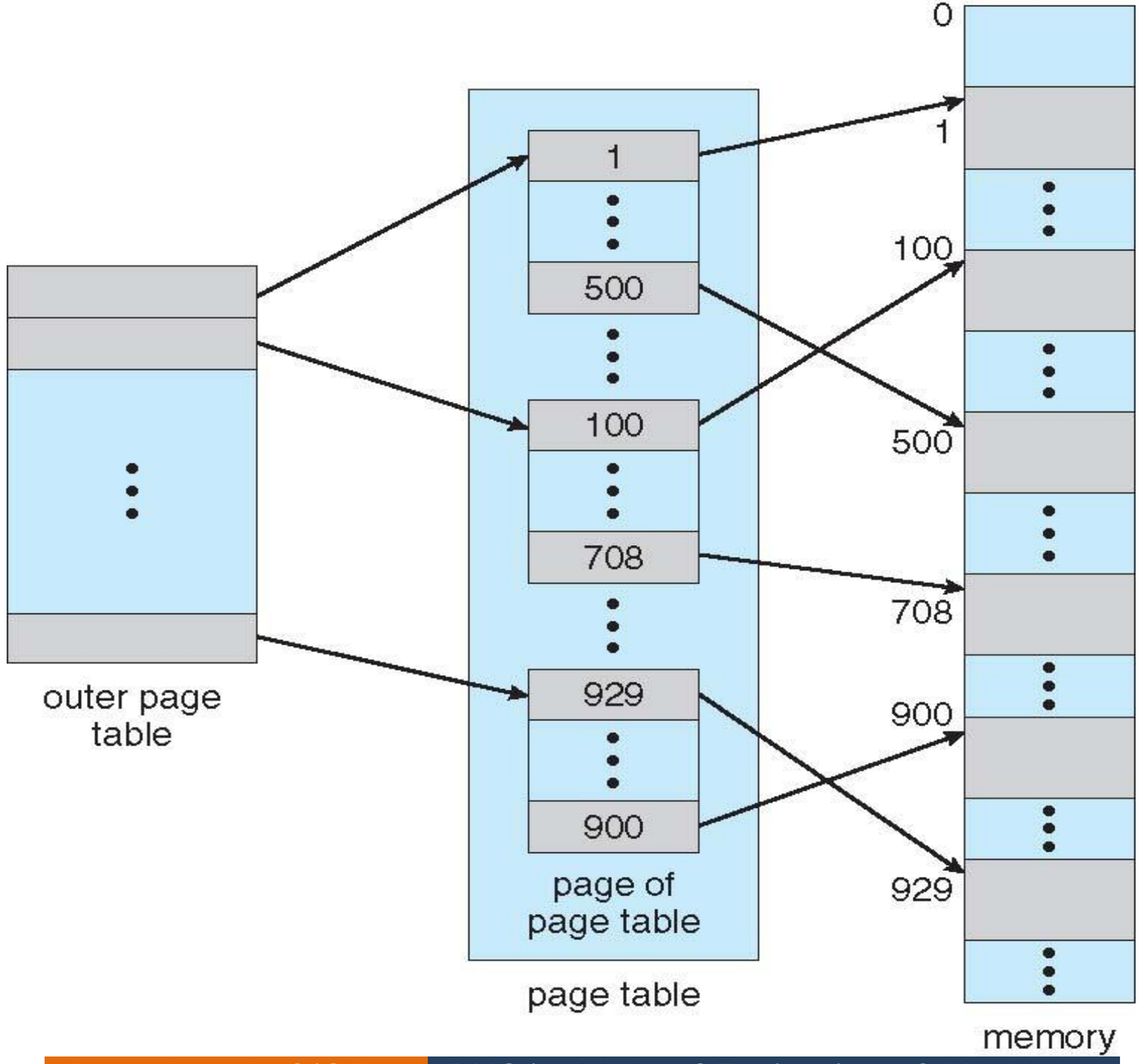
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory

- 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
- We then page the page table

Two-Level Page-Table Scheme

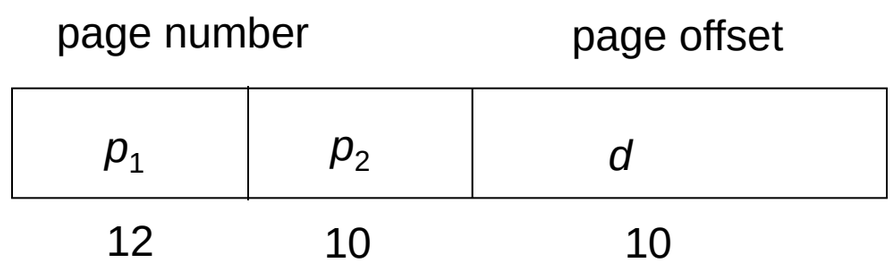


Two-Level Paging Example

- 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

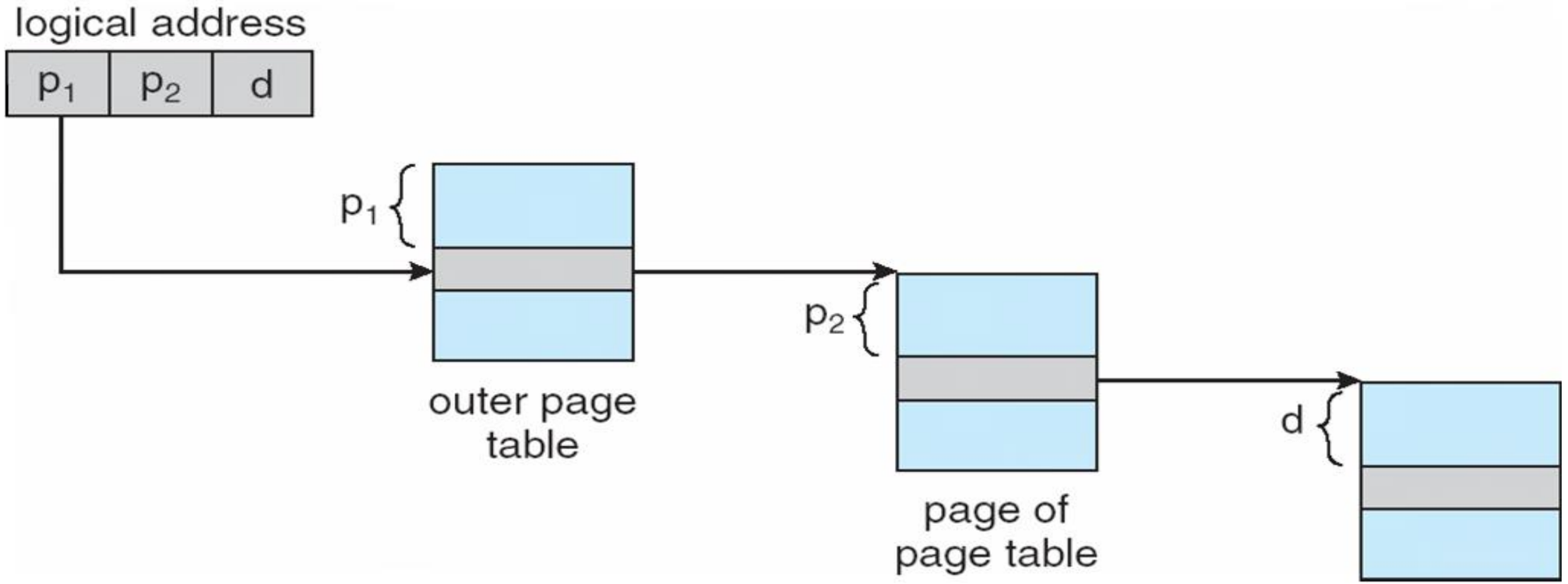
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset

- Thus, a logical address is as follows:



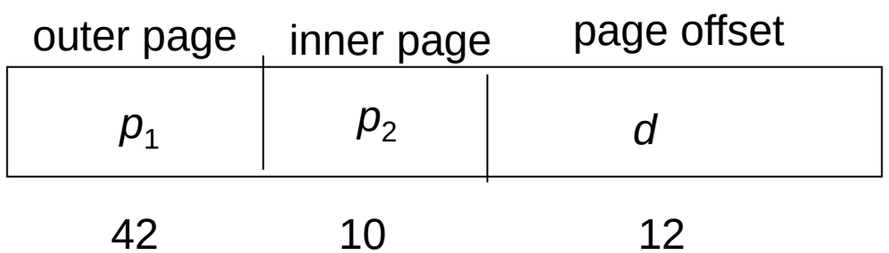
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table Known as **forward-mapped page table**

Address-Translation Scheme



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})



  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries

Address would look like

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

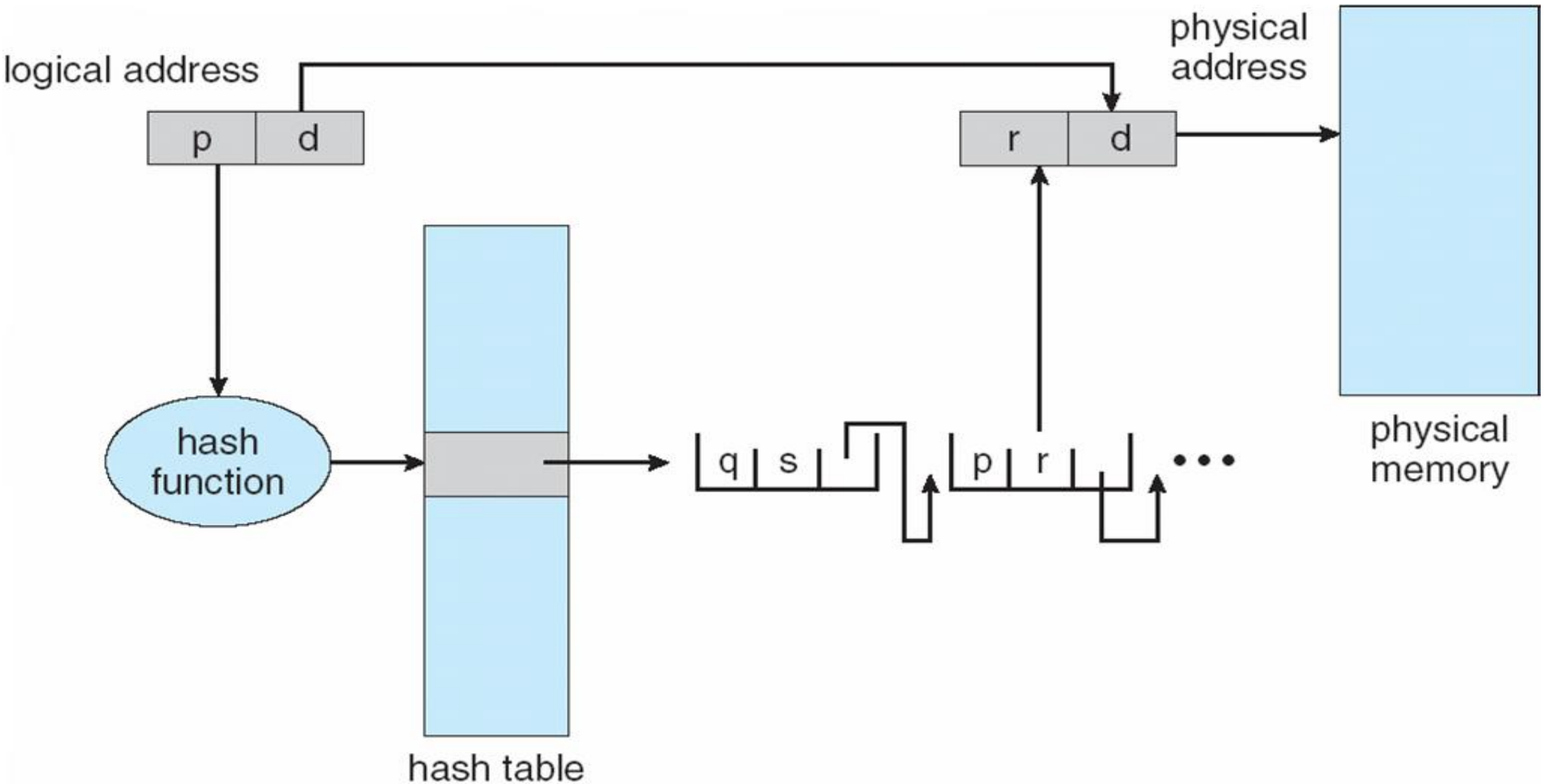
outer page	inner page	offset
p_1	p_2	d
42	10	12

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
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table

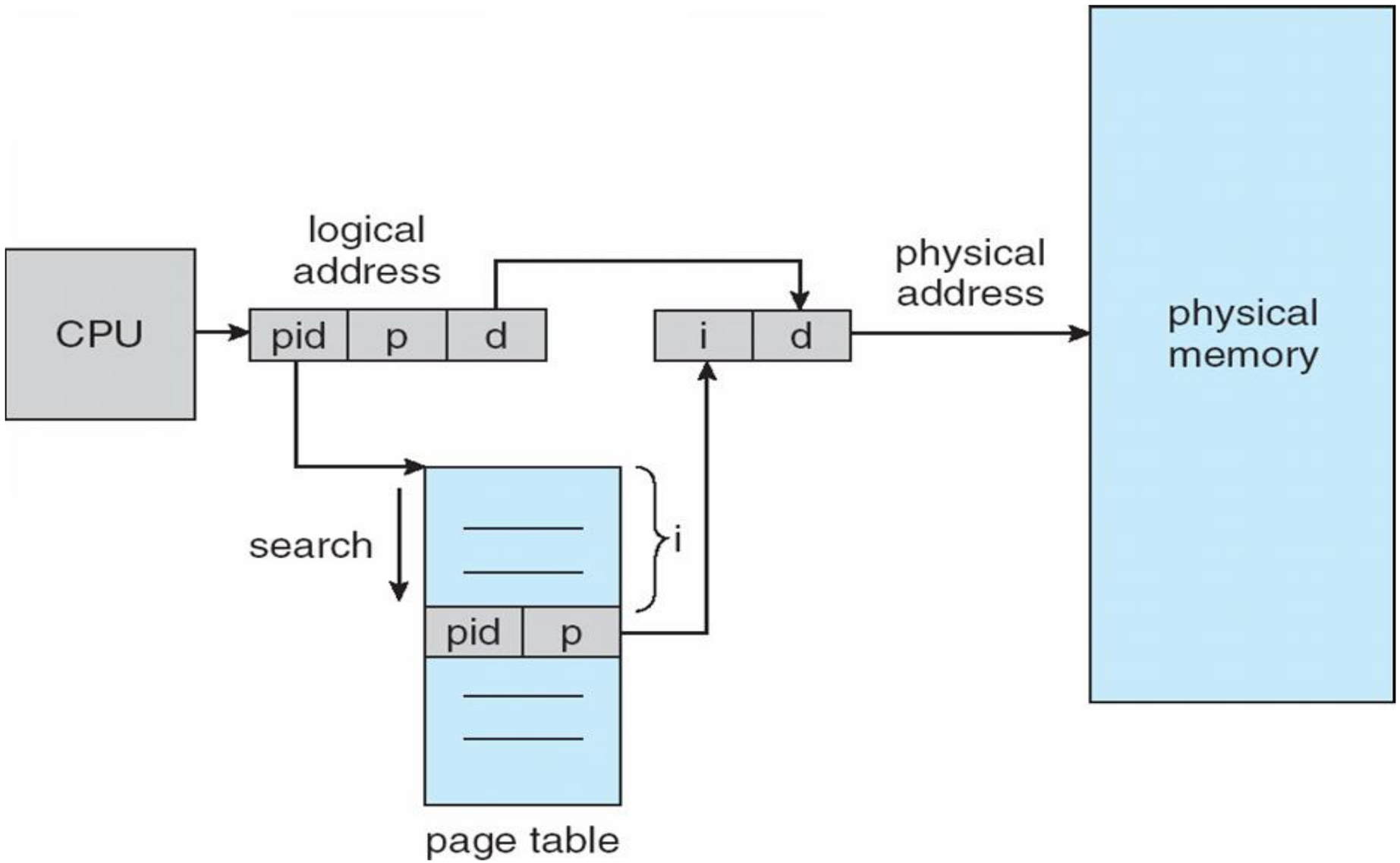


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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
  - TLB can accelerate access

- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

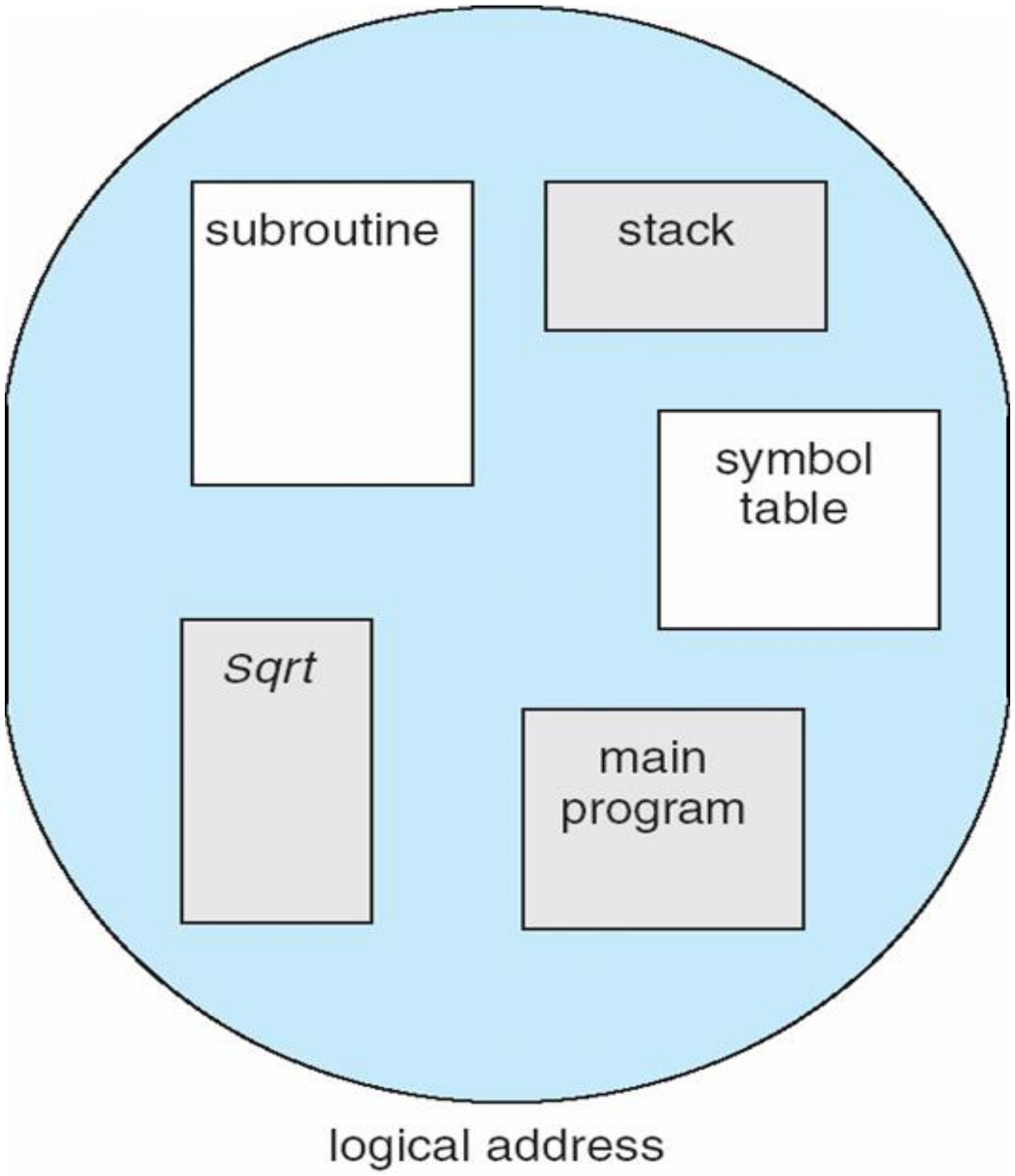
Inverted Page Table Architecture



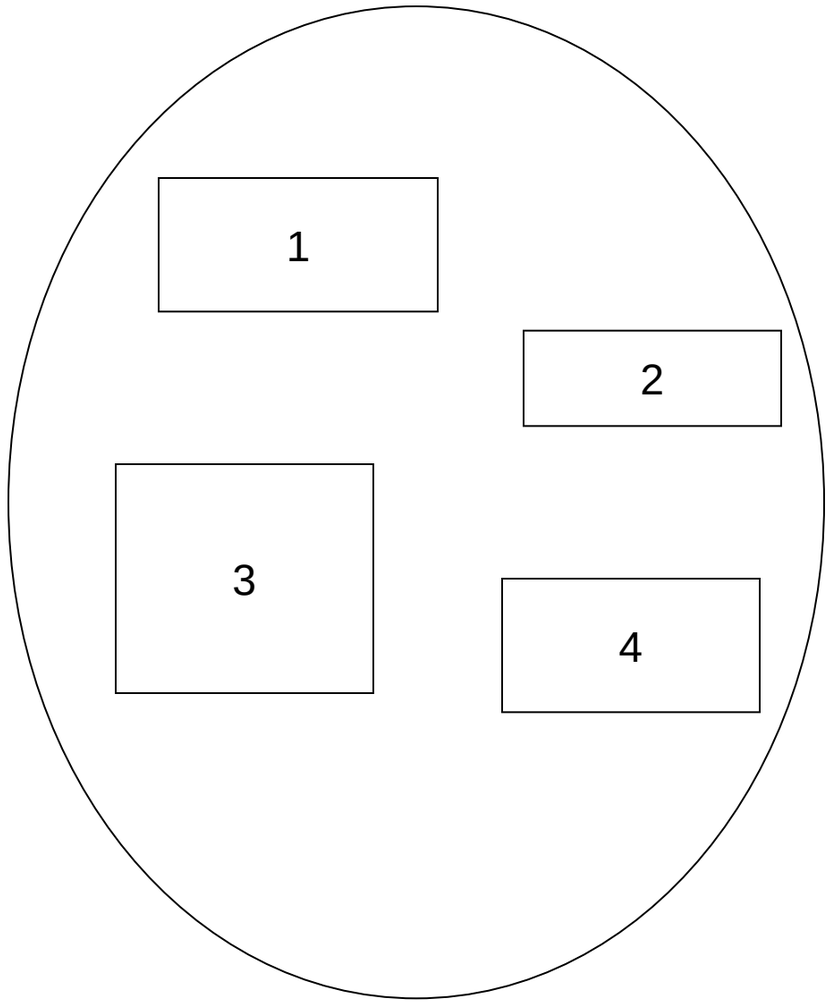
Segmentation

- Is a memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - Code
    - Global Variables
    - Heap
    - Stack
    - Standard C library

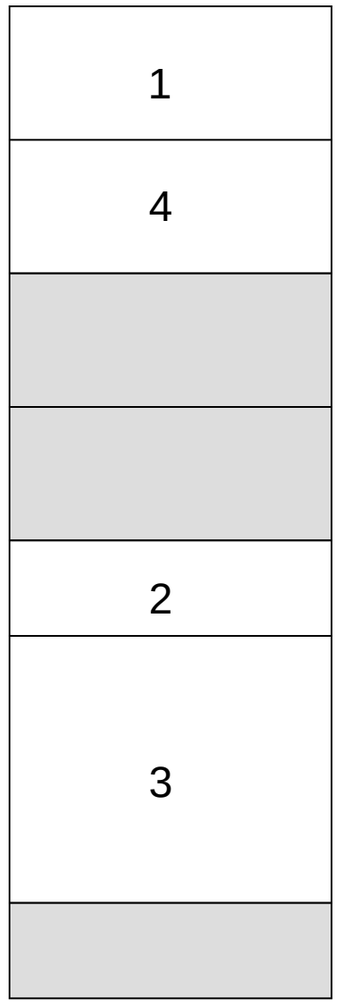
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **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.)

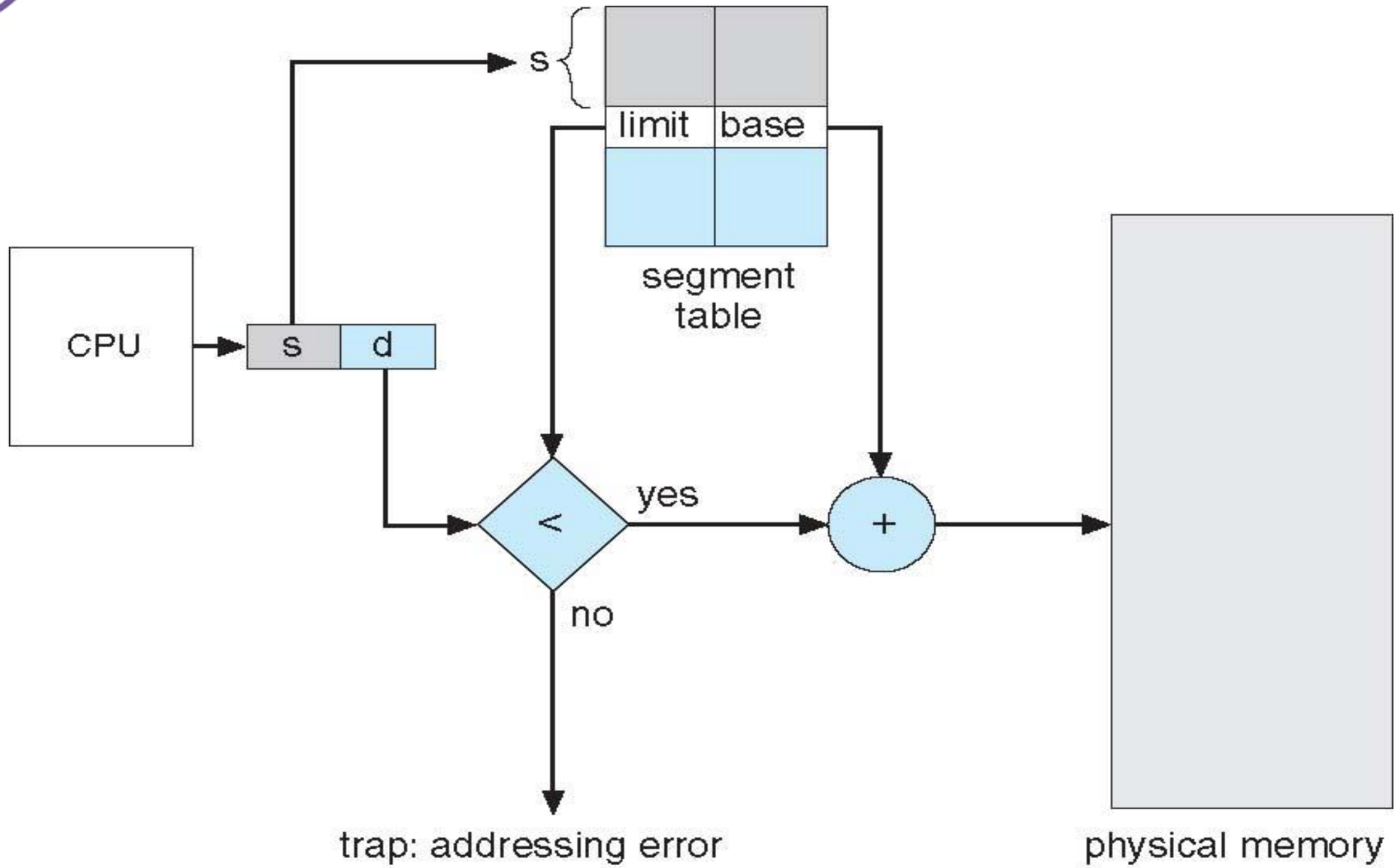
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

- Protection bits associated with segments; code sharing occurs at segment level

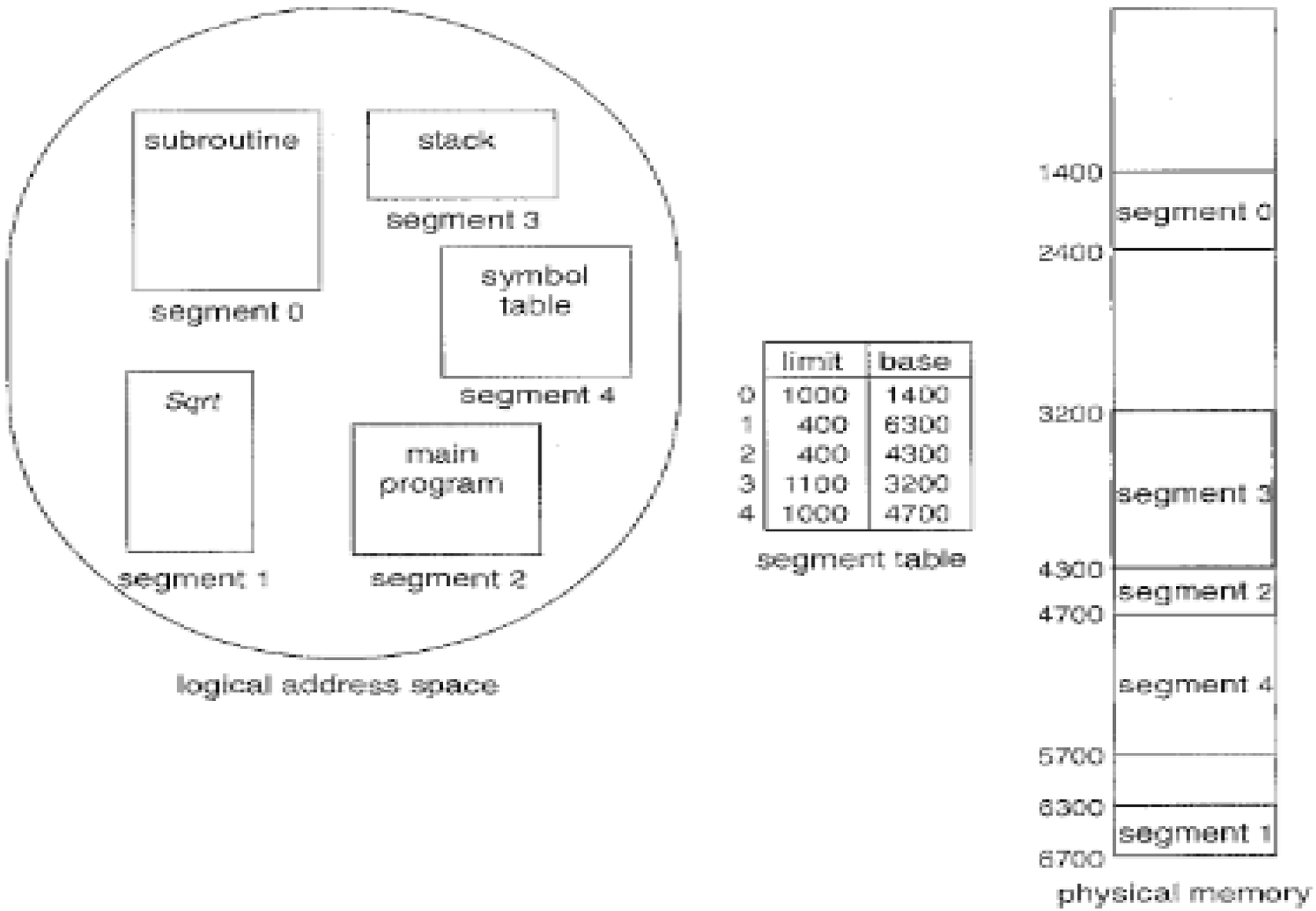
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



End of Chapter 8

Mod-4 : Virtual Memory

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

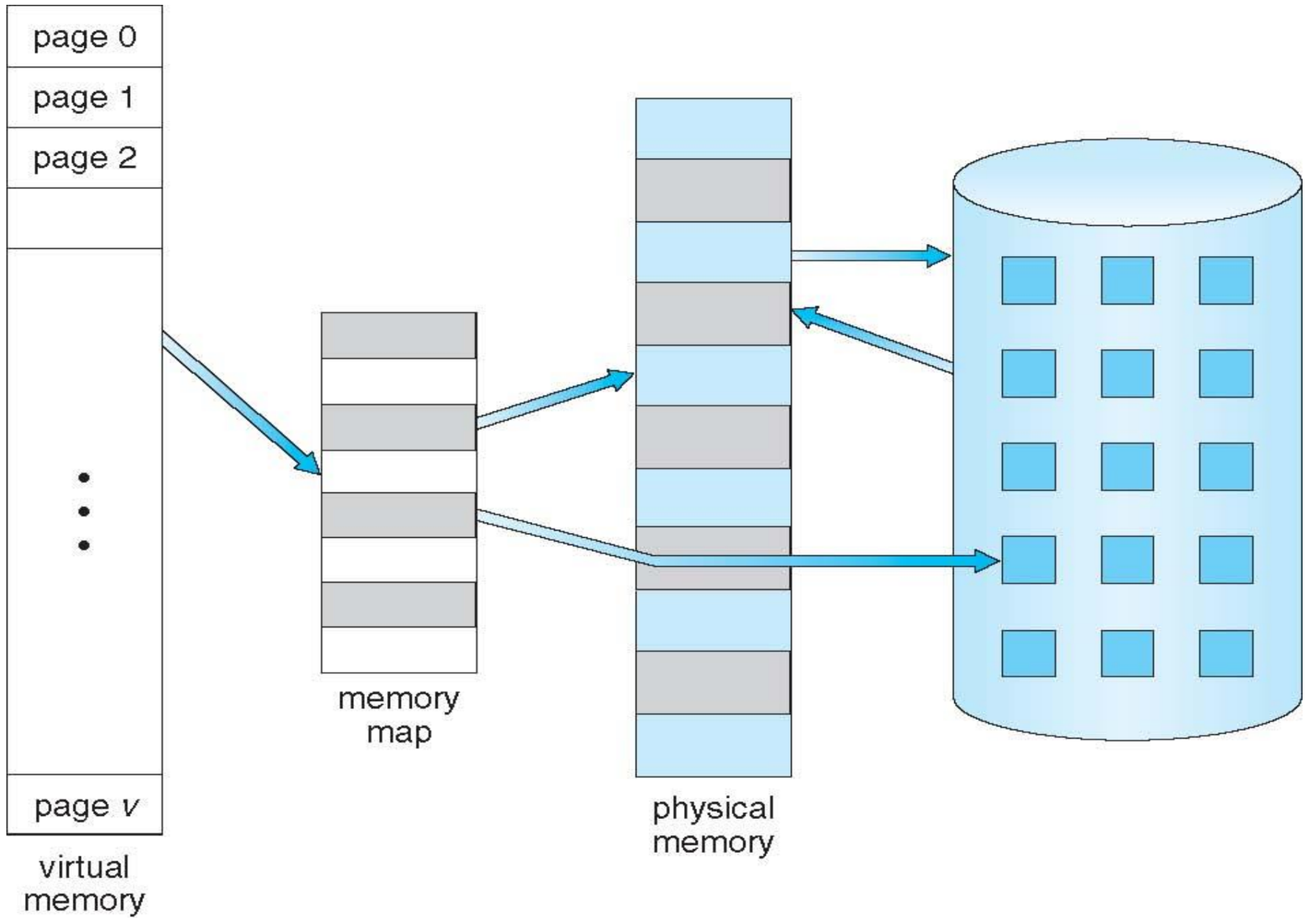
Background

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

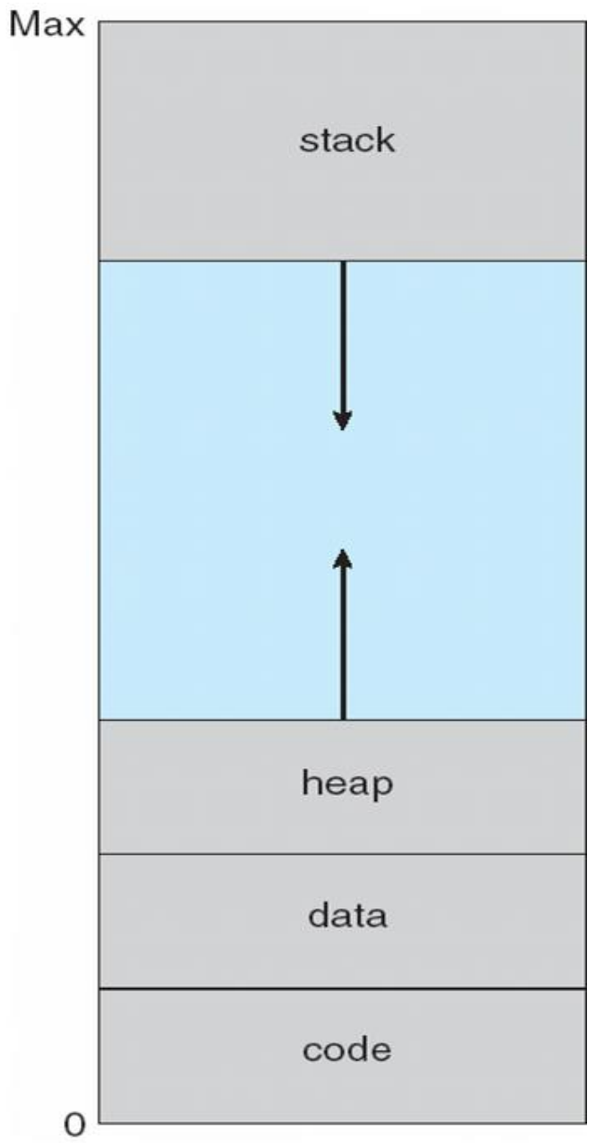
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

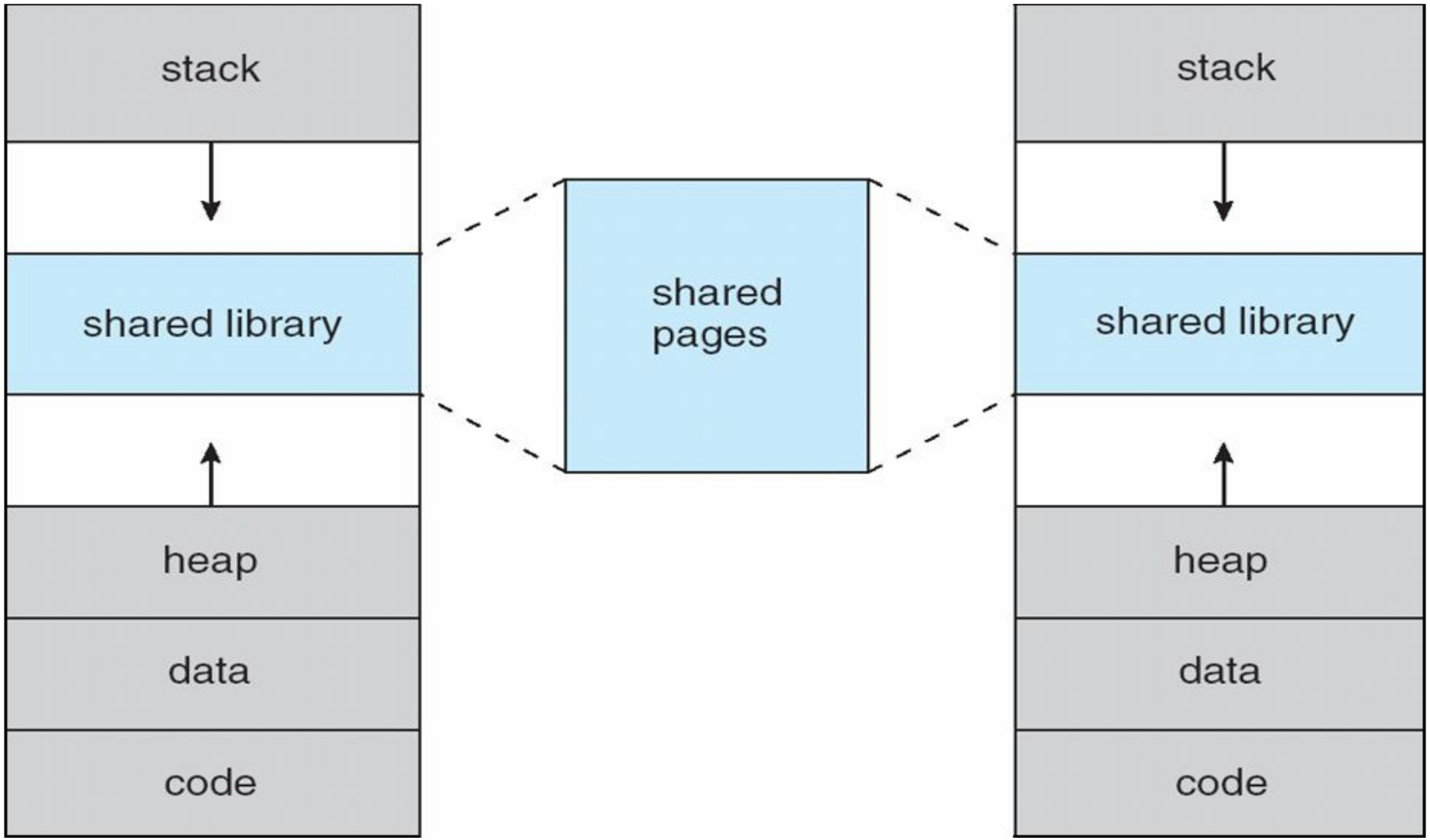


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

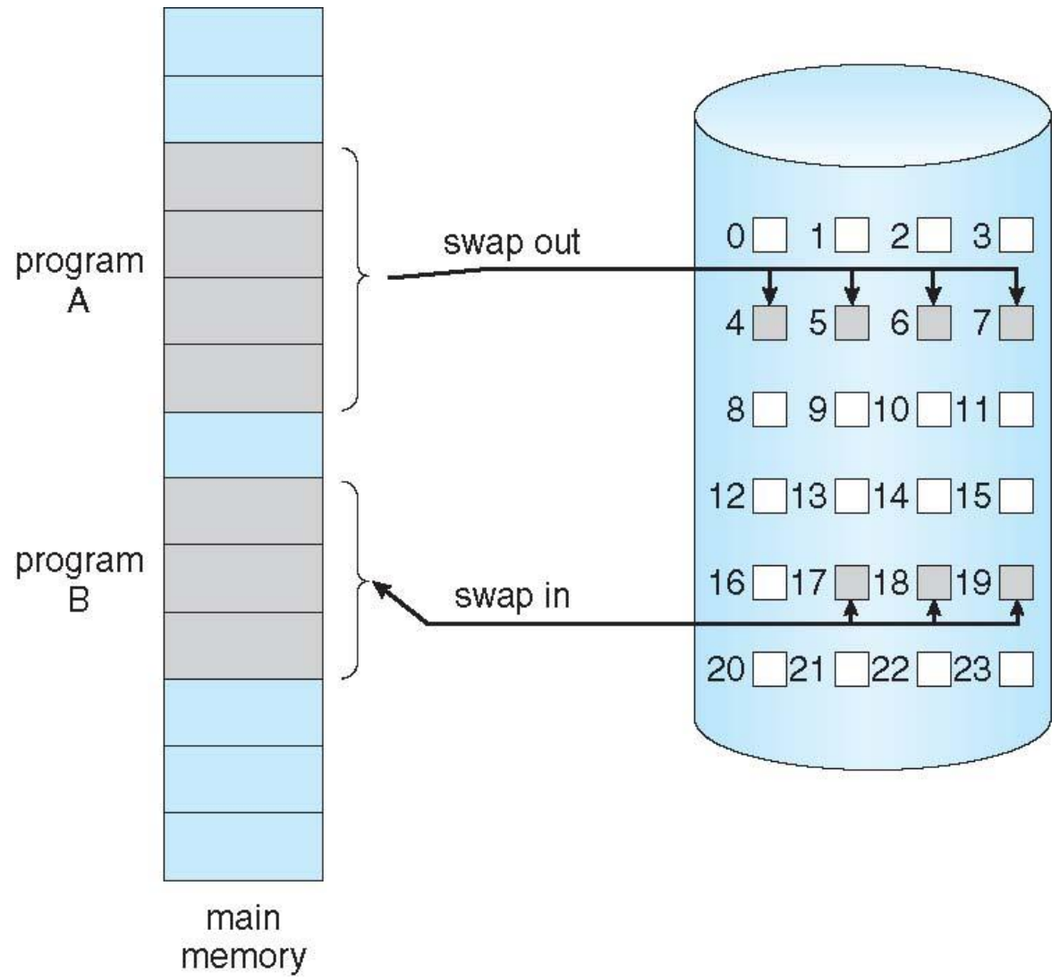


Shared Library Using Virtual Memory



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users



- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-Invalid Bit

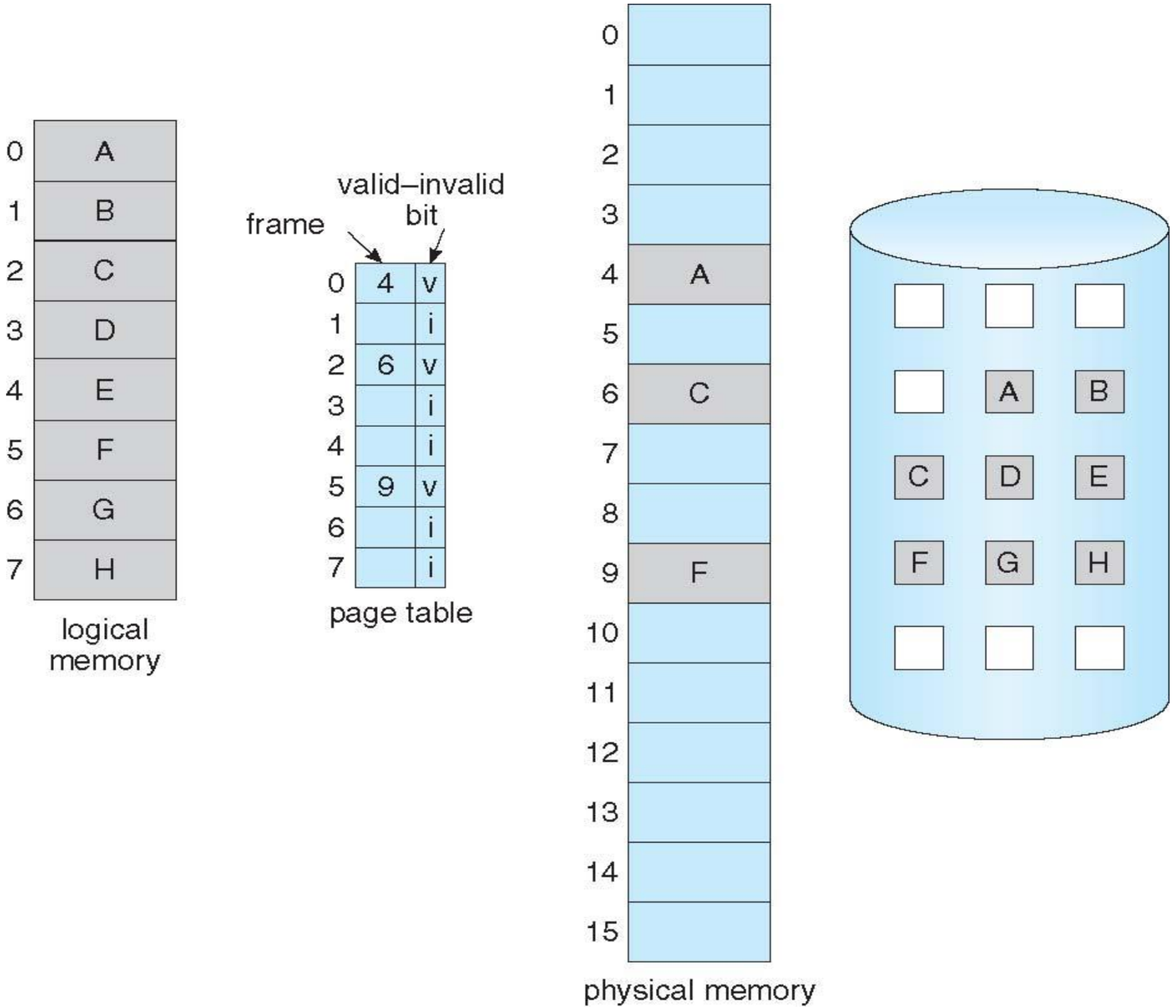
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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

page table

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

Page Table When Some Pages Are Not in Main Memory



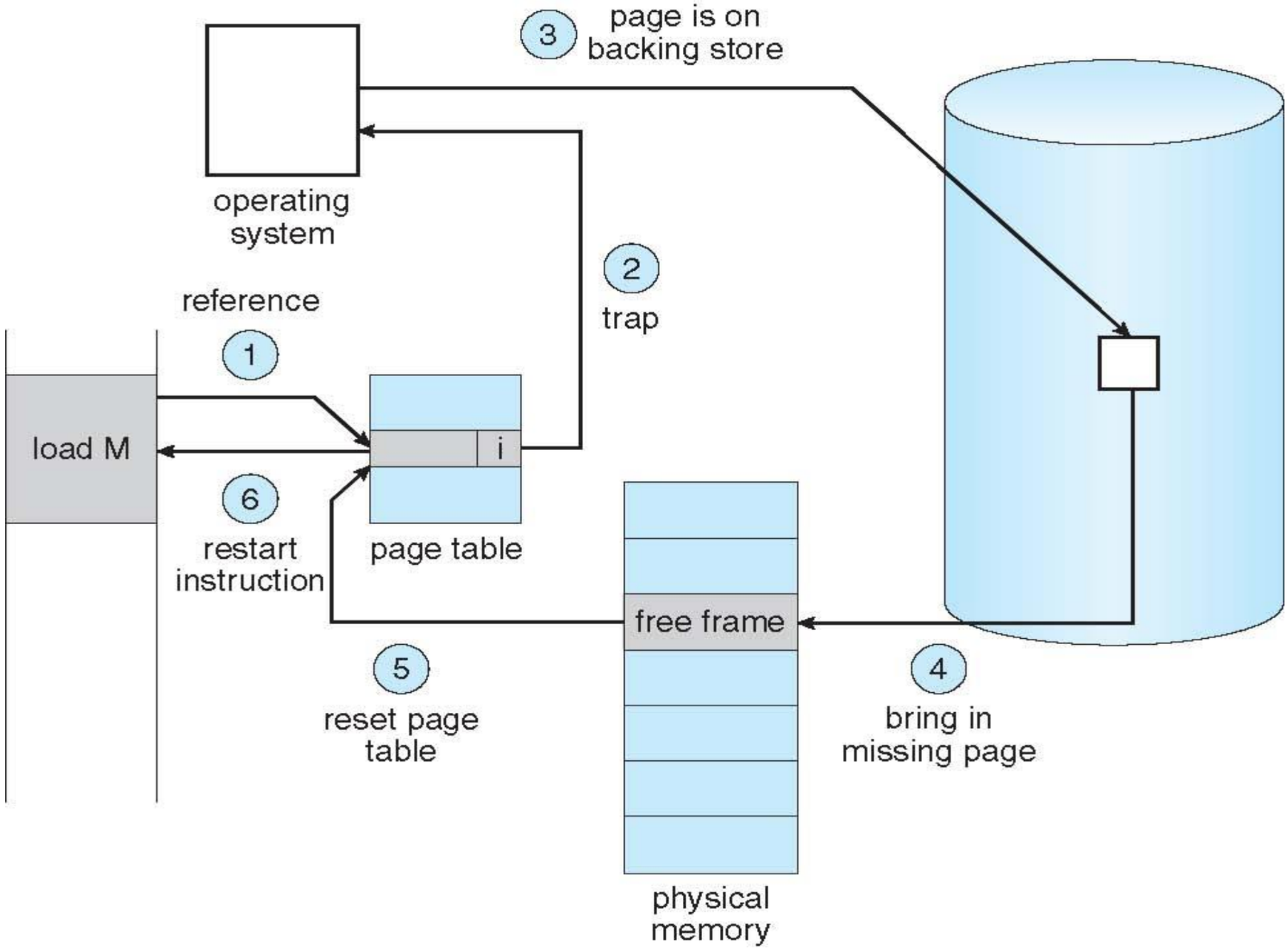
Page Fault

- 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 ⇒ abort
   - Just not in memory
2. Find free 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

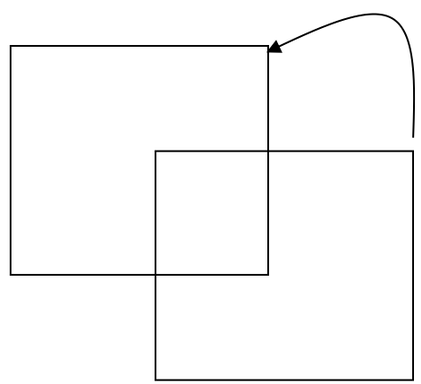


Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging (Never bring a page in to memory until it is required)**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference (same set of pages accessed over a short period of time)**
- Hardware support needed for demand paging
  1. Page table with valid / invalid bit
  2. Secondary memory (swap device with **swap space**)
  - Crucial is Instruction restart (we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible)

Instruction Restart

- Consider an instruction that could access several different locations
  - block move(256 bytes)



  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap? We cannot restart the instruction

➢ 2 methods

- ➢ The microcode computes and attempts to access both ends of both blocks. The move can then take place
- ➢ The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory so that the instruction can be repeated.

Performance of Demand Paging

- Stages in Demand Paging (worse case)

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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. 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

Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time

- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- **Effective Access Time (EAT)**

$$EAT = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system

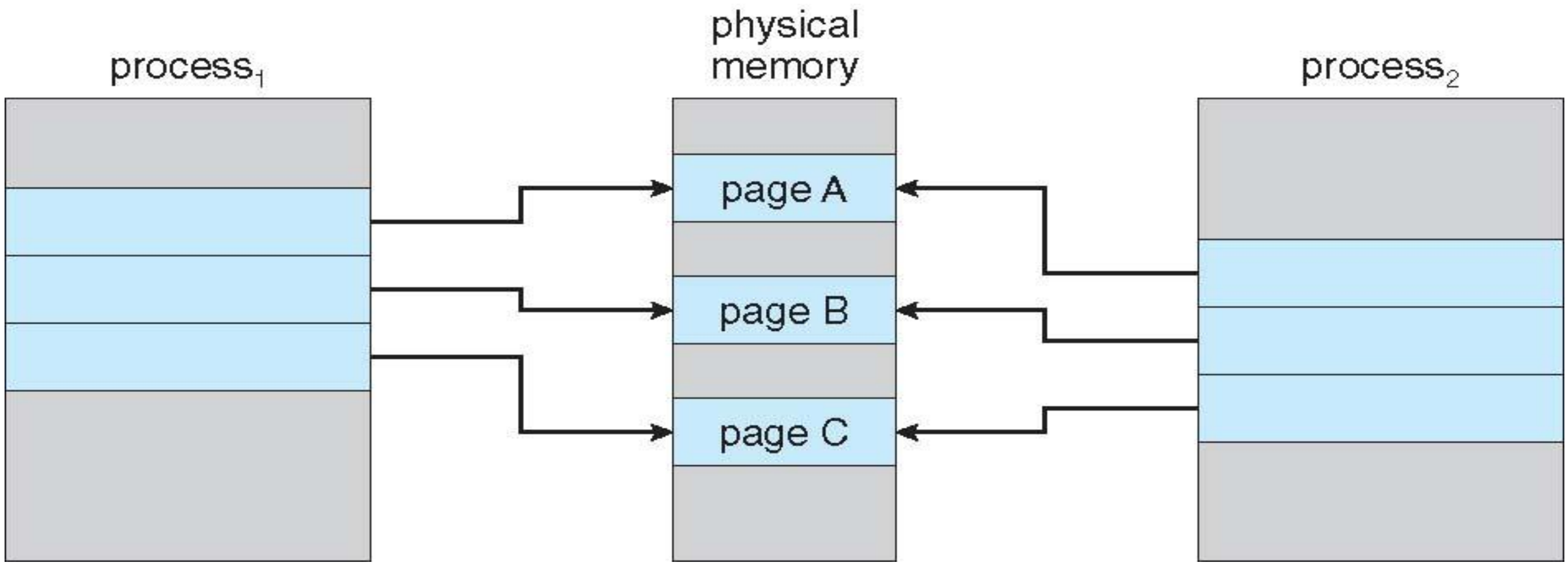
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

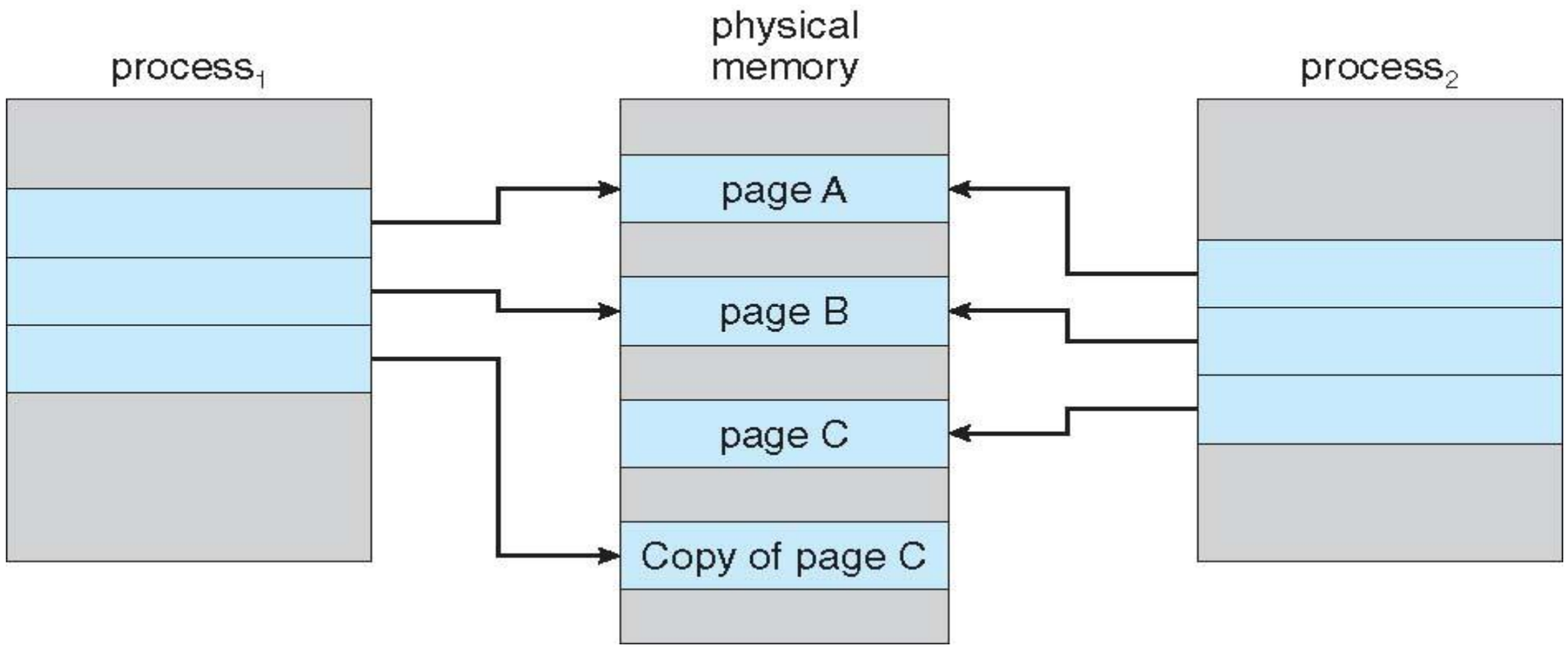
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



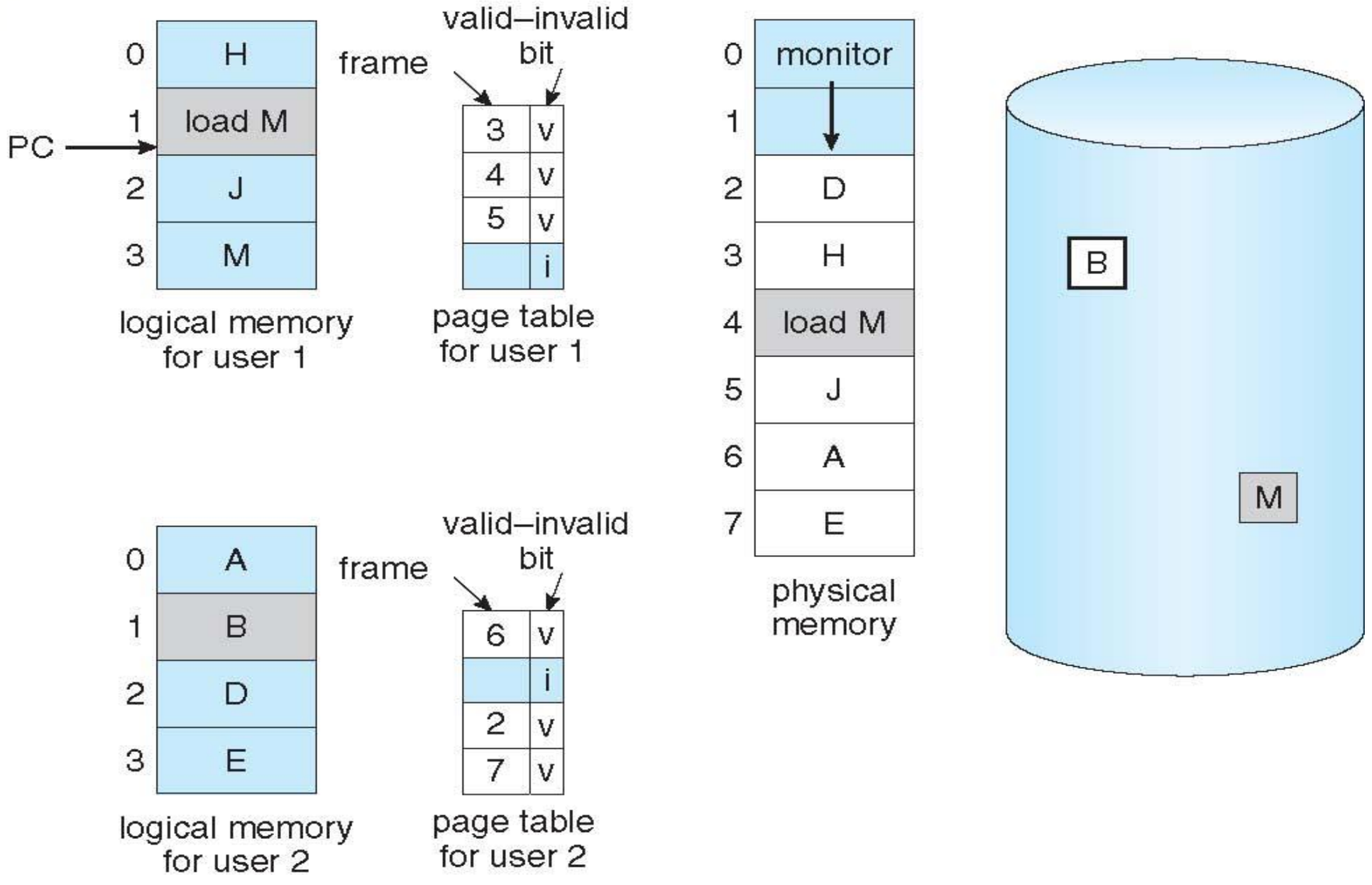
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement.
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement

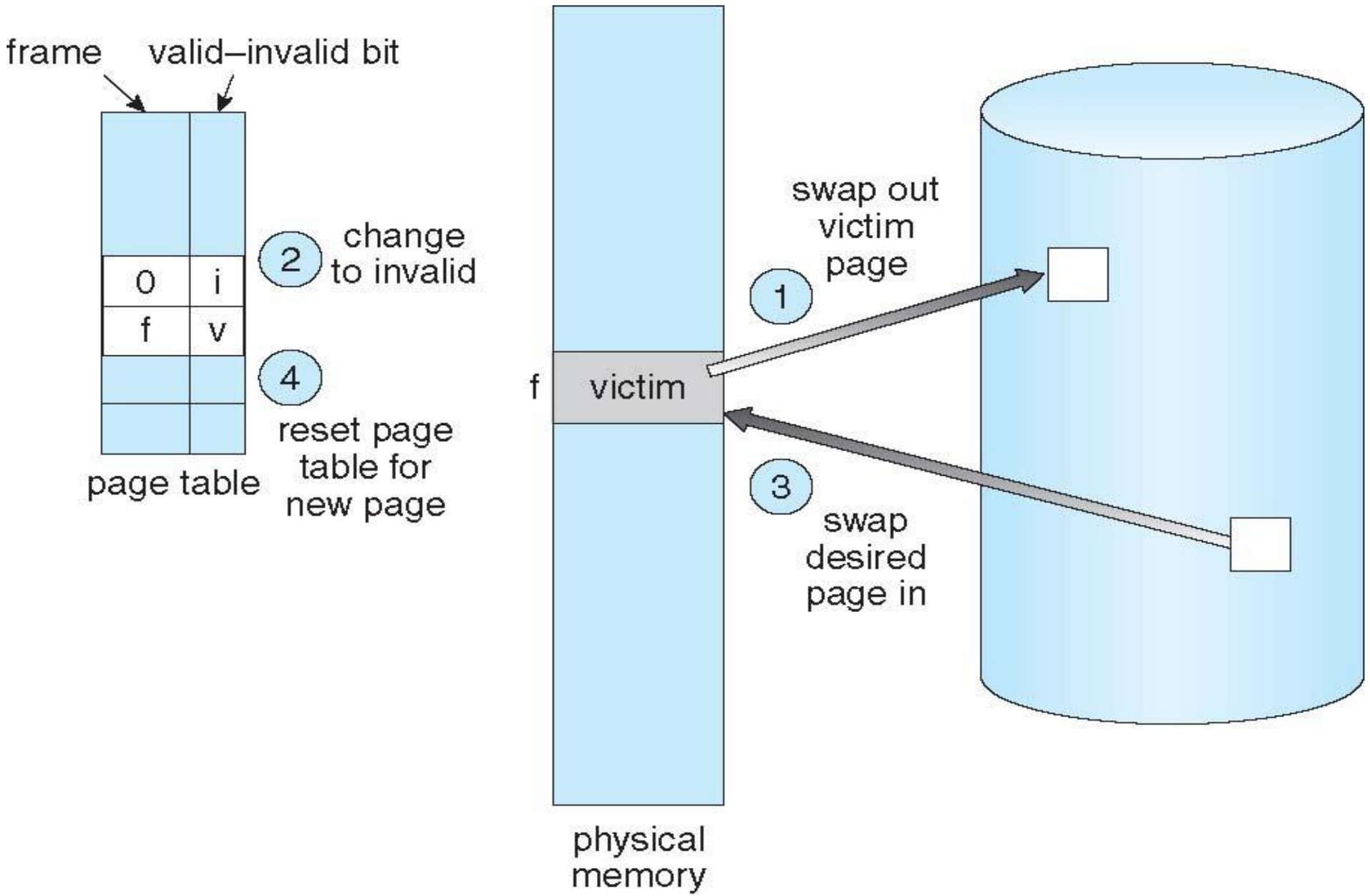


Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



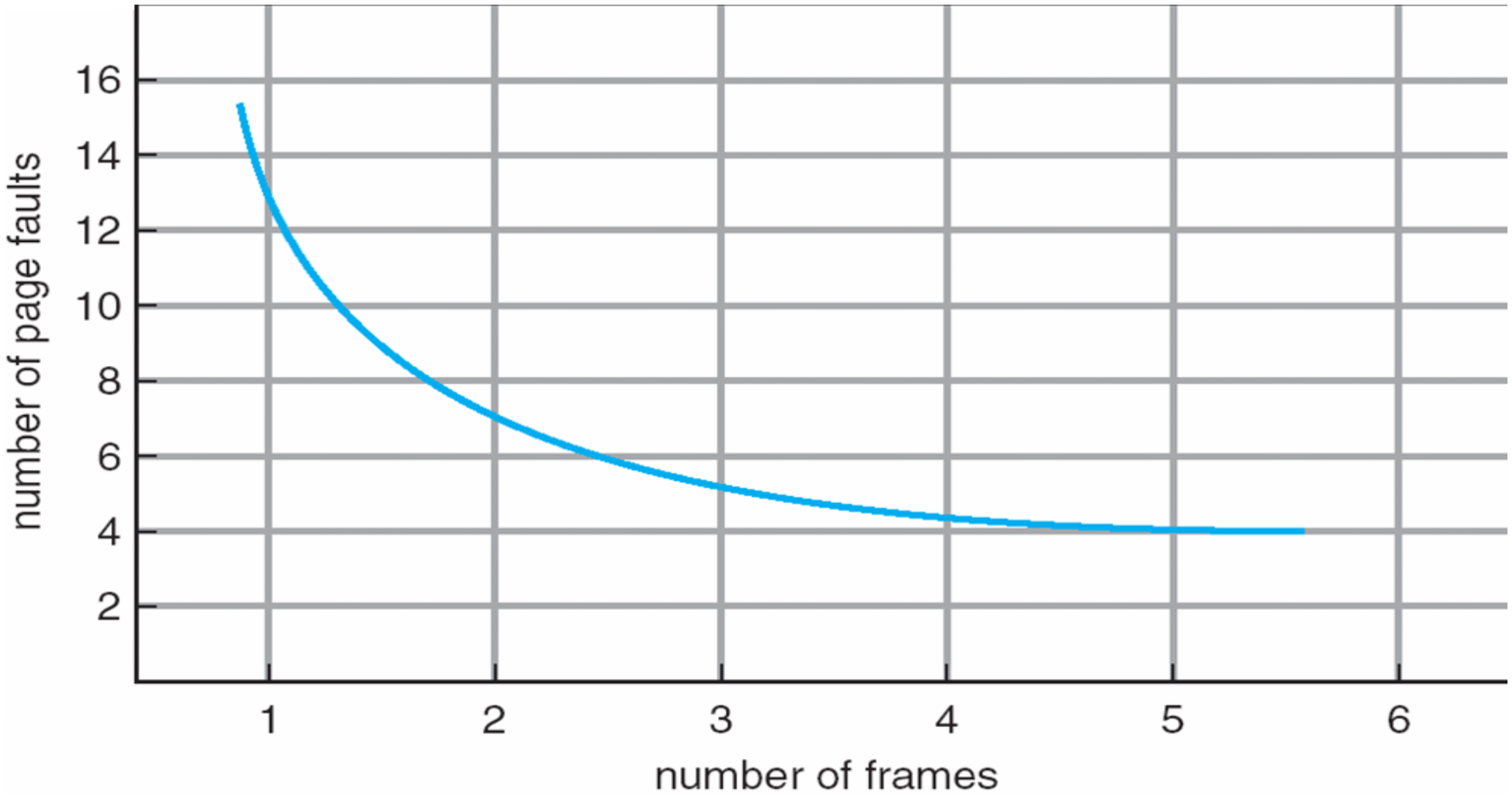
Page and Frame Replacement Algorithms

- **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
  - Results depend on number of frames available

- In all our examples, the **reference string** of referenced page numbers is

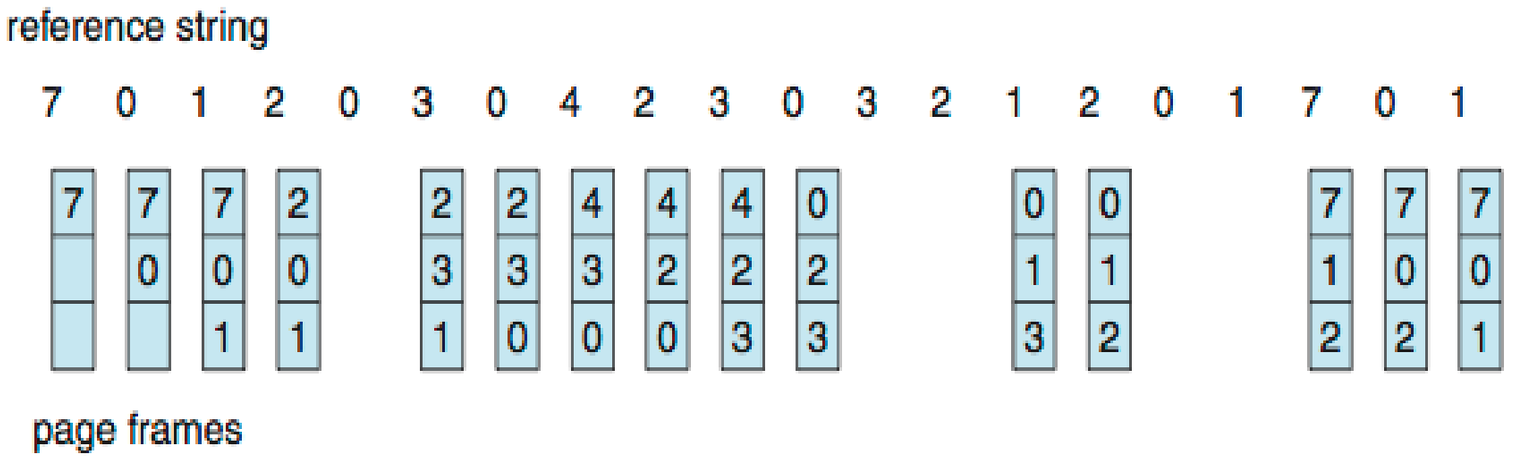
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



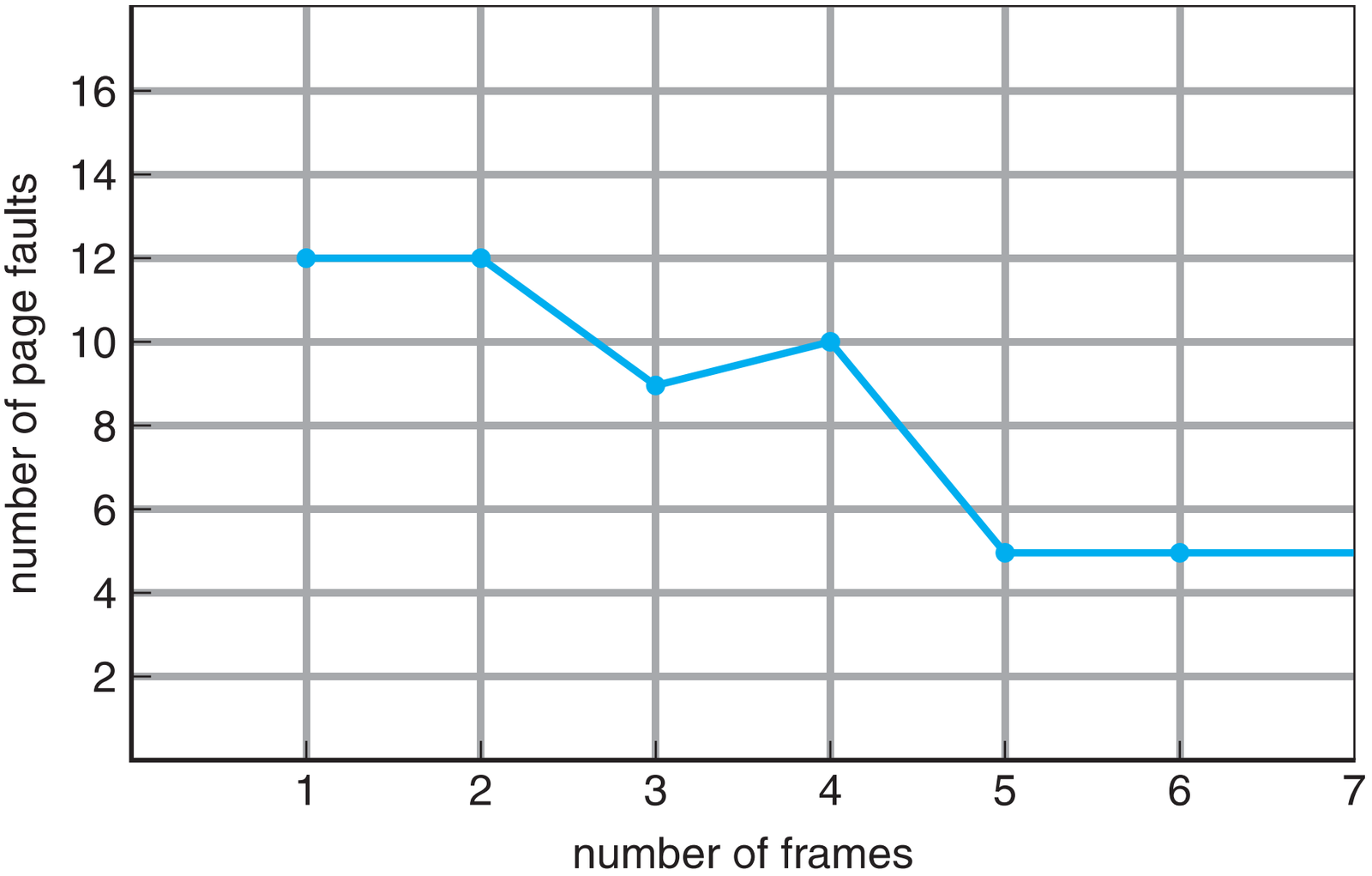
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

FIFO Illustrating Belady's Anomaly

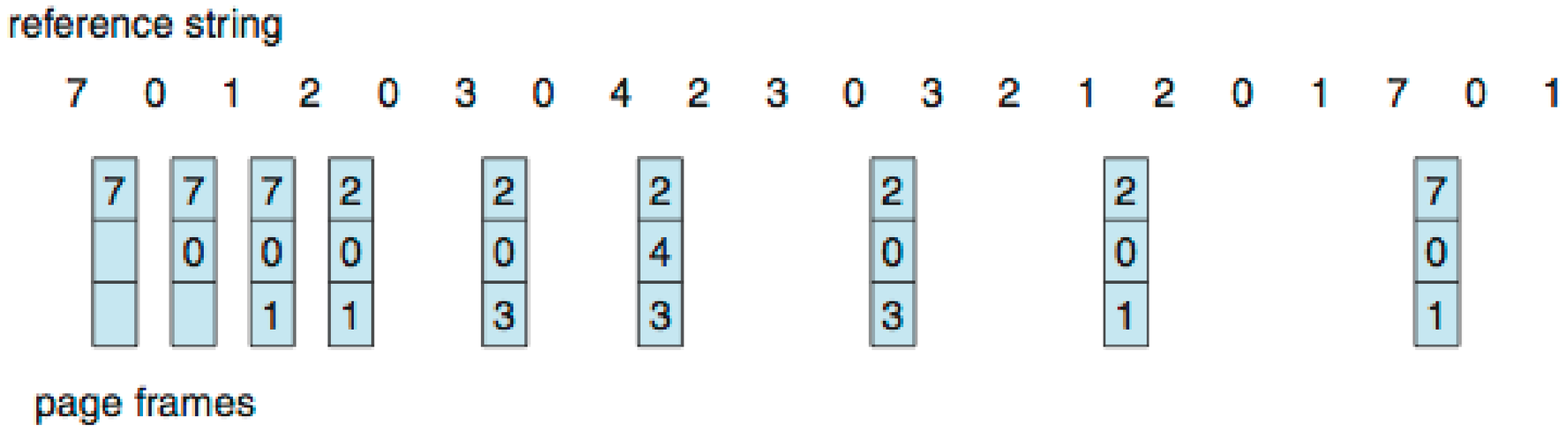


Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example

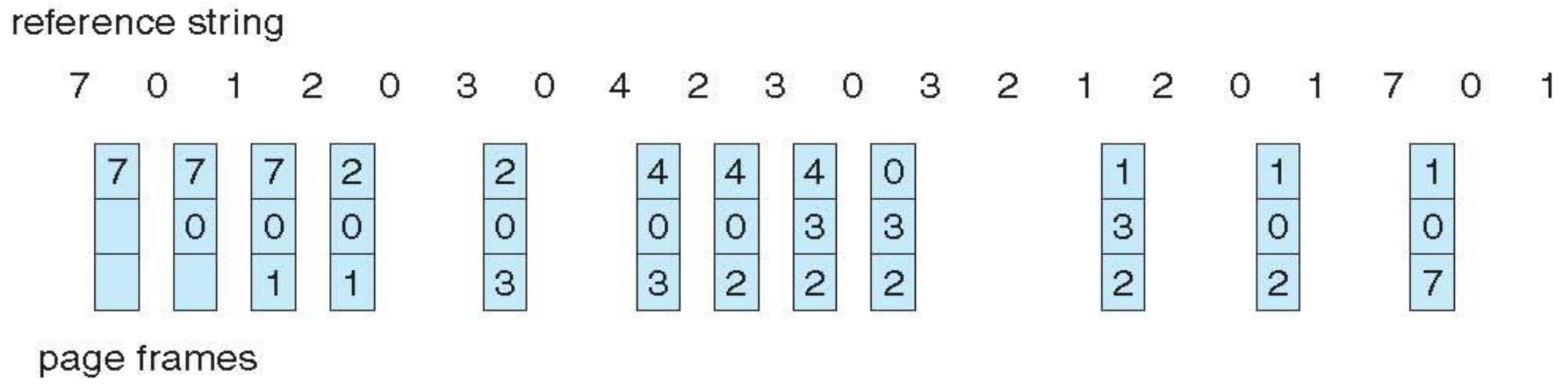
  How do you know this?
  - Can't read the future

- Used for measuring how well your algorithm performs



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



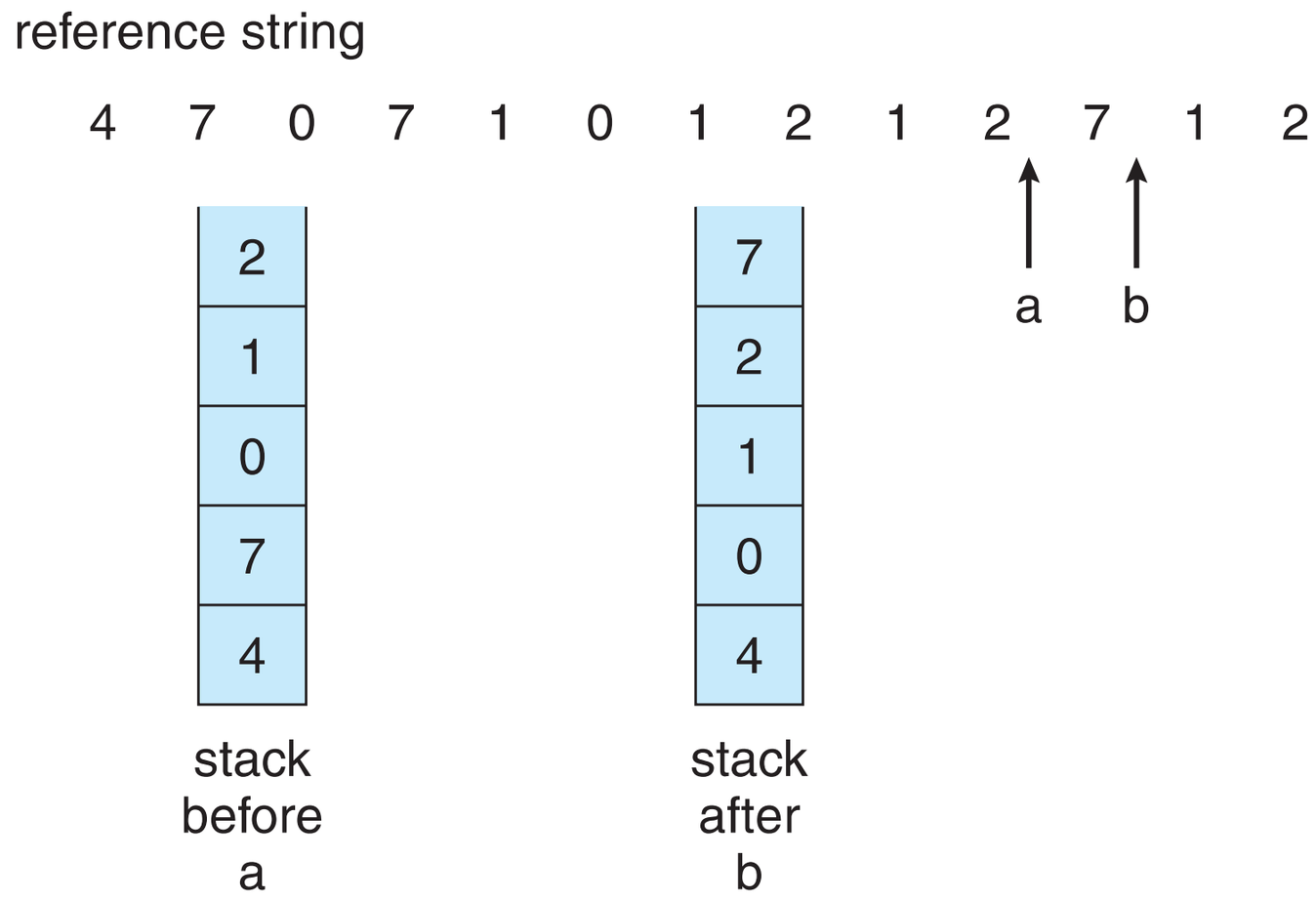
- Page falults=12

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

LRU Algorithm (Cont.)

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- **Stack implementation**
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

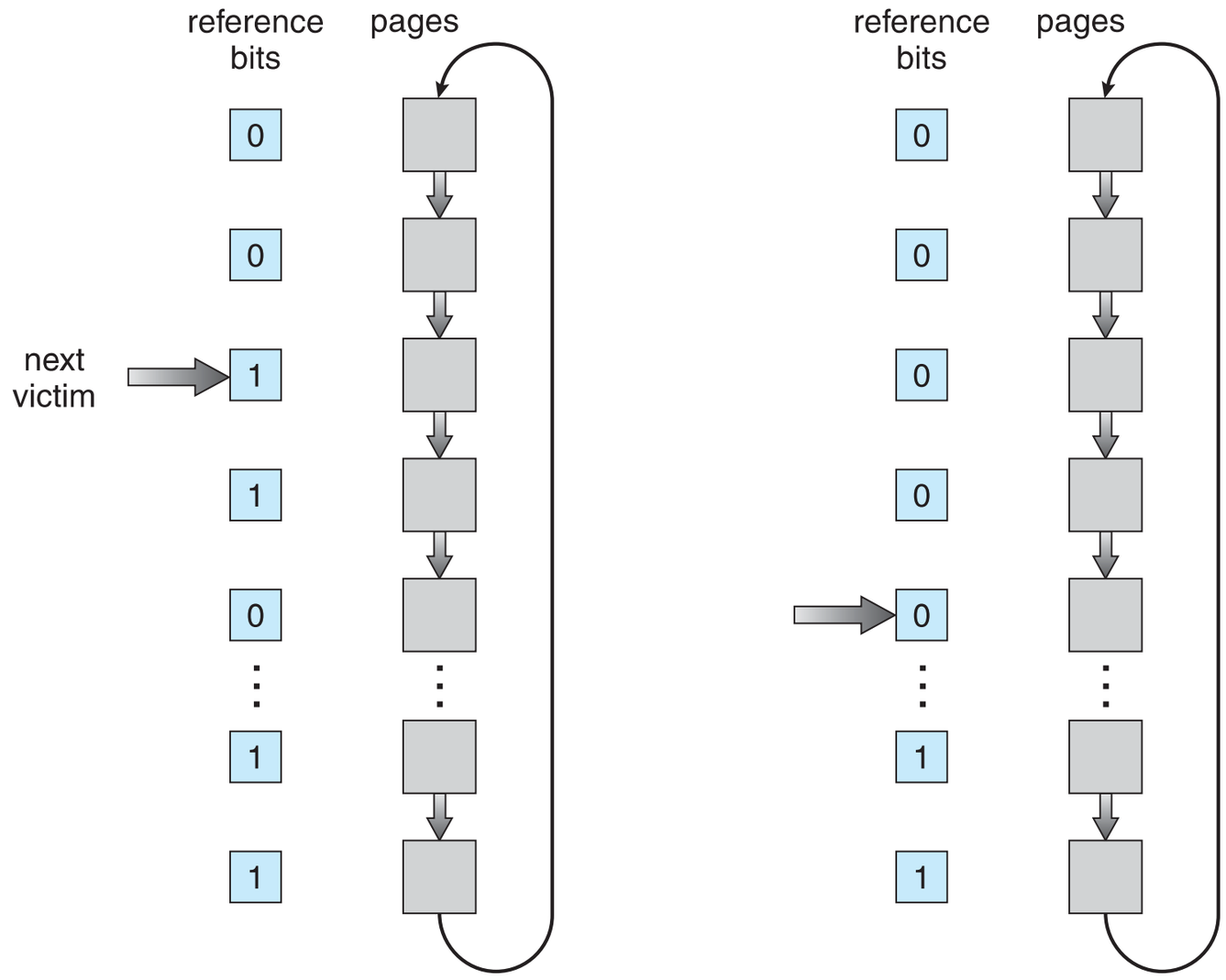
Use Of A Stack to Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm(dock algorithm)**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- **Take ordered pair (reference, modify)**

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- **Keep a pool of free frames, always**
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- **Possibly, keep list of modified pages**
  - When backing store otherwise idle, write pages there and set to non-dirty
- **Possibly, keep free frame contents intact and note what is in them**
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

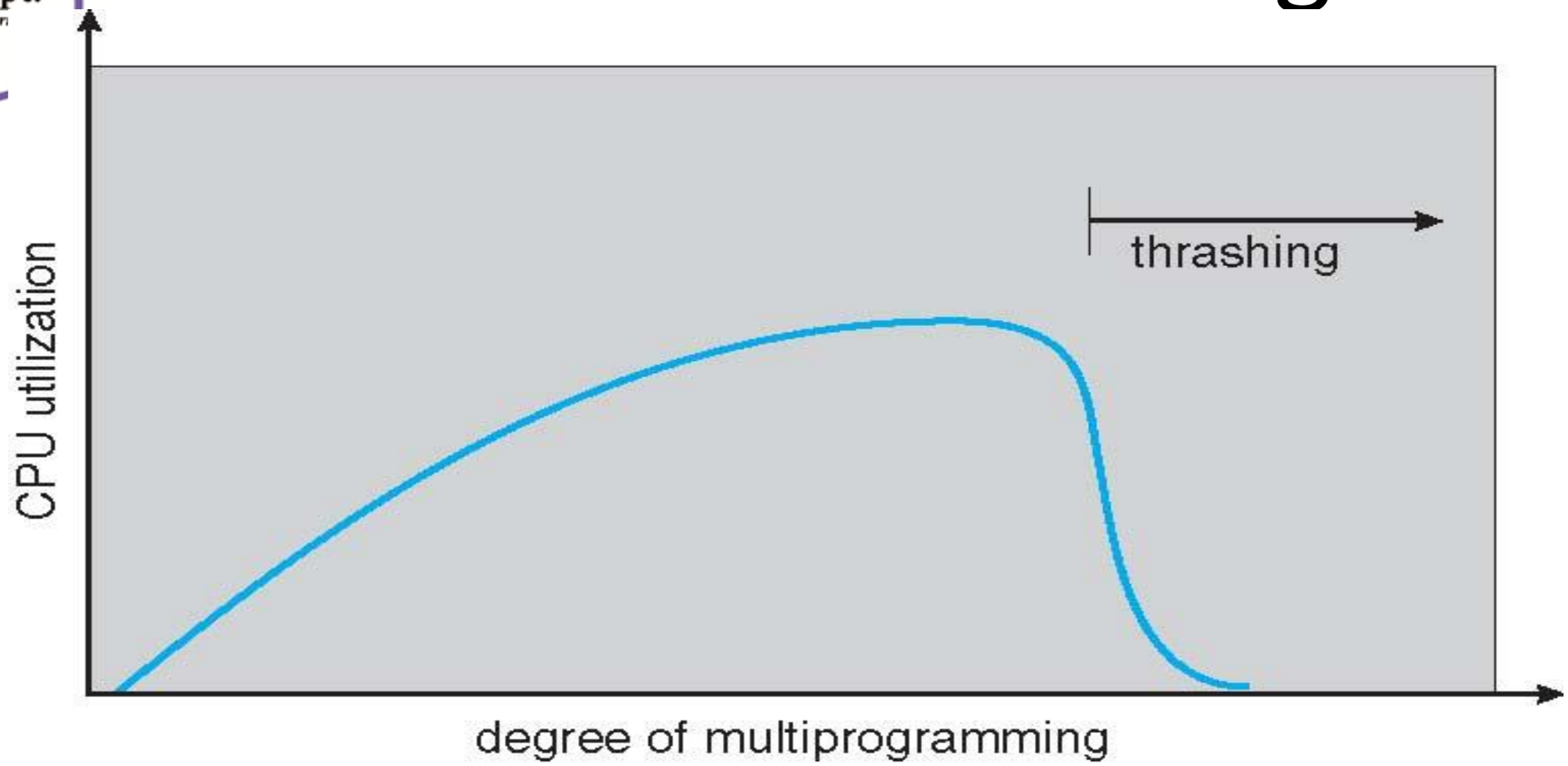
Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

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
- **A process is thrashing if it is spending more time paging than executing.**
- **This high paging activity is called thrashing**

Cause of Thrashing



- We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm).
- To prevent thrashing, we must provide a process with as many frames as it needs.

Cause of Thrashing (contd)

- **But how do we know how many frames it "needs'?**
- The working-set strategy starts by looking at how many frames a process is actually using→**Locality model**
  - Process migrates from one locality to another
  - A **locality** is a set of pages that are actively used together
  - Localities may overlap
- For example, when a function is called, it defines a new locality→ mem ref for instruction, local & global var are made
- When we exit the function, the process leaves this locality.

Cause of thrashing(contd..)

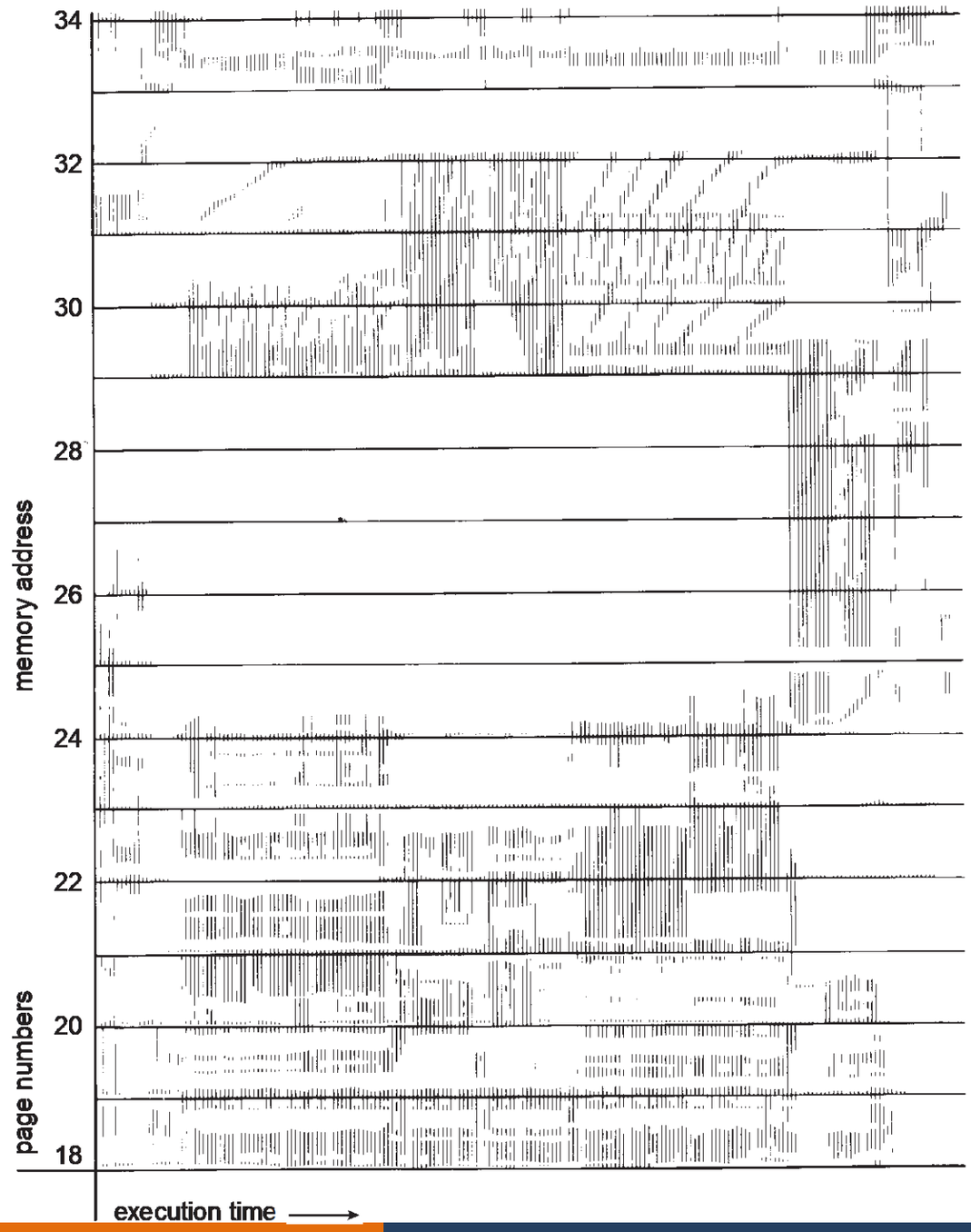
- localities are defined by the program structure and its data structures.
- If accesses to any types of data were random rather than patterned, caching would be useless
- Why does thrashing occur?

  Σ size of locality > total memory size

  – Limit effects by using local or priority page replacement

ISE Dept.
Transform Here

Locality In A Memory-Reference Pattern

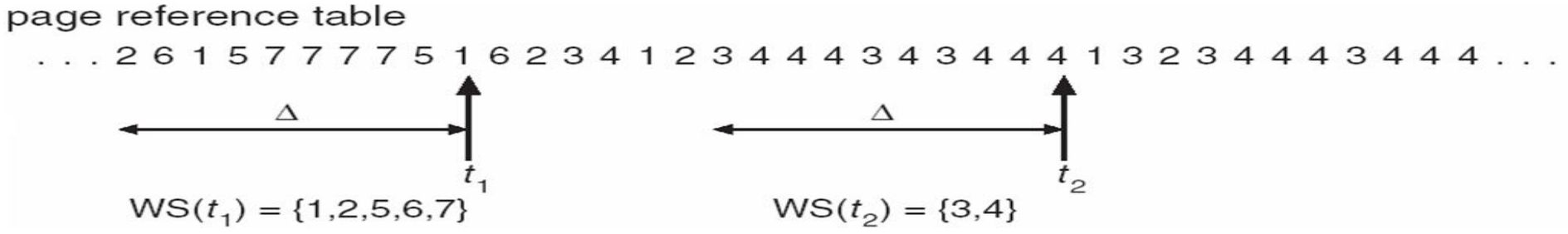


Working-Set Model

- Working-set model is based on the assumption of locality.
- This model uses a parameter, A, to define the working-set window.
- The idea is to examine the most recent A page references.
- If a page is in,active use, it will be in the working set. If it is no longer being used, it will drop from the working set A time units after its last reference.
- working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- **$D = \Sigma\ WSS_i \equiv$ total demand frames**
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes
- If A=10 memory references



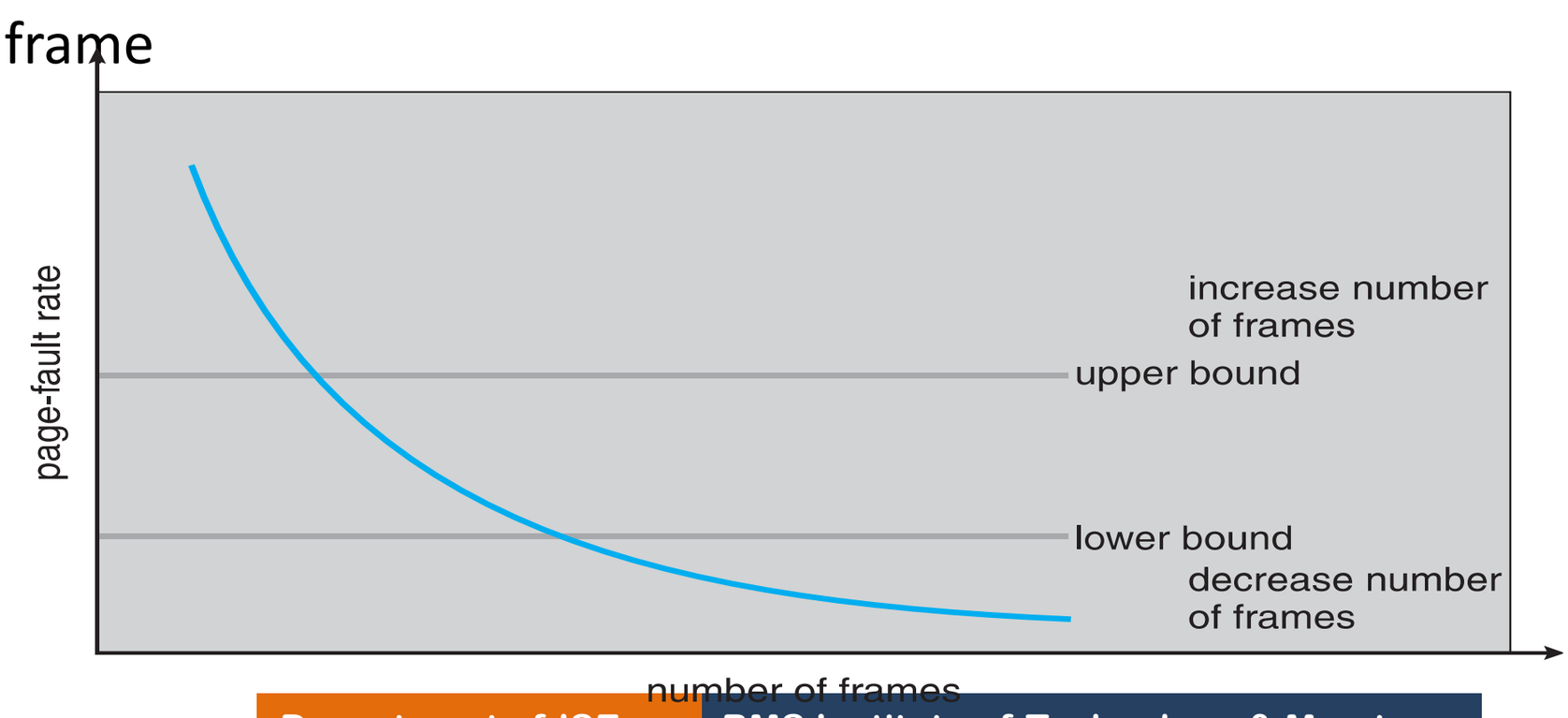
Keeping Track of the Working Set

- The difficulty with the working-set model is keeping track of the working
- set.
- The working-set window is a moving window.
- At each memory reference, a new reference appears at one end and the oldest reference drops off the other end.
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

- Why is this not completely accurate?
  - because we cannot tell where, within an interval of 5,000, a reference occurred
- Improvement = We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (eg 10 bits and interrupt every 1000 time units)
- the cost to service these more
- frequent interrupts will be correspondingly higher.

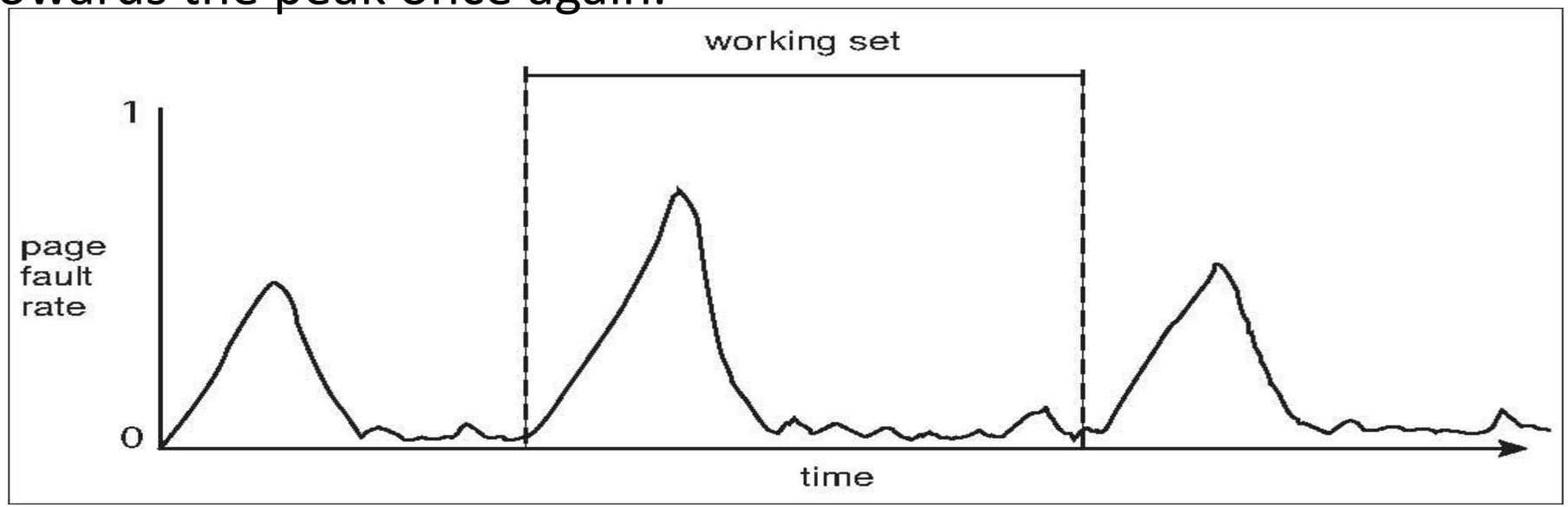
Page-Fault Frequency (PFF)

- More direct approach than WSS
- We can establish upper and lower bounds on the desired page-fault rate
  - If actual rate falls below the lower limit, too low, process loses frame
  - If actual rate exceeds the upper limit, too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time
- A peak in the page fault rate occurs when we begin demand paging a new locality
- When the process moves to the new working set, the page fault rate rises towards the peak once again.



ISE Dept.
Transform Here

End of Chapter

Chapter 11:
File-System Interface

Chapter 11: File-System Interface

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection

Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection

File Concept

- Contiguous logical address space
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider **text file, source file, executable file, object file,numeric data, graphics, images, sound…**

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum Information kept in the directory structure

File Operations

- File is an **abstract data type**
- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- ***Open(F_i)*** – search the directory structure on disk for entry ***F_i***, and move the content of entry to memory
- ***Close (F_i)*** – move the content of entry ***F_i*** in memory to directory structure on disk
- OS uses two levels of internal tables: a per-process table and a system-wide table.

Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

Open File Locking

- Provided by some operating systems and file systems
- File "locks allow one process to lock a file and prevent other processes from gaining access to it
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mandatory or advisory:
  - **Mandatory** – once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. (eg: windows)
  - **Advisory** – processes can find status of locks and decide what to do(eg: Unix)

File Types – Name. Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

Access Methods

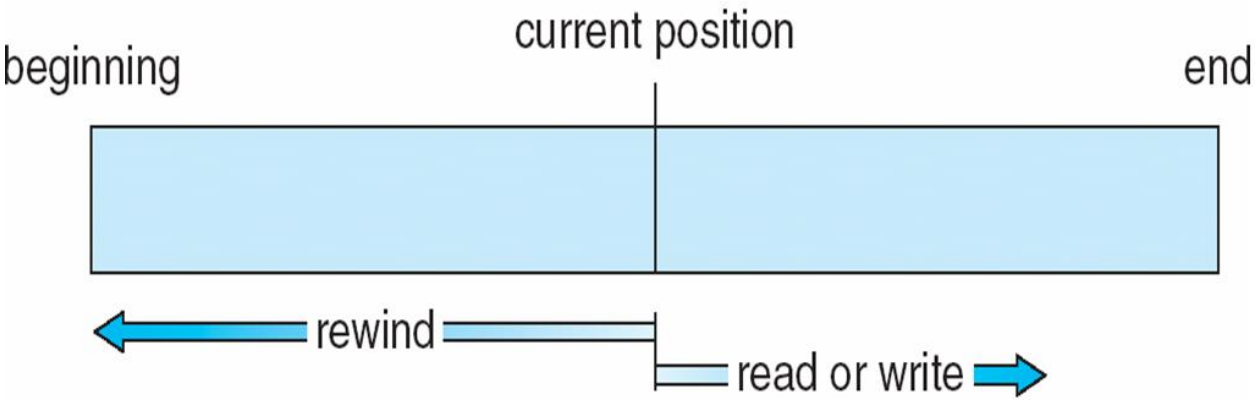
- Sequential Access
- Direct Access(Relative Access)
- Other Access Methods (Indexing)

Access Methods

- **Sequential Access**

```
read next
write next
reset
```

no read after last write
(rewrite)



Eg: text editors and compilers

Depicts tape model

Direct (Relative) access file

Direct Access – file is fixed length logical records

```
read n
write n
position to n
    read next
    write next
rewrite n
```

n = relative block number

- For direct access, the file is viewed as a numbered sequence of blocks or records.
- We may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

- **Relative block numbers** allow OS to decide where file should be placed
  See allocation problem in Ch 12

  - Eg: Databases
  - Depicts Disk model

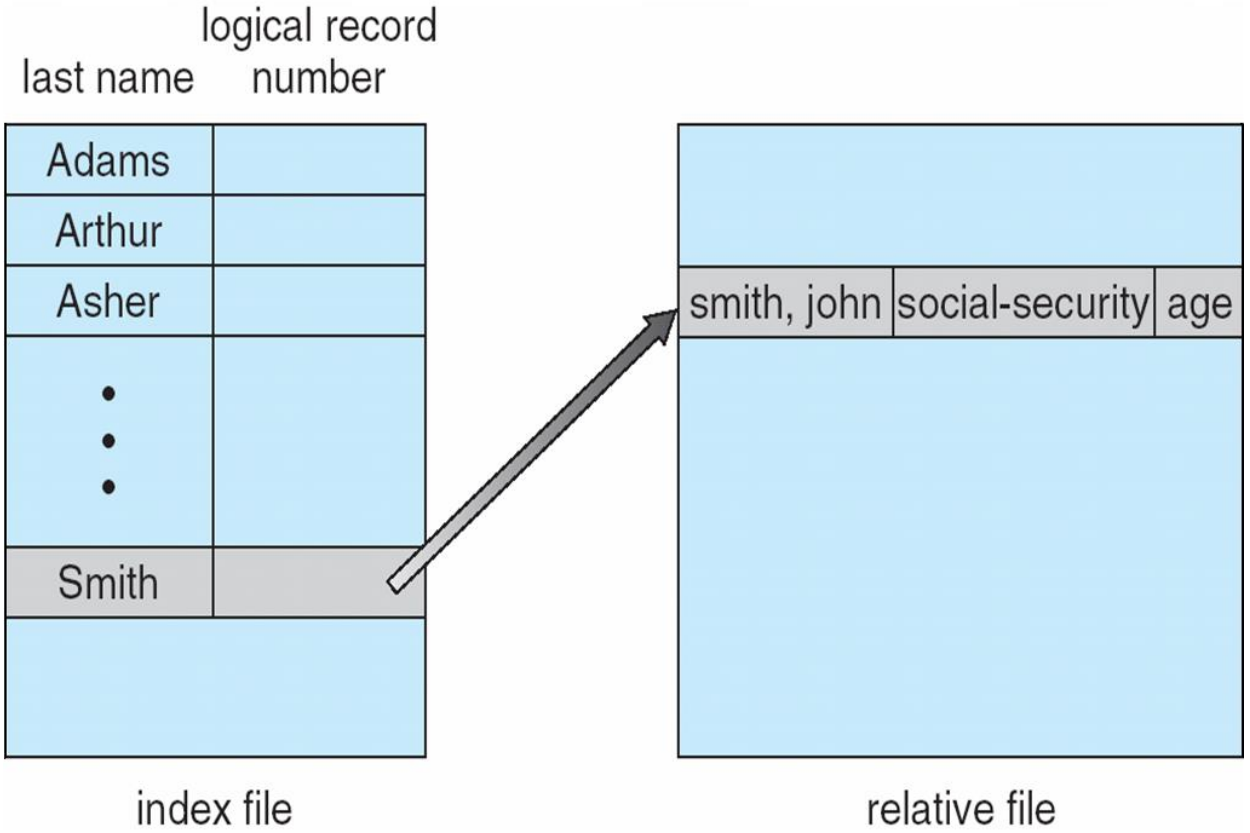
Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Other Access Methods

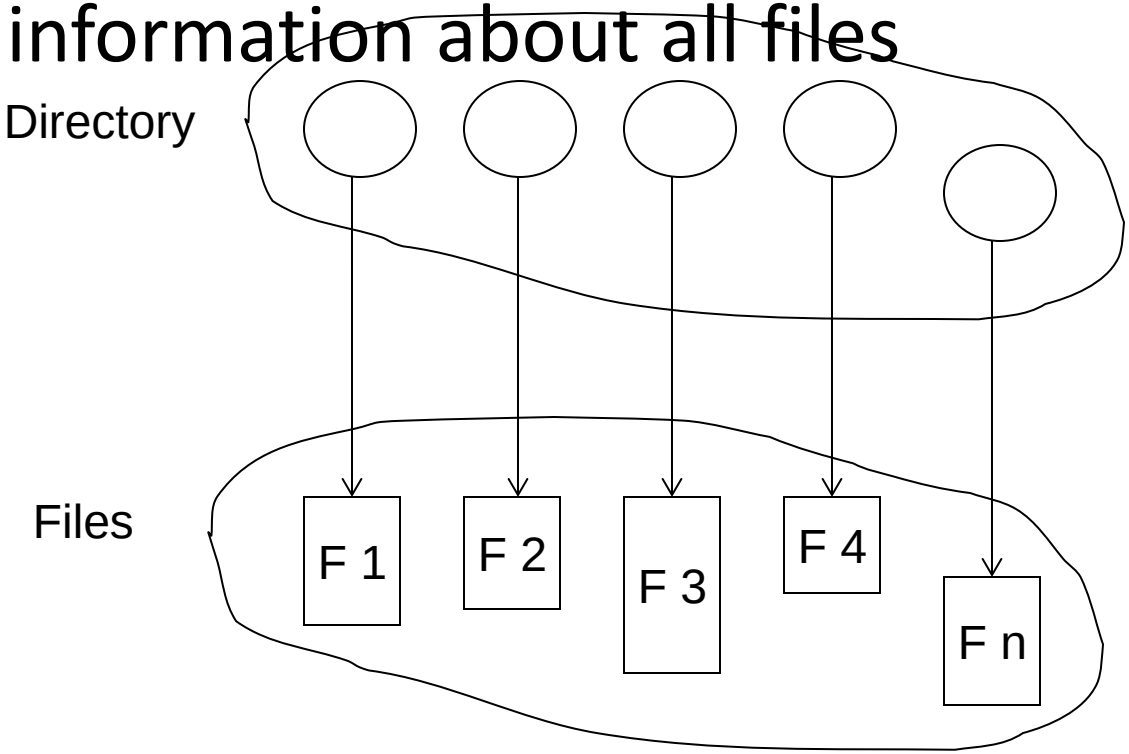
- Can be built on top of base methods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

Example of Index and Relative Files



Directory Structure

- A collection of nodes containing information about all files

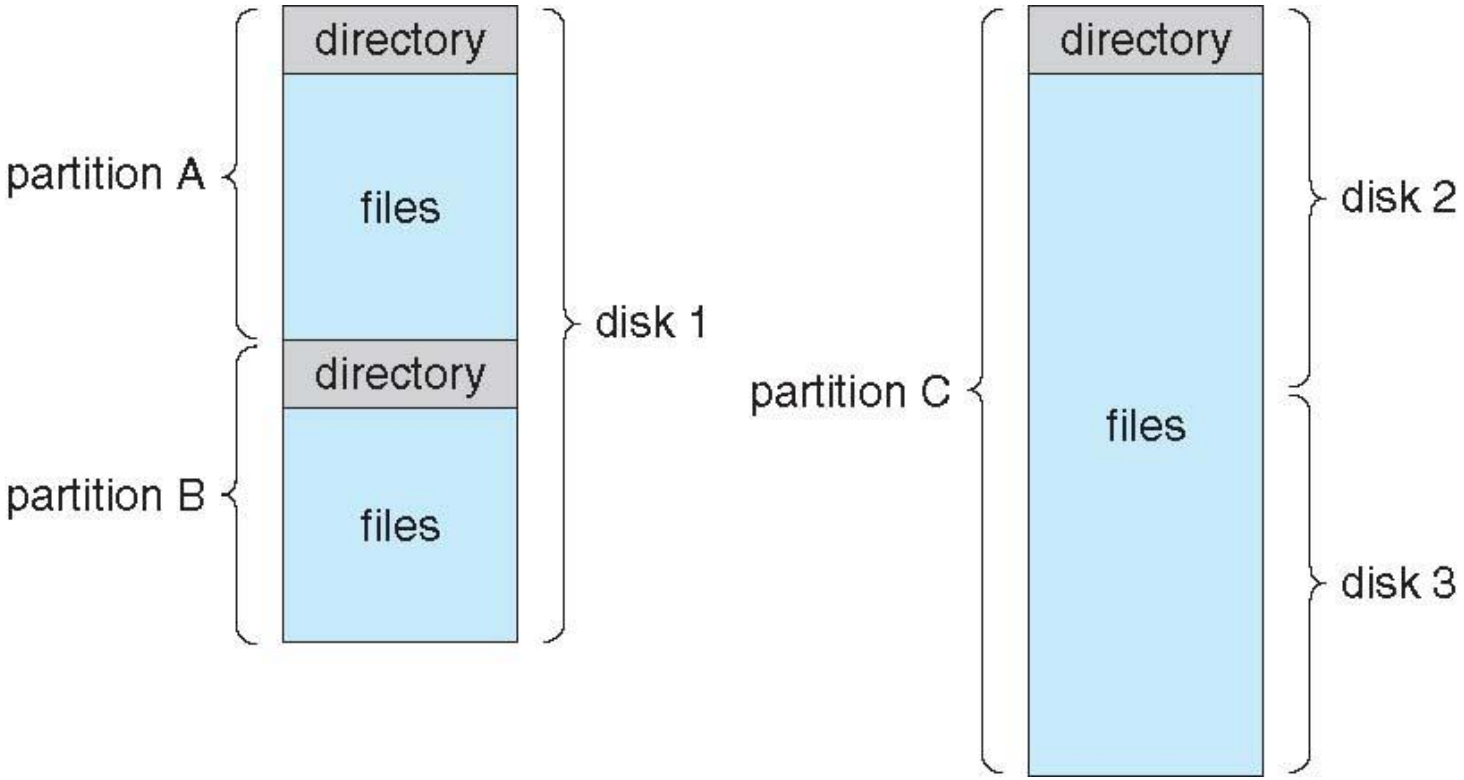


Both the directory structure and the files reside on disk

Disk Structure

- Disk can be subdivided into **partitions**
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
- As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

A Typical File-system Organization



Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Directory Organization

The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

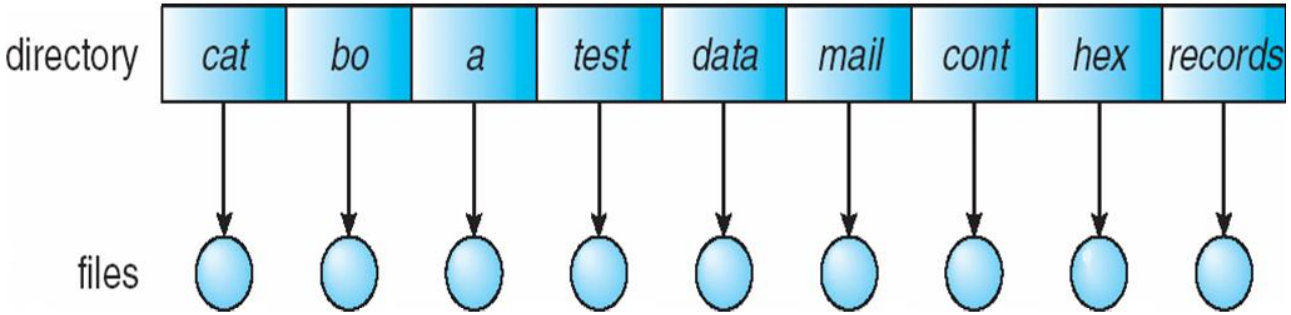
Directory structures

- Single level Directory
- Two level Directory
- Tree structured directories
- Acyclic- Graph Directories
- General Graph Directory

Single-Level Directory

- Simplest directory structure
- All the files are there in the same directory
- A single directory for all users

Adv→(easy to understand and support)

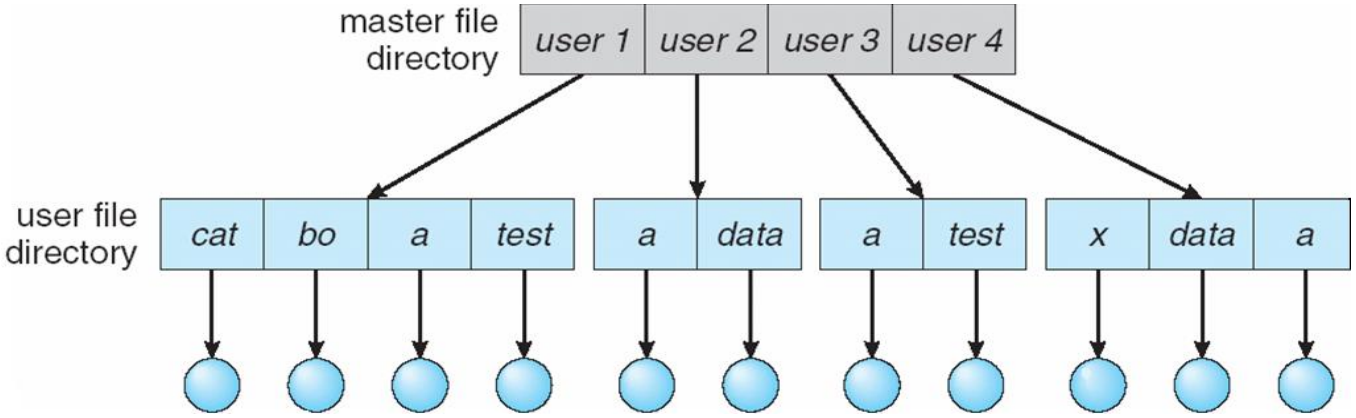


Disadvantages

- Naming problem (if file /user increases)
- Grouping problem (content is same but name is diff)
- Length limit (DOS-11, Unix-255char)

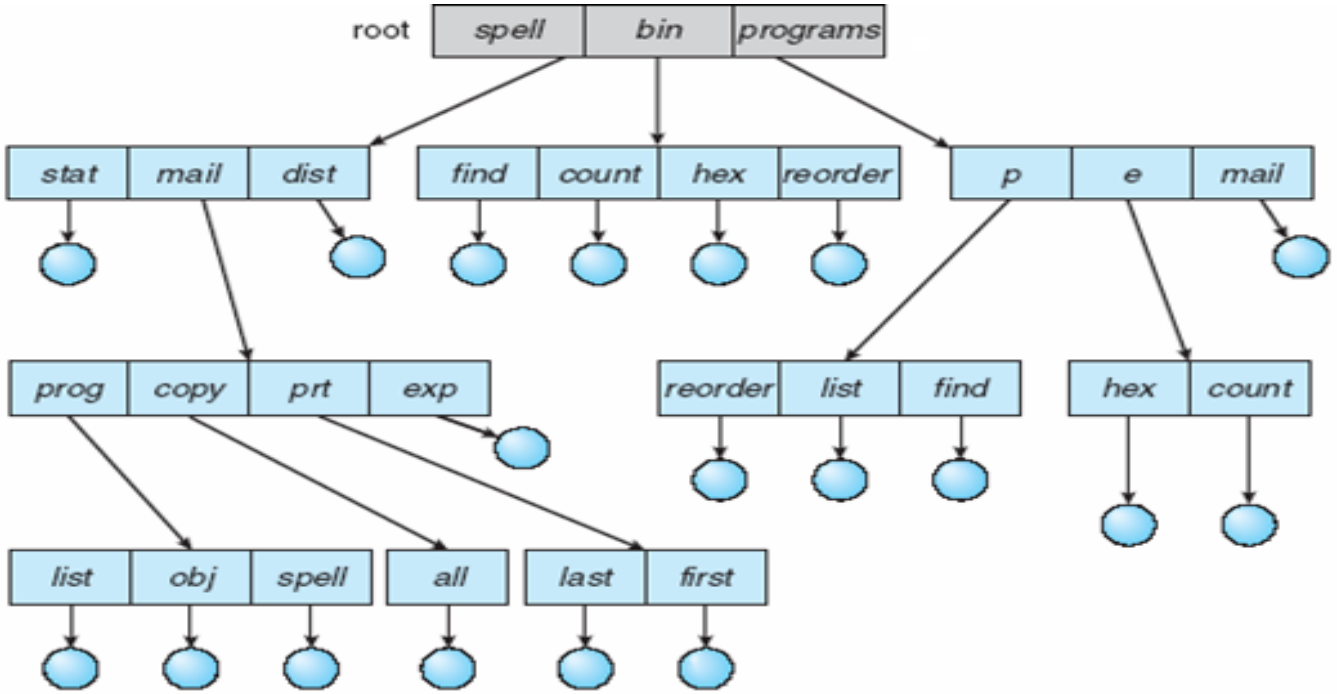
Two-Level Directory

- Separate directory for each user(UFD)



- Isolation
  - Can have the same file name for different user but do not allow cooperative task
- Tree /inverted tree of height 2
  - Path name(username, file name)
- Addition syntax needed to specify the volume
  - (C:\User2\test) u:[sst.jdeck]login.com;1
  - Efficient searching
  - Special directory for system files
- No grouping capability

Tree-Structured Directories



- This generalization allows user to create their own subdirectories
- All dir has same internal format, each bit to directory
  - File(0)/subdirectory(1)
- Spl sys call for create/delete/rename directories

Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`

Tree-Structured Directories (Cont)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

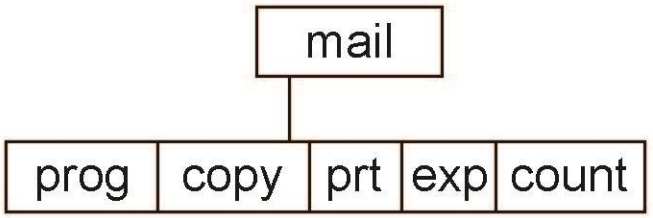
```
rm <file-name>
```

- Creating a new subdirectory is done in current directory

```
mkdir <dir-name>
```

Example: if in current directory **/mail**

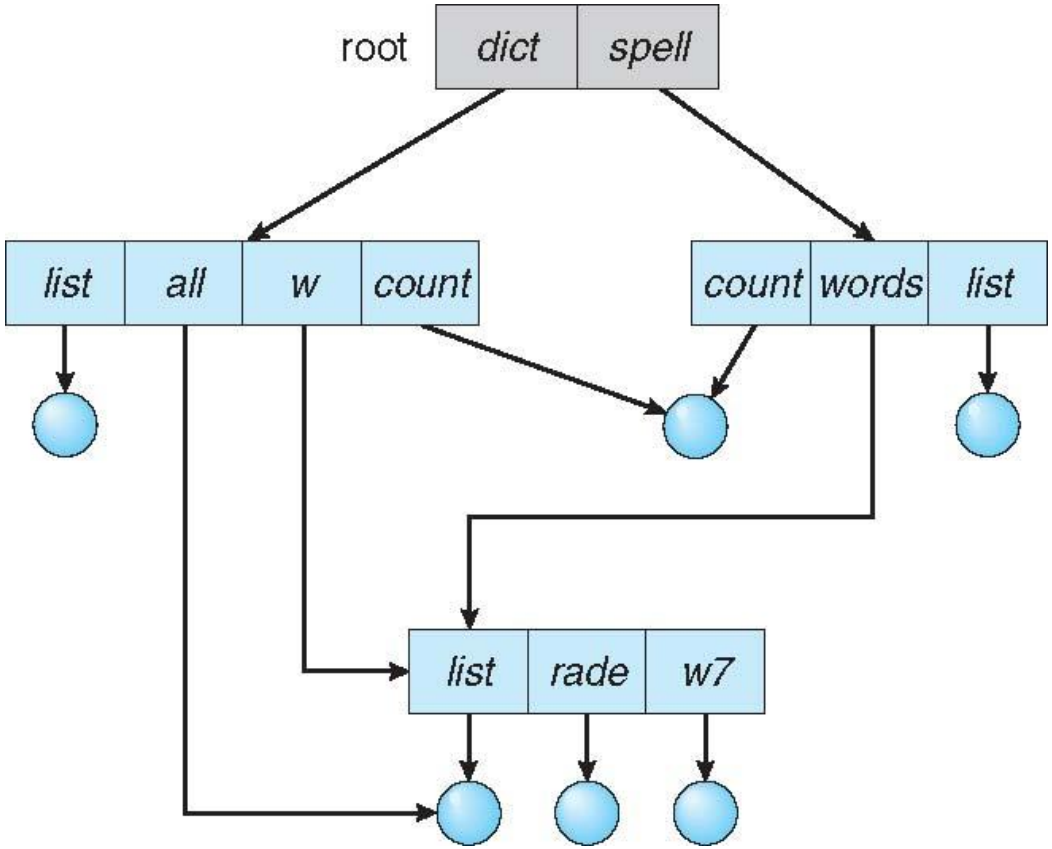
```
mkdir count
```



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files



Acyclic-Graph Directories (Cont.)

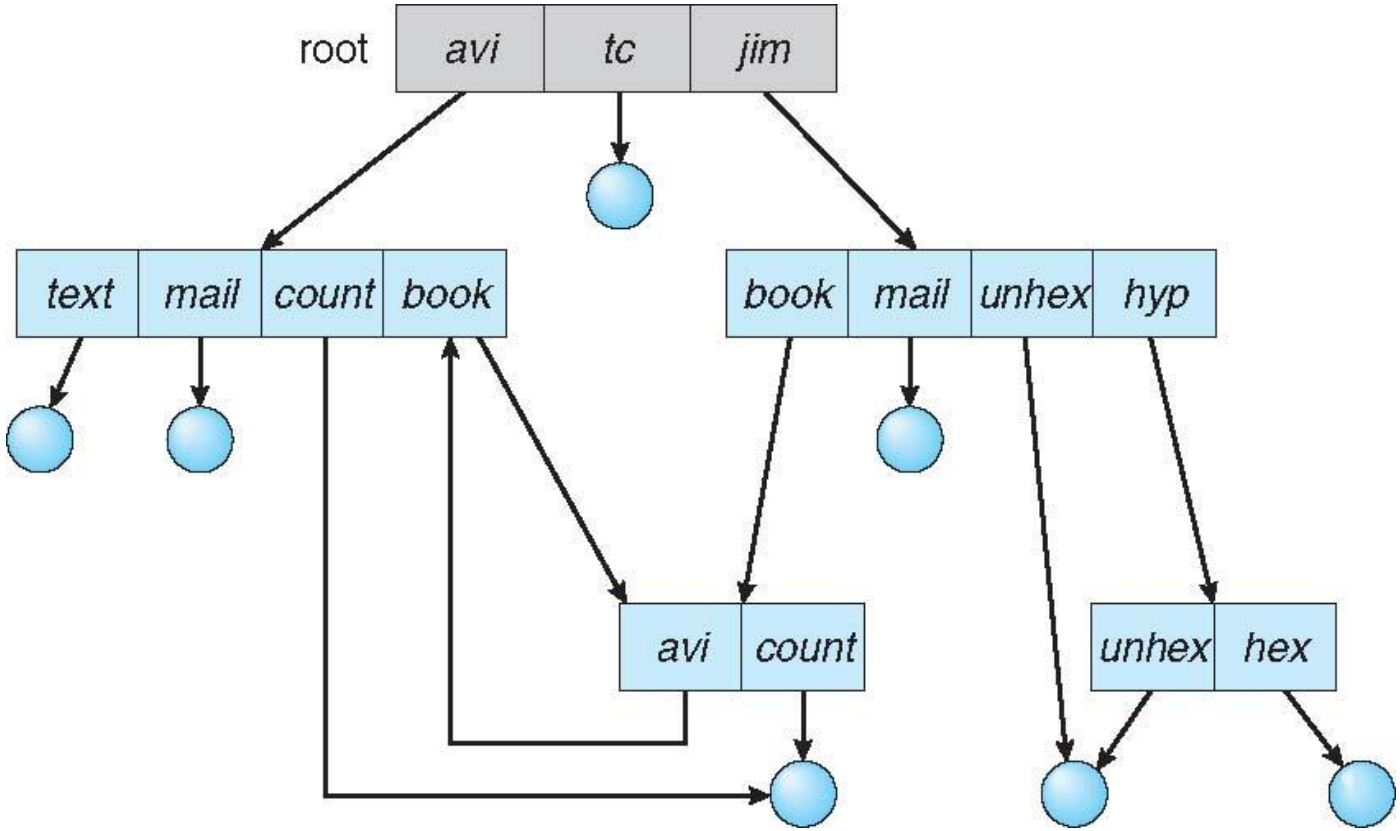
New directory entry type

- **Link** – another name (pointer) to an existing file
- **Resolve the link** – follow pointer to locate the file

Disadvantages

- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer
  - Use links
  - Preserve file until all references are deleted (hardlink count=0)

General Graph Directory

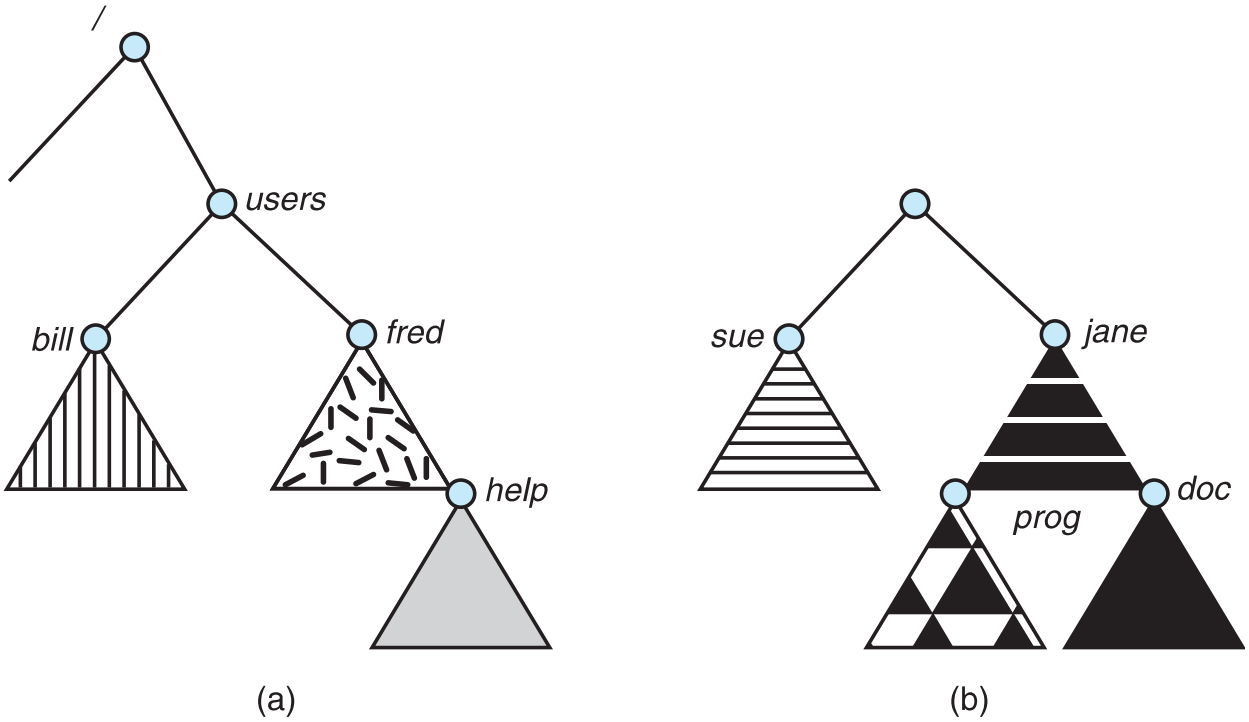


General Graph Directory (Cont.)

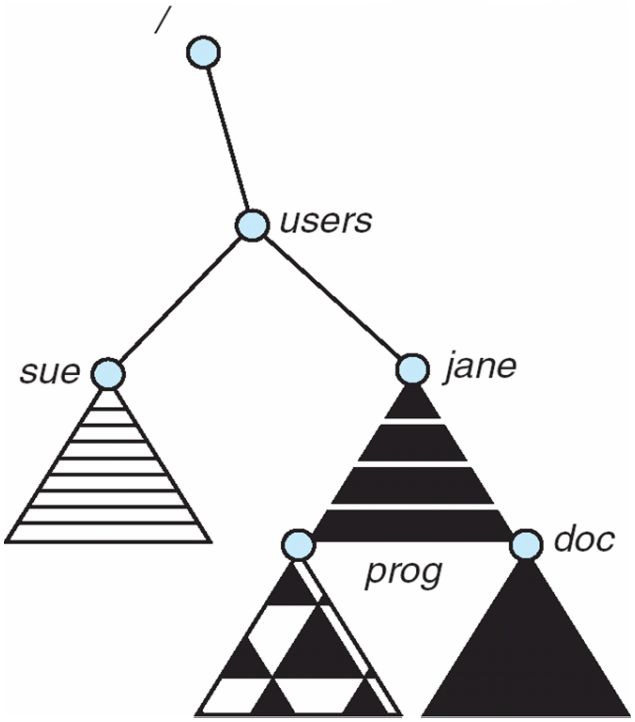
- Cycles
  - Limit the search to avoid searching in loop
  - Deleting file
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e., Fig. 11-11(b)) is mounted at a **mount point**



Mount Point



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
    Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated(use of encrypted keys)
- **Distrubuted Information Systems**
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** (common internet File sys)is standard Windows protocol
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing

File Sharing – Failure Modes

- All file systems have failure modes
- Local file sys→ failure of disk, corruption of directory structures or other non-user data, called **metadata,** disk controller failure, cable failure
  - host to crash and an error condition to be displayed, and human intervention will be required to repair the damage
- Remote file systems add new failure modes, due to network failure, server failure, the network can be interrupted between two hosts.
  - Some networks have built-in resiliency, including multiple paths between hosts, many do not.

- *EG: Consider a client in the midst of using a remote file system. It has files open from the remote host; Suddenly, the remote file system is no longer reachable*
- Most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantic

- Specify how multiple users are to access a shared file simultaneously
  - Similar to Ch 5 process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems)
  - Andrew File System (AFS) implemented complex remote file sharing semantics
  - The series of accesses between the open( ) and close () operations makes up a **file session**.
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - Sharing file pointer to allow multiple users to read and write concurrently

- **AFS has session semantics**
  - Writes only visible to sessions starting after the file is closed.
  - Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes
- In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.
- In AFS semantics a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

Immutable-Shared-Files Semantics

- Unique approach →Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has **two key properties**:
  - Its name may not be reused, and its contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined (read-only).

Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups

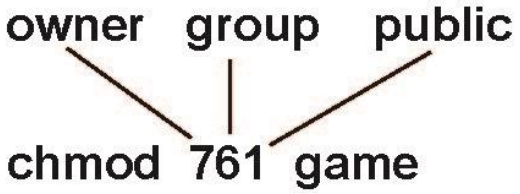
- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

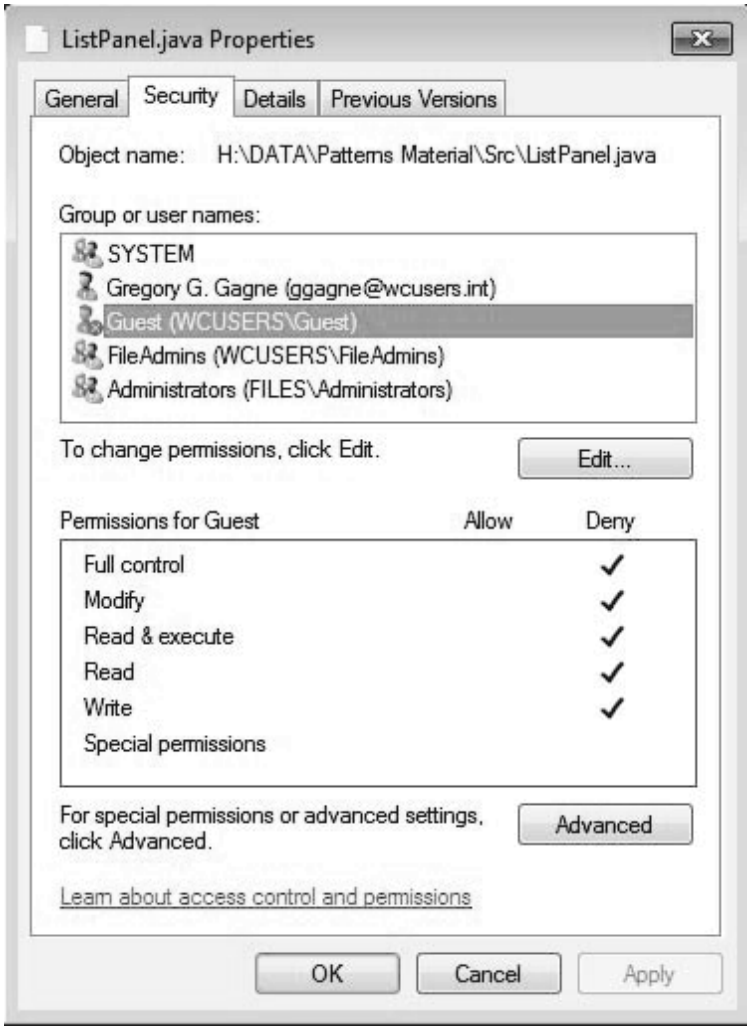
- Ask manager to create a group (unique name), say G, and add some users to the group.

```
chgrp      G      game
```

- For a particular file (say *game*) or subdirectory, define an appropriate access.



Windows 7 Access-Control List Management



A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

End of Chapter 11

Chapter 10: File System Implementation

Chapter 10: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management

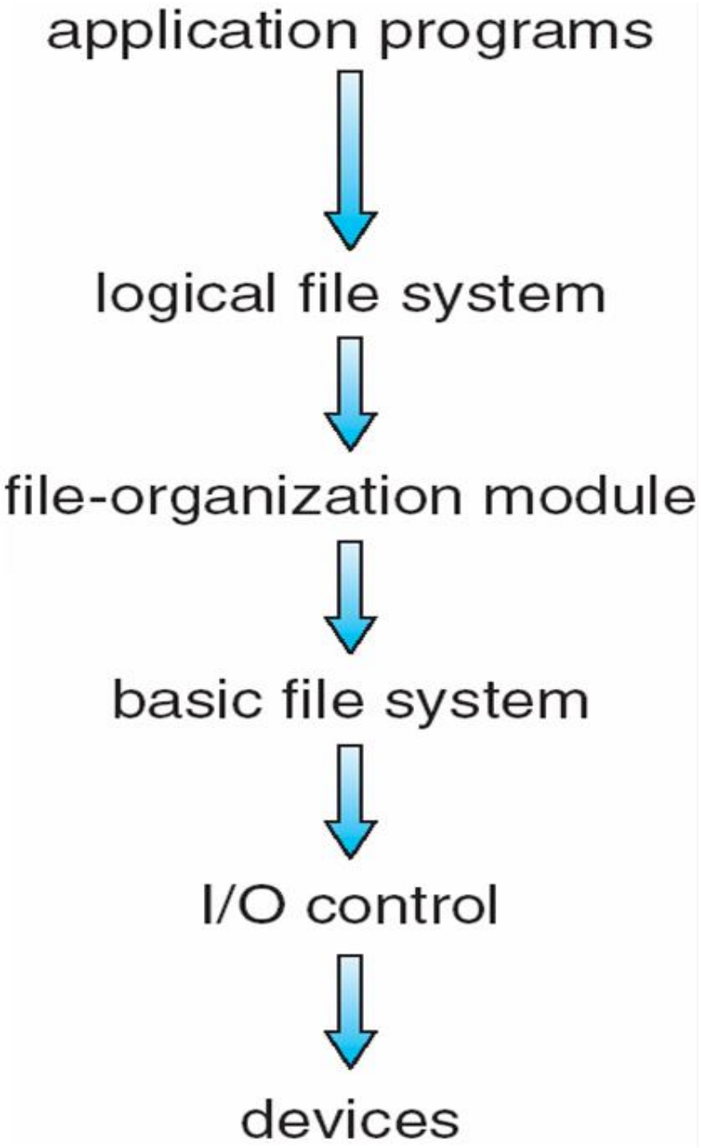
Objectives

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

File-System Structure

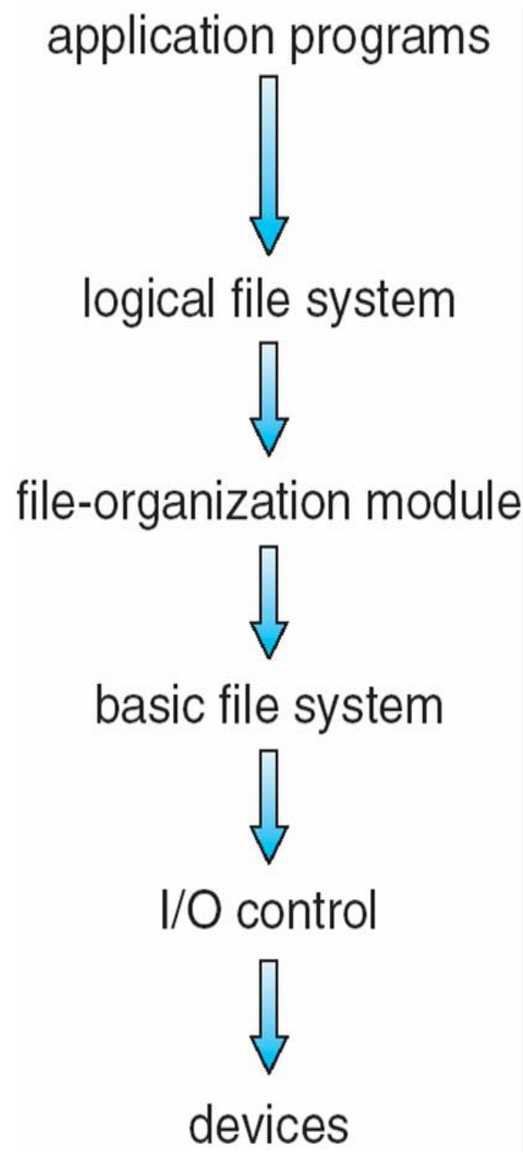
- File structure→Logical storage unit, Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- A File system poses two quite different design problems.
  1. how the file system should look to the user.
  2. Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
  - **Basic file system** given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data
  - **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation



File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in Unix)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

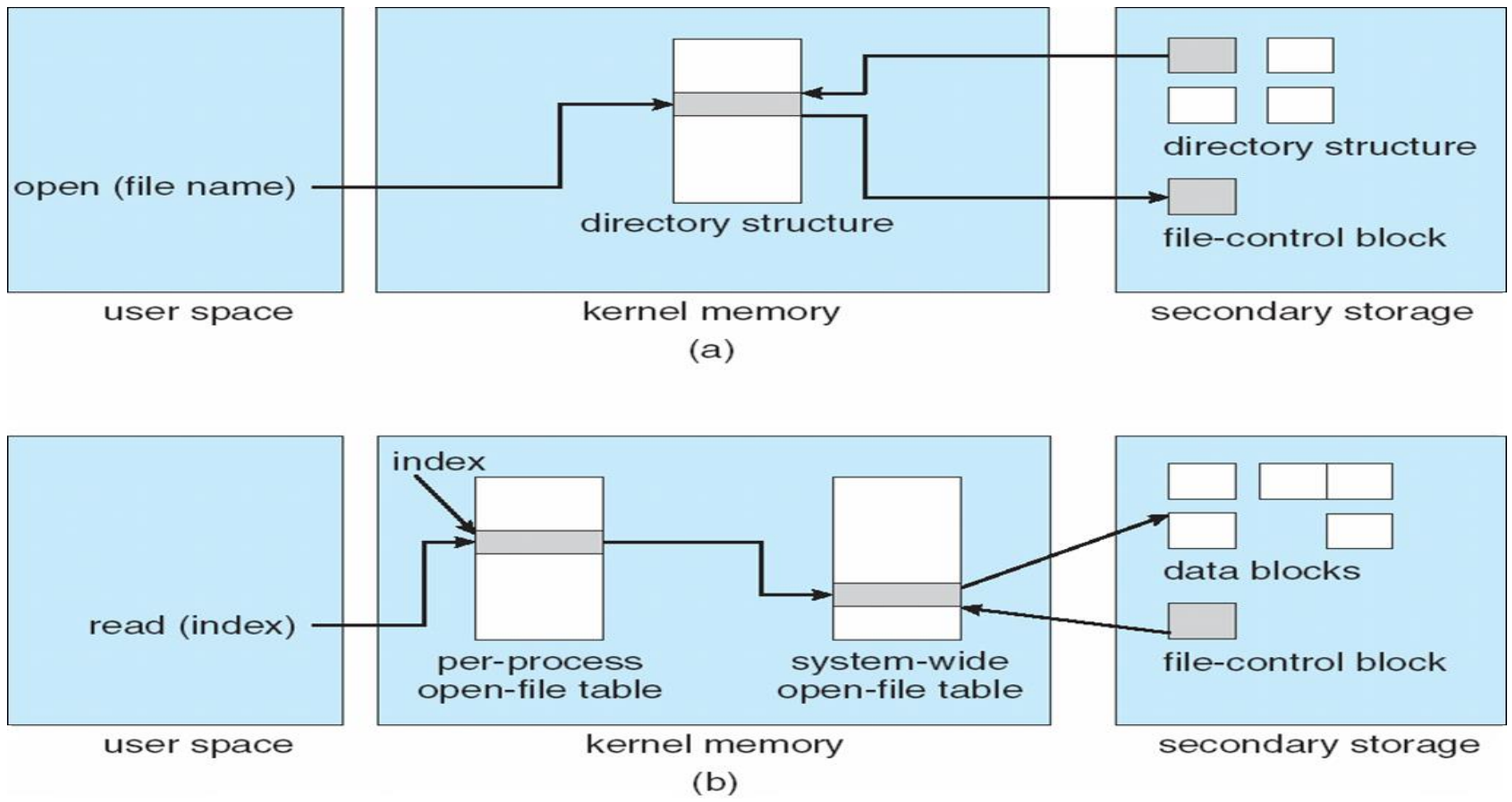
- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume(UFS→boot block, NTFS→ partition boot sector)
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - UFS→Names and inode numbers, NTFS→master file table
- Per-file **File Control Block (FCB)** contains many details about the file
  - Inode number, permissions, size, dates(UFS→ inode)
  - NTFS stores into in master file table using relational DB structures

A Typical File Control Block

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

In-Memory File System Structures

- In-memory Mount table contains information about each mounted volume.
- in-memory directory-structure cache holds the directory information of recently accessed directories.



- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

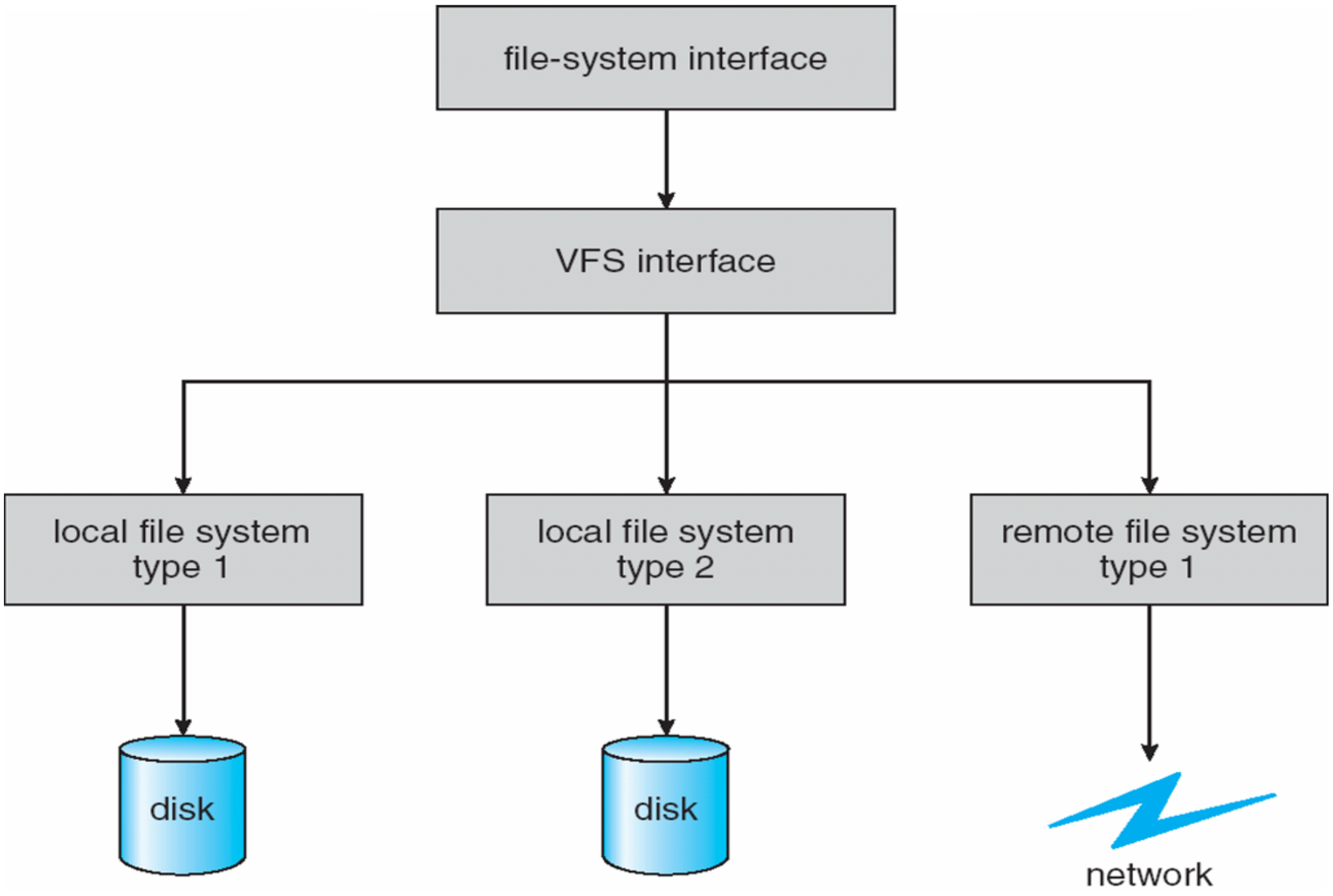
Partitions and Mounting

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

Schematic View of Virtual 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

Directory Implementation

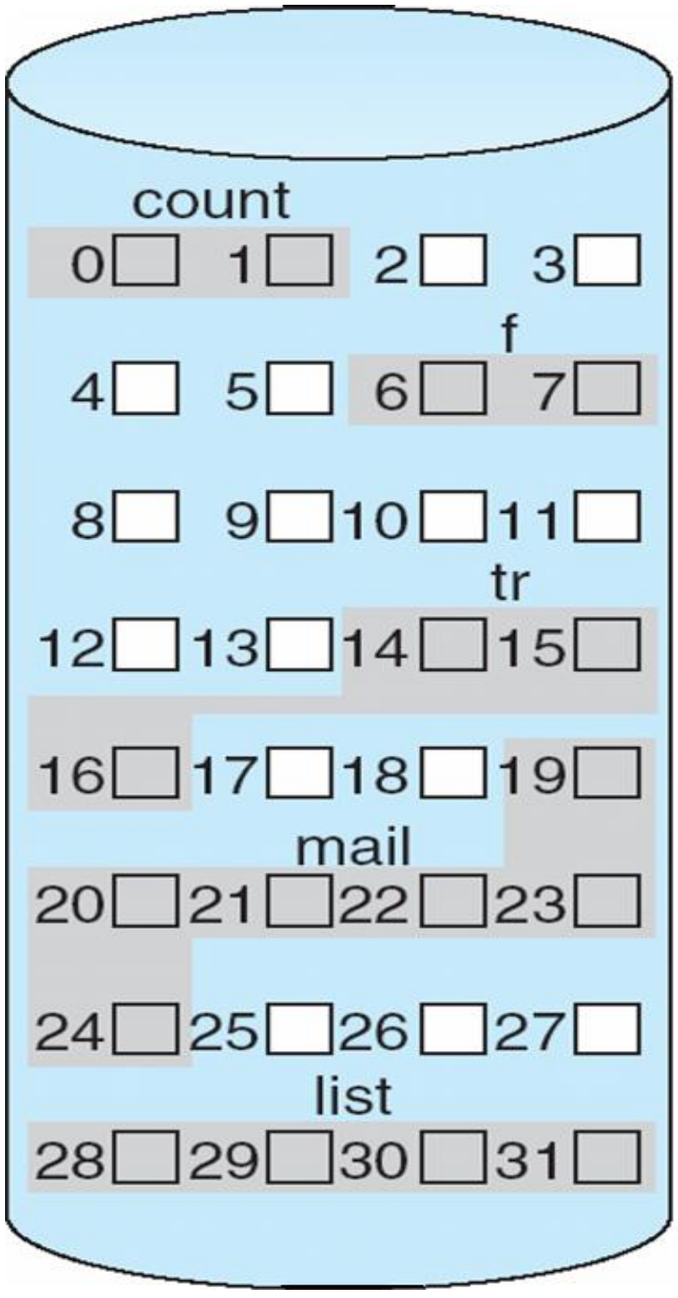
- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree

- **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 of Disk Space



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme

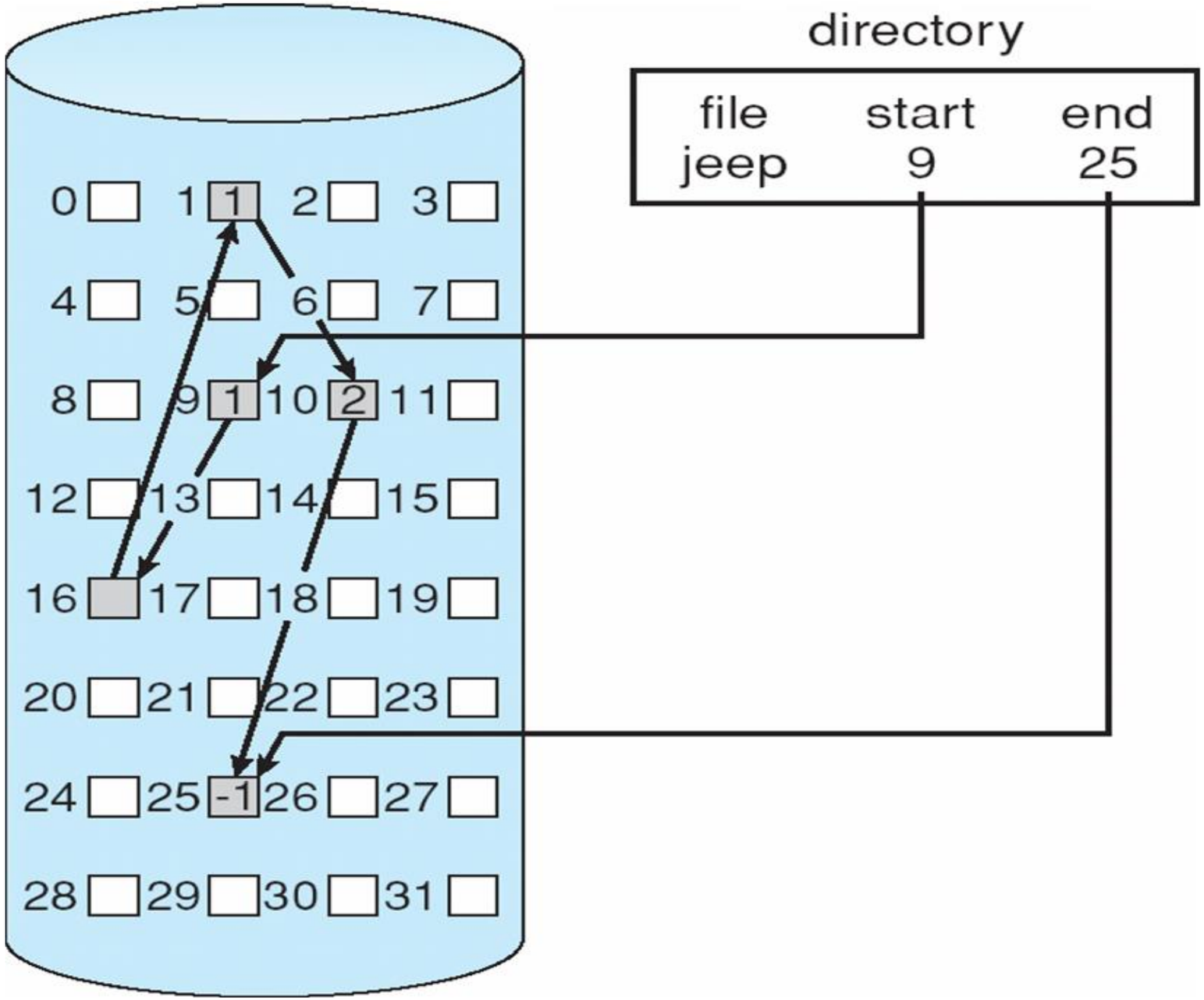
- Extent-based file systems allocate disk blocks in extents

- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

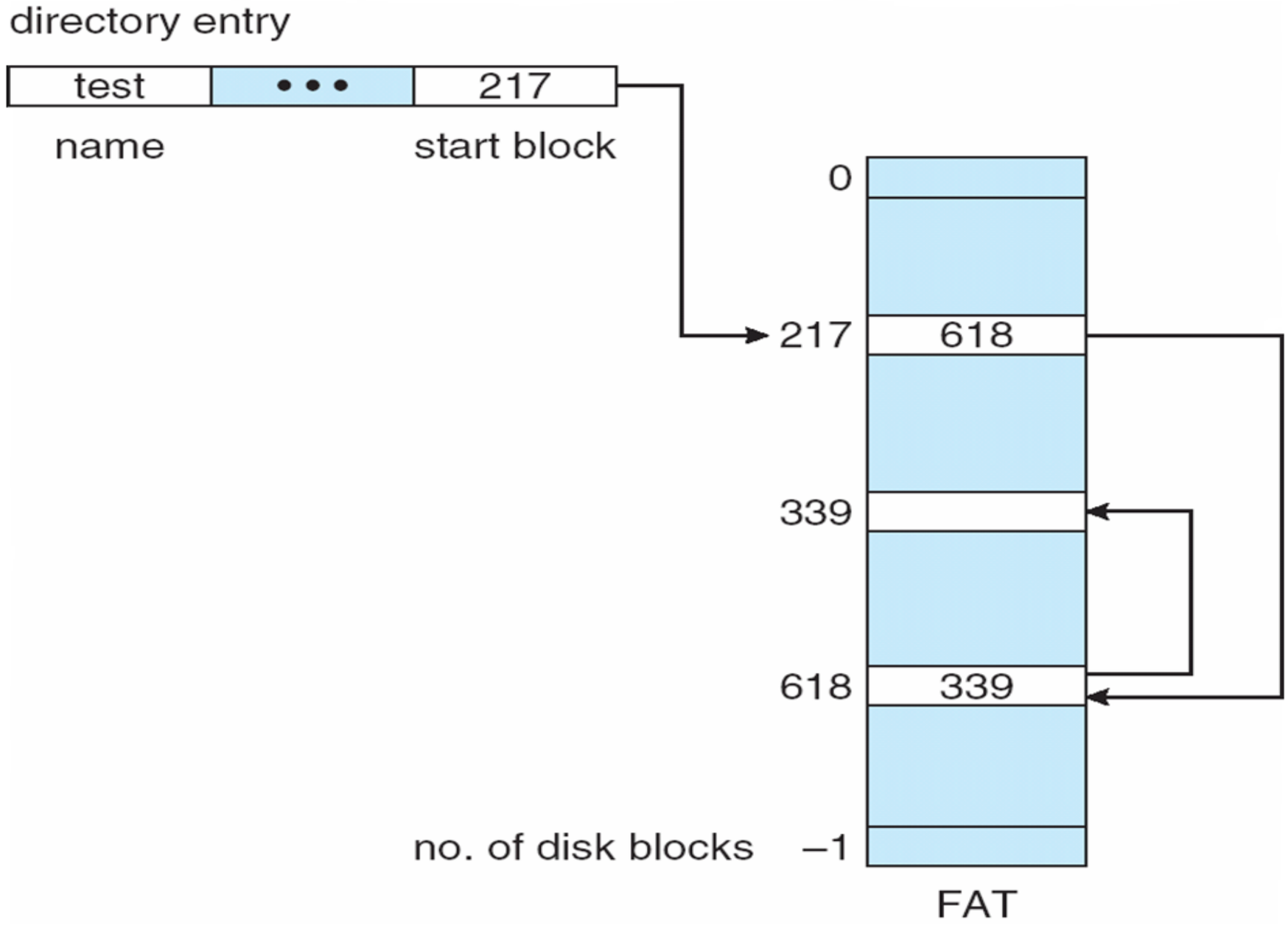
Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks(508 bytes instead of 512)
  - 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
- 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

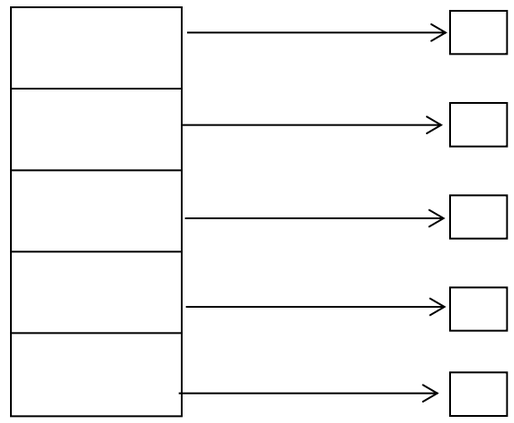


File-Allocation Table



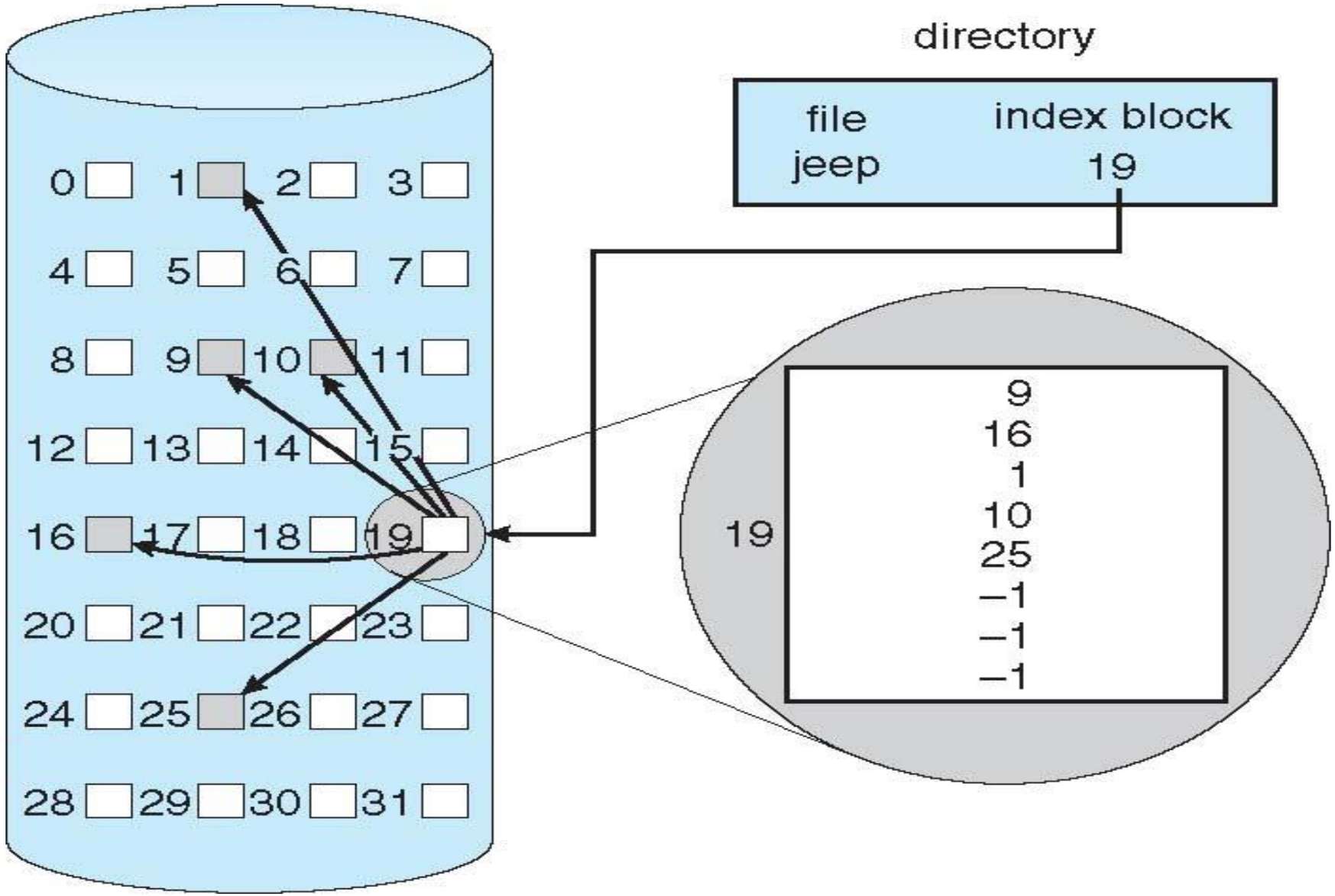
Allocation Methods - Indexed

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



index table

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
- how big the index block should be, and how it should be implemented. There are several approaches:
  - Linked Scheme
  - Multi-Level Index
  - Combined Scheme

1. Linked Scheme

- An index block is one disk block, which can be read and written in a single disk operation.
- The first index block contains some header information
- The first N block addresses, and if necessary a pointer to additional linked index blocks.

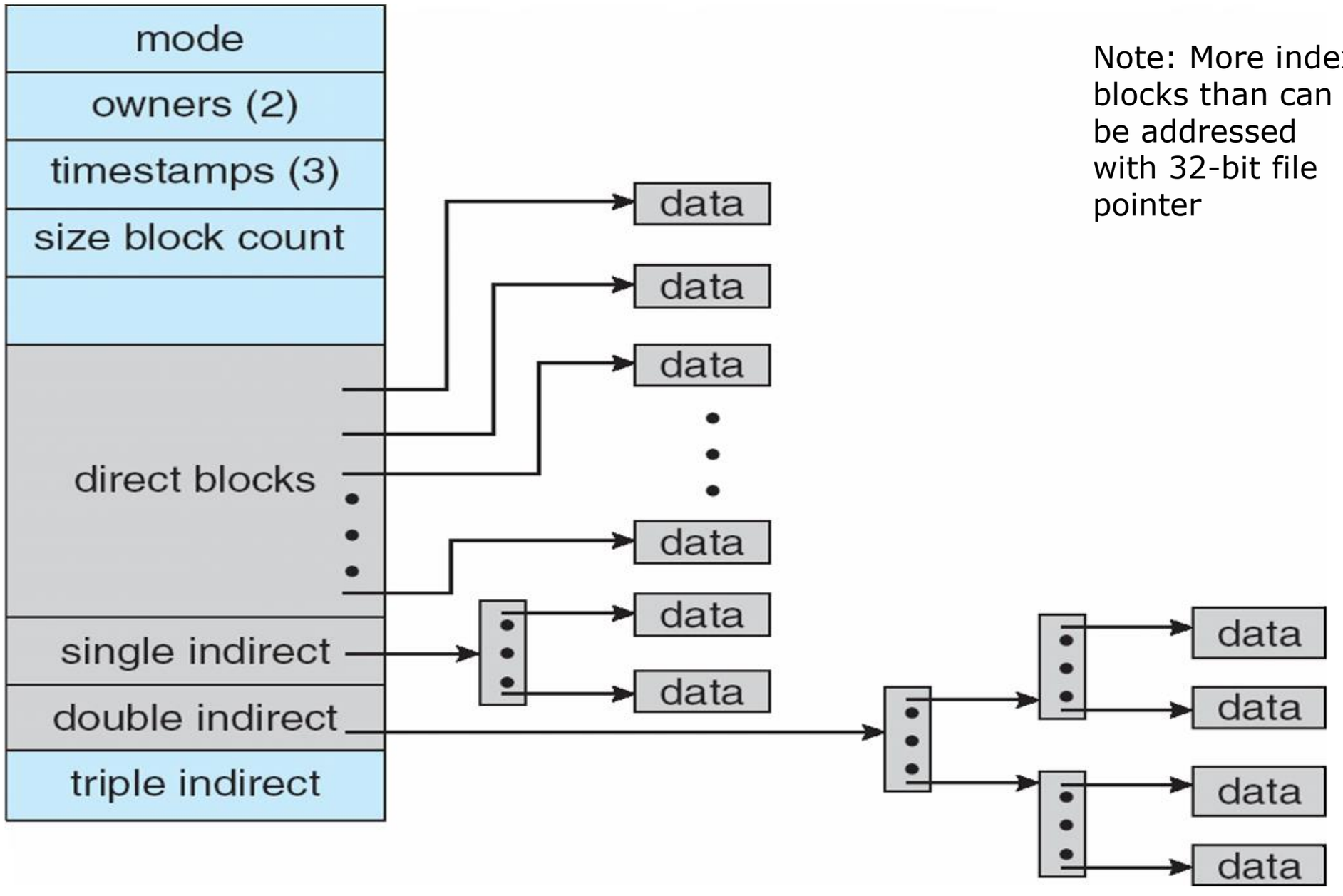
2.Multi-Level Index

- The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

3. Combined Scheme

- This is the scheme used in UNIX inodes
- The advantage of this scheme is that
  - for small files (files stored in less than 12 blocks), the data blocks are readily accessible (up to 48K with 4K block sizes);
  - files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached),
  - and huge files are still accessible using a relatively small number of disk accesses

Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)



Note: 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 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - The list which keep tracks of free space in memory
    → free space list
- To create a file
  - search the free-space list for the required amount of space and allocate that space to the new file.
  - This space is then removed from the free space list.
- When a File is deleted,
  - its disk space is added to the free-space list.

Free Space list implementation

- Bit Vector
- Linked List
- Grouping
- Counting
- Space Maps (New)

Bit vector

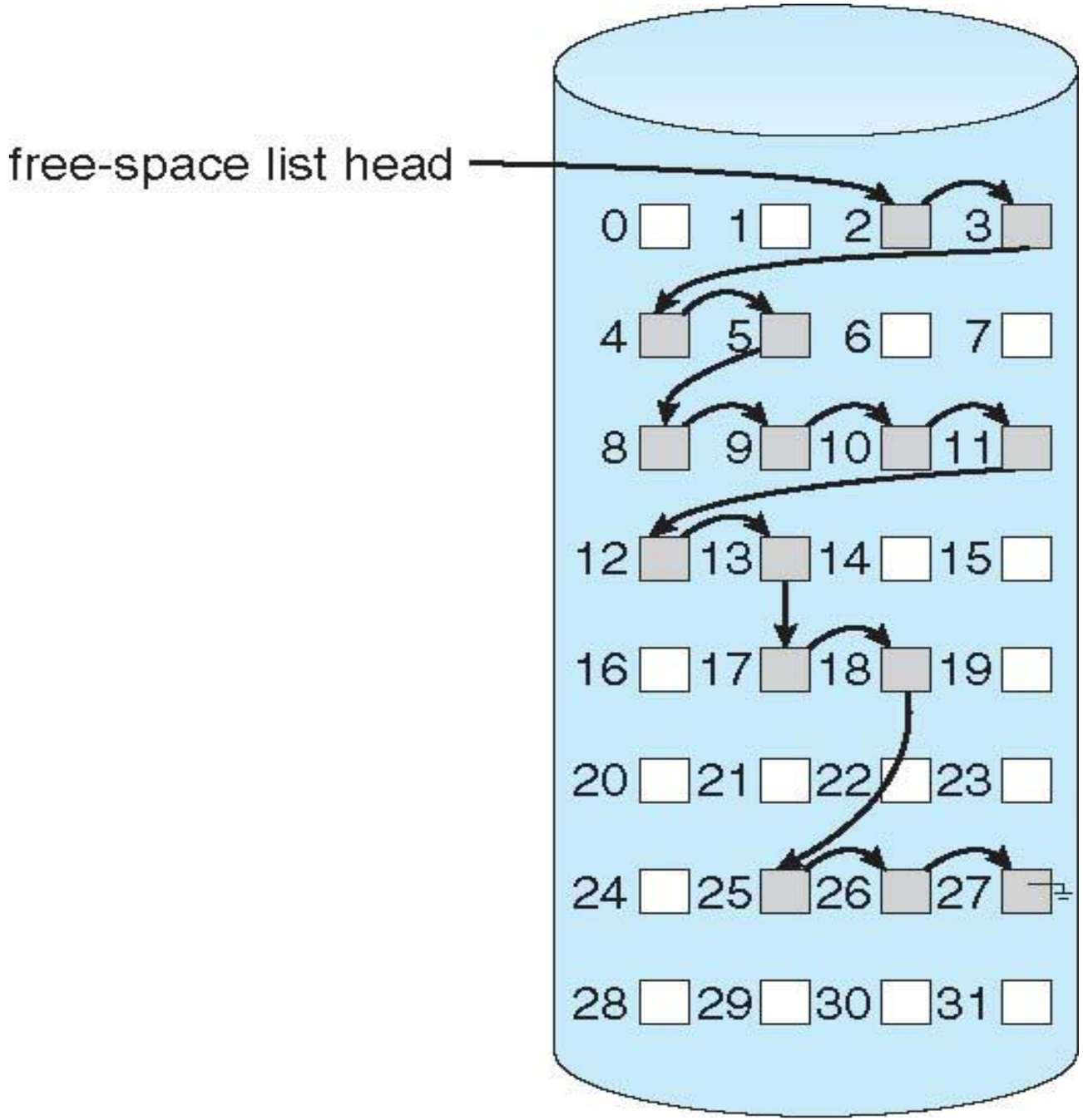
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- One simple approach is to use a bit vector, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- EG: consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be *00111100111111100011*
- Adv:
  - Easy to implement and also very efficient in finding the first free block or consecutive free blocks on the disk.
- Disadv→ Bit map requires extra space
  - Example: block size = 4KB = 2^{12} bytes
    disk size = 2^{40} bytes (1 terabyte)
    n = $2^{40}/2^{12}$ = 228 bits (or 256 MB)
    if clusters of 4 blocks -> 64MB of memory

Linked list

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations.
- Generally the system just adds and removes single blocks from the beginning of the list. (No waste of space)
- The FAT table keeps track of the free list as just one more linked list on the table. (No need to traverse the entire list (if # free blocks recorded)

ISE Dept.
Transform Here

Linked Free Space List on Disk



Grouping

- Modify linked list
- to store address of next *n-1* free blocks in first free block, The first n-1 blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on
- Adv
  - The address of a large number of free blocks can be found quickly.

Counting

– When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.

- Keep address of first free block and count of following free blocks
- Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

Space Maps

- Used in ZFS designed for huge numbers and size of files, directories and even file systems
- 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

End of Chapter 10

Mod-5 : Disk Management & Swap space management

Disk Management

- **Disk Formatting**
  - low-level formatting
  - Partition
  - logical formatting
- **Boot Block**
- **Bad Blocks**
  - sector sparing or forwarding
  - sector slipping.

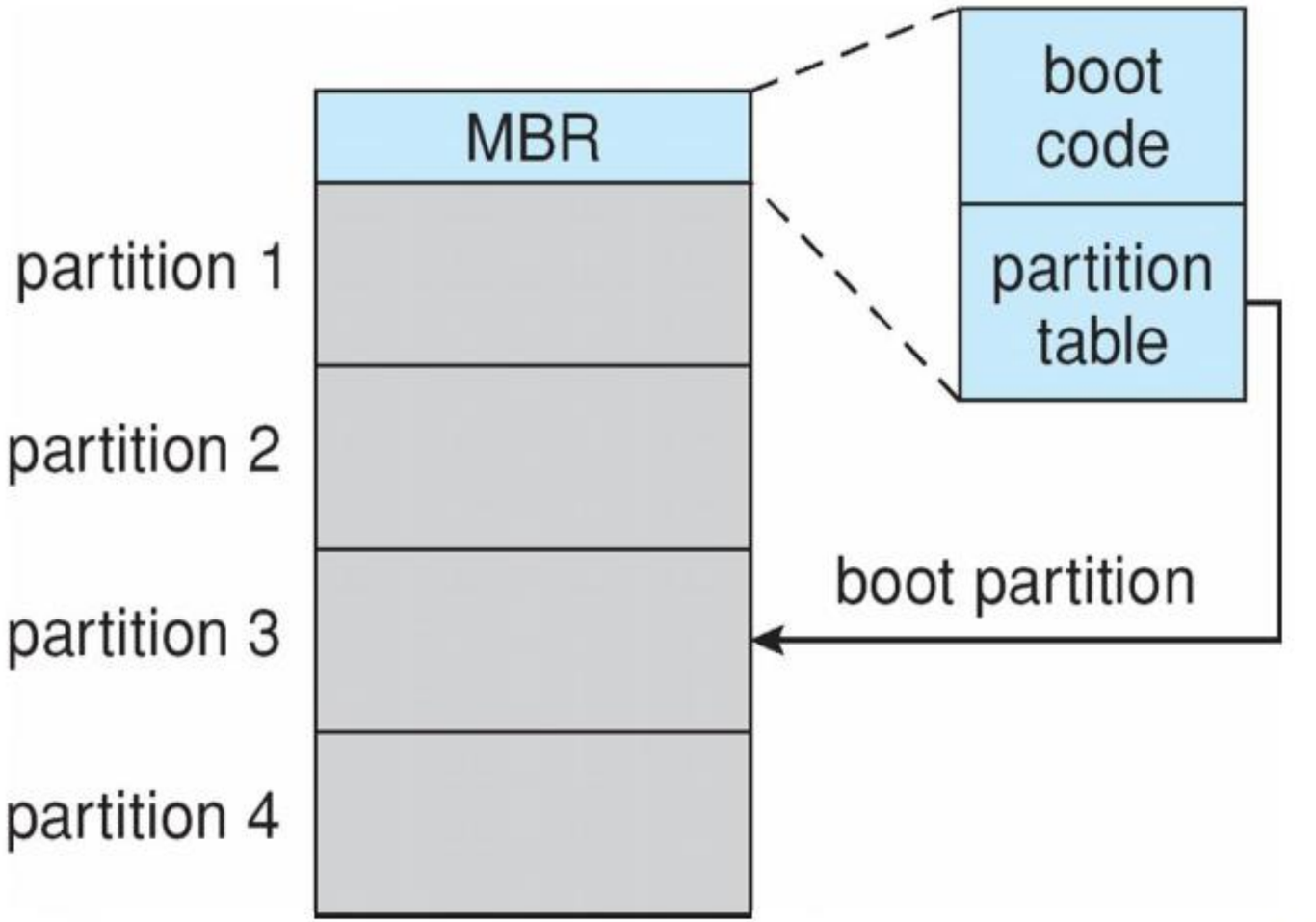
Disk Formatting

- The process of dividing the disk into sectors and filling the disk with a special data structure is called **low-level formatting.**
- Most hard disks are low-level-formatted at the factory .
- The operating system needs to record its own data structures on the disk. It does so in two steps partition **and** logical formatting.
  - **Partition**-– is to partition the disk into one or more groups of cylinders
  - **logical formatting** (or creation of a file system) - Now, the operating system stores the initial file-system data structures onto the disk
- To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.**

Boot Block

- When a computer is switched on or rebooted—it must have an initial program to run. This is called the *bootstrap* program.
- The bootstrap program
  - initializes the CPU registers, device controllers, main memory, and then starts the operating system.
  - Locates and loads the operating system from the disk
  - jumps to beginning the operating-system execution.
- The bootstrap is stored in read-only memory (ROM). →disadv
- A disk that has a boot partition is called a **boot disk** or **system disk.**

Booting from a Disk in windows



Bad Blocks

- Group of sectors that are defective are called as bad blocks.
- Different ways to overcome bad blocks are -
    - Some bad blocks are handled manually, eg. In MS-DOS.
    - Some controllers replace each bad sector logically with one of the spare sectors(extra sectors). The schemes used are **sector sparing** or **forwarding**.
    - Method utilized to push down defective sector is called **sector slipping**

SWAP SPACE MANAGEMENT

- The amount of swap space needed on a system can vary depending on
  - the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.
- The swap space can overestimated or underestimated.
- It is safer to overestimate than to underestimate the amount of swap space required.

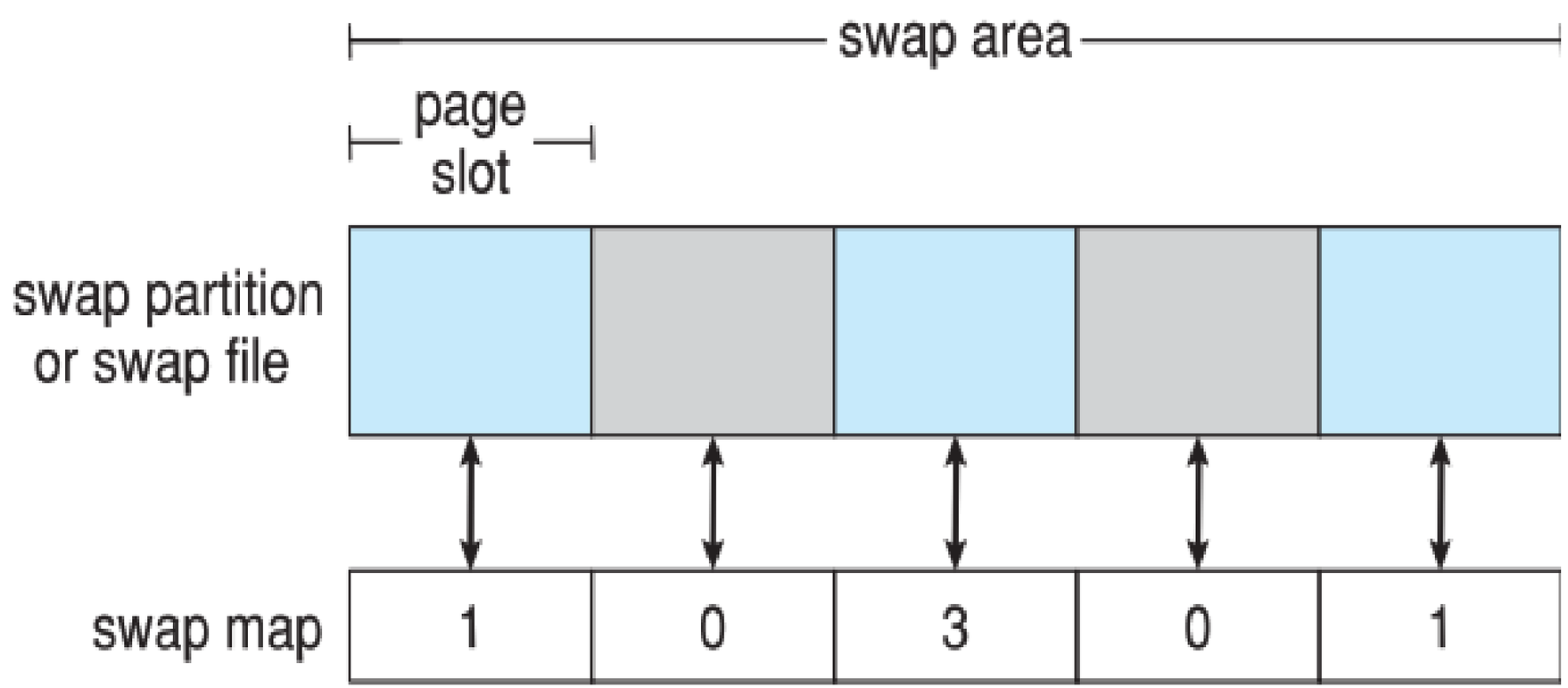
Swap-Space Location

- A swap space can reside in one of two places
  - **file system →file system routines will be used**
  - separate **disk partition →** separate swap- space storage manager is used

Swap-Space Management: An Example

- Solaris allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created.
- Linux allows one or more swap areas to be established
  - Each swap area consists of a series of 4-KB page slots
  - Associated with each swap area is a swap map —an array of integer counters, each corresponding to a page slot in the swap area
  - If the value of a counter is 0, the corresponding page slot is available.
  - Values greater than 0 indicate that the page slot is occupied by a swapped page.
  - a value of 3 indicates that the swapped page is mapped to three different processes

Data Structures for swapping in Linux systems



Mod-5 : Protection

Topics

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems

Goals of Protection

- Protection is a mechanism for **controlling the access** of programs, processes, or users to the resources defined by a computer system.
- Originally protection was conceived as an adjunct to Multiprogramming OS so that untrustworthy users ,might safely share a common logical name space.
- Modern Protection concepts have evolved to increase the reliability of any complex system that shared resources.(access restriction).
- By Providing Policies and mechanism.

Principles of Protection

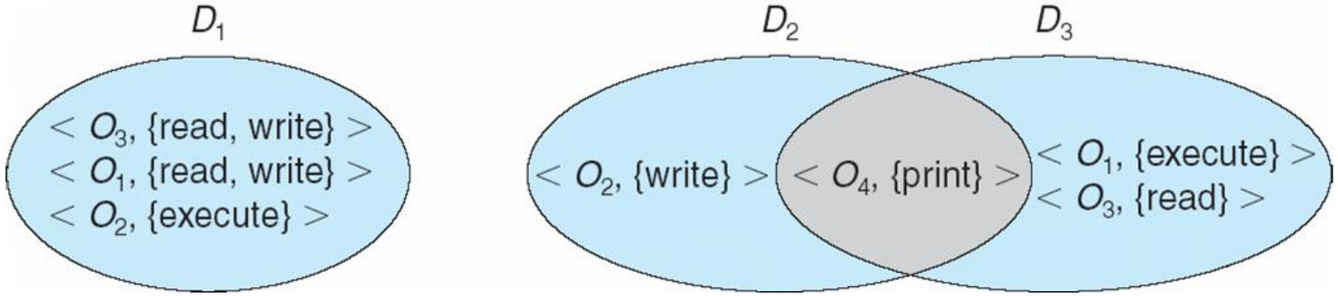
- A key, time-tested guiding principle for protection is the ‘principle of least privilege’.
- It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.
- An operating system provides mechanisms to enable privileges when they are needed and to disable them when they are not needed. Role based access control (RBAC)

DOMAIN OF PROTECTION

- A computer system is a collection of processes and objects.
  - O*bjects* are both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives)
  - **software objects** (such as files, programs, and semaphores).
- Each object (resource) has a unique name that differentiates it from all other objects in the system.
- Eg
  - CPU→ only execute
  - Mem→ read/Write
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.
- *Need to know principle*

Domain Structure

- A domain is a set of objects and types of access to these objects.
- The ability to execute an operations on an object → Access right
- Access Right=<object-name,rights-set>.
  - where *rights-set* is a subset of all valid operations that can be performed on the object.
- Domains do not need to be disjoint they may share access rights



- A domain can be realized in a variety of ways:
  - Each user may be a domain
  - Each process may be a domain
  - Each procedure may be a domain.

ACCESS MATRIX

- The model of protection can be viewed as a matrix, called an **access matrix.**
- Rows represent domains
- Columns represent objects
- *Access(i, j)* is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

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	

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix

Use of 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*

- Access matrix design separates mechanism from policy
  - Mechanism
    - Operating system provides access-matrix + rules
    - If 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

Access Matrix as Domains as Objects

domain \ object	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			

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)

Copy Right

- A right is copied from access(i,j) to access(k,j); it is then removed from access(i,j). This action is a *transfer* of a right, rather than a copy.
- Propagation of the *copy* right- limited copy. Here, when the right *R** is copied from access(i,j) to access(k,j), only the right *R* (not R**)* is created.
- A process executing in domain *Dk* cannot further copy the right *R.*

Access Matrix With *Owner* Rights

- mechanism →**addition** of new rights and **removal** of some rights (Owner)
- If access(i,j) includes the ***owner*** right, then a process executing in domain Di, can add and remove any right in any entry in column j.

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)

Access Matrix (Control)

- A mechanism is also needed to change the entries in a row.
- If access(i,j) includes the ***control*** right, then a process executing in domain *Di,* can remove any access right from row j

Access Matrix (Control)

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			

Implementation of Access Matrix

Different methods of implementing the access matrix

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

Global Table

- Simplest implementation
- A set of ordered triples *<domain, object, rights-set>* is maintained in a file
- Whenever an operation *M* is executed on an object Oj, within domain *Di,*
  - the table is searched for a triple *<Di,* Oj, *Rk>.*
  - If this triple is found, the operation is allowed to continue;
  - otherwise, an exception (or error) condition is raised.

Drawbacks

- The table is usually large and thus cannot be kept in main memory.
- Additional I/O is needed

ISE Dept.
Transform Here

Access Lists for Objects

- Each column in the access matrix can be implemented as an access list for one object
- The empty entries are discarded
- each object consists of ordered pairs *<domain, rights-set>*
- When an operation M is executed on object Oj in Di,
  - search the access list for object Oj, look for an entry *<Di,* Rj > with M ∈ Kj.
  - If the entry is found, we allow the operation;
  - if it is not, we check the default set.
  - If M is in the default set, we allow the access.
  - Otherwise, access is denied, and an exception condition occurs.
  - For efficiency, we may check the default set first and then search the access list.

Capability Lists for Domains

- Is a list of objects together with the operations allowed on those objects.
- An object is often represented by its name or address, called a **capability**
- To execute operation M on object Oj, the process executes the operation *M,* specifying the capability for object Oj as a parameter.
- Capabilities are usually distinguished from other data in one of two ways:
  i. Each object has a tag to denote its type either as a capability or as accessible data.
  ii. the address space associated with a program can be split into two parts
    a) accessible to the program and contains the program's normal data and instructions
    b) containing the capability list, is accessible only by the operating system

A Lock-Key Mechanism

- The lock-key scheme is a compromise between access lists and capability lists.
- Each object has a list of unique bit patterns, called locks.
- Similarly, each domain has a list of unique bit patterns, called keys.
- A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

ISE Dept.

Comparison

Access Matrix Type	Adv	Drawback
Global Table	simple;	however, the table can be quite large and often cannot take advantage of special groupings of objects or domains
Access lists	directly to the needs of users	• determining the set of access rights for each domain is difficult. • In a large system with long access lists, search can be time consuming.
Capability list	Useful for localizing information for a given process	• do not correspond directly to the needs of users • Revocation of capabilities, however, may be inefficient
Lock-key	• effective and flexible, depending on the length of the keys. • access privileges can be effectively revoked	

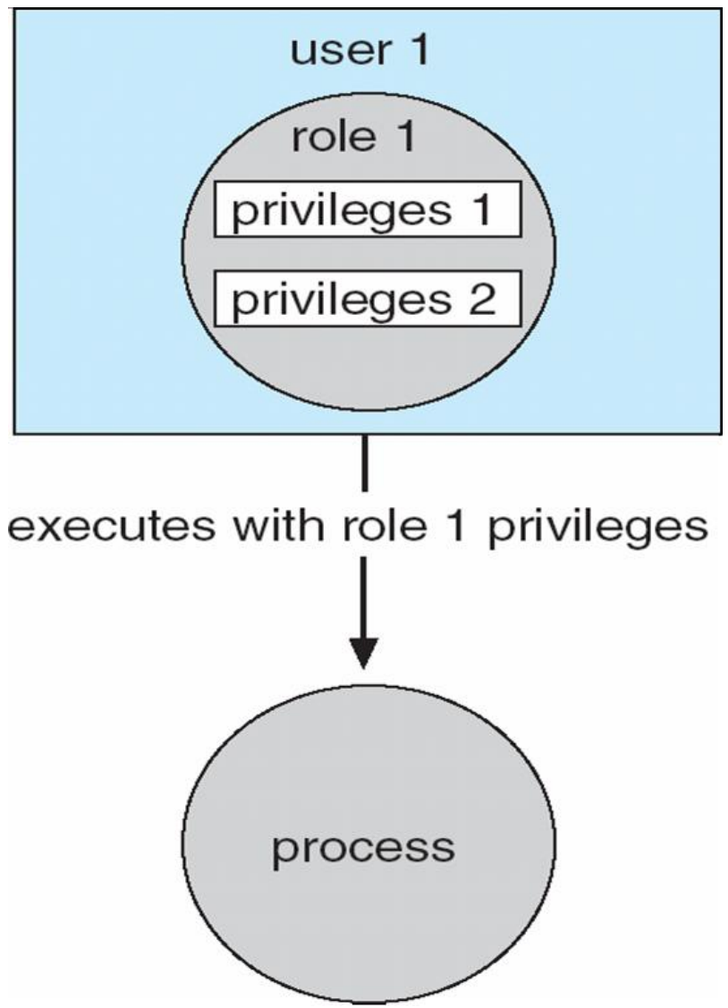
- Most systems use a combination of access lists and capabilities.
- When a process first tries to access an object, the access list is searched.
- If access is denied, an exception condition occurs.
- Otherwise, a capability is created and attached to the process.
- Additional references use the capability to demonstrate swiftly that access is allowed.
- After the last access, the capability is destroyed.
- This strategy is used in the MULTICS system and in the CAL system.

Access Control

- Protection can be applied to non-file resources
- Solaris 10 provides role-based access control (RBAC) to implement least privilege
  - Privilege is right to execute system call or use an option within a system call
  - Can be assigned to processes limiting them to exactly the access they need to perform their work
  - Privileges and programs can also be assigned to roles

Role based Access Control in Solaris10

- Users are assigned roles or can take roles based on passwords to the roles
- This implementation of privileges decreases the security risk associated with superusers and setuid programs.



Revocation of Access Rights

- In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users.
  - **Immediate Vs. delayed**
  - **Selective Vs. general.**
  - **Partial Vs. total.**
  - **Temporary Vs. permanent.**

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: Periodically, capabilities are deleted from each domain
    - If a process wants to use a capability, it may find that that capability has been deleted.
    - The process may then try to reacquire the capability.
    - If access has been revoked, the process will not be able to reacquire the capability.
  - Back-pointers: A list of pointers is maintained with each object, pointing to all capabilities associated with that object.(MULTICS)
    - When revocation is required, we can follow these pointers, changing the capabilities as necessary

Revocation of Access Rights

- Indirection: Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it.(CAL)

- Keys: A master key is associated with each object; it can be defined or replaced with the **set-key** operation ,
  - If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised
  - should not be available to all users

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

CAP's capability system is simpler and superficially less powerful than that of Hydra.

- 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